Automated Strategies for Specializing Constraint Logic Programs*

Fabio Fioravanti[1], Alberto Pettorossi[2], Maurizio Proietti[1]

(1) IASI-CNR, Viale Manzoni 30, I-00185 Roma, Italy
(2) DISP, University of Roma Tor Vergata, I-00133 Roma, Italy
{fioravanti,adp,proietti}@iasi.rm.cnr.it

Abstract. We consider the problem of specializing constraint logic programs w.r.t. constrained queries. We follow an approach based on transformation rules and strategies. The use of the rules ensures that the specialized program is equivalent to the initial program w.r.t. a given constrained query. The strategies guide the application of the rules so to derive an efficient specialized program. In this paper we address various issues concerning the development of an automated transformation strategy. In particular, we consider the problems of when and how we should unfold, replace constraints, introduce generalized clauses, and apply the contextual specialization rule. We propose a solution to these problems by adapting to our framework various techniques developed in the field of constraint programming, partial evaluation, and abstract interpretation. In particular, we use: (i) suitable solvers for simplifying constraints, (ii) well-quasi-orders for ensuring the termination of the unfoldings and for activating clause generalizations, and (iii) widening operators for ensuring the termination of the generalization process.

1 Introduction

Program specialization is a powerful methodology for automated software engineering which allows for program reuse. Program specialization, in fact, may adapt a generic program to the particular context where it has to be used and this adaptation process may be done via automatic or semi automatic techniques. One such technique is *partial evaluation*. Partial evaluation has been proposed and studied for various programming languages [12] and it has been shown to be very powerful. In particular, via partial evaluation one can perform many sophisticated program optimizations while taking advantage of the context of use of the programs. In this paper we address the problem of automating some techniques for the *contextual specialization* [7] of *constraint logic programs* over a domain $\mathcal{D}$ (CLP($\mathcal{D}$), for short) [10]. Contextual specialization is defined as follows: Given a CLP($\mathcal{D}$) program P and a constrained atom c, A, we want to derive a program P_s and an atom A_s such that, for every variable assignment σ

* In: Kung-Kiu Lau (Ed.): Proceedings of LOPSTR 2000, LNCS 2042, pp. 125-146, Springer, 2001.

that assigns an element of $\mathcal{D}$ to each free variable in the constrained atom c, A, we have that:

$$lm(P, \mathcal{D}) \models_\sigma c, A \quad \text{iff} \quad lm(P_s, \mathcal{D}) \models_\sigma c, A_s \quad (1\text{: } \textit{Contextual Specialization})$$

where $lm(P, \mathcal{D})$ denotes the *least $\mathcal{D}$-model* of P [11].
Contextual specialization is more general than the partial evaluation of CLP($\mathcal{D}$) programs based on Lloyd and Shepherdson's approach [15, 16, 24]. Indeed, partial evaluation is defined as follows: Given a CLP($\mathcal{D}$) program P and a constrained atom c, A, we want to derive a program P_{pe} and an atom A_{pe} such that, for every variable assignment σ, we have that:

$$lm(P, \mathcal{D}) \models_\sigma c, A \qquad \text{iff} \qquad lm(P_{pe}, \mathcal{D}) \models_\sigma A_{pe} \qquad (2\text{: } \textit{Partial Evaluation})$$

The following example illustrates the difference between contextual specialization of CLP programs and partial evaluation. Let us consider the following CLP($\mathcal{R}$) program P over the domain $\mathcal{R}$ of real numbers:

$$p(X) \leftarrow X \geq 0,\ q(X) \qquad \text{(Program } P\text{)}$$

where q is a predicate which does not depend on p. By contextual specialization of P w.r.t. the constrained atom $X \geq 3, p(X)$ we derive the program P_s:

$$p_s(X) \leftarrow q(X) \qquad \text{(Program } P_s\text{)}$$

together with the atom $p_s(X)$.

Instead, by partial evaluation of program P w.r.t. the same constrained atom $X \geq 3, p(X)$ we derive the program P_{pe}:

$$p_{pe}(X) \leftarrow X \geq 3,\ q(X) \qquad \text{(Program } P_{pe}\text{)}$$

together with the atom $p_{pe}(X)$.

Thus, the partially evaluated program P_{pe} is less efficient than the program P_s derived by contextual specialization, because P_{pe} redundantly checks whether or not the constraint $X \geq 3$ holds.

In [7] we have introduced some *transformation rules* and semi automatic *strategies* for contextual specialization. Here we address various issues concerning the full automation of our strategies. In particular, we consider the problems of: (i) when and how to unfold, (ii) when and how to generalize, and (iii) when and how to apply the contextual specialization rule. Our automatic strategy for contextual specialization of CLP($\mathcal{D}$) programs is based on concepts borrowed from the fields of constraint programming, partial evaluation, and abstract interpretation. In particular, we consider: (i) suitable *solvers* for simplifying constraints [10], (ii) *well-quasi-orders* for ensuring the termination of the unfolding process and for activating the clause generalization process [13, 14, 23], and (iii) *widening operators* [4] for ensuring the termination of that generalization process.

2 Rules for Contextual Specialization of CLP programs

In this section we recall the transformation rules which we use for specializing CLP($\mathcal{D}$) programs. Our rules are a slight modification of the unfold/fold rules

considered in [1, 6, 17] and they are designed for performing contextual specialization.

Let us begin by presenting some preliminary notions and notational conventions we will use. For notions not defined here the reader may refer to [10]. Throughout the paper we will consider a fixed first-order language $\mathcal{L}$. Let *Vars* be the (possibly infinite) set of variables in $\mathcal{L}$. Given a formula φ, the set of variables and the set of free variables of φ are denoted by $vars(\varphi)$ and $FV(\varphi)$, respectively. We will write $\forall X_1 \ldots X_m \, \psi$ also as $\forall X \, \psi$, where X denotes the set $\{X_1, \ldots, X_m\}$. Analogously for $\exists$, instead of $\forall$. Similarly, we will write the atom $p(t_1, \ldots, t_m)$ also as $p(t)$, where t denotes the sequence $t_1, \ldots, t_m$ of terms.

Constraints over the domain $\mathcal{D}$ are denoted by the letters $c, d, \ldots$, possibly with subscripts. We assume that the set $\mathcal{C}$ of constraints over $\mathcal{D}$ contains a given set of *primitive constraints* including *true*, *false*, and the equalities between terms. We assume that $\mathcal{C}$ is closed w.r.t. conjunction (which we denote by comma) and existential quantification. *Atoms* are denoted by the letters $H, A, B, \ldots$, possibly with subscripts. A *constrained atom* is the conjunction of a constraint and an atom. *Goals* are (possibly empty) conjunctions of atoms, and they are denoted by G, possibly with subscripts. A *constrained goal* is the conjunction of a constraint and a goal, and it is denoted by K, possibly with subscripts. Conjunction is commutative and, thus, the order of constraints and atoms in the body of a clause is immaterial. The empty conjunction (of constraints or atoms) is *true*.

We assume that every clause $A_0 \leftarrow c, A_1, \ldots, A_n$ of a CLP($\mathcal{D}$) program is in *canonical form*, that is, for every $i = 0, \ldots, n$, the atom A_i is of the form $p(X_1, \ldots, X_m)$, where $X_1, \ldots, X_m$ are distinct variables. Similarly, we assume that every constrained atom is in canonical form. A *variable renaming* is a bijective mapping from *Vars* to itself. The application of a variable renaming ρ to a syntactic expression e is denoted by $e\rho$, and $e\rho$ is obtained from e by replacing each (bound or free) variable of e by the variable $X\rho$. The expression $e\rho$ is said to be a *variant* of e. A clause C is *renamed apart* by applying a variable renaming ρ such that the (bound or free) variables of $C\rho$ do not occur in other clauses considered so far.

Any clause γ which is not in canonical form, can be transformed into a clause in canonical form which is equivalent to γ w.r.t. the least $\mathcal{D}$-model semantics. This can be done by adding suitable equality constraints. For example, the clause: $p(X+1) \leftarrow X \leq 0, \, r(X-1)$ can be transformed into the equivalent clause: $p(Y) \leftarrow X \leq 0, \, r(Z), \, Y = X+1, \, Z = X-1$, which is in canonical form.

For a clause γ of the form: $H \leftarrow K$, we say that H is the head of γ, denoted by $hd(\gamma)$, and K is the body of γ, denoted by $bd(\gamma)$. Given the constrained goals K_1 and K_2, and the clause γ of the form: $H \leftarrow K_1, K_2$, the set of the *linking variables* of K_1 in γ is $FV(K_1) \cap FV(H, K_2)$.

The semantics of a CLP($\mathcal{D}$) program P is defined as the *least $\mathcal{D}$-model* of P, denoted $lm(P, \mathcal{D})$ [11]. We assume that there exists a computable total function *solve*: $\mathcal{C} \times \mathcal{P}_{fin}(Vars) \rightarrow \mathcal{C}$ which is *sound* w.r.t. constraint equivalence, in the sense that, for every constraint c_1 and finite set X of variables, if $solve(c_1, X) =$

c_2 then $\mathcal{D} \models \forall X((\exists Y\, c_1) \leftrightarrow c_2)$ where $FV(c_2) \subseteq (FV(c_1) \cap X)$ and $Y = FV(c_1) - X$.

Moreover, we require that *solve* is *complete* w.r.t. satisfiability in the sense that, for any constraint c such that $Y = FV(c)$:
(i) $solve(c, \emptyset) = true$ if c is *satisfiable*, that is, $\mathcal{D} \models \exists Y\, c$, and
(ii) $solve(c, \emptyset) = false$ if c is *unsatisfiable*, that is, $\mathcal{D} \not\models \exists Y\, c$.
In (i) and (ii) 'if' can be replaced by 'iff' because *solve* is sound w.r.t. constraint equivalence. The completeness of the *solve* function w.r.t. satisfiability guarantees that satisfiability tests which are often required by our techniques, are decidable and they can indeed be performed by the *solve* function. Notice that we need not make any assumption about the completeness of *solve* w.r.t. non-empty sets of variables.

The process of *specializing* a given program P whereby deriving program P_s, can be formalized as a sequence $P_0, \ldots, P_n$ of programs, called a *transformation sequence*, where $P_0 = P$, $P_n = P_s$ and, for $k = 0, \ldots, n-1$, program P_{k+1} is obtained from program P_k by applying one of the following transformation rules: (R1) *constrained atomic definition*, (R2) *unfolding*, (R3) *constrained atomic folding*, (R4) *clause removal*, and (R5) *contextual constraint replacement*. Below we will present also the extra rule R6 which is an instance of rule R5.

R1. Constrained Atomic Definition. By *constrained atomic definition* (or *definition*, for short), we introduce the new clause

$$\delta: \quad newp(X) \leftarrow c, A$$

which is said to be a *definition*, where: (i) *newp* is a predicate symbol not occurring in $P_0, \ldots, P_k$, (ii) X is a sequence of distinct variables occurring in c, A, and (iii) the predicate symbol of A occurs in P_0. From program P_k we derive the new program P_{k+1} which is $P_k \cup \{\delta\}$.
For $i \geq 0$, $Defs_i$ is the set of definitions introduced during the transformation sequence $P_0, \ldots, P_i$. Obviously, $Defs_0 = \emptyset$.

R2. Unfolding. Let $\gamma: \quad H \leftarrow c, G_1, A, G_2$ be a renamed apart clause of P_k and let $\{A_j \leftarrow c_j, G_j \mid j = 1, \ldots, m\}$ be the set of *all* clauses in P_k such that A and A_j have the same predicate symbol. For $j = 1, \ldots, m$, let us consider the clause

$$\gamma_j: \quad H \leftarrow c,\ A = A_j,\ c_j,\ G_1, G_j, G_2$$

where $A = A_j$ stands for the conjunction of the equalities between the corresponding arguments. Then, by *unfolding* clause γ w.r.t. the atom A, from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_j \mid j = 1, \ldots, m\}$.

R3. Constrained Atomic Folding. Let $\gamma: \quad A \leftarrow c, G_1, B, G_2$ be a clause of P_k. Let $\delta: \quad newp(X) \leftarrow d, B$ be a variant of a clause in $Defs_k$. Suppose that: (i) $\mathcal{D} \models \forall Y\, (c \rightarrow d)$, where $Y = FV(c, d)$, and (ii) no variable in $FV(\delta) - X$ occurs in $FV(A, c, G_1, G_2)$. By *folding* clause γ w.r.t. atom B using δ, we derive the new clause

$$\gamma_f: \quad A \leftarrow c, G_1, newp(X), G_2$$

and from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{\gamma_f\}$.

In this rule R3 condition (i) may be replaced by the following weaker, but more complex condition: (i*) $\mathcal{D} \models \forall Y\, (c \rightarrow \exists Z\, d)$, where $Z = FV(d) - (X \cup vars(B))$ and $Y = FV(c,d) - Z$. However, by a suitable application of the Constraint Replacement rule R6 below, from clause δ we can derive a clause η of the form: $newp(X) \leftarrow (\exists Z\, d), B$ such that condition (i*) holds for γ and δ iff condition (i) holds for γ and η.

R4. Clause Removal. Let $\gamma:\ A \leftarrow c, G$ be a clause of P_k. If the constraint c is *unsatisfiable*, that is, $solve(c, \emptyset) = false$, then from program P_k we derive the new program P_{k+1} which is $P_k - \{\gamma\}$.

R5. Contextual Constraint Replacement. Let $\mathbf{C}$ be a set of constrained atoms. Let γ be a renamed apart clause in P_k of the form: $p(U) \leftarrow c_1, G$. Suppose that for some constraint c_2, we have that for every constrained atom $c, p(V)$ in $\mathbf{C}$,

$$\mathcal{D} \models \forall X\, ((c, U = V)\ \rightarrow\ (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2))$$

where: (i) $Y = FV(c_1) - vars(U, G)$, (ii) $Z = FV(c_2) - vars(U, G)$, and (iii) $X = FV(c, U, V, c_1, c_2) - (Y \cup Z)$. Then, we derive program P_{k+1} from program P_k by replacing clause γ by the clause: $p(U) \leftarrow c_2, G$. In this case we say that P_{k+1} has been derived from P_k by contextual constraint replacement w.r.t. $\mathbf{C}$.

The following rule is an instance of rule R5 for $\mathbf{C} = \{true, p(U)\}$.

R6. Constraint Replacement. Let $\gamma:\ A \leftarrow c_1, G$ be a renamed apart clause of P_k. Assume that $\mathcal{D} \models \forall X (\exists Y\, c_1 \leftrightarrow \exists Z\, c_2)$ where: (i) $Y = FV(c_1) - vars(A, G)$, (ii) $Z = FV(c_2) - vars(A, G)$, and (iii) $X = FV(c_1, c_2) - (Y \cup Z)$. Then from program P_k we derive the new program P_{k+1} which is $(P_k - \{\gamma\}) \cup \{A \leftarrow c_2, G\}$.

In the *Contextual Specialization Strategy* of Section 4, we will use this rule R6 by replacing a clause γ of the form $A \leftarrow c_1, G$ by the clause $A \leftarrow solve(c_1, X), G$, where X is the set of the linking variables of c_1 in γ.

3 Well-Quasi-Orders and Clause Generalization

In this section we introduce the notions of: (i) *well-quasi-orders* over constrained goals, and (ii) *clause generalization*, which we will use in our program specialization strategy. These notions are extensions to the case of CLP($\mathcal{D}$) programs of similar notions considered in the case of partial evaluation of functional and logic programs (see, for instance, [13, 14, 23]).

Definition 1. A *well-quasi-order* (*wqo*, for short) over the set of constrained goals is a reflexive, transitive, binary relation $\preceq$ such that for every infinite sequence $\{K_i \mid i = 0, 1, \ldots\}$ of constrained goals there exist two natural numbers i and j such that $i < j$ and $K_i \preceq K_j$ [5]. Given two constrained goals K_1 and K_2, if $K_1 \preceq K_2$ we say that K_1 is *embedded in* K_2.

Various examples of wqo's that are used for ensuring the termination of the unfolding process during the partial evaluation of logic and functional programs, can be found in [13, 23]. For our specialization example of Section 5 we will use

the simple wqo $\preceq_L$ defined as follows: given two constrained goals K_1 and K_2, we have that $K_1 \preceq_L K_2$ iff the leftmost atom (in the textual order) in K_1 and the leftmost atom (in the textual order) in K_2 have the same predicate symbol.

Definition 2. [*Constraint Lattice and Widening*] Given the set $\mathcal{C}$ of constraints over $\mathcal{D}$, we consider the partial order $\langle \mathcal{C}, \sqsubseteq \rangle$ such that for any two constraints c_1 and c_2 in $\mathcal{C}$, $c_1 \sqsubseteq c_2$ iff $\mathcal{D} \models \forall X(c_1 \rightarrow c_2)$ where $X = FV(c_1, c_2)$. We assume that $\langle \mathcal{C}, \sqsubseteq \rangle$ is a lattice, where: (i) the least element is *false*, (ii) the greatest element is *true*, (iii) the *least upper bound* of two constraints c_1 and c_2 is denoted by $c_1 \sqcup c_2$, and (iv) the *greatest lower bound* of two constraints c_1 and c_2 is their conjunction c_1, c_2.
A *widening operator* (see also [4]) is a binary operator ∇ between constraints such that:
(W1) $(c_1 \sqcup c_2) \sqsubseteq (c_1 \nabla c_2)$, and
(W2) for every infinite sequence $\{c_i \,|\, i = 0, 1, \ldots\}$ of constraints, the infinite sequence $\{d_i \,|\, i = 0, 1, \ldots\}$ of constraints where $d_0 = c_0$ and $d_{i+1} = d_i \nabla c_{i+1}$, *stabilizes*, that is, $\exists h \geq 0 \; \forall k \geq h \; (d_k = d_{k+1})$. Notice that, in general, ∇ is not commutative.
We say that the widening operator ∇ *agrees with* the wqo $\preceq$ iff for every atom $q(X)$ and constraints c and d, we have that $c \nabla d, q(X) \preceq c, q(X)$.

We now introduce the notion of *clause generalization*, which is based upon the widening operator ∇ and it will be used in our strategy for contextual specialization in the next section.

Definition 3. [*Clause Generalization*] Given two clauses α: $new1(U) \leftarrow c_1,\ q(X)$ and η: $new2(V) \leftarrow c_2,\ q(X)$, where each variable in $(U, FV(c_1), V, FV(c_2))$ is in X as well, we define the *generalization of* α *w.r.t.* η, denoted by $gen(\alpha, \eta)$, to be the clause: $genp(W) \leftarrow c_1 \nabla c_2,\ q(X)$, where *genp* is a new predicate symbol and W is a sequence of the distinct variables occurring in (U, V).

Example 1. [*Linear equations and inequations over the reals*] Let us consider the domain $\mathcal{R}_{lin}$ of conjunctions of linear equations ($=$) and inequations ($<, \leq, >, \geq$) over real numbers. $\mathcal{R}_{lin}$ is a lattice whose least upper bound operation is defined by the convex hull construction. Let us now introduce a widening operator for $\mathcal{R}_{lin}$ which we will use in our program specialization example of Section 5. Given a constraint c, let $ineq(c)$ be the constraint obtained by replacing every equation $t_1 = t_2$ in c by the conjunction of the two inequations $t_1 \geq t_2$, $t_1 \leq t_2$. Given two constraints c and d such that $ineq(c)$ is the conjunction of the primitive constraints $c_1, \ldots, c_n$, we define the widening $c \nabla d$ of c and d to be the conjunction of all c_i's, with $0 \leq i \leq n$, such that $d \sqsubseteq c_i$.

This widening operator satisfies Condition W1 because, by construction, $c \sqsubseteq c \nabla d$ and $d \sqsubseteq c \nabla d$, that is, $c \nabla d$ is an upper bound of c, d. Thus, we have that: $(c \sqcup d) \sqsubseteq (c \nabla d)$. Also Condition W2 holds for ∇, because for every constraint c and d the number of primitive constraints in $ineq(c)$ is not smaller than the number of primitive constraints in $ineq(c \nabla d)$.
Here is an example of clause generalization. Given the following two clauses:

α. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{\geq}0,\ mmod(I, J, M)$
η. $newp(I, J, M) \leftarrow I{=}1,\ J{>}0,\ mmod(I, J, M)$

$gen(\alpha, \eta)$ is $genp(I, J, M) \leftarrow I{\geq}0,\ J{\geq}0,\ mmod(I, J, M)$, because we have that: (i) $ineq(I{=}0,\ J{\geq}0) = (I{\geq}0,\ I{\leq}0,\ J{\geq}0)$ and (ii) $(ineq(I{=}0,\ J{\geq}0))\ \nabla\ (I{=}1,\ J{>}0) = (I{\geq}0,\ J{\geq}0)$ because it is not the case that $(I{=}1,\ J{>}0) \sqsubseteq (I{\leq}0)$.

Notice that the widening operator ∇ defined in this example agrees with the wqo $\preceq_L$ defined above. Indeed, for every atom $q(X)$ and constraints c, and d, we have that $c \nabla d,\ q(X)\ \preceq_L\ c,\ q(X)$. □

4 An Automated Strategy for Contextual Specialization

We now describe a general strategy for deriving efficient CLP($\mathcal{D}$) programs by specialization. Our strategy is a generalization of the strategies for the partial evaluation of definite logic programs presented in [8, 20, 22].

Our strategy is parameterized by: (i) the function *solve* which is used for the application of the constraint replacement rule, (ii) an *unfolding function Unfold* for guiding the unfolding process, (iii) a well-quasi-order $\preceq_u$ over constrained goals which tells us when to terminate the unfolding process, (iv) a clause generalization function *gen*, with its associated widening operator ∇, and (v) a well-quasi-order $\preceq_g$ over constrained atoms which tells us when to activate the clause generalization process.

In the following we introduce two procedures, called (i) *Unfold-Replace* procedure and (ii) *Define-Fold* procedure, which are used below to define the Contextual Specialization Strategy. When presenting these procedures and the strategy we use a tree *Defstree*, called *definition tree,* whose nodes are the clauses introduced by the definition rule during program specialization. Moreover, for each clause δ in *Defstree* we consider a tree *Utree*(δ), called *unfolding tree*. The root of *Utree*(δ) is δ itself, and the nodes of *Utree*(δ) are the clauses derived from δ by applying the unfolding and constraint replacement rules. The usual relation of *ancestor* between nodes in a tree gives us the relation of ancestor between clauses in *Defstree* and also between clauses in *Utree*(δ).

4.1 The Unfold-Replace Procedure

Let us first introduce some notions which will be useful in the definition of the *Unfold-Replace* procedure.
Let P be a CLP program and Γ be a set of clauses.

(a) A clause of the form $H \leftarrow c, A_1, \ldots, A_n$ has *non-failing* body iff c is satisfiable, and for $i = 1, \ldots, n$, the predicate of A_i occurs in the head of at least one clause in P;

(b) an *unfolding function* is a partial function *Unfold*, which takes a clause λ and an unfolding tree T of which λ is a leaf, and returns the set *Unfold*(λ, T) of clauses obtained by unfolding the clause λ in P w.r.t. an atom A in its body. We assume that the unfolding function *Unfold* is associated with the wqo $\preceq_u$.

Unfold(λ, T) is defined iff (i) λ is a leaf of T, (ii) the body of λ is non-failing and it contains at least one atom, and (iii) there exists no ancestor α of λ in T such that $bd(\alpha) \preceq_u bd(\lambda)$. Notice that if T consists of the root clause λ only, and λ has non-failing body, then *Unfold*(λ, T) is defined;

(c) *Replace*(Γ) denotes the set of clauses Γ' obtained by applying the Constraint Replacement rule to each clause γ in Γ as follows:
If γ is of the form $H \leftarrow c, G$ then the clause $H \leftarrow solve(c, X), G$ is in Γ', where X is the set of the linking variables of c in γ.

The *Unfold-Replace* procedure which we now describe, takes as input a set *NewDefs* of definition clauses and constructs a forest *UForest* of unfolding trees, one for each definition in *NewDefs*. The *Unfold-Replace* procedure is parametric w.r.t. the choice of the unfolding function *Unfold* and its associated well-quasi-order $\preceq_u$ on constrained goals.

The Procedure *Unfold-Replace*(*NewDefs*, *UForest*)**.**
Input: a set *NewDefs* of definition clauses.
Output: a forest *UForest* of unfolding trees.
for each clause $\delta \in$ *NewDefs* **do**
  Let *Utree*(δ) be the root clause δ;
  while *Unfold*$(\lambda$, *Utree*$(\delta))$ is defined for some leaf clause λ of *Utree*(δ) **do**
    $\Gamma 1 :=$ *Unfold*$(\lambda$, *Utree*$(\delta))$;
    $\Gamma 2 :=$ *Replace*$(\Gamma 1)$;
    Expand *Utree*(δ) by making every clause in $\Gamma 2$ a son of λ
  end-while
end-for
UForest $:= \{$ *Utree*$(\delta) \mid \delta \in$ *NewDefs*$\}$ □

Notice that, by the properties of the unfolding function *Unfold*, for each clause δ in *NewDefs* with non-failing body, the tree *Utree*(δ) is constructed by applying the unfolding rule at least once.

4.2 The Define-Fold Procedure

The *Define-Fold* procedure takes as input a forest *UForest* of unfolding trees and a definition tree *Defstree* and it produces as output: (i) a possibly empty set *NewDefs* of new definition clauses, and (ii) a set *FoldedCls* of clauses derived from the leaves of *UForest* which have non-failing bodies, by a (possibly empty) sequence of applications of the folding rule. The new definition clauses are added to the tree *Defstree* as new leaves.

We now introduce the notion of *folding equivalence* between definition clauses. This notion is used to avoid the introduction of unnecessary new definition clauses.

Definition 4. Given two clauses, δ_1 of the form $H_1 \leftarrow c_1, A_1$ and δ_2 of the form $H_2 \leftarrow c_2, A_2$, we say that δ_1 and δ_2 are *folding equivalent* iff there exists a variable renaming ρ such that (i) $A_1\rho = A_2$, (ii) $\mathcal{D} \models \forall X(c_1\rho \leftrightarrow c_2)$ where $X = FV(c_1\rho, c_2)$, and (iii) $vars(H_1\rho) = vars(H_2)$.

For example, the clauses $new1(X,Y) \leftarrow X > Y, p(Y)$ and $new2(V,U) \leftarrow V < U, p(V)$ are folding equivalent. We have that, if δ_1 and δ_2 are folding equivalent clauses, then a clause can be folded using δ_1 iff it can be folded using δ_2.

Notice that by the definition of the generalization function given in Section 3 we have the following property, which will be useful for proving the termination of the contextual specialization strategy: for any given clause γ_0 and infinite sequence $\delta_0, \delta_1, \ldots$ of clauses, if $\gamma_0, \gamma_1, \ldots$ is the infinite sequence of clauses such that, for all $i \geq 0$, $\gamma_{i+1} = gen(\gamma_i, \delta_i)$, then there exist two clauses γ_j and γ_k, with $0 \leq j < k$, which are folding equivalent.

For our Contextual Specialization Strategy we also need the following notion of *call patterns* of a clause or a set of clauses. Call patterns will be used for introducing new definitions and also for applying the contextual replacement rule R5.

Definition 5. Given a clause γ of the form $p(X) \leftarrow d, A_1, \ldots, A_k$, with $k > 0$, the set of *call patterns* of γ, which is denoted by $CP(\gamma)$, is the set of triples $\langle solve(d,Y), A, Y\rangle$ such that: (i) A is A_j for some $j = 1, \ldots, k$, and (ii) Y denotes the linking variables of A_j in γ. $\langle solve(d,Y), A, Y\rangle$ is said to be the call pattern of γ for A. The set of call patterns of a set Γ of clauses, denoted by $CP(\Gamma)$, is the union of the call patterns of the clauses in the set Γ.

We now introduce a *definition function Define*, which takes as input a definition tree *Defstree*, a leaf clause δ of *Defstree*, and a call pattern $\langle c, A, Y\rangle$ of a leaf clause λ of $Utree(\delta)$, and produces as output a clause to be used for folding λ w.r.t. A. *Define* is parametric w.r.t. the choices of a wqo $\preceq_g$ and a clause generalization function *gen*.

The Definition Function $Define(Defstree, \delta, \langle c, A, Y\rangle)$.
Let η be the clause: $newp(Y) \leftarrow c, A$, where *newp* is a new predicate symbol.
if η is folding equivalent to a clause ϑ in *Defstree*
then return ϑ
else Let π be the path from the root of *Defstree* to clause δ
  if in π there exists a clause of the form $H \leftarrow d, B$ with the following properties:
    (1) $(d, B) \preceq_g (c, A)$ and (2) A and B have the same predicate
  then let α be the last clause in π with properties (1) and (2)
    if $gen(\alpha, \eta)$ is folding equivalent to a clause ϑ in *Defstree*
    then return ϑ
    else return $gen(\alpha, \eta)$ (case G)
  else return η (case F). □

The Procedure $Define\text{-}Fold(UForest, Defstree, NewDefs, FoldedCls)$.
Input: a forest *UForest* of unfolding trees and a definition tree *Defstree*. Each root of the trees in *UForest* is a leaf clause of *Defstree*.
Output: a set *NewDefs* of new definition clauses and a set *FoldedCls* of derived clauses.

NewDefs := ∅; *FoldedCls* := ∅;
for each unfolding tree $Utree(\delta)$ in *UForest* **do**

for each leaf clause λ of $Utree(\delta)$ with non-failing body **do**
Let λ be of the form $A_0 \leftarrow d, A_1, \ldots, A_k$, with $k \geq 0$.
if $k = 0$ **then** $FoldedCls := FoldedCls \cup \{\lambda\}$
else begin $\gamma_1 := \lambda$ **;**
for $i = 1, \ldots, k$ **do**
Let cp_i be the call pattern of γ_i for A_i and let η be the clause $Define(Defstree, \delta, cp_i)$.
if η is not in $Defstree$ **then**
begin expand $Defstree$ by making η a son of δ;
$NewDefs := NewDefs \cup \{\eta\}$ **end ;**
Fold γ_i w.r.t. A_i by using η thereby deriving clause γ_{i+1}**;**
end-for ;
$FoldedCls := FoldedCls \cup \{\gamma_{k+1}\}$
end
end-for
end-for □

Notice that the clauses whose body consists of a satisfiable constraint only, are not folded, and they are added to *FoldedCls*. Moreover the clauses with failing body are not folded, and they are not added to the set *FoldedCls*. This treatment of the clauses with failing body can be viewed as an implicit application of the clause removal rule R4.

4.3 Contextual Specialization Strategy

We now present our strategy for contextual specialization of CLP($\mathcal{D}$) programs. It consists of two phases. During the first phase we apply the unfolding, constraint replacement, constrained atomic definition, and constrained atomic folding rules, according to the *Unfold-Replace* and *Define-Fold* procedures. During the second phase we eliminate redundant constraints by a suitable application of the contextual constraint replacement rule.

The condition $FV(c) \subseteq X$ on the input to the Contextual Specialization Strategy below, is not actually a restriction, because our constraints are closed w.r.t. existential quantification.

Contextual Specialization Strategy
Input: (i) A CLP($\mathcal{D}$) program P and
(ii) a constrained atom $c, p(X)$ such that $FV(c) \subseteq X$.
Output: A CLP($\mathcal{D}$) program P_s and an atom $p_s(X)$.

Phase 1.
By the definition rule introduce a clause δ_0 of the form $p_s(X) \leftarrow c, p(X)$**.** Let *Defstree* consist of clause δ_0 only.
$P_s := \emptyset$; $NewDefs := \{\delta_0\}$;
while $NewDefs \neq \emptyset$ **do**
Unfold-Replace$(NewDefs, UForest)$;
Define-Fold$(UForest, Defstree, NewDefs, FoldedCls)$;

$P_s := P_s \cup \mathit{FoldedCls}$
end-while

Phase 2. [*Contextual Constraint Replacement*]
Let $\mathbf{C} = \{(\mathit{solve}(c, X), p_s(X))\} \cup \{(d, A) \mid \langle d, A, Y\rangle \in \mathit{CP}(P_s)\}$.
For every predicate symbol, say q, in P_s
- take the set $\{(d_1, q(V_1)), \ldots, (d_k, q(V_k))\}$ of all constrained atoms in $\mathbf{C}$ with predicate symbol q, and
- compute the least upper bound $m = d_1\{V_1/V\} \sqcup \ldots \sqcup d_k\{V_k/V\}$, where V is a tuple of new variables.

Replace each clause $q(W) \leftarrow f_1, \ldots, f_n, G$ in P_s, where $f_1, \ldots, f_n$ are primitive constraints, by the clause $q(W) \leftarrow f, G$, where f is the conjunction of all f_j's, with $1 \leq j \leq n$, such that it is not the case that $m\{V/W\} \sqsubseteq f_j$. This replacement is performed by applying the Contextual Constraint Replacement Rule, and the use of this rule is justified by the fact that, for $i = 1, \ldots, k$, $\mathcal{D} \models \forall Z\,((d_i, V_i = W) \rightarrow ((f_1, \ldots, f_n) \leftrightarrow f))$, where $Z = \mathit{FV}(d_i, V_i, W, f_1, \ldots, f_n, f)$. □

Theorem 1. [*Correctness of the Contextual Specialization Strategy*] Let P be a CLP($\mathcal{D}$) program and $c, p(X)$ be a constrained atom with $\mathit{FV}(c) \subseteq X$. Let P_s and $p_s(X)$ be the CLP($\mathcal{D}$) program and the atom obtained by the Contextual Specialization Strategy. Then, for every variable assignment σ, we have that

$$\mathit{lm}(P, \mathcal{D}) \models_\sigma c, p(X) \qquad \text{iff} \qquad \mathit{lm}(P_s, \mathcal{D}) \models_\sigma c, p_s(X)$$

Proof. It is based on the correctness of the transformation rules [1, 6]. In particular, during the application of the Contextual Specialization Strategy, folding is applied only to clauses which have been derived by one or more applications of the unfolding rule, followed by applications of the constraint replacement rule. Thus, we may apply Theorem 4.10 of [6] which ensures the correctness of the transformation rules w.r.t. the least $\mathcal{D}$-model. □

Theorem 2. [*Termination of the Contextual Specialization Strategy*] Let P be a CLP($\mathcal{D}$) program, and $c, p(X)$ be a constrained atom with $\mathit{FV}(c) \subseteq X$. If the widening operator ∇ used for clause generalization agrees with the well-quasi-order $\preceq_g$, then the Contextual Specialization Strategy terminates.

Proof. See Appendix. □

5 An Extended Example

Let us consider the following CLP($\mathcal{R}_{lin}$) program *Mmod*:

1. $\mathit{mmod}(I, J, M) \leftarrow I \geq J,\ M = 0$
2. $\mathit{mmod}(I, J, M) \leftarrow I < J,\ I1 = I+1,\ \mathit{mod}(I, L),\ \mathit{mmod}(I1, J, M1),\ M = M1+L$
3. $\mathit{mod}(X, M) \leftarrow X \geq 0,\ M = X$
4. $\mathit{mod}(X, M) \leftarrow X < 0,\ M = -X$

$Mmod(I, J, M)$ holds if and only if $M = |I| + |I{+}1| + \cdots + |I{+}k|$ and k is the largest integer such that $I{+}k$ is smaller than J. Let us assume that we want to specialize the program $Mmod$ w.r.t. the constrained atom $I = 0,\ J \geq 0,\ mmod(I, J, M)$.

Recall that in $\mathcal{R}_{lin}$ the least upper bound operation $\sqcup$ is defined by the convex hull construction (see Section 3). In this example we instantiate our Contextual Specialization Strategy as follows. (i) The function *solve* is the simplifier of conjunctions of linear equations and inequations over the reals implemented in Holzbaur's clp(q,r) solver [9], (ii) $Unfold(\gamma, Utree(\delta))$ returns the set of clauses obtained by unfolding clause γ w.r.t. the *leftmost* (in the textual order) atom A in its body, (iii) the well-quasi-order $\preceq_u$ is the relation $\preceq_L$ over constrained goals (see Section 3), (iv) the function *gen* for clause generalization is the one introduced in Example 1 (see Section 3), and (v) the wqo $\preceq_g$ is the restriction of $\preceq_L$ to constrained atoms.

We apply the Contextual Specialization Strategy and we start off from the tree *Defstree* which consists of the root clause 5 only, where:

5. $mmod_s(I, J, M) \leftarrow I = 0,\ J \geq 0,\ mmod(I, J, M)$

We apply the *Unfold-Replace* procedure as follows. The input for the procedure consists of the set $NewDefs = \{\text{clause } 5\}$. Let $Utree(\text{clause } 5)$ consist of the root clause 5. The only atom in the body of clause 5 is $mmod(I, J, M)$, and $Unfold(\text{clause } 5, Utree(\text{clause } 5))$ is the set {clause 6, clause 7} where:

6. $mmod_s(I, J, M) \leftarrow I = 0,\ J \geq 0,\ I \geq J,\ M = 0$
7. $mmod_s(I, J, M) \leftarrow I = 0,\ J \geq 0,\ I < J,\ I1 = I{+}1,\ mod(I, L),$
   $mmod(I1, J, M1),\ M = M1{+}L$

We apply the constraint replacement rule using *solve*, thereby obtaining $Replace(\{\text{clause } 6, \text{clause } 7\}) = \{\text{clause } 8, \text{clause } 9\}$ where:

8. $mmod_s(I, J, M) \leftarrow I = 0,\ J = 0,\ M = 0$
9. $mmod_s(I, J, M) \leftarrow I = 0,\ J > 0,\ I1 = 1,\ mod(I, L),\ mmod(I1, J, M1),$
   $M = M1{+}L$

Now we expand $Utree(\text{clause } 5)$ by making clauses 8 and 9 sons of clause 5.
$Unfold(\text{clause } 8, Utree(\text{clause } 5))$ is not defined, because the body of clause 8 contains no atom, and thus, clause 8 is not unfolded.
The construction of $Utree(\text{clause } 5)$ proceeds by first unfolding clause 9 w.r.t. $mod(I, L)$ and then applying the constraint replacement rule. We get the following clauses:

10. $mmod_s(I, J, M) \leftarrow false,\ mmod(I1, J, M1),\ M = M1{+}L$
10.1 $mmod_s(I, J, M) \leftarrow I = 0,\ J > 0,\ I1 = 1,\ mmod(I1, J, M)$

We expand $Utree(\text{clause } 5)$ by making clauses 10 and 10.1 sons of clause 9.
There is no leaf clause λ of $Utree(\text{clause } 5)$ such that $Unfold(\lambda, Utree(\text{clause } 5))$ is defined. Indeed: (a) clause 10.1 is the only leaf of $Utree(\text{clause } 5)$ whose body is non-failing and contains at least one atom, and (b) clause 5 is an ancestor of clause 10.1 such that $bd(\text{clause } 5) \preceq_L bd(\text{clause } 10.1)$. Thus, the *Unfold-Replace* procedure terminates with output $UForest = \{Utree(\text{clause } 5)\}$. The leaves of $Utree(\text{clause } 5)$ are the clauses 8, 10, and 10.1.

We now apply the *Define-Fold* procedure as follows. The input for the procedure consists of the forest $\{$*Utree*(clause 5)$\}$ and the definition tree *Defstree* made out of the root clause 5 only. The leaves of *Utree*(clause 5) with non-failing body are clause 8 and clause 10.1.
The body of clause 8 contains no atom, and so we add clause 8 to *FoldedCls*.
The body of clause 10.1 contains one atom, and the only call pattern of clause 10.1 is $\langle c, A, Y\rangle$, where c is $(J > 0,\ I1 = 1)$, A is $mmod(I1, J, M)$, and Y is $\{I1, J, M\}$.
We now compute *Define*(*Defstree*, clause 5, $\langle c, A, Y\rangle$). We consider the following clause:

η. $newp(I1, J, M) \leftarrow J{>}0,\ I1{=}1,\ mmod(I1, J, M)$

Since there is no clause in *Defstree* which is folding equivalent to η and bd(clause 5) $\preceq_L$ $bd(\eta)$, we compute clause gen(clause 5, η), which is (see Example 1 at the end of Section 3):

11. $genp(I1, J, M) \leftarrow I1{\geq}0,\ J{\geq}0,\ mmod(I1, J, M)$

Since there is no clause in *Defstree* which is folding equivalent to clause 11 we are in case G. We expand *Defstree* by making clause 11 a son of clause 5 and we add clause 11 to the set *NewDefs*. Then we fold clause 10.1 by using the definition clause 11 and we get:

12. $mmod_s(I, J, M) \leftarrow I{=}0,\ J{>}0,\ I1{=}1,\ genp(I1, J, M)$

The *Define-Fold* procedure terminates with output *FoldedCls* = {clause 8, clause 12} and *NewDefs* = {clause 11}. Since *NewDefs* is non-empty, we continue the execution of the while-loop of the Contextual Specialization Strategy.
We apply the *Unfold-Replace* procedure as follows. The input for the procedure consists of the set *NewDefs* = {clause 11}. Initially, *Utree*(clause 11) consists of the root clause 11. The only atom in the body of clause 11 is $mmod(I1, J, M)$, and *Unfold*(clause 11, *Utree*(clause 11)) is the set {clause 13, clause 14} where:

13. $genp(I1, J, M) \leftarrow I1{\geq}0,\ J{\geq}0,\ I1{\geq}J,\ M{=}0$
14. $genp(I1, J, M) \leftarrow I1{\geq}0,\ J{\geq}0,\ I1{<}J,\ I2{=}I1{+}1,\ mod(I1, L),$
$mmod(I2, J, M2),\ M{=}M2{+}L$

We apply the constraint replacement rule using *solve*, and we get *Replace*({clause 13, clause 14}) = {clause 15, clause 15.1} where:

15. $genp(I1, J, M) \leftarrow I1{\geq}J,\ J{\geq}0,\ M{=}0$
15.1 $genp(I1, J, M) \leftarrow I1{\geq}0,\ I1{<}J,\ I2{=}I1{+}1,\ mod(I1, L),$
$mmod(I2, J, M2),\ M{=}M2{+}L$

Now we expand *Utree*(clause 11)) by making clauses 15 and 15.1 sons of clause 11. *Unfold*(clause 15, *Utree*(clause 11)) is not defined, because the body of clause 15 contains no atom.
The construction of *Utree*(clause 11) proceeds by first unfolding clause 15.1 w.r.t. $mod(I1, L)$ and then applying the constraint replacement rule. We get, as the reader may easily verify, the following clauses:

16 $genp(I1, J, M) \leftarrow false,\ mmod(I2, J, M2)$

16.1 $genp(I1, J, M) \leftarrow I1 \geq 0,\ I1 < J,\ I2 = I1+1,$
$mmod(I2, J, M2),\ M = M2+I1$

We expand *Utree*(clause 11) by making clauses 16 and 16.1 sons of clause 15.1. There is no leaf clause λ of *Utree*(clause 11) such that *Unfold*(λ, *Utree*(clause 11)) is defined. Indeed: (a) clause 16.1 is the only leaf of *Utree*(clause 11) whose body is non-failing and it contains at least one atom, and (b) clause 11 is an ancestor of clause 16.1 such that *bd*(clause 11) $\preceq_L$ *bd*(clause 16.1). Thus, the *Unfold-Replace* procedure terminates with output *UForest* = {*Utree*(clause 11)}.

We now apply the *Define-Fold* procedure as follows. The input for the procedure consists of the forest {*Utree*(clause 11)} and the definition tree *Defstree* made out of the root clause 5 and its son clause 11. The leaves of *Utree*(clause 11) with non-failing body are clause 15 and clause 16.1.
The body of clause 15 contains no atom, and so we add clause 15 to *FoldedCls*.
The body of clause 16.1 contains one atom and $\langle c', A', Y' \rangle$ is the only call pattern of clause 16.1 where c' is $(I2 \geq 1,\ J > I2-1)$, A' is $mmod(I2, J, M2)$, and Y' is $\{I2, J, M2\}$.
We now compute *Define*(*Defstree*, clause 11, $\langle c', A', Y' \rangle$). Let us consider the clause:

$\eta 1.\quad newq(I2, J, M2) \leftarrow I2 \geq 1,\ J > I2-1,\ mmod(I2, J, M2)$

Since there is no clause in *Defstree* which is folding equivalent to $\eta 1$ and *bd*(clause 11) $\preceq_L$ *bd*($\eta 1$), we compute clause *gen*(clause 11, $\eta 1$), which is clause 11 itself. Then we fold clause 16.1 by using the definition clause 11 and we get:

17. $genp(I1, J, M) \leftarrow I1 \geq 0,\ I1 < J,\ I2 = I1+1,$
$genp(I2, J, M2),\ M = M2+I1$

The *Define-Fold* procedure terminates with output *FoldedCls* = {clause 15, clause 17} and *NewDefs* = $\emptyset$.

Since during the last application of the *Define-Fold* procedure we did not introduce any new definition, the while-loop of the Contextual Specialization Strategy terminates and we get program P_s made out of clauses 8, 12, 15, and 17.

Now we perform the Contextual Constraint Replacement phase of the Contextual Specialization Strategy. By computing the call patterns of P_s and performing a variable renaming we get:

$\mathbf{C} = \{(I=0,\ J \geq 0,\ mmod_s(I, J, M)),$
$(I=1,\ J>0,\ genp(I, J, M)),$
$(I \geq 1,\ J > I-1,\ genp(I, J, M))\}$

Thus, the least upper bounds are:
for $mmod_s$: $m_1 = (I=0,\ J \geq 0)$ and for *genp*: $m_2 = (I \geq 1,\ J > I-1)$. We then process clauses 8, 12, 15, and 17 as indicated in the final part of the Contextual Specialization Strategy.
For clause 8, since $m_1 \sqsubseteq I=0$, we get:

8.c $mmod_s(I, J, M) \leftarrow J=0,\ M=0$

For clause 12, since $m_1 \sqsubseteq I=0$, we get:

12.c $mmod_s(I, J, M) \leftarrow J>0,\ I1=1,\ genp(I1, J, M)$

For clause 15, since $m_2 \sqsubseteq J \geq 0$, after a variable renaming, we get:

15.c $\quad genp(I, J, M) \leftarrow I \geq J,\ M = 0$

For clause 17, since $m_2 \sqsubseteq I \geq 0$, after a variable renaming, we get:

17.c $\quad genp(I, J, M) \leftarrow I < J,\ I2 = I + 1,\ genp(I2, J, M2),\ M = M2 + I$

The final program P_s we have derived, consists of clauses 8.c, 12.c, 15.c, and 17.c. Experimental results performed on a PC running Linux and SICStus Prolog, confirm the efficiency improvements of this program w.r.t. the initial program *Mmod*. The speedup is of 7.7 times. Program P_s is also more efficient than the program consisting of clauses 8, 12, 15, and 17, that is, the program derived before the application of the Contextual Constraint Replacement phase of our Contextual Specialization strategy, and the speedup in this case of 1.1 times.

6 Related Work and Conclusions

There are various methods for specializing programs w.r.t. properties of their input data [2, 3, 18, 19, 21]. Contextual specialization can be viewed as one of these methods. In a previous paper [7] we have illustrated the basic ideas of the contextual specialization of CLP($\mathcal{D}$) programs and we have presented a set of transformation rules for this form of specialization. Now we build upon our previous work and we propose a fully automated strategy for the contextual specialization of constraint logic programs.

We have assumed the existence of a solver which simplifies conjunctions of constraints in a given domain $\mathcal{D}$. We abstractly represent that solver as a computable total function *solve*. We have also assumed the existence of: (i) a function *Unfold* and a well-quasi-order $\preceq_u$ between constrained goals for guiding the unfolding process, (ii) a clause generalization function *gen* parameterized by a widening operator ∇, and (iii) a well-quasi-order $\preceq_g$ between constrained atoms, which activates the clause generalization process during program specialization. We have shown that our specialization strategy preserves the least $\mathcal{D}$-model and it terminates.

The hypothesis that the function *solve* is assumed to be complete w.r.t. satisfiability, makes our approach different from the one in [2], where program specialization is based on some undecidable properties and thus, it cannot be easily automated.

The use of our contextual constraint replacement rule R5 allows us to perform optimizations which can be performed neither by using the transformation rules presented in [1, 6, 17] nor by applying the techniques for partial evaluation of logic programs presented in [15, 24]. Indeed, our contextual constraint replacement rule is able to take advantage of the information provided by the context of use of every predicate of the programs to be specialized.

Our use of the well-quasi-orders is borrowed from techniques used for proving termination of rewriting systems [5]. Well-quasi-orders were also used in [13, 14] for ensuring termination of partial evaluation (also called *local control*) and

controlling the introduction of new atoms as generalizations of old atoms (also called *global control*).

Among the many techniques for simplifying and manipulating constraints to get more efficient specialized programs, here we want to mention the following methods which are related to ours.

In [21] the authors propose a method based on abstract interpretation, for the implementation of multiple specialization of logic programs. Particular emphasis is given to program parallelization. Similarly to their work, we specialize programs w.r.t. constrained queries which represent a possibly infinite set of queries, and, although the constrained atom produced by our strategy contains one specialized predicate only, the produced program may contain different specialized versions for the same predicate. Thus, one may consider that also our approach provides a technique for multiple program specialization.

In [18] the authors present a methodology for compiling CLP($\mathcal{D}$) languages so that the workload of the constraint solver is reduced. However, their methodology may generate *non-monotonic* CLP($\mathcal{D}$) programs, whose execution requires the ability of removing constraints from the constraint store. In contrast, in our approach we generate monotonic CLP($\mathcal{D}$) programs which at runtime do not remove constraints from the store. However, we require the solver to test for the satisfiability and entailment of constraints, and to compute the existential closure of constraints w.r.t. given sets of variables. These capabilities are indeed provided by most constraint solvers.

Finally, our work is related to the approach presented in [3] for a strict first-order functional language. As in that paper, we too specify the context of use w.r.t. which the programs should be specialized, by providing a property which may cover an infinite set of queries.

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. C. Consel and S. C. Khoo. Parameterized partial evaluation. *ACM Transactions on Programming Languages and Systems*, 15(3):463–493, 1993.
4. P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *Proceedings 4th ACM-SIGPLAN Symposium on Principles of Programming Languages (POPL '77)*, pages 238–252. ACM Press, 1977.
5. N. Dershowitz and J.-P. Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B, pages 243–320. Elsevier, 1990.
6. S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166:101–146, 1996.
7. F. Fioravanti, A. Pettorossi, and M. Proietti. Rules and strategies for contextual specialization of constraint logic programs. In M. Leuschel, editor, *Proceedings of*

the ICLP'99 Workshop on Optimization and Implementation of Declarative Programming Languages, WOID '99, Las Cruces University, New Mexico, USA, pages 1–9, December 2–3, 1999.
8. 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.
9. C. Holzbaur. OFAI clp(q,r) manual, Edition 1.3.2. Technical Report TR-95-09, Austrian Research Institute for Artificial Intelligence, Vienna, 1995.
10. J. Jaffar and M. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
11. J. Jaffar, M. Maher, K. Marriott, and P. Stuckey. The semantics of constraint logic programming. *Journal of Logic Programming*, 37:1–46, 1998.
12. N. D. Jones, C. K. Gomard, and P. Sestoft. *Partial Evaluation and Automatic Program Generation.* Prentice Hall, 1993.
13. M. Leuschel. Improving homeomorphic embedding for online termination. In P. Flener, editor, *Proceedings of LOPSTR'98, Manchester, UK, June 1998*, Lecture Notes in Computer Science 1559, pages 199–218. Springer-Verlag, 1999.
14. M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Transactions on Programming Languages and Systems*, 20(1):208–258, 1998.
15. M. Leuschel and D. De Schreye. Constrained partial deduction. In F. Bry, B. Freitag, and D. Seipel, editors, *Proceedings of the 12th Workshop Logische Programmierung (WLP'97)*, pages 116–126, Munich, Germany, September 1997.
16. J. W. Lloyd and J. C. Shepherdson. Partial evaluation in logic programming. *Journal of Logic Programming*, 11:217–242, 1991.
17. M. J. Maher. A transformation system for deductive database modules with perfect model semantics. *Theoretical Computer Science*, 110:377–403, 1993.
18. K. Marriott and P. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, Removal and Reordering. In *POPL'93: Proceedings ACM SIGPLAN Symposium on Principles of Programming Languages*, pages 334–344, 1993.
19. 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.
20. 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.
21. G. Puebla and M. Hermenegildo. Abstract multiple specialization and its application to program parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.
22. D. Sahlin. Mixtus: An automatic partial evaluator for full Prolog. *New Generation Computing*, 12:7–51, 1993.
23. M. H. Sørensen and R. Glück. An algorithm of generalization in positive supercompilation. In J. W. Lloyd, editor, *Proceedings of the 1995 International Logic Programming Symposium (ILPS '95)*, pages 465–479. MIT Press, 1995.
24. 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, Zakopane, Poland*, Lecture Notes in Computer Science 1079, pages 98–107. Springer-Verlag, 1996.

Appendix. Proof of Theorem 2.

Let us begin by showing the termination of the *Unfold-Replace* procedure. It follows from the properties of the wqo $\preceq_u$ and the hypothesis that for any leaf clause λ of an unfolding tree T, if there exists an ancestor α of λ such that $bd(\alpha) \preceq_u bd(\lambda)$, then $Unfold(\lambda, T)$ is not defined.

Also the *Define-Fold* procedure and the Contextual Replacement phase of the Contextual Specialization Strategy trivially terminate because of the absence of while-loop statements in their definitions.

To prove the termination of the Contextual Specialization Strategy we have to show that the set *NewDefs* of new definitions introduced by the *Define-Fold* procedure will eventually be empty, that is, *Defstree* is a finite tree.

Every node of *Defstree* has finite branching. Indeed, (i) each clause δ occurring in *Defstree* has a number of sons which is not greater than the number of atoms in the bodies of the leaf clauses of $Utree(\delta)$ to be folded, and (ii) for all δ, the unfolding tree $Utree(\delta)$ constructed by the *Unfold-Replace* procedure is finite.

We now show that every path starting from the root of *Defstree* is finite. Consider a generic path π of *Defstree*, of the form $\delta_0 \cdots \delta_k \cdots$, where δ_0 is the root clause of *Defstree*. We can partition the clauses of π into two sets: the set *GenDefs* of clauses which have been introduced as generalizations of one of its ancestors in π (see Case G of the definition function *Define*) and the set *FreshDefs* of all other clauses in π (see Case F of the definition function *Define*). In particular, $\delta_0 \in$ *FreshDefs*.

Let us introduce the following binary relation $\preceq_{gp}$: $(c, A) \preceq_{gp} (d, B)$ iff (1) $(c, A) \preceq_g (d, B)$ and (2) A and B have the same predicate symbol. We have that $\preceq_{gp}$ is a wqo, because $\preceq_g$ is a wqo and the set of predicate symbols is finite (recall that the predicate symbols occurring in the bodies of the definitions also occur in the initial program). We also have that *gen* agrees with $\preceq_{gp}$ because *gen* agrees with $\preceq_g$.

We will show that for any path π, we can construct a tree $T(\pi)$ whose nodes are the clauses of π, such that:
(Property F*) for any $h > 0$, a clause δ_h is the left son of a clause δ_j iff $\delta_h \in$ *FreshDefs* and δ_j is the last clause in $\delta_0 \cdots \delta_{h-1}$ which is in *FreshDefs*. Thus, $bd(\delta_i) \not\preceq_{gp} bd(\delta_h)$ for all $i = 0, \ldots, h-1$.
(Property G*) for any $h > 0$, a clause δ_h is the right son of a clause δ_j iff $\delta_h \in$ *GenDefs* and δ_h has been introduced as a generalization of δ_j, that is, $\delta_h = gen(\delta_j, \eta)$ for some clause η (which does not belong to the set *Defstree*, and thus, it is not in π). Thus, (i) δ_j is the last clause in $\delta_0 \cdots \delta_{h-1}$ such that $bd(\delta_j) \preceq_{gp} bd(\eta)$, (ii) $bd(\delta_h) \preceq_{gp} bd(\delta_j)$, because *gen* agrees with $\preceq_{gp}$, and (iii) there is no clause in $\delta_0 \cdots \delta_{h-1}$ which is folding equivalent to δ_h.

We will show that for any path π we can construct the tree $T(\pi)$ by proving that, for all finite prefixes $\delta_0 \cdots \delta_k$ of π, there exists $T(\delta_0 \cdots \delta_k)$ satisfying Property F* and Property G* above. The proof proceeds by induction on k.
The base case ($k = 0$) is trivial. For the inductive step, let us assume that Property F* and Property G* hold for $T(\delta_0 \cdots \delta_k)$ and let us show them for

$T(\delta_0 \cdots \delta_{k+1})$.
(Case F) Let $\delta_{k+1} \in \mathit{FreshDefs}$ and (A) let δ_j be the last clause in $\delta_0 \cdots \delta_k$ such that $\delta_j \in \mathit{FreshDefs}$. Let $T(\delta_0 \cdots \delta_{k+1})$ be the tree obtained from $T(\delta_0 \cdots \delta_k)$ by adding δ_{k+1} as left son of δ_j. Property F* holds for $T(\delta_0 \cdots \delta_{k+1})$: (if part) by construction; (only if part) δ_{k+1} is the only left son of δ_j in $T(\delta_0 \cdots \delta_{k+1})$. Indeed, if there exists a left son δ_h of δ_j, then by inductive hypothesis $\delta_h \in \mathit{FreshDefs}$ and $j < h \leq k$, which contradicts the assumption (A). The validity of Property G* for $T(\delta_0 \cdots \delta_{k+1})$ follows immediately from the validity of Property G* for $T(\delta_0 \cdots \delta_k)$.
(Case G) Let $\delta_{k+1} \in \mathit{GenDefs}$, that is, $\delta_{k+1} = \mathit{gen}(\delta_j, \eta)$ for some clause η, where: (B) δ_j is the last clause in $\delta_0 \cdots \delta_k$ such that $\mathit{bd}(\delta_j) \preceq_{gp} \mathit{bd}(\eta)$. Let $T(\delta_0 \cdots \delta_{k+1})$ be the tree obtained from $T(\delta_0 \cdots \delta_k)$ by adding δ_{k+1} as right son of δ_j. The validity of Property F* for $T(\delta_0 \cdots \delta_{k+1})$ follows immediately from the validity of Property F* for $T(\delta_0 \cdots \delta_k)$. Property G* holds for $T(\delta_0 \cdots \delta_{k+1})$ because: (if part) by construction; (only if part) δ_{k+1} is the only right son of δ_j in $T(\delta_0 \cdots \delta_{k+1})$. Indeed, if there exists a right son δ_h of δ_j in $T(\delta_0 \cdots \delta_k)$, then by inductive hypothesis $j < h$ and $\mathit{bd}(\delta_h) \preceq_{gp} \mathit{bd}(\delta_j)$. Thus, by transitivity of $\preceq_{gp}$ we have $\mathit{bd}(\delta_h) \preceq_{gp} \mathit{bd}(\eta)$ and $j < h$, which contradicts the assumption (B).
We now show that a generic path π of *Defstree* is finite by showing that $T(\pi)$ is finite, that is (a) $T(\pi)$ is finitely branching, and (b) each path in $T(\pi)$ is finite.
(a) By properties F* and G*, we have that each node of $T(\pi)$ has at most two sons and thus $T(\pi)$ has finite branching.
(b) Consider a path τ from the root of $T(\pi)$ of the form: $\delta_0\gamma_1 \cdots \gamma_k \cdots$. Let γ_h be a clause in τ such that $\gamma_h \in \mathit{GenDefs}$. If such a γ_h does not exist then all clauses in τ belong to *FreshDefs*, and thus, for all distinct $i, j \geq 0$ we have $\gamma_j \not\preceq_{gp} \gamma_i$. In this case τ cannot be infinite because $\preceq_{gp}$ is a wqo. If such a γ_h does exist, then by Properties F* and G* the suffix of τ of the form: $\gamma_h\gamma_{h+1} \cdots$ is such that for all $i \geq h$, $\gamma_{i+1} = \mathit{gen}(\gamma_i, \eta_i)$. By Point (iii) of Property G* we have that: for all $i \geq h$ and for all $j < i$, clause γ_j is not folding equivalent to γ_i. The path $\gamma_h\gamma_{h+1} \cdots$ is finite because, as pointed out in Section 4, if it were infinite, then there exist two folding equivalent clauses γ_s and γ_t, with $h \leq s < t$. Thus τ is finite. □