Specialization with Clause Splitting for Deriving Deterministic 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— **The reduction of nondeterminism can increase efficiency when specializing programs. We consider constraint logic programs and we propose a technique which by making use of a new transformation rule, called clause splitting, allows us to generate efficient, specialized programs which are deterministic. We have applied our technique to the specialization of pattern matching programs.**

Keywords— **Program Specialization, Program Transformation, Constraint Logic Programming, Pattern Matching.**

I. Introduction

Programs are often written in a parametric form so that one can reuse them in different contexts. When one reuses parametric programs, one may want to transform those programs for taking advantage of the contexts of use and, indeed, by doing so, often program efficiency is improved. This program transformation is usually called *program specialization* [15] and it can be performed by using well established techniques such as *partial evaluation* [4], [10], [15], [16], [18].

Various program specialization methods have been proposed in the literature for different programming languages. In this paper we consider a program specialization method for constraint logic programming (CLP) and we use the *rules + strategies* transformation approach. This approach was first suggested by Burstall-Darlington for functional languages [3] and later applied to logic languages by Tamaki-Sato [21]. We will present a program transformation technique which allows us to increase program efficiency by deriving deterministic, specialized programs starting from nondeterministic, general programs.

The paper is structured as follows. We first present the rules for transforming constraint logic programs. These rules are an extension of the ones presented in [7], [19]. They include extensions of the familiar unfolding and folding rules, and an extra rule, called *clause splitting*, which generalizes the *case splitting* rule presented in [19]. Given a clause $H \leftarrow Body$ and a constraint c, by the clause splitting rule we can generate the clauses: $H \leftarrow c \wedge Body$ and $H \leftarrow \neg c \wedge Body$. Since these clauses have mutual exclusive bodies, we are able to derive efficient programs with reduced nondeterminism. The correctness of the derived programs follows from the fact that the transformation rules preserve the least model semantics [14].

In: Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics (SMC'02), 6 - 9 October, 2002, Hammamet, Tunisia, IEEE Computer Society Press, 2002.

We also present an automatic specialization strategy for guiding the application of the transformation rules. This strategy is an enhancement of the strategy presented in [19] and includes a specific treatment of constraints. It consists of the following steps: (i) the introduction of an initial definition, corresponding to the goal w.r.t. which we want to specialize the initial program, (ii) the execution of some unfolding steps and constraint manipulations, and (iii) the execution of some folding steps. If these folding steps require the introduction of new definitions, we introduce them, and we continue the specialization process by executing unfolding, constraint manipulations, and folding steps starting from each of these new definitions. On the contrary, if the folding steps do not require the introduction of new definitions, we terminate the specialization process.

II. An Introductory Example: Specialization of Constrained Matching

We present an example of program specialization using the rules + strategies approach. Starting from a nondeterministic, general program which specifies a pattern matcher on strings, we derive a deterministic, specialized pattern matcher for a given pattern. In this example we define a more general matching relation between strings which is expressed as a constraint logic program. Our derivation generalizes the derivations of the Knuth-Morris-Pratt matcher which were presented, among others, in [8], [9], [10], [13], [19], [20]. As in the case of that matcher, we derive a program which is a deterministic finite automaton with transitions labelled by constraints, rather than symbols of the strings. We improve over the derivations of specialized pattern matchers presented in [8], [9], [10], [13], [20] because we start from a *nondeterministic* specification of the matcher, while in those papers the initial programs are deterministic. As already mentioned, the improvement over [19] is that we now deal with a general pattern matcher presented as a constraint logic program.

In our example we define a matching relation $m(P, S)$ between a pattern $P = [p_1, \ldots, p_n]$ and a string S, which holds iff in S there exists a substring $Q = [q_1, \ldots, q_n]$ and for all $i = 1, \ldots, n$, we have that $p_i \leq q_i$. The following CLP program can be taken as the specification of the general pattern matching problem:

1. $m(P, S) \leftarrow a(B, C, S) \wedge a(A, Q, B) \wedge le(P, Q)$
2. $a([\,], Ys, Ys) \leftarrow$
3. $a([X|Xs], Ys, [X|Zs]) \leftarrow a(Xs, Ys, Zs)$
4. $le([\,], [\,]) \leftarrow$
5. $le([X|Xs], [Y|Ys]) \leftarrow X \leq Y \wedge le(Xs, Ys)$

where a denotes the list concatenation. Now let us suppose that we want to specialize this general program w.r.t. the pattern $P=[1,0,2]$. We start off by introducing the following definition:

6. $m_{sp}(S) \leftarrow m([1,0,2], S)$

Clauses 1–6 constitute the initial program P_0 from which we begin our program specialization process. We generate a sequence of programs, each of which is derived from the previous one by applying a transformation rule (Section IV) according to the Determinization Strategy (Section V). As indicated in Section V, we will get the following final program:

9. $m_{sp}(S) \leftarrow new1(S)$
16. $new1([X|Xs]) \leftarrow 1 \leq X \wedge new2(Xs)$
17. $new1([X|Xs]) \leftarrow 1 > X \wedge new1(Xs)$
18. $new2([X|Xs]) \leftarrow 1 \leq X \wedge new3(Xs)$
19. $new2([X|Xs]) \leftarrow 0 \leq X \wedge 1 > X \wedge new4(Xs)$
20. $new2([X|Xs]) \leftarrow 0 > X \wedge new1(Xs)$
21. $new3([X|Xs]) \leftarrow 2 \leq X$
22. $new3([X|Xs]) \leftarrow 1 \leq X \wedge 2 > X \wedge new3(Xs)$
23. $new3([X|Xs]) \leftarrow 0 \leq X \wedge 1 > X \wedge new4(Xs)$
24. $new3([X|Xs]) \leftarrow 0 > X \wedge new1(Xs)$
25. $new4([X|Xs]) \leftarrow 2 \leq X$
26. $new4([X|Xs]) \leftarrow 1 \leq X \wedge 2 > X \wedge new2(Xs)$
27. $new4([X|Xs]) \leftarrow 1 > X \wedge new1(Xs)$

This final program is deterministic in the sense that at most one clause can be applied during the evaluation of every ground goal.

III. Preliminaries

In this section we recall some basic notions of constraint logic programming. For notions not defined here the reader may refer to [1], [14], [17].

A. Syntax of Constraint Logic Programs

We consider a first order language $\mathcal{L}$ generated by an infinite set *Vars* of *variables*, a set *Funct* of *function symbols* with arity, and a set *Pred* of *predicate symbols* with arity. We assume that *Pred* is the union of two disjoint sets: (i) the set $Pred_c$ of *constraint* predicate symbols, including *true*, *false*, and the *equality* symbol $=$, and (ii) the set $Pred_u$ of *user defined* predicate symbols. Terms and formulas of $\mathcal{L}$ are constructed from the element of *Vars*, *Funct*, and *Pred*, by means of connectives ($\neg$, $\wedge$, $\vee$) and quantifiers ($\forall, \exists$), as usually done in first order logic.

Given a sequence of terms or formulas $e_1, \ldots, e_n$ (for $n > 0$), the set of variables occurring in that sequence is denoted by $vars(e_1, \ldots, e_n)$. Given a formula φ, the set of the *free variables* in φ is denoted by $FV(\varphi)$. A term or a formula is *ground* iff it contains no variable. Given a set $X = \{X_1, \ldots, X_n\}$ of variables, by $\forall X\, \varphi$ we denote the formula $\forall X_1 \ldots \forall X_n\, \varphi$. By $\forall(\varphi)$ we denote the *universal closure* of φ, that is, the formula $\forall X\, \varphi$, where $FV(\varphi) = X$. Analogously, by $\exists(\varphi)$ we denote the *existential closure* of φ.

A *primitive constraint* is an atomic formula $p(t_1, \ldots, t_n)$ where p is a predicate symbol in $Pred_c$ and $t_1, \ldots, t_n$ are terms. The set $\mathcal{C}$ of *constraints*, ranged over by $c, d, \ldots$, is the smallest set of formulas of $\mathcal{L}$ which contains all primitive constraints and it is closed w.r.t. all connectives and quantifiers.

An *atom* A is an atomic formula $p(t_1, \ldots, t_n)$ where p is an element of $Pred_u$ and $t_1, \ldots, t_n$ are terms. A *goal* G is the conjunction of $m\,(\geq 0)$ atoms. A *constrained goal* $c \wedge G$ is the conjunction of a constraint and a goal. The empty conjunction of constraints or atoms is identified with *true*.

A *clause* γ is a formula of the form $H \leftarrow c \wedge G$, where: (i) H is an atom, called the *head* of γ, and (ii) $c \wedge G$ is a constrained goal, called the *body* of γ. Clauses of the form $H \leftarrow c$ are called *constrained facts*. Clauses of the form $H \leftarrow true$ are also written as $H \leftarrow$.

A *constraint logic program* (or *program*, for short) is a finite set of clauses. (Here we do not allow for negated atoms in the bodies of clauses.)

Given a program P, we say that a predicate p *depends on* a predicate q iff *either* there exists in P a clause whose head predicate is p and whose body contains an occurrence of q *or* there exists a predicate r such that p depends on r and r depends on q.

Given two atoms $p(t_1, \ldots, t_n)$ and $p(u_1, \ldots, u_n)$, we denote by $p(t_1, \ldots, t_n) = p(u_1, \ldots, u_n)$ the conjunction of the constraints: $t_1 = u_1 \wedge \ldots \wedge t_n = u_n$.

A *variable renaming* is a bijective mapping from *Vars* to *Vars*. The application of a variable renaming ρ to a formula φ returns the formula $\rho(\varphi)$, called a *variant* of φ, obtained by replacing each (bound or free) occurrence of X in φ by the variable $\rho(X)$. A *renamed apart* clause is a variant of a clause such that all its (bound or free) variables of do not occur elsewhere.

We will feel free to apply to clauses the following two transformations which, as the reader may verify, preserve program semantics (see below): (1) application of variable renamings, and (2) replacement of a clause of the form $H \leftarrow X = t \wedge c \wedge G$, where $X \notin vars(t)$, by the clause $(H \leftarrow c \wedge G)\{X/t\}$, and vice versa.

B. Least $\mathcal{D}$-model Semantics

We assume that we are given an interpretation $\mathcal{D}$ for the constraints in $\mathcal{C}$. Let D be the carrier of $\mathcal{D}$. $\mathcal{D}$ assigns a subset of D^n to each n-ary constraint predicate symbol in $Pred_c$. In particular, $\mathcal{D}$ assigns the whole carrier D to *true*, the empty set to *false*, and the identity over D to the equality symbol $=$.

A *$\mathcal{D}$-interpretation* is an interpretation for the formulas of $\mathcal{L}$ which extends the interpretation $\mathcal{D}$. In particular, a $\mathcal{D}$-interpretation assigns a subset of D^n to each n-ary user defined predicate symbol in $Pred_u$. Thus, a $\mathcal{D}$-interpretation is isomorphic to a subset of the following set $\mathcal{B}_\mathcal{D}$:

$$\mathcal{B}_\mathcal{D} = \{p(d_1, \ldots, d_n) \mid p \text{ is a predicate symbol in } Pred_u \text{ and } (d_1, \ldots, d_n) \in D^n\}$$

A $\mathcal{D}$-model of a program P is a $\mathcal{D}$-interpretation I such that $I \models \forall(P)$. It can be shown that for every CLP program P there exists a *least $\mathcal{D}$-model* (w.r.t. set inclusion), denoted by $lm(P, \mathcal{D})$ [14].

C. Operational Semantics

In order to define the operational semantics of constraint logic programs, we assume that there is a computable total function *solve*: $\mathcal{C}\times\mathcal{P}_{fin}(Vars)\rightarrow\mathcal{C}$, where $\mathcal{P}_{fin}(Vars)$ is the set of all finite subsets of *Vars*, by which we can simplify the constraints in $\mathcal{C}$. We assume that *solve* is *sound* w.r.t. constraint equivalence, that is, for every constraint c_1 and for every finite set X of variables, if $solve(c_1,X)=c_2$ then $\mathcal{D}\models\forall X((\exists Y\, c_1)\leftrightarrow(\exists Z\, c_2))$, where $Y=FV(c_1)-X$ and $Z=FV(c_2)-X$.

We also assume that *solve* is *complete* w.r.t. satisfiability, in the sense that, for any constraint c,
(i) $solve(c,\emptyset)=true$ iff c is *satisfiable*, i.e., $\mathcal{D}\models\exists(c)$, and
(ii) $solve(c,\emptyset)=false$ iff c is *unsatisfiable*, i.e., $\mathcal{D}\models\neg\exists(c)$.

The totality and the soundness of the *solve* function guarantee the correctness of the transformation strategy (see Section V). The assumption that *solve* is complete w.r.t. satisfiability guarantees that constraint satisfiability tests, which are required in our transformation method, are decidable. Moreover, the completeness w.r.t. satisfiability guarantees that for any constraints c_1 and c_2, by evaluating $solve(\forall(c_1\rightarrow c_2),\emptyset)$ we can check whether or not $\mathcal{D}\models\forall(c_1\rightarrow c_2)$ holds.

Now we define the operational semantics of a CLP program P by introducing a derivability relation $\mapsto_P$ between constrained goals as follows.

$c\wedge A\wedge G \;\mapsto_P\; c\wedge A=H_1\wedge c_1\wedge G_1\wedge G$
iff $H_1\leftarrow c_1\wedge G_1$ is a renamed apart clause of P and $c\wedge A=H_1\wedge c_1$ is satisfiable.

The relation $\mapsto_P^*$ is the reflexive and transitive closure of $\mapsto_P$. We say that the constrained goal $c\wedge G$ *succeeds* in P iff $c\wedge G\mapsto_P^* d$ for some satisfiable constraint d.

IV. Rules for Transforming CLP Programs

The process of transforming a given program P whereby deriving a program Q, can be formalized as a sequence $P_0,\ldots,P_n$ of programs, called a *transformation sequence*, where $P_0=P$, $P_n=Q$ and, for $k=0,\ldots,n-1$, program P_{k+1} is obtained from program P_k by applying one of the transformation rules listed below.

R1. Definition. We introduce a set of clauses

$$\begin{array}{ll}\delta_1: & newp(X_1,\ldots,X_h)\leftarrow c_1\wedge G_1\\ & \ldots\\ \delta_m: & newp(X_1,\ldots,X_h)\leftarrow c_m\wedge G_m\end{array}$$

where: (i) *newp* is a predicate symbol not occurring in $P_0,\ldots,P_k$, (ii) $\{X_1,\ldots,X_h\}\subseteq FV(c_1\wedge G_1,\ldots,c_m\wedge G_m)$, and (iii) the predicates occurring in $G_1,\ldots,G_m$ occur also in P_0.
We derive the new program $P_{k+1}=P_k\cup\{\delta_1,\ldots,\delta_m\}$. For $i\geq 0$, $Defs_i$ is the set of clauses introduced by the definition rule during the transformation sequence $P_0,\ldots,P_i$. In particular, $Defs_0=\emptyset$.

R2. Unfolding. Let $\gamma:\; H\leftarrow c\wedge G'\wedge A\wedge G''$ be a renamed apart clause of P_k. By unfolding γ w.r.t. A we derive the set of clauses

$$\Gamma:\; \{H\leftarrow c\wedge A=H_1\wedge c_1\wedge G'\wedge G_1\wedge G'' \mid H_1\leftarrow c_1\wedge G_1 \text{ is a clause in } P_k \text{ and } c\wedge A=H_1\wedge c_1 \text{ is satisfiable}\}$$

and the new program $P_{k+1}=(P_k-\{\gamma\})\cup\Gamma$.

R3. Folding. Let

$$\begin{array}{ll}\gamma_1: & H\leftarrow c\wedge c_1\vartheta\wedge G'\wedge G_1\vartheta\wedge G''\\ & \ldots\\ \gamma_m: & H\leftarrow c\wedge c_m\vartheta\wedge G'\wedge G_m\vartheta\wedge G''\end{array}$$

be $m\,(>0)$ clauses in P_k and let *newp* be a predicate such that

$$\begin{array}{ll}\delta_1: & newp(X_1,\ldots,X_h)\leftarrow c_1\wedge G_1\\ & \ldots\\ \delta_m: & newp(X_1,\ldots,X_h)\leftarrow c_m\wedge G_m\end{array}$$

are the clauses in $Defs_k$ which have *newp* as head predicate. Suppose that, for $i=1,\ldots,m$ and for every variable $X\in(FV(c_i\wedge G_i)-\{X_1,\ldots,X_h\})$, we have that: (i) $X\vartheta$ is a variable not occurring in (H,c,G',G''), and (ii) for every variable $Y\in(FV(c_i\wedge G_i))-\{X\})$, $X\vartheta$ does not occur in $Y\vartheta$. By folding $\gamma_1,\ldots,\gamma_m$ using $\delta_1,\ldots,\delta_m$ we derive the clause

$$\eta:\; H\leftarrow c\wedge G'\wedge newp(X_1,\ldots,X_h)\vartheta\wedge G''$$

and the new program $P_{k+1}=(P_k-\{\gamma_1,\ldots,\gamma_m\})\cup\{\eta\}$.

R4. Clause Removal. Let γ be a clause in P_k. We derive the new program $P_{k+1}=P_k-\{\gamma\}$ if one of the following cases occurs:

(*Unsatisfiable Constraint*) γ is the clause $H\leftarrow c\wedge G$ and c is unsatisfiable, that is, $\mathcal{D}\models\neg\exists(c)$;

(*Subsumed Clause*) γ is the clause $(H\leftarrow c_1\wedge G_1)\vartheta$ and there exists a clause in $P_k-\{\gamma\}$ of the form $H\leftarrow c_2\wedge G_2$ such that $\mathcal{D}\models\forall(c_1\rightarrow\exists X c_2)$, where $X=FV(c_2)-vars(H,G_2)$ and G_2 is a subconjunction of G_1.

R5. Constraint Replacement. Let $\gamma_1:\; H\leftarrow c_1\wedge G$ be a clause in P_k. Suppose that for some constraint c_2, we have that: $\mathcal{D}\models\forall(\exists Y\, c_1\leftrightarrow\exists Z\, c_2)$ where: (i) $Y=FV(c_1)-vars(H,G)$, and (ii) $Z=FV(c_2)-vars(H,G)$. In particular, we may take $c_2=solve(c_1,vars(H,G))$. Then we derive the clause

$$\gamma_2:\; H\leftarrow c_2\wedge G$$

and the new program $P_{k+1}=(P_k-\{\gamma_1\})\cup\{\gamma_2\}$.

R6. Clause Fusion. Let

$$\gamma_1:\; H\leftarrow c\wedge G \qquad\qquad \gamma_2:\; H\leftarrow d\wedge G$$

be clauses in P_k. Then we derive the clause

$$\gamma:\; H\leftarrow(c\vee d)\wedge G$$

and the new program $P_{k+1}=(P_k-\{\gamma_1,\gamma_2\})\cup\{\gamma\}$.

R7. Clause Splitting. Let

$$\gamma:\; H\leftarrow(c\vee d)\wedge G$$

be a clause in P_k. Then we derive the clauses

$$\gamma_1:\; H\leftarrow c\wedge G \qquad\qquad \gamma_2:\; H\leftarrow d\wedge G$$

and the new program $P_{k+1}=(P_k-\{\gamma\})\cup\{\gamma_1,\gamma_2\}$.

The following result ensures the correctness of the transformation rules w.r.t. the least model semantics.

Theorem 1: Let $P_0,\ldots,P_n$ be a transformation sequence. Suppose that, for every $k\in\{0,\ldots,n-1\}$ such that P_{k+1} is derived by folding clauses $\gamma_1,\ldots,\gamma_m$ in P_k using clauses $\delta_1,\ldots,\delta_m$ in $Defs_k$, one of the following conditions holds:

(1) for $i = 1, \ldots, m$, clause δ_i is unfolded during the construction of $P_0, \ldots, P_n$; or (2) the head predicate of $\delta_1, \ldots, \delta_m$ does not depend on itself in P_n.
Then $lm(P_0 \cup Defs_n, \mathcal{D}) = lm(P_n, \mathcal{D})$.

The rules listed above are an extension of the rules for transforming logic programs and constraint logic programs presented in [2], [6], [7], [12], [19], [21]. In particular, the folding rules considered in [2], [6], [7], [21] allow us to fold only one clause at a time, while by using our rule R3 we can fold m (≥ 1) clauses simultaneously. Our rule R3 is an adaptation to the case of CLP programs of the folding rules considered in [12], [19]. Our clause splitting rule R7 generalizes to constraint logic programs the case splitting rule for logic programs presented in [19]. The folding and clause splitting rule play a crucial role in the strategy for deriving deterministic programs presented in the next section.

V. A Strategy for Deriving Deterministic Specialized Programs

In this section we present the *Determinization Strategy* for guiding the application of the transformation rules. By applying this strategy we can derive deterministic, specialized programs starting from nondeterministic, general ones.

A. Determinism and Modes

We say that a program P is *deterministic w.r.t. a constrained atom* $c_0 \wedge A_0$ iff for all constrained goals $c \wedge A \wedge G$ such that $c_0 \wedge A_0 \mapsto_P^* c \wedge A \wedge G$, there exists *at most one* clause γ in P with a renamed apart variant $H_1 \leftarrow c_1 \wedge G_1$ such that the constraint $c \wedge A = H_1 \wedge c_1$ is satisfiable.

Given a constrained atom, the determinism of a program may depend on whether or not the variables in the atom are grounded by the constraint [14]. Recall that a variable X is said to be *grounded* by a constraint c iff $\mathcal{D} \models \exists Y \forall Z\, (c \rightarrow X = Y)$, where Y is a new variable and $Z = FV(c) \cup \{X\}$ (i.e., there is at most one value for X which makes c satisfiable). For instance, the following program over integers:

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

is deterministic w.r.t. the constrained atom $X = 1 \wedge p(X, Y)$ (where X is grounded by $X = 1$), while it is not deterministic w.r.t. the constrained atom $X \leq 1 \wedge p(X, Y)$ (where X is not grounded by $X \leq 1$). For this reason we now introduce the notion of *mode* which provides information about the groundness of the variables occurring in constrained atoms.

A *mode* M is a set of expressions of the form $p(m_1, \ldots, m_h)$, called a *mode for the predicate* p, such that: (i) p is a user defined predicate, (ii) for each p there exists at most one expression $p(m_1, \ldots, m_h)$, and (iii) for $i = 1, \ldots, h$, m_i is either $+$ (meaning that every variable in the i-th argument of p is grounded by some constraint) or ? (meaning that the i-th argument of p is any term). A mode M is a *mode for a program* P iff there exists in M a mode for each user defined predicate occurring in P.

Given an atom $p(t_1, \ldots, t_h)$ and a mode M with the element $p(m_1, \ldots, m_h)$, (1) for $i = 1, \ldots, h$, the term t_i is said to be an *input argument* of p (relative to M) iff m_i is $+$, and (2) a variable of $p(t_1, \ldots, t_h)$ which occurs in an input argument of p, is said to be an *input variable* of $p(t_1, \ldots, t_h)$.

Definition 1: Let P be a program and M be a mode for P. We say that a constrained atom $c \wedge p(t_1, \ldots, t_h)$ *satisfies* M iff $p(m_1, \ldots, m_h) \in M$ and for $i = 1, \ldots, h$, if m_i is $+$ then every variable in t_i is grounded by c. We say that P *satisfies* M iff for each constrained atom $c_0 \wedge A_0$ which satisfies M, and for each constrained goal $c \wedge A \wedge G$ such that $c_0 \wedge A_0 \mapsto_P^* c \wedge A \wedge G$, we have that $c \wedge A$ satisfies M.

Often the property that a program satisfies a mode can be automatically verified by abstract interpretation methods [11].

We say that a program P is *deterministic w.r.t. a mode* M iff P is deterministic w.r.t. every constrained atom $c_0 \wedge A_0$ which satisfies M. Now we give a sufficient condition which ensures that a program is deterministic w.r.t. a mode. We need the following definition.

Definition 2: Let us consider the following two clauses without variables in common:

$\gamma_1 : p(t_1, \ldots, t_h, u_1, \ldots, u_k) \leftarrow c_1 \wedge G_1$
$\gamma_2 : p(v_1, \ldots, v_h, w_1, \ldots, w_k) \leftarrow c_2 \wedge G_2$

where p is a k-ary predicate whose first h arguments are input arguments relative to a given mode M. We say that γ_1 and γ_2 are *mutually exclusive* w.r.t. M iff $\mathcal{D} \models \neg\exists (t_1 = v_1 \wedge \ldots \wedge\ t_h = v_h \wedge c_1 \wedge c_2)$.

Proposition 1: Let P be a program and M be a mode for P. If P satisfies M and the clauses of P are pairwise mutually exclusive w.r.t. M, then P is deterministic w.r.t. M.

B. The Determinization Strategy

Our Determinization Strategy is based upon the following three subsidiary strategies: (i) *Unfold-Simplify*, which uses the unfolding, clause removal, and constraint replacement rules, (ii) *Partition*, which uses the clause removal, constraint replacement, clause fusion, and clause splitting rules, and (iii) *Define-Fold*, which uses the definition and folding rules.

Let us consider an initial program P, a mode M for P, and a constrained atom $c \wedge p(t_1, \ldots, t_h)$, with $FV(c) \subseteq vars(t_1, \ldots, t_h)$. In order to specialize P w.r.t. $c \wedge p(t_1, \ldots, t_h)$, we introduce, by the definition rule, the clause

δ_{sp}: $p_{sp}(X_1, \ldots, X_r) \leftarrow c \wedge p(t_1, \ldots, t_h)$

where $X_1, \ldots, X_r$ are the distinct variables occurring in $p(t_1, \ldots, t_h)$. The mode $p_{sp}(m_1, \ldots, m_r)$ for the predicate p_{sp} is the following: for $j = 1, \ldots, r$, m_j is $+$ iff X_j is an input variable of $p(t_1, \ldots, t_h)$ relative to M. We assume that P satisfies M and thus, the program $P \cup \{\delta_{sp}\}$ satisfies $M \cup \{p_{sp}(m_1, \ldots, m_r)\}$.

Our Determinization Strategy is an iterative procedure that at each iteration manipulates the following three sets of clauses: (1) *Defs*, which is the set of clauses introduced so far by the definition rule, (2) *Cls*, which is the set of clauses to be transformed during the current iteration, and (3) P_{sp}, which is the specialized program derived so far. Initially, both *Defs* and *Cls* consist of the single clause δ_{sp}. From the set *Cls* a new set of deterministic clauses is derived by applying the transformation rules according to the *Unfold-Simplify*, *Partition*, and *Define-Fold* subsidiary strategies. This new set of deterministic clauses is added to P_{sp}. During each iteration, in order to derive deterministic clauses, we may need to introduce new predicates, whose defining clauses are stored in the set *NewDefs*. At the end of each iteration *NewDefs* is added to *Defs*, and the value of the set *Cls* is updated to *NewDefs*. The transformation strategy terminates when $Cls=\emptyset$, that is, when no new predicate is introduced during the current iteration.

The following definition is needed for presenting the *Unfold-Simplify* subsidiary strategy.

Definition 3: Let $H \leftarrow c \wedge G' \wedge A \wedge G''$ be a clause in a program P and let M be a mode for P. We say that A is a *consumer atom* iff for every clause $H_1 \leftarrow c_1 \wedge G_1$ in P, we have that one of the following conditions holds:
(i) G_1 is the empty conjunction; or
(ii) $c \wedge A = H_1 \wedge c_1$ is unsatisfiable; or
(iii) $\mathcal{D} \models \forall (c \rightarrow \exists Y (A = H_1))$ where $Y = \{X \in FV(A = H_1) \mid X$ is not an input variable of A relative to $M\}$.

During the *Unfold-Simplify* subsidiary strategy we unfold w.r.t. consumer atoms. In particular, when Condition (iii) of Definition 3 holds, we unfold w.r.t. atoms whose input arguments are instances of the corresponding arguments in the heads of the clauses of P.

Determinization Strategy

Input: A program P, a mode M for P such that P satisfies M, and a clause
δ_{sp}: $p_{sp}(X_1, \ldots, X_r) \leftarrow c \wedge p(t_1, \ldots, t_h)$

Output: A specialized program P_{sp} and a mode M_{sp} for P_{sp}.

Initialize: $Defs := \{\delta_{sp}\}$; $Cls := \{\delta_{sp}\}$; $P_{sp} := \emptyset$;
$M_{sp} := \{p_{sp}(m_1, \ldots, m_r)\}$;

while $Cls \neq \emptyset$ **do**

(1) *Unfold-Simplify*:
$UnfCls := \{\eta \mid \eta$ is a constrained fact in *Cls* or it is derived by unfolding a clause in *Cls* w.r.t. the leftmost atom in its body$\}$;
while there exists γ in *UnfCls* with a leftmost consumer atom A in the body of γ **do**
$UnfCls := (UnfCls - \{\gamma\}) \cup \{\eta \mid \eta$ is derived by unfolding γ w.r.t. $A\}$;
$UnfCls := \{H \leftarrow \widetilde{c} \wedge G \mid$ there exists $H \leftarrow c \wedge G$ in *UnfCls* such that: (i) $\widetilde{c} = solve(c, vars(H, G))$, (ii) c is satisfiable, and (iii) $H \leftarrow c \wedge G$ is not subsumed by any other clause in $UnfCls\}$.

(2) *Partition*: We apply the clause removal, constraint replacement, clause fusion, and clause splitting rules, and from *UnfCls* we derive a set *PartCls* of clauses which is the union of disjoint subsets, called *packets*, such that the following two properties hold.
(i) Each packet is a set of clauses of the form:

$H \leftarrow c \wedge d_1 \wedge G_1$
$\ldots$
$H \leftarrow c \wedge d_m \wedge G_m$

In particular, if for $i=1, \ldots, m$, G_i is the empty conjunction, then by clause fusion we derive a packet consisting of one constrained fact only.
(ii) Any two clauses belonging to different packets are mutually exclusive w.r.t. mode M_{sp}.

(3) *Define-Fold*: Let *CFacts* be the union of the packets in *PartCls* consisting of constrained facts only, and let *NonCFacts* be the union of all other packets. Let *NewDefs* be a (possibly empty) set of new clauses introduced by the definition rule such that each packet in *NonCFacts* can be folded by using clauses in $Defs \cup NewDefs$ of the form:

$newp(X_1, \ldots, X_r) \leftarrow d_1 \wedge G_1$
$\ldots$
$newp(X_1, \ldots, X_r) \leftarrow d_m \wedge G_m$

whereby deriving a single clause of the form:

$H \leftarrow c \wedge newp(X_1, \ldots, X_r)$

When we introduce *NewDefs* and perform folding, we also make sure that Condition (1) or (2) of Theorem 1 holds.
For each new predicate *newp* in *NewDefs*, we add to M_{sp} the mode $newp(m_1, \ldots, m_r)$ defined as follows: for $i = \ldots r$, $m_i = +$ iff X_i is either an input variable of H or an input variable of the leftmost atom of one of the goals $G_1, \ldots, G_m$.
Let *FldCls* be the set of clauses derived by folding the packets in *NonCFacts*.

(4) $Defs := Defs \cup NewDefs$; $Cls := NewDefs$;
$P_{sp} := P_{sp} \cup CFacts \cup FldCls$

end-while

Now we see the Determinization Strategy in action on the matching example of Section II. This will explain how the specialized program (clauses 9, 16–27) has been automatically derived.

We are given the clauses 1–5, the mode $M = \{m(+,+), a(?,?,+), le(+,+)\}$, and $\delta_{sp} =$ clause 6. Thus, initially, $Defs = Cls = \{$clause 6$\}$ and $M_{sp} = \{m_{sp}(+)\}$. Since $Cls \neq \emptyset$, we execute the body of the while-loop and we unfold clause 6 w.r.t. $m([1, 0, 2], S)$ and we get:

7. $m_{sp}(S) \leftarrow a(B, C, S) \wedge a(A, Q, B) \wedge le([1, 0, 2], Q)$

Clause 7 is a packet in itself and in order to fold it, we introduce the following definition:

8. $new1(S) \leftarrow a(B, C, S) \wedge a(A, Q, B) \wedge le([1, 0, 2], Q)$

and then we fold clause 7, whereby getting:

9. $m_{sp}(S) \leftarrow new1(S)$

Now $Defs = \{$clause 6, clause 8$\}$, $Cls = \{$clause 8$\}$, and $M_{sp} = \{m_{sp}(+), new1(+)\}$. Since $Cls \neq \emptyset$, we execute once more the body of the while-loop and we unfold clause 8 w.r.t. the atoms a and le. We get:

10. $new1([X|Xs]) \leftarrow 1 \leq X \wedge a(Q, C, Xs) \wedge le([0, 2], Q)$
11. $new1([X|Xs]) \leftarrow a(B, C, Xs) \wedge a(A, Q, B) \wedge le([1, 0, 2], Q)$

Since clause 10 and 11 are not mutually exclusive w.r.t. M_{sp}, we apply the clause splitting rule to clause 11, whereby getting:

12. $new1([X|Xs]) \leftarrow 1 \leq X \wedge a(B,C,Xs) \wedge$
$a(A,Q,B) \wedge le([1,0,2],Q)$

13. $new1([X|Xs]) \leftarrow 1 > X \wedge a(B,C,Xs) \wedge$
$a(A,Q,B) \wedge le([1,0,2],Q)$

We have two packets: (i) {clause 10, clause 12} and (ii) {clause 13}. In order to fold the first packet we introduce the following definition:

14. $new2(Xs) \leftarrow a(Q,C,Xs) \wedge le([0,2],Q)$

15. $new2(Xs) \leftarrow a(B,C,Xs) \wedge a(A,Q,B) \wedge$
$le([1,0,2],Q)$

We fold clauses 10 and 12 by using clauses 14 and 15, and we fold clause 13 by using clause 8. We get the following mutually exclusive clauses:

16. $new1([X|Xs]) \leftarrow 1 \leq X \wedge new2(Xs)$

17. $new1([X|Xs]) \leftarrow 1 > X \wedge new1(Xs)$

Now $Defs = \{$clause 6, clause 8, clause 14, clause 15$\}$, $Cls = \{$clause 14, clause 15$\}$, and $M_{sp} = \{m_{sp}(+), new1(+), new2(+)\}$. Since $Cls \neq \emptyset$, the derivation continues by executing again the body of the while-loop. Thus, we unfold the clauses 14 and 15. We will not give all the details of the derivation here. We eventually get the specialized, deterministic program of Section II.

The termination of our Determinization Strategy depends on the finiteness of (i) the unfolding subsidiary strategy and (ii) the set of definitions which are introduced for performing folding steps. In particular, for ensuring termination it may be necessary to consider suitable generalizations of the bodies of the clauses to be folded (see, for instance, [5], [7], [10], [15], [16], [22]).

As a consequence of Theorem 1, if the Determinization Strategy terminates, then the specialized program P_{sp} is equivalent to the initial program P in the following sense: for every constraint d,

$lm(P,\mathcal{D}) \models \exists(d \wedge c \wedge p(t_1,\ldots,t_h))$ iff
$lm(P_{sp},\mathcal{D}) \models \exists(d \wedge p_{sp}(X_1,\ldots,X_r))$.

Moreover, by construction, P_{sp} satisfies M_{sp} and its clauses are pairwise mutually exclusive w.r.t. M_{sp}. Thus, by Proposition 1, P_{sp} is deterministic w.r.t. M_{sp}. In particular, for every constraint d such that $d \wedge c \wedge p(t_1,\ldots,t_h)$ satisfies M_{sp}, we have that P_{sp} is deterministic w.r.t. $d \wedge p_{sp}(X_1,\ldots,X_r)$.

VI. Conclusions

We have introduced a new transformation rule, called *clause splitting*, which can be used for reducing the nondeterminism when specializing constrained logic programs. This rule allows us to *reason by cases* as often done in various specialization techniques (see, for instance, [9], [13], [22]). Clause splitting, together with the other familiar unfolding and folding rules, is applied according to the *Determinization Strategy* which is an enhancement of *conjunctive partial deduction* [5]. Indeed, we allow new predicates to be defined in terms of *disjunctions of conjunctions* of constrained atoms. The Determinization Strategy is an extension to constraint logic programs of the strategy presented in [19]. We have used our strategy for specializing constrained matching algorithms and we have derived efficient programs which correspond to deterministic finite automata with transitions labelled by constraints.

References

[1] K. R. Apt. Introduction to logic programming. In J. van Leeuwen, ed., *Handbook of Theoretical Computer Science*, 493–576. Elsevier, 1990.

[2] N. Bensaou and I. Guessarian. Transforming constraint logic programs. *Theor. Comp. Sci.*, 206:81–125, 1998.

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

[4] O. Danvy, R. Glück, and P. Thiemann, eds. *Partial Evaluation*, LNCS 1110. Springer, 1996.

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

[6] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theor. Comp. Sci.*, 166:101–146, 1996.

[7] F. Fioravanti, A. Pettorossi, and M. Proietti. Automated strategies for specializing constraint logic programs. *LOPSTR 2000*, LNCS 2042, 125–146. Springer, 2001.

[8] H. Fujita. An algorithm for partial evaluation with constraints. Tech. Memo. 0367, ICOT, Tokyo, Japan, 1987.

[9] Y. Futamura, K. Nogi, and A. Takano. Essence of generalized partial computation. *Theor. Comp. Sci.*, 90:61–79, 1991.

[10] J. P. Gallagher. Tutorial on specialisation of logic programs. *PEPM '93, Copenhagen, Denmark*, 88–98. ACM Press, 1993.

[11] M. Garcia de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global analysis of constraint logic programs. *ACM Toplas*, 18(5):564–614, 1996.

[12] M. Gergatsoulis and M. Katzouraki. Unfold/fold transformations for definite clause programs. *PLILP '94*, LNCS 844, 340–354. Springer, 1994.

[13] R. Glück and A.V. Klimov. Occam's razor in metacomputation: the notion of a perfect process tree. *WSA '93*, LNCS 724, 112–123. Springer, 1993.

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

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

[16] M. Leuschel, B. Martens, and D. De Schreye. Controlling generalization and polyvariance in partial deduction of normal logic programs. *ACM Toplas*, 20(1):208–258, 1998.

[17] J. W. Lloyd. *Foundations of Logic Programming*. Springer, 1987. Second Edition.

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

[19] A. Pettorossi, M. Proietti, and S. Renault. Reducing nondeterminism while specializing logic programs. *POPL '97, Paris, France*, 414–427. ACM Press, 1997.

[20] D. A. Smith. Partial evaluation of pattern matching in constraint logic programming languages. *PEPM '91*, SIGPLAN Notices, 26, 9, 62–71. ACM Press, 1991.

[21] H. Tamaki and T. Sato. Unfold/fold transformation of logic programs. *ICLP '84*, 127–138. Uppsala, Sweden, 1984.

[22] V. F. Turchin. The concept of a supercompiler. *ACM Toplas*, 8(3):292–325, 1986.