

# A Transformation System for CLP with Dynamic Scheduling and CCP

Sandro Etalle<sup>†</sup>

Maurizio Gabbrielli<sup>§</sup>

Elena Marchiori<sup>¶</sup>

## Abstract

In this paper we study unfold/fold transformations for constraint logic programs (CLP) with dynamic scheduling and for concurrent constraint programming (CCP). We define suitable applicability conditions for these transformations which guarantee that the original and the transformed program have the same results of successful derivations and the same deadlock free queries.

The possible applications of these results are twofold. On one hand we can use the unfold/fold system to optimize CLP and CCP programs while preserving their intended meaning and in particular without the risk of introducing deadlocks. On the other hand, unfold/fold transformations can be used for proving deadlock freeness of a class of queries in a given program: to this aim it is sufficient to apply our transformations and to specialize the resulting program with respect to the given queries in such a way that the obtained program is trivially deadlock free. As shown by several interesting examples, this yields a methodology for proving deadlock freeness which is simple and powerful at the same time.

*Keywords:* Transformation, Concurrent Constraint Logic Programming, Coroutining, Deadlock.

## 1 Introduction

Constraint Logic Programming (CLP for short) is a powerful declarative programming paradigm which has been successfully applied to several diverse fields such as financial analysis [22], circuit synthesis [15] and combinatorial search problems [35]. The main reason for the interest in this paradigm is that it combines in a clean and effective way the power of constraint solving with logical deduction. In this way complex practical problems can be tackled by using reasonably simple and concise programs.

<sup>†</sup>WINS, University of Amsterdam, Kruislaan 403 1098SJ Amsterdam, The Netherlands. [sandro@wins.uva.nl](mailto:sandro@wins.uva.nl)

<sup>§</sup>Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy. [gabbri@di.unipi.it](mailto:gabbri@di.unipi.it).

<sup>¶</sup>Dipartimento di Matematica Applicata e Informatica, Università Ca' Foscari di Venezia, Via Torino 153, Venezia-Mestre, Italy. [elena@dsi.unive.it](mailto:elena@dsi.unive.it).

Permission to make digital/hard copy of part or all this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage, the copyright notice, the title of the publication and its date appear, and notice is given that copying is by permission of ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists, requires prior specific permission and/or a fee. PEPM '97 Amsterdam, ND

© 1997 ACM 0-89791-917-3/97/0006...\$3.50

In nearly all the practical CLP systems [19] the flexibility of CLP computation is further enhanced by allowing a dynamic selection rule which allows the computation to dynamically delay (or suspend) the selection of an atom until its arguments are sufficiently instantiated. This *dynamic scheduling* is obtained by adding the so-called *delay declarations* next to program clauses. Delay declarations, advocated by van Emden and de Lucena [34] and introduced explicitly in logic programming by Naish in [27], allow one to improve the efficiency of programs, to prevent run-time errors and to enforce termination. In many CLP systems they are used also to postpone the evaluation of constraints which are “too hard” for the solver. For example, in CLP( $\mathbb{R}$ ) [18] non-linear arithmetic constraints are delayed until they become linear. More generally, delay declarations provide the programmer with a better control over the computation and, similarly to guards in concurrent logic languages, allow one to express some degree of synchronization, usually called coroutining, among the different processes (i.e. atoms) in a program. In this sense, even though the underlying computational model uses a different kind of non-determinism, CLP with dynamic scheduling can be considered as an intermediate language between the purely sequential CLP and the concurrent language CCP.

The increased expressiveness of the language mentioned above comes with a price: in presence of dynamic scheduling the computation can end up into a state in which no atom can be selected for resolution. This situation, analogous to the one which can arise when considering concurrent (constraint) languages, is called *deadlock* and is clearly undesirable, since it corresponds to a programming error. Checking whether the computation for a generic query in a program will end up in a deadlock is an undecidable yet crucial problem. In order to give it a partial solution, several techniques have been employed. Among them we should mention those based on abstract interpretation (in [6, 7]), mode and type analysis (in [1, 12]) and assertions (in [25, 4]).

A central issue for the development of large, correct and efficient applications is the study of optimization techniques. When considering (constraint) logic programs, the literature on this subject can be divided into two main branches. On one hand, we find several methods for compile-time optimization based on abstract interpretation. These methods have been recently applied also to CLP with dynamic scheduling [10, 26, 9] with promising results.

On the other hand, there are techniques based on transformation of programs such as unfold/fold and replacement (cf. [28]). As shown by several applications, these tech-

niques provide a powerful methodology for program development, synthesis and optimization. Historically, the unfold/fold transformation rules, were first introduced for functional programs in [3], and then adapted to logic programs both for program synthesis [5, 16], and for program specialization and optimization [20]. Soon after, Tamaki and Sato [32] proposed an elegant framework for the unfold/fold transformation of logic programs which has recently been extended to CLP in [2, 24, 13] (for an overview of the subject, see the survey by Pettorossi and Proietti [28]).

However, these systems cannot be in general applied as they are to languages with dynamic scheduling. This is due to the following reasons. First, as we have already mentioned, a peculiar feature of such languages is that the computation might deadlock. It is therefore crucial that the transformation system does not introduce deadlock derivations in the resulting program for a query that was deadlock free in the original program. For this to happen, new applicability conditions and new correctness results are needed. Secondly, languages with dynamic scheduling (often) use special constructs (similar to the ask of CCP) for providing simple yet powerful suspension tools. In order to handle these tools one needs new, specific transformations.

In this paper we handle these problems as follows. First, we concentrate on CLP with dynamic scheduling, enhanced with an if - then - else construct, and we extend the transformation system defined in [13]. The extension is twofold: on one hand we provide new applicability conditions which guarantee that the initial and the transformed program have exactly the same answer constraints and the same deadlock free queries. On the other hand we define a new transformation for handling the if - then - else parts. Thanks to these new operations we can now perform optimization which were not possible in [13]. Further, we show that the resulting transformation system can be smoothly extended to CCP. Actually, in this case, due to the more expressive language, we can weaken the applicability conditions and still have a correctness result which ensures us that results of successful computations and deadlock free queries are preserved.

These results for CLP and CCP have applications to both the practical issues mentioned above.

Firstly, they can be used for optimizing programs while preserving their intended meaning and, in particular, without the risk of introducing deadlocked derivations. We will show examples of such optimizations.

Secondly, unfold/fold transformations can be used for proving deadlock freeness of a class of (intended) queries for a given program. To this aim, it is sufficient to apply our transformations, to restrict the transformed program with respect to the considered queries, and to prove deadlock freeness of the resulting program. This latter proof is in many cases trivial, either because the resulting CLP program has no delay declarations at all or because the guards of the resulting CCP program are trivially satisfied. In this way we obtain a method for proving deadlock freeness that is simple and powerful at the same time. Its simplicity stems from the fact that it only uses transformation operations, thus it does not have to introduce other formalisms, like modes, types or assertions, as in the above mentioned proof methods, or like abstract domains, as in the methods based on abstract interpretation. Its power will be demonstrated in the paper by applying it to several non trivial programs whose execution uses full corouting.

This paper is organized as follows. The next section contains some preliminaries on CLP with dynamic scheduling

and on the if - then- else construct (some more details are deferred to the Appendix). Section 3 contains two examples which illustrate the use of transformations for optimizing programs and proving deadlock freeness. In Section 4 we define formally the transformation system and Section 5 shows its application to the proof of deadlock freeness. Finally in Section 6 we extend the previous system and results to the CCP languages.

## 2 Preliminaries

We recall here some basic notions on CLP with dynamic scheduling, deferring to the Appendix some more details concerning the computational model of this language. For a thorough treatment of the theory of constraint logic programming, the reader is referred to the original paper [17] by Jaffar and Lassez and to the survey [19] by Jaffar and Maher.

A *constraint*  $c$  is a (first-order) formula built using pre-defined predicates (called primitive constraints) over a computational domain  $\mathcal{D}$ . Formally,  $\mathcal{D}$  is a *structure* which determines the interpretation of the constraints, and the formula  $\mathcal{D} \models \forall c$  denotes that  $c$  is true under the interpretation provided by  $\mathcal{D}$ , for every binding of its free variables. The empty conjunction of primitive constraints will be identified with true. A CLP rule is a formula of the form

$$H \leftarrow c, B_1, \dots, B_n.$$

where  $c$  is a constraint,  $H$  (the head) is an atom and  $B_1, \dots, B_n$  (the body) is a sequence of atoms<sup>1</sup> and the connective “,” denotes conjunction. Analogously, a *query* is denoted by  $c, B_1, \dots, B_n$ . In the sequel,  $\vec{t}$  and  $\vec{X}$  denote a tuple of terms and a tuple of distinct variables, respectively, while  $\vec{B}$  denotes a (finite, possibly empty) conjunction of atoms. For a formula  $\phi$ , its existential closure is denoted by  $\exists \phi$ , while  $\exists_{-\vec{x}} \phi$  stands for the existential closure of  $\phi$  *except* for the variables in  $\vec{x}$  which remain unquantified. Moreover, for two atoms  $A$  and  $H$ , the expression  $A = H$  is used as shorthand for:

-  $s_1 = t_1 \wedge \dots \wedge s_n = t_n$ , if, for some predicate symbol  $p$ ,  $A \equiv p(s_1, \dots, s_n)$  and  $H \equiv p(t_1, \dots, t_n)$  (where  $\equiv$  denotes syntactic equality)

- false, otherwise. This notation extends to conjunctions of atoms in the expected way.

If  $\vartheta$  is a valuation, (i.e., a function mapping variables into elements of  $\mathcal{D}$ ), then  $c\vartheta$  denotes the application of  $\vartheta$  to the free variables of  $c$  further, if  $\mathcal{D} \models c\vartheta$  holds, then  $\vartheta$  is called a  *$\mathcal{D}$ -solution* of  $c$ . By naturally extending the usual notion used for pure logic programs, we say that a query  $c, \vec{C}$  is an *instance* of the query  $d, \vec{D}$  iff for any solution  $\gamma$  of  $c$  there exists a solution  $\delta$  of  $d$  such that  $\vec{C}\gamma \equiv \vec{D}\delta$ .

The standard operational model of CLP is obtained from SLD resolution by simply substituting  $\mathcal{D}$ -solvability for unifiability. As previously mentioned, in many CLP systems the programmer can dynamically control the selection of the atoms in a derivation by augmenting a program with delay declarations. These allow one to specify that the selection of an atom can be delayed, or suspended, until its arguments become sufficiently instantiated. So we consider declarations of the form

delay  $p(\vec{X})$  until Condition

<sup>1</sup>It is assumed that atoms use predicate symbols different from those of the primitive constraints.

where Condition is a formula (in some assertion language) whose free variables are contained in  $\tilde{X}$ .

We consider as delay conditions formulas consisting of conjunctions and disjunctions of primitive constraints and of “selection predicates” (for example  $\text{ground}(X)$  is such a predicate). The resulting assertion language, as well as its interpretation, is formalized in the Appendix. The interpretation simply defines for each delay condition  $\phi$  the set  $[\phi]$  of constraints which satisfy  $\phi$ , where  $[\phi]$  is assumed being closed under entailment. This assumption corresponds to the fact that in most of the existing programming systems, if an atom is not delayed then adding more information will never cause it to become delayed. Furthermore, delay conditions are usually assumed to be consistent wrt renaming, so we make the same assumption here. The Appendix contains a more precise formalization of these conditions.

We suppose that for each predicate symbol exactly one delay declaration is given. This is not restrictive, since multiple declarations can be obtained by using logical connectives in the syntax of conditions. When the condition holds vacuously (i.e., when it is satisfied by all the constraints) we simply omit the corresponding declaration.

Intuitively, the meaning of a delay declaration  $\text{delay } p(\tilde{X})$  until Condition is that the atom  $p(\tilde{X})$  can be selected in a query  $c, Q$  only if the constraint  $c$  satisfies Condition (see Definition A.3 in the Appendix). This notion is generalized to an atom  $p(\tilde{t})$  in the expected way as follows: let  $\text{delay } p(\tilde{X})$  until  $\phi$  be the delay declaration for  $p$ , and  $c$  be a constraint, we say that  $p(\tilde{t})$  satisfies its delay declaration in the context  $c$  if  $(c \wedge \tilde{t} = \tilde{Y})$  satisfies  $\phi[\tilde{X}/\tilde{Y}]^2$ , where  $\tilde{Y}$  are new distinct variables which do not appear in  $c$  and in  $\tilde{t}$ .

For example, consider a numerical constraint system like the one used in CLP( $\mathcal{R}$ ): the declaration  $\text{delay } p(X)$  until  $X > 0$  means that  $p(X)$  can be selected in a query  $c, Q$  only if  $c$  forces  $X$  to assume values greater than 0 (that is if  $c$  entails  $X > 0$ ), while the declaration  $\text{delay } q(X)$  until  $\text{ground}(X)$  means that  $q(X)$  can be selected in a query  $c, Q$  only if  $c$  forces  $X$  to assume a unique value (that is, if all valuations which render  $c$  true in  $\mathcal{D}$  bind  $X$  to the same value)<sup>3</sup>.

A derivation step for a query  $c, Q$  in a program  $P$  consists in replacing an atom  $A$  selected in  $Q$  with the body of a (suitably renamed version of a) clause  $cl$ , provided that the constraint in  $cl$  is consistent with  $c$ , no variable clash occurs, and  $A$  in the context of  $c$  satisfies its delay declaration.

A derivation for a query  $Q$  in the program  $P$  is a maximal sequence of queries, starting from  $Q$ , such that every next query is obtained from the previous one by means of a derivation step. In the following a derivation  $Q, Q_1, \dots, Q_i$  in  $P$  will be denoted by  $Q \xrightarrow{P} Q_i$ .

A successful derivation (or refutation) for the query  $Q$  is a finite derivation whose last element is of the form  $c$ , i.e. consisting only of a constraint. In this case,  $\exists\text{-vars}(Q) c$  is called the answer constraint of  $Q$ . Since the answer constraint represents the result of a successful computation, it is considered the standard observable property for CLP programs. Therefore we will consider it as the notion of observable to be preserved by our transformation system.

A *deadlocked derivation* is a finite derivation  $Q \xrightarrow{P} c, \tilde{B}$

<sup>2</sup> $\phi[\tilde{X}/\tilde{Y}]$  denotes the formula obtained from  $\phi$  by replacing the variables  $\tilde{X}$  for the variables  $\tilde{Y}$ .

<sup>3</sup>The notion of groundness here considered is the generalization to CLP of the notion of groundness used in LP, as shown e.g. in [23] (there called determinateness).

such that  $\tilde{B}$  is a non-empty sequence of atoms, and each atom in  $\tilde{B}$  does not satisfy its delay declaration. Note that  $\tilde{B}$  does not contain constraints: In fact, usually the case in which a derivation ends in a query containing only suspended constraints is considered successful. A query  $Q$  *deadlocks (in P)* if there exists a deadlocked derivation for  $Q$ , while  $Q$  is *deadlock free* if it does not deadlock.

Observe that the addition of delay declarations to CLP does not affect its declarative nature. In fact, it has been shown [8] that CLP with dynamic scheduling admits a simple compositional denotational semantics that can be translated into a logic for proving partial correctness, where the logical meaning of delay declarations can be understood as implication.

## 2.1 Adding the if - then - else

The if - then - else is present in logic programming languages since their appearance, but it has always been rather overlooked by theoreticians and considered only as “syntactic sugar” for programs fragments involving cuts or negation-as-failure. The use of delay declarations in a CLP language allows us to usefully employ a restricted form of if - then - else construct, which still retains a simple declarative meaning. In this context the presence of this construct is crucial for two reasons: (i) it allows transformations which otherwise would not be possible; (ii) it allows us to introduce in the queries new suspension points without having to create new artificial predicates. Thus in the following we shall allow in a query constructs of the form

$$\text{if } c \text{ then } \tilde{A} \text{ else } \tilde{B}$$

where  $c$  is a constraint and  $\tilde{A}$  and  $\tilde{B}$  are queries. Operationally, the meaning of such a construct is that the execution of the queries  $\tilde{A}$  and  $\tilde{B}$  is delayed until the current store either entails  $c$  or entails  $\neg c$ , i.e., until it is known which branch has to be selected. As soon as one of the two conditions is satisfied, the corresponding branch is selected and the other is discarded.

From a declarative viewpoint, the meaning of the previous construct is simply obtained by considering an occurrence of  $\text{if } c \text{ then } \tilde{A} \text{ else } \tilde{B}$  as a shorthand for an occurrence of the predicate  $\text{if\_then\_else}(\tilde{X})$  defined by the following program

$$\begin{aligned} \text{if\_then\_else}(\tilde{X}) &\leftarrow c, \tilde{A}. \\ \text{if\_then\_else}(\tilde{X}) &\leftarrow \neg c, \tilde{B}. \end{aligned}$$

$$\text{delay if\_then\_else}(\tilde{X}) \text{ until } c \vee \neg c.$$

Here  $\text{if\_then\_else}$  is a new predicate symbol and the free variables appearing in  $c$  are contained in  $\tilde{X}$ , with  $\tilde{X} = \text{vars}(\tilde{A}, \tilde{B})$ , where  $\text{vars}(E)$  denotes the set of variables of the expression  $E$ . Note that in the above clauses  $\neg c$  is a basic constraint: in fact negation is allowed at the “constraint level” and not at the level of delay declarations. Thus  $c \vee \neg c$  is a legal delay condition and according to the interpretation formally defined in the Appendix it is satisfied by any  $d$  such that either  $d$  entails  $c$  or  $d$  entails  $\neg c$ . Given this interpretation, clearly the delay condition  $c \vee \neg c$  in general is not equivalent to true. Notice also that the above delay declaration forces the if - then - else to suspend itself until it is known which branch is to be selected. We assume here that for each occurrence of an  $\text{if } c \text{ then } \tilde{A} \text{ else } \tilde{B}$  construct a different

if\_then\_else predicate symbol is introduced. It can be easily checked that the computational behavior of this program is equivalent to the one previously described for the if - then - else. The reader familiar with concurrent constraint programming has probably recognized in the above constructs a form of ask operator. We will discuss the relation existing between the two paradigms in Section 6.

### 3 Motivating Examples

In this section we illustrate what are the possibilities offered by program transformation in the context of CLP with dynamic scheduling. We will do this by giving two simple yet significant examples. In order to avoid making the reading too heavy, we postpone the formal definitions of the single operations (and their applicability conditions) until the next section.

To simplify the notation, here and in the sequel, when the constraint in a query or in a clause is true we omit it. So the notation  $H \leftarrow \tilde{B}$  actually denotes the CLP clause  $H \leftarrow \text{true}, \tilde{B}$ .

Further, for the sake of readability every now and then we'll have to rename some variables and to "clean up" some constraints. Formally, these operations can be seen as replacements of a clause with an equivalent one, where the notion of equivalence ( $\simeq$ ) is defined as follows on clauses which do not contain if - then - else:  $H_1 \leftarrow c_1, \tilde{B}_1 \simeq H_2 \leftarrow c_2, \tilde{B}_2$  iff for  $i, j \in [1, 2]$  and any  $\mathcal{D}$ -solution  $\vartheta$  of  $c_i$  there exists a  $\mathcal{D}$ -solution  $\gamma$  of  $c_j$  such that  $H_i\vartheta \equiv H_j\gamma$  and  $\tilde{B}_i\vartheta$  and  $\tilde{B}_j\gamma$  are equal as multisets<sup>4</sup>. This notion of equivalence is extended to clauses containing the if - then- else by considering, in the obvious way, the  $\simeq$  equivalence of the clauses obtained by their translation in standard clauses shown above. It can be proven that replacing a clause by a  $\simeq$ -equivalent one does not affect the applicability and the results of the transformations, and that all the observable properties we refer to are invariant under  $\simeq$ .

**Example 3.1** The following program DELMAX allows one to delete all the occurrences of the maximum from a given list of integers. The relation  $\text{del\_max}(Xs, Zs)$  defines  $Zs$  to be obtained from  $Xs$  by deleting all the occurrences of its maximum element.

```
d1:del_max(Xs, Zs) ← find_max(Xs, Max), del_el(Xs, Max, Zs).

find_max([], 0).
find_max([X|Xs], Max) ← find_max(Xs, Max'),
    if Max' < X then Max = X
    else Max = Max'.

del_el([], -, []).
del_el([X|Xs], El, OutList) ← del_el(Xs, El, OutList'),
    if El = X then OutList = OutList'
    else OutList = [X|OutList'].
```

Given the list  $Xs$  of integer values,  $\text{del\_max}(Xs, Zs)$  produces the list  $Zs$  by deleting all the occurrences of its maximum element and in order to do this it scans the input list  $Xs$

<sup>4</sup>It is worth noticing that considering sets here would not be correct, since we are interested in equivalences which preserve the answer constraints. For example, assume that the predicate  $q$  is defined by the two unit clauses  $q(a, Z)$  and  $q(Z, a)$ . Then the clause  $p(X, Y) \leftarrow q(X, Y), q(X, Y)$  allows us to obtain the answer  $X = a, Y = a$  for the query  $p(X, Y)$ , while this is not the case for the clause  $p(X, Y) \leftarrow q(X, Y)$ .

twice. To this aim, first  $\text{find\_max}(Xs, \text{Max})$  computes the maximum  $\text{Max}$  of  $Xs$ , and then  $\text{del\_el}(Xs, \text{Max}, Zs)$  computes  $Zs$  from  $Xs$  by deleting all the occurrences of  $\text{Max}$  from it.

Now, via an unfold/fold transformation we can obtain an equivalent program which scans the list only once. For this we apply a transformation strategy known as *tupling* (cf. [28]) or as *procedural join* (cf. [21]). First, we add to the program the *new* predicate  $\text{find\_max\_and\_del}(\text{InList}, \text{Max}, \text{El}, \text{OutList})$  which given a list  $\text{InList}$ , computes its maximum element  $\text{Max}$  and the list  $\text{OutList}$  obtained from  $\text{InList}$  by deleting all the occurrences of  $\text{El}$  from it.

```
d2:find_max_and_del(InList, Max, El, OutList) ←
    find_max(InList, Max),
    del_el(InList, El, OutList).
```

Now, we *unfold*  $\text{find\_max}(\text{InList}, \text{Max})$  in the body of  $d2$ . This basic operation corresponds to applying a resolution step (in all possible ways). In this case it originates the following two clauses.

```
find_max_and_del([], 0, El, OutList) ← del_el([], El, OutList).
find_max_and_del([X|Xs], Max, El, OutList) ←
    find_max(Xs, Max'),
    if Max' < X then Max = X else Max = Max',
    del_el([X|Xs], El, OutList).
```

By further unfolding  $\text{del\_el}$  in both the above clauses, we obtain,

```
find_max_and_del([], 0, -, []).
d3:find_max_and_del([X|Xs], Max, El, OutList) ←
    find_max(Xs, Max'),
    if Max' < X then Max = X else Max = Max',
    del_el(Xs, El, OutList'),
    if El = X then OutList = OutList'
    else OutList = [X|OutList'].
```

Now, we can *fold*  $\text{find\_max}(Xs, \text{Max}')$ ,  $\text{del\_el}(Xs, \text{El}, \text{OutList}')$  in the body of  $d3$ , using  $d2$  as folding clause. Intuitively, the folding operation corresponds to the inverse of the unfolding one. In this case, first notice that part of the body of  $d3$ , corresponds to a renaming of the body of  $d2$ , thus the folding operation will replace this part with the corresponding renaming of the head of  $d2$ . The result is a recursive definition:

```
find_max_and_del([], 0, -, []).
find_max_and_del([X|Xs], Max, El, OutList) ←
    find_max_and_del(Xs, Max', El, OutList'),
    if Max' < X then Max = X else Max = Max',
    if El = X then OutList = OutList'
    else OutList = [X|OutList'].
```

This definition traverses the input list only once. Now, in order to let  $\text{del\_max}$  exploit the efficiency achieved by  $\text{find\_max}$ , we have to fold  $\text{find\_max}(Xs, \text{Max})$ ,  $\text{del\_el}(Xs, \text{Max}, Zs)$  in the body of the  $d1$ , thus obtaining the following final program DELMAX\_EFF

```
d8:del_max(Xs, Zs) ← find_max_and_del(Xs, Max, Max, Zs).

find_max_and_del([], 0, -, []).
find_max_and_del([X|Xs], Max, El, OutList) ←
    find_max_and_del(Xs, Max', El, OutList'),
    if Max' < X then Max = X else Max = Max',
    if El = X then OutList = OutList'
    else OutList = [X|OutList'].
```

plus other clauses, which are no longer useful.

As we have already mentioned, this program has the advantage of needing to traverse the list only once. At the same time, its dynamic behavior is now much more complex than at the beginning. For instance, the `Max` variable of clause `d8` is now used as an asynchronous communication channel between processes, in fact the atom `find_max_and_del(Xs, Max, Max, Zs)` in the body of `d8` uses `Max` as input value that it has to produce itself. This is a typical situation in which we run the risk of entering in a deadlock.

Indeed, while the initial program can be run by using a dummy left-to right selection rule, the final program certainly cannot. Further, in the final program it is not at all immediate to check that the query `del_max(ns, Zs)` (`ns` being a list of positive integers) generates a non-deadlocking computation.

Summarizing, this program shows that Unfold/Fold transformations can be used to improve program efficiency, but that they can lead to programs whose dynamic behavior is more complicate. To cope with this problem we need to guarantee that the transformation does not introduce deadlocked derivations.

Next, we show an opposite example, in which the unfold/fold transformation simplifies the dynamic behavior of a program.

**Example 3.2** Consider the following simple Reader-Writer program, which uses a buffer of length one: `reader(Xs)` reads the input stream and puts the various tokens in the list `Xs`, while `writer(Xs)` writes the elements of the list `Xs` to the standard output and stops doing so when it encounters the token `eof`.

```
d1:read-write ← reader(Xs). writer(Xs).

reader([Y|Ys]) ← read(Y), reader(Ys).
reader([]).

delay reader(Xs) until nonvar(Xs)

writer([Y|Ys]) ← if Y=eof then Ys=[]
                else (write(Y), writer(Ys)).
```

The dynamic behavior of this program is not elementary. `reader(Xs)` behaves as follows: (a) waits until `Xs` is partially instantiated, (b1) when `Xs` is instantiated to `[Y|Ys]` then it instantiates `Y` with the value it reads from the input, (b2) when `Xs` is instantiated to `[]` it stops. On the other hand, the actions of `writer(Xs)` are the following ones: (a) it instantiates `Xs` to `[Y|Ys]` (this activates `reader(Xs)`), (b) it waits until `Y` is instantiated (via the delay declaration embedded in the `if` construct), (c1) if `Y` is the end of file character then it instantiates `Ys` to `[]` (this will also stop the reader), (c2) otherwise it proceeds with the recursive call (which will further activate `reader` and so on). Thus, `read-write` actually implements a communication channel with a buffer of length one, and `Xs` is actually a bidirectional communication channel. We now proceed with the transformation. First we unfold `writer(Xs)` in the body of `d1`, obtaining

```
read-write ← reader([Y|Ys]),
            if Y=eof then Ys=[] else (write(Y), writer(Ys))
```

Then, we unfold `reader([Y|Ys])` in the resulting clause,

```
read-write ← read(Y),
            reader(Ys),
            if Y=eof then Ys=[] else (write(Y), writer(Ys)).
```

Now, we have to apply a new operation, that we'll call *distribution*, in order to bring `reader(Ys)` inside the scope of the `if` construct.

```
read-write ← read(Y),
            if Y=eof then (Ys=[], reader(Ys))
            else (write(Y), reader(Ys), writer(Ys)).
```

We can now *fold* `reader(Ys)`, `writer(Ys)`, using `d1` as folding clause, and obtain,

```
read-write ← read(Y),
            if Y=eof then (Ys=[], reader(Ys)) else (write(Y), read-write).
```

Finally, we can now *evaluate* the constraint `Ys=[]` in the first branch of the `if - then - else` construct (this operation corresponds to substituting the clause for a  $\simeq$ -equivalent one)

```
read-write ← read(Y),
            if Y=eof then reader([]) else (write(Y), read-write).
```

In the end, we can unfold `reader([])`

```
read-write ← read(Y),
            if Y=eof then (true) else (write(Y), read-write).
```

Which is our final program □

Three aspects are important to notice.

First, that the resulting program is more efficient than the initial one: in fact it does not need to use the heap as heavily as the initial one for passing the parameters between `reader` and `writer`. Furthermore, modern optimization techniques such as the ones implemented for the logic language MERCURY (for bibliography and more informations, see <http://www.cs.mu.oz.au/mercury/papers.html>), exploit the fact that `Y` has no aliases in order to avoid saving it on the heap altogether, and further compiling `reader-writer` as a simple `while` program, dramatically saving memory space while speeding up the compiled code (these latter techniques are not yet implemented for programs with dynamic scheduling).

Second, that – as opposed to the initial program – the resulting one has a straightforward dynamic behavior. For instance it can be run with a simple left to right selection rule. Further, if we consider the query `reader-writer`, one can easily see it to be deadlock-free in the latter program (to prove it formally, a straightforward extension of the tools of [1] is sufficient), while in the original program this is not at all immediate. After proving that the transformation does not introduce *nor eliminates* any deadlocking branch in the semantics of the program, we'll be able to state that “if the resulting program specialized with respect to a query is deadlock-free then this query is also deadlock-free in the initial program”. Thus program transformations can be profitably used as a tool for proving deadlock freeness of queries: it is in fact often easier to prove deadlock freeness of a query in a transformed version of a program than in the original one.

Third, that in order to achieve these goals, we have introduced two new operations, strictly extending the existing transformation systems.

## 4 The Transformations

As customary for unfold/fold systems, we start with some requirements on the original (i.e., initial) program that one wants to transform. Here we say that a predicate `p` is *defined* in a program `P`, if `P` contains at least one clause whose head has predicate symbol `p`.

**Definition 4.1 (Initial program)** We call a CLP program  $P_0$  an *initial program* if the following two conditions are satisfied:

- (I1)  $P_0$  is partitioned into two disjoint sets  $P_{\text{new}}$  and  $P_{\text{old}}$ ,
- (I2) the predicates defined in  $P_{\text{new}}$  do not occur neither in  $P_{\text{old}}$  nor in the bodies of the clauses in  $P_{\text{new}}$ .

In presence of delay declarations, we also need the following:

- (D1) No atom defined in  $P_{\text{new}}$  is subject to a delay declaration.  $\square$

Following the notation above, we call *new* predicates those predicates that are defined in  $P_{\text{new}}$ . As we shall discuss in the following, previous conditions (I2) and (D1) are needed to ensure the correctness of the folding operation.

The first transformation we consider is the *unfolding*. Since all the observable properties we consider are invariant under reordering of the atoms in the bodies of clauses, the definition of unfolding, as well as those of the other operations, is given modulo reordering of the bodies atoms. To simplify the notation, in the following definition we also assume that the clauses of a program and its delay declarations have been renamed so that they do not share variables pairwise.

**Definition 4.2 (Unfolding)** Let  $cl : A \leftarrow c, H, \tilde{K}$  be a clause in the program  $P$ , and  $\{H_1 \leftarrow c_1, \tilde{B}_1, \dots, H_n \leftarrow c_n, \tilde{B}_n\}$  be the set of the clauses in  $P$  (called *unfolding clauses*) such that  $c \wedge c_i \wedge (H = H_i)$  is  $\mathcal{D}$ -satisfiable. For  $i \in [1, n]$ , let  $cl'_i$  be the clause

$$A \leftarrow c \wedge c_i \wedge (H = H_i), \tilde{B}_i, \tilde{K}$$

Then *unfolding  $H$  in  $cl$  in  $P$*  consists of replacing  $cl$  by  $\{cl'_1, \dots, cl'_n\}$  in  $P$ . Moreover we say that the unfolding operation is *allowed* iff

- (D2)  $H$  in the context  $c$  satisfies its delay declaration.  $\square$

The unfolding of an atom occurring inside an if - then - else construct is defined in the same way as the unfolding of an atom occurring in the body of a clause. One can easily check that this is consistent with the formal definition of if - then - else given in Section 2.1.

Condition (D2) is clearly needed to avoid the transformation of a deadlocked derivation into a successful one, since once the atom  $H$  has been unfolded its delay declaration is not anymore visible.

Note that, as usual, we consider here a notion of unfolding which is defined “locally” on single clauses, independently from the computation context. Clearly, using some assumptions on the context in which the clause will be evaluated we could weaken the applicability condition (D2). However, this would not be in the line of the standard program transformation practice. Moreover, as illustrated by the examples given in this paper, condition (D2) is sufficient for several interesting applications.

We are now ready to introduce the folding operation. This operation is a fundamental one, as it allows one to introduce recursion in the new definitions. Here we use a formalization of the applicability conditions given in [13]. As in [32], the applicability conditions of the folding operations depend on the history of the transformation. Therefore we define a transformation sequence as follows.

**Definition 4.3** A *transformation sequence* is a sequence of programs  $P_0, \dots, P_n$ , in which  $P_0$  is an initial program and each  $P_{i+1}$ , is obtained from  $P_i$  via a transformation operation of unfolding, clause removal, splitting, constraint replacement and folding, whose applicability conditions are satisfied.

As usual, in the following definition we assume that the folding ( $cl_f$ ) and the folded ( $cl$ ) clause are renamed apart and, as a notational convenience, that the body of the folded clause has been reordered so that the atoms that are going to be folded are found in the leftmost positions. Moreover, recall that the initial program  $P_0$  is partitioned into  $P_{\text{new}}$  and  $P_{\text{old}}$ .

**Definition 4.4 (Folding)** Let  $P_0, \dots, P_i, i \geq 0$ , be a transformation sequence. Let also

$$cl : A \leftarrow c_A, \tilde{K}, \tilde{J} \text{ be a clause in } P_i,$$

$$cl_f : D \leftarrow c_D, \tilde{H} \text{ be a clause in } P_{\text{new}}.$$

If  $c_A, \tilde{K}$  is an instance of *true*,  $\tilde{H}$  and  $e$  is a constraint such that  $\text{vars}(e) \subseteq \text{vars}(D) \cup \text{vars}(cl)$ , then *folding  $\tilde{K}$  in  $cl$  via  $e$*  consists of replacing  $cl$  by

$$cl' : A \leftarrow c_A \wedge e, D, \tilde{J}$$

provided that the following three conditions hold:

- (F1) “If we unfold  $D$  in  $cl'$  using  $cl_f$  as unfolding clause, then we obtain  $cl$  back” (modulo  $\simeq$ ),
- (F2) “ $cl_f$  is the only clause of  $P_{\text{new}}$  that can be used to unfold  $D$  in  $cl'$ ”, that is, there is no clause  $cl_{f'} : B \leftarrow c_B, \tilde{L}$  in  $P_{\text{new}}$  such that  $cl_{f'} \neq cl_f$  and  $c_A \wedge e \wedge (D = B) \wedge c_B$  is  $\mathcal{D}$ -satisfiable.
- (F3) “No self-folding is allowed”, that is
  - (a) either the predicate in  $A$  is an old predicate;
  - (b) or  $cl$  is the result of at least one unfolding in the sequence  $P_0, \dots, P_i$ .  $\square$

The constraint  $e$  acts as a bridge between the variables of  $cl_f$  and  $cl$ . As shown in [13], conditions (F1) and (F2) ensure that the folding operation behaves, to some extent, as the inverse of the unfolding one; the underlying idea is that if we unfolded the atom  $D$  in  $cl'$  using only clauses from  $P_{\text{new}}$  as unfolding clauses, then we would obtain  $cl$  back. In this context condition (F2) ensures that in  $P_{\text{new}}$  there exists no clause other than  $cl_f$  that can be used as *unfolding clause*. Condition (F3) is from the original Tamaki-Sato definition of folding for logic programs. Its purpose is to avoid the introduction of loops which can occur if a clause is folded by itself.

Finally, note that since the folding clause  $cl_f$  has to be in  $P_{\text{new}}$ , condition (D1) in the definition of initial program implies that the head of the folding clause is not subject to any delay declaration. This condition is needed in order to avoid the introduction of deadlocked derivations, as shown by the following example. Consider the program  $P$ :

$$\begin{aligned} p(X) &\leftarrow q(X), r(X). \\ m(X) &\leftarrow q(X). \\ q(X). \\ r(X). \end{aligned}$$

$$\text{delay } m(X) \text{ until } \text{ground}(X)$$

If we fold the first clause using the second one as folding clause, we obtain the program  $P'$ :

$$\begin{aligned} p(X) &\leftarrow m(X), r(X). \\ m(X) &\leftarrow q(X). \\ q(X) &. \\ r(X) &. \end{aligned}$$

delay  $m(X)$  until  $\text{ground}(X)$

which is not equivalent to  $P$ . In fact the query  $p(X)$  has a successful derivation in  $P$  while it deadlocks in  $P'$ .

Finally, we consider a transformation consisting in distributing an atom over the if - then - else construct in the body of a clause. This operation is often needed in order to apply the folding operation.

**Definition 4.5 (Distribution)** Let

$$cl : H \leftarrow c, K, \bar{J}, \text{ if } d \text{ then } \bar{A} \text{ else } \bar{B}.$$

be a clause in a program  $P$ , then applying the *distribution operation* to  $K$  in  $cl$  means replacing  $cl$  with

$$cl' : H \leftarrow c, \bar{J}, \text{ if } d \text{ then } (\bar{A}, K) \text{ else } (\bar{B}, K).$$

This operation is *allowed* provided that

(D3) there is no constraint  $d$ ,  $\text{vars}(d) = \text{vars}(H, \bar{J})$ , such that

- $c \wedge d$  is  $\mathcal{D}$ -satisfiable, and
- $K$  in the context  $c \wedge d$  satisfies its delay declaration.  $\square$

Condition (D3) states that  $K$  can be brought inside an if - then - else construct if it can not become selectable neither by means of an instantiation of the variables of  $H$  nor of the variables of the other body atoms  $\bar{J}$ . Intuitively, this is needed to ensure that  $K$  will not play a role in determining which of the branches of the if - then - else construct will be followed. (D3) prevents the introduction of deadlocked derivations, as illustrated by the following example. Consider the program.

$$\begin{aligned} p(X) &\leftarrow q(X), \text{ if } X=a \text{ then } r(X) \text{ else } t(X). \\ q(a) &. \\ r(a) &. \end{aligned}$$

If we disregard condition (D3) we could apply the distributive operation and transform it into

$$\begin{aligned} p(X) &\leftarrow \text{if } X=a \text{ then } (q(X), r(X)) \text{ else } (q(X), t(X)). \\ q(a) &. \\ r(a) &. \end{aligned}$$

However, in the first program the query  $p(X)$  succeeds, while in the second one it deadlocks (being  $X$  free the if - then - else construct suspends).

The correctness of the introduced transformations is stated by the following.

**Theorem 4.6 (Correctness)** Let  $P_0, \dots, P_n$  be a transformation sequence and  $Q$  be a query. Then for any  $P_i, P_j \in P_0, \dots, P_n$  we have that

- (i)  $Q$  has a deadlocked derivation in  $P_i$  iff it has a deadlocked derivation in  $P_j$ .
- (ii)  $Q$  has the same answer constraints in  $P_i$  and  $P_j$ , i.e., there exists a successful derivation  $Q \stackrel{P_i}{\rightsquigarrow} d_i$  iff there exists a successful derivation  $Q \stackrel{P_j}{\rightsquigarrow} d_j$ , and  $\mathcal{D} \models \exists_{-\text{vars}(Q)} d_i \leftrightarrow \exists_{-\text{vars}(Q)} d_j$ .

**Other Operations.** Other less prominent operations which can nevertheless be useful in program transformation are the *splitting* and the *constraint replacement*. Due to space limitations we are not able to provide significant examples of these operations (although, the splitting one is employed in the CCP example at the end of the paper). Therefore, we now simply report their definitions and we refer to [13] for further information on them. In the following definition, just like for the unfolding operation, it is assumed that program clauses do not share variables pairwise.

**Definition 4.7 (Splitting)** Let  $cl : A \leftarrow c, H, \bar{K}$  be a clause in the program  $P$ , and  $\{H_1 \leftarrow c_1, \bar{B}_1, \dots, H_n \leftarrow c_n, \bar{B}_n\}$  be the set of the clauses in  $P$  such that  $c \wedge c_i \wedge (H = H_i)$  is  $\mathcal{D}$ -satisfiable. For  $i \in [1, n]$ , let  $cl'_i$  be the clause

$$A \leftarrow c \wedge c_i \wedge (H = H_i), H, \bar{K}$$

If, for any  $i, j \in [1, n]$ ,  $i \neq j$ , the constraint  $(H_i = H_j) \wedge c_i \wedge c_j$  is unsatisfiable then *splitting  $H$  in  $cl$  in  $P$*  consists of replacing  $cl$  by  $\{cl'_1, \dots, cl'_n\}$  in  $P$ . This operation is *allowed* iff

(D4)  $H$  in the context  $c$  satisfies its delay declaration.  $\square$

Differently from previous cases, the applicability condition for *constraint replacement* relies on the semantics of the program. In fact, as opposed to the "cleaning up" of constraints previously discussed, this operation allows us to replace a clause for another one which is not  $\simeq$ -equivalent: the equivalence is ensured only with respect to the given program.

**Definition 4.8 (Constraint Replacement)** Let  $P$  be a program,  $cl : H \leftarrow c_1, \bar{B}$  be a clause  $P$  and  $c_2$  be a constraint. If,

(D5) for each successful or deadlocked derivation  $\text{true}, \bar{B} \stackrel{P}{\rightsquigarrow} d, \bar{D}$  we have that  $H \leftarrow c_1 \wedge d, \bar{D} \simeq H \leftarrow c_2 \wedge d, \bar{D}$

then *replacing  $c_1$  by  $c_2$  in  $c_1$*  consists in substituting  $cl$  by  $H \leftarrow c_2, \bar{B}$  in  $P$ .  $\square$

The correctness of the newly introduced transformations is stated by the following.

**Corollary 4.9 (Correctness)** Theorem 4.6 holds also if in the transformation sequence we have additionally employed splitting and constraint replacement steps.  $\square$

## 5 Proving Deadlock Freeness

As we have already seen in Example 3.2, unfold/fold transformations can be employed for proving deadlock-freeness of a certain query. The underlying idea is simple: we transform (specialize) the program until the query we are interested in becomes independent from those predicates which are subject to a delay declaration. If we manage, then by Theorem 4.6 we have proven that the query is deadlock-free also in the initial program. This can be extended to a class of queries in the obvious way.

More in general, program specialization can be employed to prove absence of deadlock in combination with known methods. For instance, in the program Reader-Writer, one can easily extend the tools of Apt and Luitjes [1] to CLP programs with if - then - else in order to prove that the

query read-write is deadlock-free in the *last* program of the sequence. Thus by Theorem 4.6 read-write is deadlock-free also in the *first* program of the sequence. Proving this latter statement with the (extended) tools of [1] is not possible. Thus, despite its simplicity, program specialization is a rather powerful tool for proving deadlock freeness.

In order to further substantiate our assertions, we now consider a more complicated example, implementing a static network of stream transducers. This program solves the so-called Hamming problem introduced by Dijkstra: generate an ordered stream of all numbers of the form  $2^i 3^j 5^k$  without repetitions. To this end, five processes are used, that interleave their execution, thus giving rise to a number of different schedulings.

**Example 5.1** In [34], van Emden and de Lucena proposed a solution for the Hamming problem based on the incorporation of corouting in the execution of a logic program. Using our formalism with delay declarations, such a solution is implemented by the following program HAMMING

```

hammings(X) ← mult([1|X],2,X2),
             mult([1|X],3,X3),
             mult([1|X],5,X5),
             merge(X2,X3,X23),
             merge(X5,X23,X).

merge([X|Xi1],[X|Xi2],[X|Xo]) ← merge(Xi1,Xi2,Xo).
merge([X|Xi1],[Y|Xi2],[X|Xo]) ← X<Y, merge(Xi1,[Y|Xi2],Xo).
merge([X|Xi1],[Y|Xi2],[Y|Xo]) ← X>Y, merge([X|Xi1],Xi2,Xo).

mult([X|Xs],Y,[X*Y|Z]) ← mult(Xs,Y,Z).

delay mult(X,Y,Z) until nonvar(X) and ground(Y)
delay merge(X,Y,Z) until nonvar(X) and nonvar(Y)

```

We prove that `hammings(X)` is deadlock free in HAMMING. The problem with this program is that the execution of `hammings(X)` uses various schedulings of the body atoms of its first clause. For instance, initially, only the leftmost three atoms are selectable. After one resolution step, all these atoms become suspended. At this point, the leftmost `merge` atom is allowed to proceed. After one execution step, its execution is also suspended. However, at this point the first two arguments of the last `merge` atom have been instantiated in such a way that it can be selected. After one more resolution step, we obtain the initial situation, with the three `mult` processes active and the other two processes suspended.

Our proof consists of the following steps:

1. We introduce the new clause:

```

ham1([X|Xs],[Y|Xs],[Z|Xs],X2,X3,X5,X23) ←
  mult([X|Xs],2,X2),
  mult([Y|Xs],3,X3),
  mult([Z|Xs],5,X5),
  merge(X2,X3,X23),
  merge(X5,X23,Xs).

```

2. By folding the body of the clause defining `hammings` via the above clause we obtain the clause:

```

hammings(Xs) ← ham1([1|Xs],[1|Xs],[1|Xs],X2,X3,X5,X23).

```

3. Unfolding the first three body atoms of the definition of `ham1`, we obtain:

```

ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],X23) ←
  mult(Xs,2,X2),
  mult(Xs,3,X3),
  mult(Xs,5,X5),
  merge([X*2|X2],[Y*3|X3],X23),
  merge([Z*5|X5],X23,Xs).

```

4. Unfolding `merge([X*2|X2],[Y*3|X3],X23)` we get the three clauses:

```

ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23])
  ← X*2 < Y*3,
  mult(Xs,2,X2),
  mult(Xs,3,X3),
  mult(Xs,5,X5),
  merge(X2,[Y*3|X3],X23),
  merge([Z*5|X5],[X*2|X23],Xs).

ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[Y*3|X23])
  ← X*2 > Y*3,
  mult(Xs,2,X2),
  mult(Xs,3,X3),
  mult(Xs,5,X5),
  merge([X*2|X2],X3,X23),
  merge([Z*5|X5],[Y*3|X23],Xs).

ham1([X|Xs],[Y|Xs],[Z|Xs],[X*2|X2],[Y*3|X3],[Z*5|X5],[X*2|X23])
  ← X*2 = Y*3,
  mult(Xs,2,X2),
  mult(Xs,3,X3),
  mult(Xs,5,X5),
  merge(X2,X3,X23),
  merge([Z*5|X5],[X*2|X23],Xs).

```

5. Unfolding the last body atom of each of the clauses obtained in the previous step, we get nine clauses. We give here only the three obtained by unfolding the last clause in the previous step; the other ones have a similar structure.

```

ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],
      [Z*5|X5],[X*2|X23]) ←
  X*2 = Y*3, Z*5 > X*2,
  mult([X*2|Xs],2,X2),
  mult([X*2|Xs],3,X3),
  mult([X*2|Xs],5,X5),
  merge(X2,X3,X23),
  merge([Z*5|X5],X23,Xs).

ham1([X,Z*5|Xs],[Y,Z*5|Xs],[Z,Z*5|Xs],[X*2|X2],[Y*3|X3],
      [Z*5|X5],[X*2|X23]) ←
  X*2 = Y*3, Z*5 < X*2,
  mult([Z*5|Xs],2,X2),
  mult([Z*5|Xs],3,X3),
  mult([Z*5|Xs],5,X5),
  merge(X2,X3,X23),
  merge(X5,[X*2|X23],Xs).

ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],
      [Z*5|X5],[X*2|X23]) ←
  X*2 = Y*3, Z*5 = X*2,
  mult([X*2|Xs],2,X2),
  mult([X*2|Xs],3,X3),
  mult([X*2|Xs],5,X5),
  merge(X2,X3,X23),
  merge(X5,X23,Xs).

```

6. By folding each clause obtained in the previous step – using the clause introduced in step 1 – we get nine clauses. Here we show only those obtained from the three clauses of the previous step. Those for the other clauses are similar.

```

ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],
      [Z*5|X5],[X*2|X23]) ←
  X*2 = Y*3, Z*5 > X*2,
  ham1([X*2|Xs],[X*2|Xs],[X*2|Xs],X2,X3,[Z*5|X5],X23).

ham1([X,Z*5|Xs],[Y,Z*5|Xs],[Z,Z*5|Xs],[X*2|X2],[Y*3|X3],
      [Z*5|X5],[X*2|X23]) ←
  X*2 = Y*3, Z*5 < X*2,
  ham1([Z*5|Xs],[Z*5|Xs],[Z*5|Xs],[X*2|X2],X3,X5,X23).

ham1([X,X*2|Xs],[Y,X*2|Xs],[Z,X*2|Xs],[X*2|X2],[Y*3|X3],

```

```

{Z*5|X5},{X*2|X23} ←
X*2 = Y*3, Z*5 = X*2,
ham1([X*2|Xs],[X*2|Xs],[X*2|Xs],X2,X3,X5,X23).

```

The resulting program – restricted to the definition of `hammings` – is the clause of step 2 plus the clauses obtained in the last step. This program is readily deadlock free (it does not have any delay declaration), hence, by Theorem 4.6 `hammings(Xs)` is deadlock free in `HAMMING`.  $\square$

It is worth noticing that the proof methods for proving deadlock freeness defined in [1, 12, 25, 4] cannot cope with this program. Thus the only available methods for proving deadlock freeness of `hammings(Xs)` could be those based on abstract interpretation. However, these methods require the definition of suitable abstract domains which for deadlock analysis are rather complicated. We believe that our method based on transformations – possibly combined with the techniques which use mode and type information as in [1] – has the advantage of combining simplicity with usefulness. Furthermore, there are cases which cannot be handled by using (practical) tools based on abstract interpretation while they can be handled with other techniques. For instance, the system defined in [6], as discussed in that paper, does not allow one to prove absence of deadlock for the query `p(a, Y), test(Y)` in the following program implementing the so called *short circuit technique*:

```

p(X,Y) ← p(X,Z), p(Z,Y).
p(X,X).
test(X).

delay test(X) until ground(X)

```

On the other hand, such a query can be easily proved deadlock free by simply using modes. For this reason we believe that our method based on transformations, possibly combined with the techniques which use mode and type information as in [1], provides a simple and powerful tool.

## 6 Extension to CCP

The CLP paradigm we have considered in the previous sections is probably the logic language which has the greatest impact on practical applications. Nevertheless, in the field of concurrent programming, there exist more expressive languages which allow for more sophisticated synchronization mechanisms. In particular, concurrent constraint programming (CCP) [30] can be considered the natural extension of CLP. In fact, in [8] it has been shown that CLP with dynamic scheduling can be embedded into a strict sublanguage of CCP. Such an increased expressive power allows us to define a more flexible transformation system. As an example, consider the clause `cl : H ← c, B̄, A`, and assume that `A` is defined by the single clause `A ← C̄` and that in the context `c`, `A` does not satisfy its delay declaration `delay A until φ`. In this situation, we cannot unfold `A` in `cl` (it is forbidden by condition (D2)). On the other hand, this would be possible in CCP, in fact the definition of `A` would be `A ← (ask(φ) → C̄)`, and the result of the unfolding operation would be `cl' : H ← c, B̄, ask(φ) → (C̄)`. In other words, in CCP the `ask` construct allows us to report in the unfolded clause the delay declaration of the unfolded atom, and this enhances the flexibility of the transformation system.

In this section we'll show how our transformation system can be extended to the CCP language. Due to space limitation we'll restrict ourselves to the essential definitions and one example.

Both CLP and CCP frameworks are defined parametrically wrt to some notion of constraint system. Usually a more abstract view is followed in CCP by formalizing the notion of constraint system [31] along the guidelines of Scott's treatment of information systems. Here, for the sake of uniformity, we will assume that also in CCP constraints are first order formulae as previously defined. The basic idea underlying CCP languages is that computation progresses via monotonic accumulation of information in a global store. Information is produced by the concurrent and asynchronous activity of several agents which can add a constraint `c` to the store by performing the basic action `tell(c)`. Dually, agents can also check whether a constraint `c` is entailed by the store by using an `ask(c)` action, thus allowing synchronization among different agents. A notion of locality is obtained in CCP by introducing the agent  $\exists_X A$  which behaves like `A`, with the variable `X` considered *local* to `A`. We do not use such an explicit operator: analogously to the standard CLP setting, locality is introduced implicitly by assuming that if a process is defined by `p(X̄) ← A` and a variable `Y` in `A` does not appear in `X̄`, then `Y` has to be considered local to `A`. The `||` operator allows one to express parallel composition of two agents and it is usually described in terms of interleaving. Finally non-determinism arises by introducing a (global) choice operator  $\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i$ : the agent  $\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i$  nondeterministically selects one `ask(c_i)` which is enabled in the current store, and then behaves like `A_i`. Thus, the syntax of CCP *declarations* and *agents* is given by the following grammar:

<i>Declarations</i>	<code>D ::= ε   p(X̄) ← A   D, D</code>
<i>Agents</i>	<code>A ::= stop   tell(c)   <math>\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i</math>   A    A   p(X̄)</code>
<i>Processes</i>	<code>P ::= D.A</code>

where `c` is a constraint.

Due to the presence of an explicit choice operator, as usual we assume (without loss of generality) that each predicate symbol is defined by exactly one declaration. The operational model of CCP is described by a transition system  $T = (\text{Conf}, \rightarrow)$  where configurations (in `Conf`) are pairs consisting of a process and a constraint (representing the common store), while the transition relation  $\rightarrow \subseteq \text{Conf} \times \text{Conf}$  is described by the (least relation satisfying the) rules **R1-R4** of Table 1. Here and in the following we assume given a set `D` of declarations and we denote by `defD(H)` the set of variants<sup>5</sup> of declarations in `D` which have `H` as head. We assume also the presence of a renaming mechanism that takes care of using fresh variables each time a clause is considered<sup>6</sup>. In order to simplify the definitions, we also assume that only variables occur as arguments of predicates. This does not cause any loss of generality, since the procedure call (the head) `p(t̄)` can be equivalently rewritten as `p(X̄) || tell(X̄ = t̄)` (as `p(X̄) ← tell(X̄ = t̄)`). We denote by

<sup>5</sup>A variant of a clause `cl` is obtained by replacing the tuple `X̄` of all the variables appearing in `cl` for another tuple `Ȳ`.

<sup>6</sup>For the sake of simplicity we do not describe this renaming mechanism in the transition system. The interested reader can find in [31, 30] various formal approaches to this problem.

<b>R1</b>	$\langle D.\text{tell}(c), d \rangle \rightarrow \langle D.\text{stop}, c \wedge d \rangle$
<b>R2</b>	$\langle D.\sum_{i=1}^n \text{ask}(c_i) \rightarrow A_i, d \rangle \rightarrow \langle D.A_j, d \rangle$ if $j \in [1, n]$ and $\mathcal{D} \models d \rightarrow c_i$
<b>R3</b>	$\frac{\langle D.A, c \rangle \rightarrow \langle D.A', c' \rangle}{\langle D.A \parallel B, c \rangle \rightarrow \langle D.A' \parallel B, c' \rangle}$ $\langle D.B \parallel A, c \rangle \rightarrow \langle D.B \parallel A', c' \rangle$
<b>R4</b>	$\langle D.p(\tilde{X}), c \rangle \rightarrow \langle D.A, c \rangle$ if $p(\tilde{X}) \leftarrow A \in \text{def}_D(p(\tilde{X}))$

Table 1: The (standard) transition system.

$\rightarrow^*$  the reflexive and transitive closure of the relation  $\rightarrow$  defined by the transition system, and we denote by *Stop* any agent which contains only *stop*,  $\parallel$  and  $+$  constructs. A finite derivation (or computation) is called *successful* if it is of the form  $\langle D.A, c \rangle \rightarrow^* \langle \text{Stop}, d \rangle \not\rightarrow$  while it is called *deadlocked* if it is of the form  $\langle D.A, c \rangle \rightarrow^* \langle B, d \rangle \not\rightarrow$  with  $B$  different from *Stop* (i.e.,  $B$  contains at least one suspended agent). As it results from the transition system above, we consider here the so called “eventual tell” CCP, i.e. when adding constraints to the store (via *tell* operations) there is no consistency check. Therefore *false* is also a possible result of successful computation (clearly, since *false* entails any constraint, no deadlock can arise when the store is *false*).

The syntax of CCP agents allow us to define unfolding and the other transformations in a very simple way by using the notion of context. A context, denoted by  $C[\ ]$ , is simply an agent with a “hole”.  $C[A]$  denotes the agent obtained by replacing the hole in  $C[\ ]$  for the agent  $A$ .

**Definition 6.1 (Unfolding)** Consider a set of declarations  $D$  containing the declaration  $cl : q(\tilde{Y}) \leftarrow C[p(\tilde{X})]$  and suppose that  $p(\tilde{X}) \leftarrow B \in \text{def}_D(p(\tilde{X}))$ <sup>7</sup>. Then unfolding  $p(\tilde{X})$  in  $C[p(\tilde{X})]$  consists simply in replacing  $C[p(\tilde{X})]$  by  $C[B]$ .  $\square$

The notions of initial program and of transformation sequence are defined as in the previous Section (a CCP program is a set of declarations). Here, since we don’t have external delay declarations, we do not have the condition (D1), thus (I2) is the only condition imposed on the initial program. So, we can define folding as follows.

**Definition 6.2 (Folding)** Let  $P_0, \dots, P_i, i \geq 0$ , be a transformation sequence. Let also

$$cl : p(\tilde{X}) \leftarrow C[A] \text{ be a declaration in } P_i,$$

$$cl_f : q(\tilde{Y}) \leftarrow A \text{ be a declaration in } P_{\text{new}}.$$

(it is assumed here that  $q(\tilde{Y}) \leftarrow A$  is suitably renamed to avoid variables names clashes). Then *folding*  $A$  in  $cl$  consists of replacing  $cl$  by

$$cl' : p(\tilde{X}) \leftarrow C[q(\tilde{Y})]$$

provided that the conditions (F1’), and (F3) are satisfied, where (F1’) is the following modification of (F1):

(F1’) *If we unfold  $q(\tilde{Y})$  in  $cl'$  using  $cl_f$  as unfolding clause, then we obtain  $cl$  back.*  $\square$

<sup>7</sup>As before, it is assumed that  $p(\tilde{X}) \leftarrow B$  uses fresh variables.

Analogously to the case of the if - then - else, in some cases we need to distribute a procedure call over an ask action. The corresponding of the distribution operation for CCP can be formalized as follows.

**Definition 6.3 (Distribution)** Consider a set of declarations  $D$  containing the declaration

$$cl : p(\tilde{X}) \leftarrow C[q(\tilde{Y}) \parallel \sum_i \text{ask}(c_i) \rightarrow A_i]$$

and assume that, for any input constraint  $c \neq \text{false}$  whose free variables are contained in  $\tilde{X}$ ,  $\langle D.C[q(\tilde{Y})], c \rangle$  has only deadlocked derivations of the form  $\langle D.C[q(\tilde{Y})], c \rangle \rightarrow^* \langle D.B, d \rangle \not\rightarrow$  (with  $B$  different from *Stop*) such that  $\mathcal{D} \models \exists_{\tilde{z}} c \leftrightarrow \exists_{\tilde{z}} d$ , where  $\tilde{z} = \text{vars}(C[q(\tilde{Y})], c)$ . In this case we can transform  $cl$  into the declaration

$$cl' : p(\tilde{X}) \leftarrow C[\sum_i \text{ask}(c_i) \rightarrow (q(\tilde{Y}) \parallel A_i)]. \quad \square$$

The applicability condition ensures that it is correct to bring  $q(\tilde{Y})$  in the scope of the  $\text{ask}(c_i)$ ’s. In fact,  $q(\tilde{Y})$  can proceed in the computation only if some  $A_i$  produces more information.

Finally, in some cases we need also an operation which allows us to simplify the ask guards occurring in a program. Consider in fact an agent of the form  $C[\text{ask}(c) \rightarrow A + \text{ask}(d) \rightarrow B]$  and a given set of declarations. If  $c'$  is the conjunction of all the constraints which appear both in *ask* and *tell* actions in the “path” leading in the context  $C[\ ]$  from the top-level to the agent  $(\text{ask}(c) \rightarrow A + \text{ask}(d) \rightarrow B)$  and  $\mathcal{D} \models c' \rightarrow c$ , then clearly we can simplify the previous agent to  $C[\text{ask}(\text{true}) \rightarrow A + \text{ask}(d) \rightarrow B]$  (note that in general it is not correct to transform the previous agent into  $C[A + \text{ask}(d) \rightarrow B]$ ). This operation, which can easily be defined by structural induction on the agents, is often used to “clean up” the CCP agents, analogously to the previous  $\simeq$  equivalence. We omit its formal definition for space reasons.

The correctness of the transformation for CCP is expressed by the following result which, analogously to the case of CLP, shows that both answer constraints and deadlocked derivations are preserved by the transformations.

**Theorem 6.4 (Correctness)** Let  $P_0, \dots, P_n$  be a transformation sequence of CCP programs and let  $A$  be a generic agent. Then for any  $P_i, P_j \in P_0, \dots, P_n$  we have that, for any constraint  $c$ :

- (i)  $P_i.A$  has a deadlocked derivation iff  $P_j.A$  has a deadlocked derivation;
- (ii) there exists a successful derivation  $\langle P_i.A, c \rangle \rightarrow^* \langle P_i.\text{Stop}, d \rangle \not\rightarrow$  iff there exists a successful derivation  $\langle P_j.A, c \rangle \rightarrow^* \langle P_j.\text{Stop}, d' \rangle \not\rightarrow$  and  $\mathcal{D} \models \exists_{\text{vars}(A,c)} d \leftrightarrow \exists_{\text{vars}(A,c)} d'$ .  $\square$

We conclude with an example in order to illustrate the application of our methodology based on unfold/fold. We consider a stream protocol problem where two input streams are merged into an output stream. An input stream consists of lines of messages, and each line has to be passed to the output stream without interruption. Input and output streams are dynamically constructed by a reader and a monitor process, respectively. A reader communicates with the monitor by means of a buffer of length one, and is synchronized in such a way that it can read a new message only

when the buffer is empty (i.e., when the previous message has been processed by the monitor). On the other hand, the monitor can access a buffer only when it is not empty (i.e., when the corresponding reader has put a message into its buffer).

In order to keep the notation simple, we adopt here a more liberal syntax, similar to the one used for CLP. We allow terms as arguments to predicates since, as previously mentioned,  $p(t)$  can be seen as a shorthand for  $p(X) \parallel \text{tell}(X = t)$  (or  $p(X) \leftarrow \text{tell}(X = t)$ ). We also simply write  $c$  to mean  $\text{tell}(c)$ . Finally we allow multiple declarations for a predicate. Also in this case the syntax is equivalent to the previous one, since the two declarations  $p \leftarrow D_1$  and  $p \leftarrow D_2$  can be seen as a shorthand for  $p \leftarrow (\text{ask}(\text{true}) \rightarrow D_1) + (\text{ask}(\text{true}) \rightarrow D_2)$ . Clearly, these simplifications do not affect the applicability conditions and the results of the transformations we apply.

The following program STREAMER implements the stream protocol discussed before.

```
streamer(Os) ← reader(left,Ls),
  reader(right,Rs),
  monitor(Ls,Rs,idle,Os).

reader(Channel,Xs) ←
  ask(∃x,xs, Xs=[X|Xs']) →
  (read(Channel,X), reader(Channel,Xs'))
  + ask(Xs=[]) → true.

monitor([L|Ls],[R|RS],idle,Os) ← % waiting for an input
  ask(ground(L)) → monitor([L|Ls],[R|RS],left,Os)
  + ask(ground(R)) → monitor([L|Ls],[R|RS],right,Os)

monitor([L|Ls],RS,left,[L|Os]) ← % processing the left stream
  ask(ground(L)) →
  ( ask(L=eof) → (Ls=[], onestream(Rs,Os))
  + ask(L=eol) → monitor(Ls,RS,idle,Os)
  + ask(L=/=eol AND L=/=eof) → monitor(Ls,RS,left,Os))

% plus an analogous clause for processing the right stream

onestream([X|Xs],[X|Os]) ←
  ask(ground(X)) →
  ( ask(X = eof) → Os=[]
  + ask(X /= eof) → onestream(Xs,Os))
```

The predicate `reader(Channel,Xs)` waits to process `Channel` until `Xs` is instantiated; `read(Channel,msg)` is a primitive defined in order to read from the stream `Channel`; the predicate `monitor(Ls,Rs,State,Os)` takes care of merging `Ls` and `Rs`; and the predicate `onestream(Xs,Os)` takes care of handling the single stream `Xs`. The constants `eol` and `eof` denote the *end of line* and the *end of file* characters, respectively. Moreover, the other constants `left`, `right` and `idle` describe the state of the monitor, i.e., if it is processing a message from the left stream, from right stream, or if it is in an idle situation, respectively.

We now improve the efficiency of the computation of the query `streamer(Ys)`. We do this by transforming STREAMER via the following steps.

1. We introduce the following new predicate:

```
handle_two(L,R,State,Os) ← reader(left,Ls),
  reader(right,Rs),
  monitor([L|Ls],[R|RS],State,Os).
```

2. We unfold the monitor atom in the new clause. We obtain three definitions

```
handle_two(L,R,idle,Os) ← reader(left,Ls),
```

```
  reader(right, Rs),
  ( ask(ground(L)) → monitor([L|Ls],[R|RS],left,Os)
  + ask(ground(R)) → monitor([L|Ls],[R|RS],right,Os))
```

```
handle_two(L,R,left,[L|Os]) ← reader(left,Ls),
  reader(right, Rs),
  ask(ground(L)) →
  ( ask(L=eof) → (Ls=[], onestream([R|RS],Os))
  + ask(L=eol) → monitor(Ls,[R|RS],idle,Os)
  + ask(L=/=eol AND L=/=eof) →
    monitor(Ls,[R|RS],left,Os))
```

% plus a third clause, symmetric to the second one.

3. Now, we can (a) apply the *distribution* operation to the reader atoms. (b) apply the *splitting* operation to the last two occurrences of monitor in the last definition and (c) *evaluate* the expression `Ls = []`. We obtain the following definition.

```
handle_two(L,R,idle,Os) ←
  ask(ground(L)) → (reader(left,Ls),
  reader(right,Rs),
  monitor([L|Ls],[R|RS],left,Os))
  + ask(ground(R)) → (reader(left,Ls),
  reader(right,Rs),
  monitor([L|Ls],[R|RS],right,Os))
```

```
handle_two(L,R,left,[L|Os]) ←
  ask(ground(L)) →
  ( ask(L=eof) → (onestream([R|RS],Os)
  reader(left,[]),
  reader(right,Rs))
  + ask(L=eol) → (monitor([L'|Ls'],[R|RS],idle,Os),
  reader(left,[L'|Ls']),
  reader(right,Rs))
  + ask(L=/=eol AND L=/=eof) →
  (monitor([L'|Ls'],[R|RS],left,Os),
  reader(left,[L'|Ls']),
  reader(right,Rs)))
```

% plus a third clause, symmetric to the second one

4. Now we can (a) fold `handle_two` in the first of these definitions and (b) unfold the various instances of `reader(left, -)`. We obtain:

```
handle_two(L,R,idle,Os) ←
  ask(ground(L)) → handle_two(L,R,left,Os)
  + ask(ground(R)) → handle_two(L,R,right,Os)
```

```
handle_two(L,R,left,[L|Os]) ←
  ask(ground(L)) →
  ( ask(L=eof) →
  (onestream([R|RS],Os),
  ( ask(∃x,xs, []=[X|Xs']) → (read(left,X),
  reader(left,Xs'))),
  + ask([]=[]) → true ),
  reader(right,Rs))
  + ask(L=eol) →
  (monitor([L'|Ls'],[R|RS],idle,Os),
  ( ask(∃x,xs, [L'|Ls']=[X|Xs']) → read(left,X),
  reader(left,Xs')),
  + ask([L'|Ls']=[]) → true.),
  reader(right,Rs))
  + ask(L=/=eol AND L=/=eof) →
  (monitor([L'|Ls'],[R|RS],left,Os) ,
  ( ask(∃x,xs, [L'|Ls']=[X|Xs']) →
  (read(left,X),reader(left,Xs'))),
  + ask([L'|Ls']=[]) → true.),
  reader(right,Rs)))
```

5. We can now (a) evaluate the newly introduced ask constructs, and (b) fold twice `handle_two` in the last definition

```

handle_two(L,R,idle,Os) ←
  ask(ground(L)) → handle_two(L,R,left,Os)
+ ask(ground(R)) → handle_two(L,R,right,Os)

handle_two(L,R,left,[L|Os]) ←
  ask(ground(L)) →
  ( ask(L=eof) →
    (onestream([R|Rs],Os), reader(right,Rs)) %(**)
  + ask(L=eol) →
    (read(left,L'), handle_two(L',R,idle,Os))
  + ask(L=/=eol AND L=/=eof) →
    (read(left,L'),handle_two(L',R,left,Os)))

% plus a third clause, symmetric to the second one

```

6. Now, the above program is *almost completely* independent from the definition of `reader`. In order to eliminate the atom `reader(right,Rs)`, we use an unfold/fold transformation similar to (but simpler than) the previous one. We start with the new predicate:

```

handle_one(X, Channel, Os) ← reader(Channel,Xs),
  onestream([X|Xs],Os).

```

7. After the application of transformations analogous to the one used for transforming `handle_two`, we obtain

```

handle_one(X, Channel, [X|Os]) ←
  ask(ground(X)) →
  ( ask(X=eof) → Os=[]
  + ask(X=/=eof) → (read(Channel,X'),
    handle_one(X', Channel,Os)))

```

8. By folding `handle_one` into the second definition of `handle_two` (the part marked with **(\*\*)**), we obtain

```

handle_two(L,R,left,[L|Os]) ←
  ask(ground(L)) →
  ( ask(L=eof) → handle_one(R,right,Os)
  + ask(L=eol) →
    (read(left,L'), handle_two(L',R,idle,Os))
  + ask(L=/=eol AND L=/=eof) →
    (read(left,L'),handle_two(L',R,left,Os)))

% plus a third clause, symmetric to the second one

```

9. We now want to let `streamer` benefit from the improvements we have obtained via this transformation. First, we transform its definition by splitting `monitor(Ls,Rs,idle,Os)`, and obtain:

```

streamer(Os) ← reader(left,[L|Ls]),
  reader(right,[R|Rs]),
  monitor([L|Ls],[R|Rs],idle,Os).

```

10. Next, we unfold the two `reader` atoms, and eliminate the redundant `ask` guards.

```

streamer(Os) ← read(left,L),
  reader(left,Ls),
  read(right,R),
  reader(right,Rs),
  monitor([L|Ls],[R|Rs],idle,Os).

```

11. We can now fold `handle_two` in it, obtaining:

```

streamer(Os) ← read(left,L),
  read(right,R),
  handle_two(L,R,idle,Os).

```

Now, the definition of `streamer` is more efficient than the original one. In particular it benefits from a straightforward left-to-right dataflow and is now independent of the definition of `reader` which is the one that most negatively influences the performances of the program, having to suspend and awaken itself virtually at each input token. Moreover, it is worth noticing that by suitably extending the notion of modes to CCP languages, one could also easily obtain a deadlock freeness result for the transformed program above, while this would be more difficult for the original program.

## 7 Conclusions

In this paper we have presented a unfold/fold transformation system for CLP with dynamic scheduling and for CCP which can be used both for program optimization and for proving deadlock freeness.

Since CLP with dynamic scheduling can be embedded into a sublanguage of CCP, one could think that we could have better considered the latter paradigm alone. However, this is not the case for the following two reasons: first, that CLP with dynamic scheduling would in any case require a thorough restatement of the operations and their applicability conditions (leaving it as a “simple special case” would not work). Secondly, that CLP has a far greater practical impact, as it is much more employed as a programming language than CCP is. Therefore, from a practical perspective, it is worthwhile to focus primarily on this paradigm.

The results contained in this paper should provide the basis to develop a more general transformation system for CLP with dynamic scheduling and CCP. In particular, we are now investigating how to extend the results in [11] to these languages in order to obtain also a replacement transformation.

We conclude with some discussion on related work. To the best of our knowledge, the problem of transforming concurrent logic programs has been first tackled by Ueda and Furukawa in [33], where unfold/fold transformations of Guarded Horn Clauses (GHC) are considered. The main difference between the present paper and [33] lies in the fact that our systems takes advantage of the greater flexibility of the CCP (wrt GHC) and allows transformations not possible with the tools of [33]. For instance, we can define the unfolding as a simple body replacement operation without any additional applicability condition, while this is not the case for GHC. Moreover, the distributive operation is not present in [33], while it is of basic importance in our method. Another work dealing with unfold/fold techniques in a concurrent setting is [14], where De Francesco and Santone investigate transformations of CCS programs. The results in [14] are rather different from ours, since they define an applicability condition for the folding operation based on the notion of “guardness” which does not depend on the history of the transformation. Moreover, the correctness result in [14] is given with respect to a bisimulation semantics. Finally, partial evaluation of the AKL language (a language similar to CCP) has been defined by Sahlin in [29].

## References

- [1] K. R. Apt and I. Luitjes. Verification of logic programs with delay declarations. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Fourth*

- International Conference on Algebraic Methodology and Software Technology, (AMAST'95)*, Lecture Notes in Computer Science, Berlin, 1995. Springer-Verlag.
- [2] N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.
  - [3] R.M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.
  - [4] P. Chambre, P. Deransart, and J. Małuszynski. A proof method for safety properties of clausal concurrent constraint programs. In F. de Boer and M. Gabbrielli, editors, *Proc. JICSLP'96 Post-Conference Workshop on Verification and Analysis of Logic Programs*, 1996. Technical Report TR-96-31, Dipartimento di Informatica di Pisa.
  - [5] K.L. Clark and S. Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of IJCAI'77*, pages 419–420, 1977.
  - [6] M. Codish, M. Falaschi, and K. Marriott. Suspension Analyses for Concurrent Logic Programs. *ACM Transactions on Programming Languages and Systems*, 16(3):649–686, 1994.
  - [7] C. Codognot, P. Codognot, and M. Corsini. Abstract Interpretation for Concurrent Logic Languages. In S. Debray and M. Hermenegildo, editors, *Proc. North American Conf. on Logic Programming'90*, pages 215–232. MIT Press, 1990.
  - [8] F.S. de Boer, M. Gabbrielli, and C. Palamidessi. Proving correctness of constraint logic programs with dynamic scheduling. In *Static Analysis, Third International Static Analysis Symposium, (SAS'96)*, LNCS. Springer-Verlag, Berlin, 1996.
  - [9] M. Garcia de la Banda, K. Marriott, and P. Stuckey. Efficient analysis of logic programs with dynamic scheduling. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*, pages 417–431. MIT Press, 1995.
  - [10] S. Debray, D. Gudemann, and P. Bigot. Detection and optimization of suspension-free logic programs. In M. Bruynooghe, editor, *Proc. Eleventh International Logic Programming Symposium*, pages 487–504. MIT Press, 1994.
  - [11] S. Etalle and M. Gabbrielli. The replacement operation for CLP modules. In N. Jones, editor, *ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '95)*, pages 168–177. ACM press, 1995.
  - [12] S. Etalle and M. Gabbrielli. Layered modes. In F. de Boer and M. Gabbrielli, editors, *Proc. JICSLP'96 Post-Conference Workshop on Verification and Analysis of Logic Programs*, 1996. Technical Report TR-96-31, Dipartimento di Informatica di Pisa.
  - [13] S. Etalle and M. Gabbrielli. Transformations of CLP modules. *Theoretical Computer Science*, 166(1):101–146, 1996.
  - [14] N. De Francesco and A. Santone. Unfold/fold transformation of concurrent processes. In H. Kuchen and S. Doaitse Swierstra, editors, *Proc. 8th Int'l Symp. on Programming Languages: Implementations, Logics and Programs*, volume 1140, pages 167–181. Springer-Verlag, 1996.
  - [15] Nevin Heintze, Spiro Michaylov, and Peter J. Stuckey. CLP( $\mathcal{R}$ ) and some electrical engineering problems. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press. Also in *Journal of Automated Reasoning* vol. 9, pages 231–260, October 1992.
  - [16] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.
  - [17] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.
  - [18] J. Jaffar, S. Michaylov, and P. Stuckey. The clp( $\nabla$ ) language and system. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.
  - [19] Joxan Jaffar and Michael J. Maher. Constraint logic programming: A survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
  - [20] H. Komorowski. Partial evaluation as a means for inferring data structures in an applicative language: A theory and implementation in the case of Prolog. In Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, pages 255–267. ACM, 1982.
  - [21] A. Lakhota and L. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2,3):211–225, 1988.
  - [22] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Option Trading. *IEEE Expert*, 2(3), 1987.
  - [23] M. J. Maher. A CLP view of Logic Programming. In H. Kirchner and G. Levi, editors, *Proc. of the Third Int'l Conf. on Algebraic and Logic Programming*, volume 632 of *Lecture Notes in Computer Science*, pages 364–383. Springer-Verlag, Berlin, 1992.
  - [24] M.J. Maher. A transformation system for deductive databases with perfect model semantics. *Theoretical Computer Science*, 110(2):377–403, March 1993.
  - [25] E. Marchiori and F. Teusink. Proving termination of logic programs with delay declarations. In J. Lloyd, editor, *Proc. Twelfth International Logic Programming Symposium*. MIT Press, 1995.
  - [26] K. Marriott, M. Garcia de la Banda, and M. Hermenegildo. Analyzing logic programs with dynamic scheduling. In *Proc. 21st Annual ACM Symp. on Principles of Programming Languages*, pages 240–253. ACM Press, 1994.
  - [27] L. Naish. An introduction to mu-prolog. Technical Report 82/2, The University of Melbourne, 1982.

- [28] A. Pettorossi and M. Proietti. Transformation of logic programs: Foundations and techniques. *Journal of Logic Programming*, 19,20:261–320, 1994.
- [29] D. Sahlin. Partial Evaluation of AKL. In *Proceedings of the First International Conference on Concurrent Constraint Programming*, 1995.
- [30] V.A. Saraswat, M. Rinard, , and P. Panangaden. Semantics foundations of concurrent constraint programming. In *Proc. Eighteenth Annual ACM Symp. on Principles of Programming Languages*. ACM Press, 1991.
- [31] V.A. Saraswat and M. Rinard. Concurrent constraint programming. In *Proc. of the Seventeenth ACM Symposium on Principles of Programming Languages*, pages 232–245. ACM, New York, 1990.
- [32] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.
- [33] K. Ueda and K. Furukawa. Transformation rules for GHC Programs. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 582–591. Institute for New Generation Computer Technology, Tokyo, 1988.
- [34] M.H. van Emden and G.J. de Lucena. Predicate logic as a language for parallel programming. In K.L. Clark and S.-A. Tärnlund, editors, *Logic Programming*, London, 1982. Academic Press.
- [35] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.

## A CLP with dynamic scheduling

In this appendix, following [8], we give a more precise formalization of the computational model of CLP with dynamic scheduling. To this aim we first define the assertion language for delay conditions as follows. We assume a given set  $\Delta$  of selection predicates (e.g.  $\text{ground}(X)$ ) to be used in the conditions of the delay declarations, a given set  $\mathcal{C}$  of constraints and a given structure  $\mathcal{D}$ .

**Definition A.1** The language of delay conditions is defined by the following grammar, where  $c \in \mathcal{C}$  and  $\delta \in \Delta$ :

$$\phi ::= c \mid \delta \mid \phi \wedge \psi \mid \phi \vee \psi$$

Note that we do not allow negation, since we require that delay conditions are interpreted as sets of constraints closed under entailment.

An interpretation for delay conditions is given by a function which, for each condition, returns the set of constraints which satisfy it. A constraint  $c$  viewed as a delay condition is satisfied by all the constraints  $d$  that entail  $c$ . For selection predicates, we assume given an interpretation  $I : \Delta \rightarrow \wp(\mathcal{C})$  which defines their meaning. A constraint  $c$  satisfies a selection predicate  $\delta$  if  $c \in I(\delta)$ .

For example, the interpretation of  $\text{ground}(X)$  is the set of all the constraints which force  $X$  to assume a unique value, that is

$$\text{ground}(X) = \{c \mid \forall \vartheta, \vartheta' \text{ if } c\vartheta \text{ and } c\vartheta' \text{ are true in } \mathcal{D} \text{ then } X\vartheta = X\vartheta'\}.$$

The logical operations of conjunction and disjunction are interpreted in the classical way by their corresponding set theoretic operations of intersection and union. Formally, the interpretation of delay conditions is then defined as follows.

**Definition A.2** An interpretation  $[ \cdot ]$  is a function from delay conditions into non-empty sets of constraints such that:

$$\begin{aligned} [c] &= \{d \mid \mathcal{D} \models d \rightarrow c\} & [\delta] &= I(\delta) \\ [\phi \wedge \psi] &= [\phi] \cap [\psi] & [\phi \vee \psi] &= [\phi] \cup [\psi]. \end{aligned}$$

Satisfaction of delay condition is then formalized as follows.

**Definition A.3** A constraint  $c$  satisfies a delay condition  $\phi$  if  $c \in [\phi]$ . An atom  $p(\bar{t})$  in the context of a constraint  $c$  satisfies its delay condition if, assuming that  $\text{DELAY } p(\bar{X}) \text{ UNTIL } \phi$  is the delay declaration for  $p$  in  $P$ , we have that  $(c \wedge \bar{t} = \bar{Y}) \in [\phi[\bar{X}/\bar{Y}]]$  holds, where  $\bar{Y}$  are new distinct variables which do not appear in  $c$  and in  $\bar{t}$ .

As previously mentioned, we impose three assumptions on delay declarations. The first one describes a property which is satisfied by the majority of the (C)LP systems with delay declarations: if an atom is not delayed then adding more information will never cause it to delay. This amounts to say that the interpretation of any selection predicate  $\delta$  is closed under entailment, i.e.,

(i) for each  $\delta \in \Delta$ , if  $c \in I(\delta)$  and  $\mathcal{D} \models d \rightarrow c$  then  $d \in I(\delta)$ .

It is straightforward to check that this requirement ensures that the interpretation  $[ \cdot ]$  of a delay condition  $\phi$  is also closed under entailment. Generic negated conditions are not allowed because they could violate previous requirements. For example, this is case for the assertion  $\neg \text{ground}(X)$ , which is satisfied by all the constraints which do not force  $X$  to assume a unique value.

Furthermore, delay conditions are usually assumed to be consistent wrt renaming, so we require that

(ii) for each delay condition  $\phi$  whose free variables are  $\bar{X}$ , if  $c \in [\phi]$  then  $c \wedge \bar{Y} = \bar{X} \in [\phi[\bar{X}/\bar{Y}]]$ .

where  $\bar{Y}$  contains new fresh variables ( $\phi[\bar{X}/\bar{Y}]$  denotes the formula obtained from  $\phi$  by replacing the variables  $\bar{X}$  by  $\bar{Y}$ ).

Finally, we assume that for each predicate symbol exactly one delay declaration is given. This is not restrictive, since multiple declarations can be obtained by using logical connectives in the syntax of conditions.

Now, we can specify more precisely a derivation step for CLP with dynamic scheduling as follows.

Then, a goal  $c, p(\bar{t}_1), \dots, p(\bar{t}_n)$  in the program  $P$  results in the goal

$$c \wedge (\bar{t}_i = \bar{s}_i) \wedge d, p_1(\bar{t}_1), \dots, p_{i-1}(\bar{t}_{i-1}), \bar{B}, p_{i+1}(\bar{t}_{i+1}), \dots, p(\bar{t}_n)$$

provided that

1. the atom  $p_i(\bar{t}_i)$  in the context of the constraint  $c$  satisfy its delay declaration in  $P$ ,
2. there exists a (renamed version of a) clause  $p_i(\bar{s}_i) \leftarrow d, \bar{B}$  in  $P$  which does not share variables with the given goal,
3.  $\mathcal{D} \models \exists (c \wedge (\bar{t}_i = \bar{s}_i) \wedge d)$ .

A derivation for the query  $Q$  in the program  $P$  is therefore a finite or infinite sequence of goals, starting in  $Q$ , such that every next goal is obtained from the previous one by means of a derivation step defined as above.