A Transformation System for Modular CLP Programs[1]

Sandro Etalle[1,2] and Maurizio Gabbrielli[1,3]

[1] CWI
Kruislaan 413, 1098 SJ Amsterdam, The Netherlands

[2] D.I.S.I, Università di Genova
Viale Benedetto XV 3, 16132 Genova, Italy

[3] Dipartimento di Informatica, Università di Pisa
Corso Italia 40, 56125 Pisa, Italy
email: sandro@disi.unige.it, gabbri@di.unipi.it

Abstract

We propose a transformation system for CLP programs and modules. The framework is inspired by the one of Tamaki and Sato for pure logic programs [19]. Here, the use of CLP allows us to introduce some new operations such as splitting and constraint replacement. We provide two sets of applicability conditions. The first one guarantees that the original and the transformed programs have the same computational behavior, in terms of answer constraints. The second set contains more restrictive conditions that ensure *compositionality*: we prove that under these conditions the original and the transformed modules have the same answer constraints also when they are composed with other modules. As corollaries we obtain the correctness of both the modular and the non-modular system w.r.t. the least model semantics.

1 Introduction

As shown by a number of applications, programs transformation is a powerful methodology for the development and optimization of large programs. In this field, the unfold/fold transformation rules were first introduced by Burstall and Darlington [5] for transforming clear, simple functional programs into equivalent, more efficient ones. Afterwards, Tamaki and Sato [19] proposed an elegant framework for the transformation of logic programs based on the same rules. Their system was proven to be correct w.r.t. the least Herbrand model semantics [19] and the computed answer substitution semantics [13]. The system was later extended to logic programs with negation and serious research effort has been devoted to proving its correctness w.r.t. the various semantics available for normal programs.

All the (unfold/fold) transformation systems proposed so far for (constraint) logic programs, with the only exception of [15], assume that the en-

[1] To appear in Proc. ICLP '95.

tire program is available at the time of transformation. This is often an unpractical assumption, either because not all program components have been defined, or because for handling the complexity a large program has been broken into several smaller *modules*. Indeed, the incremental and modular design is by now a well established software-engineering methodology which helps to verify and maintain large applications. Modularity has received a considerable attention also in the field of logic programming, as the recent survey [4] shows. Adhering to the above mentioned methodology, we consider here CLP programs as a combination of separate modules. Each module partially defines some predicates, and different modules are combined together by a simple composition operator. Now, a transformation system for modules requires ad-hoc applicability conditions: when we transform P into P' we don't just want P and P' to have the same (answer constraint) semantics: we want them to be observationally equivalent *whatever the context in which they are employed*. When this condition is satisfied we say that P and P' are *observationally* congruent.

In this paper, we develop a transformation system for the optimization of CLP modules. This is accomplished in two steps. First, we generalize the Unfold/Fold system of Tamaki and Sato [19] to CLP programs. The full use of CLP allows us to introduce some new operations, such as splitting and constraint replacement, which broaden the range of possible optimizations. We also define new applicability conditions for the folding operation which avoid the use of substitutions and which are simpler that the ones used previously.

Afterwards, we define a (compositional) transformation system for modules. This is obtained by adding some further applicability conditions, which we prove sufficient to guarantee that the transformed module is observationally congruent to the original one. This system allows us to transform independently the components of an application, and then to combine together the results while preserving the original meaning of the program in terms of answer constraints. This is useful when a program is not completely specified in all its parts, as it allows us to optimize on the available modules. When a new module is added, we can just compose it (or its transformed version) with the already optimized parts, being sure that the composition of the transformed modules and the composition of the original ones have the same computational behavior in terms of answer constraints. From a particular case of this correctness result it follows that also the non-modular transformation system preserves the computational behavior and the least model (on the relevant algebraic structure) of programs.

The proofs of the results are contained in the extended version [7].

2 Preliminaries: CLP programs and Modules

The reader is assumed to be familiar with the terminology and the main results on the semantics of Constraint Logic Programming (CLP for short). Here we introduce some notations that we will use in the sequel. The original

paper [10] by Jaffar and Lassez and the recent survey [11] by Jaffar and Maher provide the necessary background material. We assume programs defined on a signature with predicates where the set of predicate symbols, denoted by Π, is partitioned into two disjoint sets: Π_c (containing predicate symbols used for constraints) which contains also the equality symbol "=", and Π_b (containing symbols for user definable predicates). The notations $\tilde{t}$ and $\tilde{x}$ will denote a tuple of terms and of distinct variables respectively, while $\tilde{B}$ will denote a (finite, possibly empty) conjunction of atoms. The connectives "," and $\Box$ will often be used instead of $\wedge$ to denote conjunction. We find convenient to use the notation $\exists_{-\tilde{x}}\ \phi$ from [11] to denote the existential closure of the formula ϕ *except* for the variables $\tilde{x}$ which remain unquantified.

A *constraint* is a first order formula whose predicate symbols are all in Π_c. A CLP rule is denoted by $H \leftarrow c \Box B_1, \ldots, B_n$ where c is a constraint, H (the head) and $B_1, \ldots, B_n$ (the body) are atomic formulas which use predicate symbols from Π_b only. Analogously a *goal* (or query) is denoted by $c \Box B_1, \ldots, B_n$. Given two atoms A and H, we write $A = H$ as a shorthand for the conjunction $a_1 = t_1 \wedge \ \ldots \ \wedge a_n = t_n$ if, for some predicate symbol p and natural n, $A = p(a_1, \ldots, a_n)$ and $H = p(t_1, \ldots, t_n)$. Otherwise $A = H$ denotes *false*. This notation readily extends to conjunctions of literals. By naturally extending the usual notion used for pure logic programs, we say that a query $c \Box \tilde{C}$ is *an instance* of the query $d \Box \tilde{D}$ iff for any solution γ of c there exists a solution δ of d such that $\tilde{C}\gamma = \tilde{D}\delta$. Finally, we will use the following definition of equivalence on clauses. Intuitively, it generalizes to the CLP case the notion of variance when bodies of clauses are viewed as multisets.

Definition 2.1 Let $cl_1 : \ A_1 \leftarrow c_1 \Box \tilde{B}_1$ and $cl_2 : \ A_2 \leftarrow c_2 \Box \tilde{B}_2$ be two clauses. We write $cl_1 \simeq cl_2$ iff for any $i, j \in [1, 2]$ and for any $\mathcal{D}$-solution ϑ of c_i there exists an $\mathcal{D}$-solution γ of c_j such that $A_i\vartheta = A_j\gamma$ and $\tilde{B}_i\vartheta$ and $\tilde{B}_j\gamma$ are equal as multisets. $\Box$

The semantics of CLP programs is based on a *structure* $\mathcal{D}$ which determines an interpretation for the constraints. Given a structure $\mathcal{D}$ and a constraint c, $\mathcal{D} \models c$ denotes that c is *true* under the interpretation provided by $\mathcal{D}$. If ϑ is a valuation (i.e. a mapping of variables on the domain of $\mathcal{D}$), and $\mathcal{D} \models c\vartheta$ holds, then ϑ is called a $\mathcal{D}$-*solution* of c ($c\vartheta$ denotes the application of ϑ to the variables in c). We recall that there exists [10] the least $\mathcal{D}$-model of a program P which is the natural CLP counterpart of the least Herbrand model for logic programs.

The operational model of CLP is obtained from SLD resolution by simply substituting $\mathcal{D}$-solvability for unifiability. More precisely, a derivation step for a goal $G : \ c_0 \Box B_1, \ldots, B_n$ in the program P results in a goal of the form $c_1 \Box B_1, \ldots, B_{i-1}, \tilde{B}, B_{i+1}, \ldots, B_n$ if B_i is the atom selected by the selection rule and there exists a clause in P standardized apart (i.e. with no variables in common with G) $H \leftarrow c \Box \tilde{B}$ such that $c_1 : \ (c_0 \wedge (B_i = H) \wedge c)$ is $\mathcal{D}$-satisfiable, that is, $\mathcal{D} \models \exists\, c_1$. The notion of derivation (in the program P) of

a goal G_i from a goal G is the usual one and in the following will be denoted by $G \overset{P}{\rightsquigarrow} G_i$. A derivation is *successful* if it is finite and its last element is a goal of the form $(c\ \Box)$. In this case, $\exists_{-Var(G)}\ c$ is called the *answer constraint*. Since the result of a CLP computation is an *answer constraint*, it is natural to say that two programs are *observationally equivalent* to each other iff they produce the same answer constraints (up to logical equivalence in the structure $\mathcal{D}$) for any query. This concept is formalized in the following Definition.

Definition 2.2 (Program's Equivalence) Let P_1, P_2 be CLP programs. We say that P_1 and P_2 are *(observationally) equivalent*, $P_1 \approx P_2$, iff, for any query Q and for any $i, j \in [1,2]$, if there exists a derivation $Q \overset{P_i}{\rightsquigarrow} c_i\ \Box$ then there exists a derivation $Q \overset{P_j}{\rightsquigarrow} c_j\ \Box$ such that $\mathcal{D} \models \quad \exists_{-Var(Q)}\ c_i \quad \leftrightarrow \exists_{-Var(Q)}\ c_j$. $\Box$

This notion is particularly relevant in the field of program's transformation: in fact we say that a transformation is *correct* iff it preserves program's equivalence, that is, if it maps programs into equivalent ones.

2.1 Modular CLP Programs

Following the original paper of R. O'Keefe [17], the approach to modular programming we consider here is based on a *meta-linguistic* programs composition mechanism. This provides a formal background to the usual software engineering techniques for the incremental development of programs. Viewing modularity in terms of *meta-linguistic* operations on programs has several advantages. In fact it leads to the definition of a simple and powerful methodology for structuring programs which does not require to extend the CLP theory (this is not the case if one tries to extend CLP programs by *linguistic* mechanisms richer than those offered by clausal logic). Moreover, *meta-linguistic* operations are quite powerful, indeed the typical mechanisms of the object-oriented paradigm, such as encapsulation and information hiding, can be realized by means of simple composition operators [2]. Here we follow [3] and say that a module is a CLP program P together with a set $Op(\mathcal{P})$ of predicate symbols specifying the *open* predicates.

Definition 2.3 (Module) A CLP module $\mathcal{P}$ is a pair $\langle P, Op(\mathcal{P})\rangle$ where P is a CLP program and $Op(\mathcal{P})$ is a set of predicate symbols. $\Box$

The idea underlying the previous definition is that the *open* predicates, specified in $Op(\mathcal{P})$, behave as an interface for composing $\mathcal{P}$ with other modules. On one hand, the definition of open predicates could be partially given in $\mathcal{P}$ and further specified by *importing* it from other modules. Symmetrically, the definitions of open predicates may be *exported* and used by other modules.

To compose CLP modules we again follow [3] and use a simple program union operator. We denote by $Pred(E)$ set of predicate symbols which appear in the expression E.

Definition 2.4 (Module Composition) Let $\mathcal{P} = \langle P, Op(\mathcal{P})\rangle$ and $\mathcal{Q} = \langle Q, Op(\mathcal{Q})\rangle$ be modules. We define $\mathcal{P} \oplus \mathcal{Q} = \langle P \cup Q, Op(\mathcal{P}) \cup Op(\mathcal{Q})\rangle$ provided that $Pred(P) \cap Pred(Q) \subseteq Op(\mathcal{P}) \cap Op(\mathcal{Q})$ holds. Otherwise $\mathcal{P} \oplus \mathcal{Q}$ is undefined. □

So, when composing $\mathcal{P}$ and $\mathcal{Q}$, we require the common predicate symbols to be open in both modules. As previously mentioned, more sophisticated compositions (like encapsulation, inheritance and information hiding) can be obtained from the one defined above by suitably modifying the treatment of the interfaces (essentially by introducing renamings to simulate hiding and overriding).

Now, in order to define the correctness of a transformation system for modules, we need a notion of module's equivalence. Here, the relation $\approx$ given for non-modular programs in Definition 2.2 is too weak for our purposes. For instance, consider the modules $\mathcal{P}: \langle\{p \leftarrow r\}, \{r\}\rangle$ and $\mathcal{Q}: \langle\{p \leftarrow q\}, \{q\}\rangle$. $\mathcal{P}$ and $\mathcal{Q}$ are $\approx$-equivalent, while they clearly have quite different behaviors when composed with other modules (consider for example $\mathcal{R}: \langle\{r\}, \{r\}\rangle$). The notion of equivalence which we need when transforming CLP modules has to take into account also the contexts given by the $\oplus$ composition. In other words, we have to strengthen the relation $\approx$ to obtain a congruence wrt the $\oplus$ operator. This is done as follows.

Definition 2.5 (Module's Congruence) Let $\mathcal{P}$ and $\mathcal{Q}$ be CLP modules. We say that $\mathcal{P}$ is *observationally congruent* to $\mathcal{Q}$, $\mathcal{P} \approx_c \mathcal{Q}$, iff $Op(\mathcal{P}) = Op(\mathcal{Q})$ and for every module $\mathcal{R}$ such that $\mathcal{P}\oplus\mathcal{R}$ and $\mathcal{Q}\oplus\mathcal{R}$ are defined, $\mathcal{P}\oplus\mathcal{R} \approx \mathcal{Q}\oplus\mathcal{R}$ holds. □

So $\mathcal{P} \approx_c \mathcal{Q}$ iff they have the same open predicates and, for any query, they produce the same answer constraints in any $\oplus$-context. By taking $\mathcal{R}$ as the empty module we immediately see that if $\mathcal{P} \approx_c \mathcal{Q}$ then $\mathcal{P} \approx \mathcal{Q}$.

3 A transformation system for CLP

We are now ready to introduce the transformation system. First, it is important to notice first that all the observable properties of computations we refer to are invariant under $\simeq$. Moreover, it can be proven (see [7]) that the applicability conditions and the result (up to $\simeq$) of the transformation operations are also invariant under $\simeq$. This implies that we can always replace any clause cl in a program P by a clause cl', provided that $cl' \simeq cl$. This operation is often useful to *clean up* the constraints, and, in general, to present a clause in a more readable form. We start from 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 3.1 (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_{new} and P_{old},

(I2) the predicates defined in P_{new} don't occur in P_{old} nor in the bodies of the clauses in P_{new}. □

Following this notation, we call *new* predicates those predicates that are defined in P_{new}. We also call *transformation sequence* a sequence of programs $P_0, \ldots, P_n$, in which P_0 is an initial program and each P_{i+1}, is obtained from P_i via a transformation operation.

Throughout this section we will use the following working example to illustrate our transformation system. To simplify the notation, when the constraint in a goal or in a clause is *true* we omit it. So the notation $H \leftarrow \tilde{B}$ actually denotes the CLP clause $H \leftarrow \mathit{true} \square \tilde{B}$.

Example 3.2 (Computing an average) Consider the following CLP($\Re$)[2] program `AVERAGE` computing the average of the values in a list. Values may be given in different currencies, for this reason each element of the list contains a term of the form $\langle$`Currency, Amount`$\rangle$; the applicable exchange rates may be found by calling predicate `exchange_rates`, which will return a list containing terms of the form $\langle$`Currency, Exchange_Rate`$\rangle$, where `Exchange_Rate` is the exchange rate relative to `Currency`.

```
c1: average(Xs, Av)  ← Len > 0 ∧ Av*Len = Sum  □
      exchange_rates(Rates),
      weighted_sum(Xs, Rates, Sum),
      len(Xs, Len).

    weighted_sum([], 0).
    weighted_sum([ ⟨Currency, Amount⟩ | Rest], Rates, Sum)  ←
      Sum = Amount*Value+Sum'  □
      member(⟨Currency, Value⟩, Rates),
      weighted_sum(Rest, Rates, Sum').

    len([], 0 ).
    len([H|Rest], Len)  ← Len = Len'+1  □ len(Rest, Len').
```

Notice that the definition of `average` needs to scan the list `Xs` twice. This is a source of inefficiency that can be fixed via a transformation sequence. □

The first transformation we consider is *unfolding*. This operation is basic to all the transformation systems and essentially consists in applying a derivation step to an atom in the body of a program clause, in all possible ways.

[2] CLP($\Re$) [12] is the CLP language obtained by considering the constraint domain $\Re$ of arithmetic over the real numbers.

Note Since all the observable properties we consider are invariant under $\simeq$, to simplify the notation all the definitions of the transformation operations will be given modulo reordering of the bodies. Moreover, we also assume that the clauses of a program have been renamed so that they are variable disjoint.

Definition 3.3 (Unfolding) Let $cl : A \leftarrow c \sqcap H, \tilde{K}$ be a clause in the program P, and $\{H_1 \leftarrow c_1 \sqcap \tilde{B}_1, \ldots, H_n \leftarrow c_n \sqcap \tilde{B}_n\}$ 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 $A \leftarrow c \wedge c_i \wedge (H = H_i) \sqcap \tilde{B}_i, \tilde{K}$. Then *unfolding H in cl in P* consists of replacing cl by $\{cl'_1, \ldots, cl'_n\}$ in P. In this situation we also say that $\{H_1 \leftarrow c_1 \sqcap \tilde{B}_1, \ldots, H_n \leftarrow c_n \sqcap \tilde{B}_n\}$ are the *unfolding* clauses. □

Example 3.2 (part 2) The transformation strategy which we use to optimize `AVERAGE` is often referred to as *tupling* (see [18]) or as *procedural join* (see [14]). First, we introduce a *new* predicate `avl` defined by the following clause

```
c2: avl(XS, RATES, AV, LEN) ← LEN > 0 ∧ AV*LEN = SUM ⊓
      exchange_rates(RATES),
      weighted_sum(Xs, RATES, SUM),
      len(XS, LEN).
```

`avl` differs from `average` only in the fact that it reports also the list of exchange rates and the length of the list `Xs`. Notice that `avl`, as it is now, needs to traverse the list twice as well. Now let P_0 be the *initial* program consisting of `AVERAGE` augmented by `c2` and assume that `avl` is the only *new* predicate. We start to transform P_0 by performing some unfolding operations. First we unfold `weighted_sum(XS, RATES, SUM)` in the body of `c2`. The resulting clauses, after having cleaned up the constraints and renamed some variables, are the following ones

```
c3: avl([], Rates, Average, Len) ← Len > 0 ∧ Average*Len = 0 ⊓
      exchange_rates(Rates),
      len([], Len).
c3': avl([⟨Currency,Amount⟩|Rest], Rates, Average, Len) ←
      Len > 0 ∧ Average*Len = Amount*Value+Sum' ⊓
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates),
      weighted_sum(Rest, Rates, Sum'),
      len([⟨Currency,Amount⟩|Rest], Len).
```

Now, observe that `c3` can be deleted by unfolding `len([], Len)`. Furthermore, in `c3'` we can unfold `len`$([\langle$`Currency`$,$`Amount`$\rangle|$`Rest`$]$, `Len`$)$. This yields the following clause:

```
c4: avl([⟨Currency,Amount⟩|Rest], Rates, Average, Len) ←
      Len > 0 ∧ Len = Len'+1 ∧ Average*Len = Amount*Value+Sum' ⊓
      exchange_rates(Rates),
```

```
      member(⟨Currency, Value⟩, Rates),
      weighted_sum(Rest, Rates, Sum'),
      len(Rest, Len').                                          □
```

We now introduce the *splitting* operation.

Definition 3.4 (Splitting) Let $cl : A \leftarrow c \,\square\, H, \tilde{K}$ be a clause in the program P, and $\{H_1 \leftarrow c_1 \,\square\, \tilde{B}_1,\ \ldots,\ H_n \leftarrow c_n \,\square\, \tilde{B}_n\}$ 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 $A \leftarrow c \wedge c_i \wedge (H = H_i) \,\square\, H, \tilde{K}$. 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 $\{cl'_1, \ldots, cl'_n\}$ in P. □

In other words, the splitting operation is just an unfolding operation in which we do not replace the atom H by the bodies of the unfolding clauses. The condition that for no two distinct i, j, $(H_i = H_j) \wedge c_i \wedge c_j$ is satisfiable is easily seen needed in order to obtain $\approx$ equivalent programs.

Example 3.2 (part 3) By applying the splitting operation to $\texttt{len}(\texttt{Rest}, \texttt{L}')$ in clause `c4` we obtain the following two clauses:

```
c5: avl([⟨Currency,Amount⟩],Rates, Average, Len)  ←
      Len > 0 ∧ Len = 1 ∧ Average*Len = Amount*Value+Sum'  □
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates),
      weighted_sum([], Rates, Sum'),
      len([], 0).
c6: avl([⟨Currency,Amount⟩,J|Rest],Rates,Average,Len) ← Len > 0 ∧
      Len = Len'+1 ∧ Len' = Len''+1 ∧ Average*Len = Amount*Value+Sum' □
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates),
      weighted_sum([J|Rest], Rates, Sum'),
      len([J|Rest], Len').
```

In clause `c6` we can now remove the superfluous constraint $\texttt{Len'} = \texttt{Len''+1}$, and in `c5` we can unfold both $\texttt{weighted\_sum}([\,], \texttt{Rates}, \texttt{Sum}')$ and $\texttt{len}([\,], 0)$. After this operations (and some cleaning-up) we end up with:

```
c7: avl([⟨Currency,Amount⟩],Rates, Average, 1) ←
      Average = Amount*Value  □
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates).
c8: avl([⟨Currency,Amount⟩,J|Rest], Rates, Average, Len)  ←
      Len > 0 ∧ Len = Len'+1 ∧ Average*Len = Amount*Value+Sum'  □
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates),
      weighted_sum([J|Rest], Rates, Sum'),
      len([J|Rest], Len').                                      □
```

In order to be able to perform the folding operation on clause c8 we need now a last, preliminary operation: the *constraint replacement*. In fact, as we will discuss later, to apply such a folding, c8 should contain also the constraint $\texttt{Len}' > 0$. Clearly, adding $\texttt{Len}' > 0$ to the body of c8 cannot be done via a simple cleaning-up of the constraints: it would transform c8 in a non $\simeq$-equivalent clause. However, notice that the variable $\texttt{Len}'$ in the atom $\texttt{len}([\texttt{J}|\texttt{Rest}], \texttt{Len}')$ (in the body of c8) represents the length of the list $[\texttt{J}|\texttt{Rest}]$ which obviously contains at least one element. Indeed, every time that c8 is used in a refutation its internal variable $\texttt{Len}'$ will eventually be bounded to a numeric value greater than zero. We can then safely add the redundant constraint $\texttt{Len}' > 0$ to body of c8. This type of operation is formalized by the following definition.

Definition 3.5 (Constraint Replacement) Let $cl : H \leftarrow c_1 \Box \tilde{B}$ be a clause of a program P and let c_2 be a constraint. If, for each successful derivation $true \Box \tilde{B} \overset{P}{\rightsquigarrow} d \Box$, we have that $\mathcal{D} \models \exists_{-Var(H)}\ c_1 \wedge d \leftrightarrow \exists_{-Var(H)}\, c_2 \wedge d$, then *replacing* c_1 *by* c_2 *in* cl consists in substituting cl by $H \leftarrow c_2 \Box \tilde{B}$ in P. □

Constraint replacement has some similarity with the refinement operation as defined in [16], however the optimization technique introduced in [16] is totally orthogonal to the one discussed here.

Example 3.2 (part 4) By performing a constraint replacement of $\texttt{Len} > 0 \wedge \texttt{Len} = \texttt{Len}' + 1 \wedge \texttt{Average} * \texttt{Len} = \texttt{Amount} * \texttt{Value} + \texttt{Sum}'$ by $\texttt{Len} > 0 \wedge \texttt{Len} = \texttt{Len}' + 1 \wedge \texttt{Average} * \texttt{Len} = \texttt{Amount} * \texttt{Value} + \texttt{Sum}' \wedge \texttt{Len}' > 0$ we can add the constraint $\texttt{Len}' > 0$ to the body of clause c8, obtaining the clause

```
c9: avl([⟨Currency,Amount⟩,J|Rest],Rates,Average,Len) ← Len > 0 ∧
      Len = Len'+1 ∧ Average*Len = Amount*Value+Sum' ∧ Len' > 0 □
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates),
      weighted_sum([J|Rest], Rates, Sum'),
      len([J|Rest], Len').
```

As we said before, the applicability conditions for the constraint replacement operations are satisfied because each time that the query $\texttt{len}([\texttt{J}|\texttt{Rest}], \texttt{Len}')$ succeeds in the current program the variable $\texttt{Len}'$ is constrained to a value greater than zero. □

We are now ready for the folding operation. This operation is a fundamental one, as it allows to introduce recursion in the new definitions. Intuitively, folding can be seen as the inverse of unfolding. Here, we take advantage of this intuitive idea in order to give a different formalization of its applicability conditions which we hope will be more easily readable than those existing in the literature. Recall that the initial program of a transformation sequence is partitioned into P_{new} and P_{old}. As a notational convenience in the following

definition we assume that the body of the folded clause has been reordered so that the atoms that are going to be folded are found on its left hand side.

Definition 3.6 (Folding) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence. Let also

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

$d : D \leftarrow c_D \sqcap \tilde{H}$ be a clause in P_{new}.

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

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

provided that the following three conditions hold:

(F1) (i) *"If we unfold* D *in* cl' *using* d *as unfolding clause, then we obtain* cl *back" (modulo* $\simeq$*),* or, equivalently,

(ii) $\mathcal{D} \models \ \exists_{-Var(A,\tilde{J},\tilde{H})} \ c_A \wedge e \wedge c_D \ \leftrightarrow \ \exists_{-Var(A,\tilde{J},\tilde{H})} \ c_A \wedge (\tilde{H} = \tilde{K})$.

(F2) *"*d *is the only clause of* P_{new} *that can be used to unfold* D *in* cl'*"*, that is

there is no clause $b : \ B \leftarrow c_B \sqcap \tilde{L}$ in P_{new} such that b is different from d 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 an unfolding in the sequence $P_0, \ldots, P_i$. □

Here, the constraint e acts as a bridge between the variables of d and cl. For this reason in the sequel we'll often refer to it as *bridge* constraint. 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_{new} as unfolding clauses, then we would obtain cl back[3]. It can be shown (see [7]) that conditions **F1**(i) and **F1**(ii) are equivalent to each other. Condition **F2** ensures that in P_{new} there exists no clause other than d that can be used as *unfolding clause*. Finally, the purpose of **F3** is to avoid the introduction of loops which can occur if a clause is folded by itself.

Example 3.2 (part 5) We can now fold `exchange_rates(Rates)`, `sum([J|Rest], Rates, Sum')`, `len([J|Rest], Len')` in `c9`, using `c2` as folding clause. In this case, the bridge constraint e has to be `XS` $=$ `[J|Rest]` $\wedge$ `RATES` $=$ `Rates` $\wedge$ `LEN` $=$ `Len'` $\wedge$ `AV` $=$ `Sum'/Len'`. In the resulting program, after cleaning up the constraints, the predicate `avl` is defined by the following clauses:

[3]Note however that the folding clause is always found in P_0 and usually does not belong to the "current" program, therefore in practice "undoing" a fold via an unfolding operation is usually not possible.

```
c7: avl([⟨Currency,Amount⟩],Rates, Average, 1) ←
      Average = Amount*Value  □
      exchange_rates(Rates),
      member(⟨Currency, Value⟩, Rates).
c10:avl([⟨Currency,Amount⟩,J|Rest], Rates, Average, Len) ←
      Len > 0 ∧ Len = Len'+1 ∧
      Average*Len = Amount*Value+(Average'*Len') ∧ Len' > 0  □
      avl([J|Rest], Rates, Average',Len'),
      member(⟨Currency, Value⟩, Rates).
```

Notice that, because of this last operation, the definition of `avl` is now recursive and it needs to traverse the list only once. Here, checking **F1** is a trivial task: what we have to do is to unfold `c10` using `c2` as unfolding clause, and check that the resulting clause is $\simeq$-equivalent to `c9`. Finally, in order to let also the definition of `average` enjoy of these improvements, we simply fold `weighted_sum(Xs, Rates, Sum)`, `len(Xs, Len)` in the body of `c1`, using `c2` as folding clause. The bridge constraint e is now `Xs` $=$ `XS` $\wedge$ `RATES` $=$ `Rates` $\wedge$ $\wedge$ `AV` $=$ `Av` $\wedge$ `LEN` $=$ `Len` and the resulting clause is, after the cleaning-up

```
c11:average(List, Av) ← Len > 0  □ avl(List, Rates, Av, Len).
```

Again, we could eliminate the constraint `Len` $>$ `0` in the body of `c11`, by applying a constraint replacement operation. In any case, we now have a definition of `average` which needs to scan the list only once. □

The transformation system given by the previous four operations is correct w.r.t. $\approx$, that is any transformed program together with a generic query Q will produce the same answer constraints of the original one. This is the content of the following Theorem.

Theorem 3.7 (Correctness) If $P_0, \ldots, P_n$ is a transformation sequence then $P_0 \approx P_n$. □

This result is proven by showing that the transformation system preserves a semantics modeling answer constraints called *answer constraint semantics* [9]. Since the least $\mathcal{D}$-model can be seen as an abstraction of such a semantics, we also have the following.

Corollary 3.8 If $P_0, \ldots, P_n$ is a transformation sequence then the least $\mathcal{D}$-models of P_0 and P_n coincide. □

4 A transformation system for CLP modules

The above results show the correctness of the transformation system when viewing each CLP program as an autonomous unit. However, as pointed out in the introduction, an essential requirement for *programming-in-the-large* is modularity: a program should be structured as a composition of interacting

modules. In this framework Theorem 3.7 falls short from the minimal requirement since it does not guarantee that a module P will be transformed into observationally congruent one P'. Transforming CLP modules requires then a strengthening of the applicability conditions given in the previous section. In what follows, we discuss such modifications considering the various operations one by one. In the sequel we call *open* atoms those atoms whose predicate symbol belong $Op(\mathcal{P})$. Moreover, we assume that the transformed version of a module has the same open predicates as the original one.

We start with the *unfolding* operation. In order to preserve the compositional equivalence, we need the following additional applicability condition:

(O1) The unfolding can not be applied to an open atom.

This condition is clearly needed. Consider for instance the module $\mathcal{P}_0$: $\langle\{\mathtt{c1}: \ \mathtt{p} \leftarrow \mathtt{q}.\}, \{\mathtt{q}\}\rangle$. Since $\mathcal{P}_0$ contains no clause whose head unifies with $\mathtt{q}$, unfolding $\mathtt{q}$ in $\mathtt{c1}$ will return an empty module $\mathcal{P}_1 = \emptyset$. Obviously $\mathcal{P}_0$ and $\mathcal{P}_1$ are not observationally congruent.

Being closely connected to the unfolding, the *splitting* operation requires the same kind of precautions when is applied to a modular program. Namely we need the following condition:

(O2) The splitting operation may not be applied to an open atom.

The example used to show the need for condition **O1** for the unfolding operation can be applied here to demonstrate the necessity of **O2**.

In order to apply the *constraint replacement* to modules we need to restate completely its applicability conditions. As am example showing the need of such a change, let us consider the module $\mathcal{P}_0$: $\langle\{\mathtt{c1}: \ \mathtt{p(X)} \leftarrow \mathit{true} \ \Box \ \mathtt{q(X)}.,$ $\mathtt{c2}: \ \mathtt{q(a)}\}, \ \{\mathtt{q}\}\rangle$. The only answer constraint to the query $\mathtt{q(X)}$ in $\mathcal{P}_0$ is $\mathtt{X} = \mathtt{a}$. Therefore, if we refer to the applicability conditions of Definition 3.5, we could add the constraint $\mathtt{X} = \mathtt{a}$ to the body of $\mathtt{c1}$ thus obtaining $\mathcal{P}_1$: $\langle\{\mathtt{p(X)} \leftarrow \mathtt{X} = \mathtt{a} \ \Box \ \mathtt{q(X)}., \ \ \mathtt{q(a)}\}, \ \{\mathtt{q}\}\rangle$. Now $\mathcal{P}_0$ and $\mathcal{P}_1$ are not observationally congruent since, for $\mathcal{Q}$: $\langle\{\mathtt{q(b)}.\}, \ \{\mathtt{q}\}\rangle$, the query $\mathtt{p(b)}$ succeeds in $\mathcal{P}_0 \oplus \mathcal{Q}$ and fails in $\mathcal{P}_1 \oplus \mathcal{Q}$.

Definition 4.1 (Constraint Replacement for Modules) Let $\mathcal{P}$ be a module, $cl : H \leftarrow c_1 \ \Box \ \tilde{B}$ be a clause of $\mathcal{P}$ and c_2 be a constraint. If

(O3) for each derivation $\mathit{true} \ \Box \ \tilde{B} \overset{\mathcal{P}}{\leadsto} d \ \Box \ \tilde{D}$ such that $\tilde{D}$ is either empty or contains only open atoms, we have $H \leftarrow c_1 \wedge d \ \Box \ \tilde{D} \ \simeq \ H \leftarrow c_2 \wedge d \ \Box \ \tilde{D}$

then *replacing* c_1 *by* c_2 *in* c_l consists in substituting cl by $H \leftarrow c_2 \ \Box \ \tilde{B}$ in $\mathcal{P}$. $\Box$

The difference between Definitions 4.1 and the previous one lies in the fact that here we cannot just refer to the successful derivations $\mathit{true} \ \Box \ \tilde{B} \overset{P}{\leadsto} d \ \Box$, but we also have to take into account those partial derivations that end in a

tuple of open atoms, whose definition could eventually be modified. It is easy to see when the set of open atoms is empty, Definitions 3.5 and 4.1 coincide, while if $Op(\mathcal{P}) \neq \emptyset$ then this definition is more restrictive than the previous one.

Finally, in order for the *folding* operation to preserve the observational congruence we require the head of the folding clause not to be an open atom. Consider for instance the initial module $\mathcal{P}_0 : \langle \{\texttt{c1}: \ \texttt{p} \leftarrow \texttt{q}., \ \texttt{c2}: \ \texttt{r} \leftarrow \texttt{q}.\}, \{\texttt{p}\}\rangle$ and assume $\mathcal{P}_{new} : \{\texttt{p} \leftarrow \texttt{q}.\}$. Since `r` is an old atom, we can fold `q` in `c2` using `c1` as folding clause. The resulting module is $\mathcal{P}_1 : \langle \{\texttt{c3}: \ \texttt{p} \leftarrow \texttt{q}., \ \texttt{c4}: \ \texttt{r} \leftarrow \texttt{p}.\}, \{\texttt{p}\}\rangle$. Again $\mathcal{P}_0$ and $\mathcal{P}_1$ are not observationally congruent (consider the composition with the module $\mathcal{Q} = \langle \{\texttt{p}.\}, \{\texttt{p}\}\rangle$). Since the *new* predicates are the only ones that can be used in the heads of folding clauses, we can express this additional applicability condition for folding as follows:

(O4) No open predicate is also a *new* predicate.

It is worth noticing that open atoms may still be *folded*. Below (Example 3.2, part 6), we report an example of such a case.

Using the additional applicability conditions just introduced, we can define now the transformation sequence for CLP modules (for short, modular transformation sequence).

Definition 4.2 (Modular transformation sequence) Consider a module $\mathcal{P}_0 = \langle P_0, Op(\mathcal{P}_0)\rangle$ and a transformation sequence $P_0, \ldots, P_n$. We say that $\mathcal{P}_0, \ldots, \mathcal{P}_n$ is a *modular* transformation sequence iff, for $i \in [0, n]$, $\mathcal{P}_i = \langle P_i, Op(\mathcal{P}_0)\rangle$ and the conditions **O1**...**O4** are satisfied by all the operations used in $P_0, \ldots, P_n$. □

As expected, for a modular transformation sequence we can prove a correctness theorem stronger than Theorem 3.7. Indeed, the system transforms a module into an observationally congruent one.

Theorem 4.3 (Correctness of the modular transformation sequence) Let $\mathcal{P}_0, \ldots, \mathcal{P}_n$ be a modular transformation sequence, then $\mathcal{P}_0 \approx_c \mathcal{P}_n$.

In other words, for any module $\mathcal{Q}$ such that $\mathcal{P}_0 \oplus \mathcal{Q}$ is defined, $\mathcal{P}_n \oplus \mathcal{Q}$ is also defined[4] and a generic query has the same answer constraints in $\mathcal{P}_0 \oplus \mathcal{Q}$ and $\mathcal{P}_n \oplus \mathcal{Q}$.

Example 3.2 (part 6) Program `AVERAGE` can be used in a modular context. Indeed, if we consider that the exchange rates between currencies are typically fluctuating ratios, it comes natural to assume `exchange_rates` as an open predicate which may refer to some external "information server" to access always the most up-to-date information. In this context, it is easy to

[4]This follows immediately from the fact that $\mathcal{P}_0$ and $\mathcal{P}_n$ contain definitions for the same predicate symbols.

check that all the transformations we performed satisfied **O1**...**O4**. Therefore Theorem 4.3 guarantees that the final program will behave exactly as the initial one, even in this modular setting. □

5 Conclusions

The works most closely related to this paper are Maher's [15] and the one of Bensaou and Guessarian [1]. Maher considers several transformations for deductive databases modules with constraints (allowing negation in the bodies of the clauses) and refers to the perfect model semantics. However the folding operation proposed in [15] is quite restrictive since it lacks the possibility of introducing recursion. Indeed it is a particular case of the one defined here. Moreover, our notion of module composition is more general than the one considered in [15], since the latter assumes each predicate is defined within a single module and does not allow mutual recursion among modules.

An extension of the Tamaki-Sato method to CLP programs has also been proposed by Bensaou and Guessarian [1], yet there are some substantial differences between [1] and our proposal. Firstly, since in an Unfold/Fold transformation sequence we allow more operations, we obtain a more powerful system. For instance, the transformation performed in Example 3.2 is not feasible with the tools of [1][5]. Secondly, the semantics they refer to is an extension to the CLP case of the so called C-semantics [6]. Since this semantics can be obtained as an abstraction of the answer constraint semantics, the result on the correctness of the Unfold/Fold system of [1] is a particular case of our Theorem 3.7 (see [7]). A third relevant difference is due to the fact that since modularity is not take into account in [1], the system introduced in that paper does not produce observationally congruent programs.

To conclude, our contributions can be summarized as follows. We have defined a transformation system for CLP based on the Unfold/Fold framework of Tamaki and Sato for logic programs [19]. Here, the use of CLP allowed us to define some new operations and to express the applicability conditions for the folding operation without the use of substitutions. Our definition of folding emphasizes its nature of being a quasi-inverse of the unfolding. We hope that this will provide a more intuitive explanation of its applicability conditions. The system is then proven to preserve the answer constraints and the least $\mathcal{D}$-model of the original program. A definition of a modular transformation sequence is given by adding some further applicability conditions. These conditions are shown to be sufficient to guarantee the correctness of the system w.r.t. the observational congruence. As previously argued, this provides a useful tool for the development of real software since it allows incremental

[5] In [1] it is used also a replacement operation which allows transformations that cannot be obtained with our system. However, such an operation cannot be fitted in a Unfold/Fold sequence, in particular no folding is allowed when the transformation sequence contains a replacement. Therefore the replacement in [1] is an operation orthogonal to Unfold/Fold transformations and beyond the scope of this paper. This issue is investigated in [8].

and modular optimizations of large programs.

Acknowledgments The authors want to thank Krzysztof Apt, Annalisa Bossi and the anonymous referees for their useful comments. This research has been supported by the ERCIM Fellowship Program and by the EC/HCM network EUROFOCS.

References

[1] N. Bensaou and I. Guessarian. Transforming Constraint Logic Programs. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*, 1994.

[2] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential logic programming. In *Proc. POPL'93*, pages 359–370. ACM Press, 1993.

[3] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. A Compositional Semantics for Logic Programs. *Theoretical Computer Science*, 122(1-2):3–47, 1994.

[4] M. Bugliesi, E. Lamma, and P. Mello. Modularity in logic programming. *Journal of Logic Programming*, 19-20:443–502, 1994.

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

[6] K. L. Clark. Predicate logic as a computational formalism. Res. Report DOC 79/59, Imperial College, Dept. of Computing, London, 1979.

[7] S. Etalle and M. Gabbrielli. Transformations of CLP Modules. CWI Technical report, 1995. Available at http://www.cwi.nl/cwi/publications/#AP.

[8] S. Etalle and M. Gabbrielli. Replacement of CLP Modules. In Proc. *ACM SIGPLAN Symposium* PEPM'95. ACM Press, 1995.

[9] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. ICLP'91*, pages 238– 252. The MIT Press, 1991.

[10] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. POPL'87*, pages 111–119. ACM Press, 1987.

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

[12] J. Jaffar, S. Michayov, P.J. Stuckey, and R.H. C. Yap. The CLP($\mathcal{R}$) language and system. *ACM Trans. on Programming Languages and Systems*, 14(3):339–395, 1992.

[13] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. In *Proc. Int'l Conf. on Fifth Generation Computer Systems*, pages 413–422. ICOT, Tokyo, 1988.

[14] A. Lakhotia and L. Sterling. Composing recursive logic programs with clausal join. *New Generation Computing*, 6(2,3):211–225, 1988.

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

[16] K. Marriott and P.J. Stuckey. The 3 R's of optimizing constraint logic programs: Refinement, removal and reordering. In *Proc. POPL'93.* ACM Press, 1993.

[17] R. A. O'Keefe. Towards an Algebra for Constructing Logic Programs. In *Proc. IEEE Symp. on Logic Programming*, pages 152–160, 1985.

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

[19] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. ICLP '84*, pages 127–139, 1984.