The Replacement Operation for CLP Modules

Sandro Etalle
CWI and Università di Genova

Maurizio Gabbrielli
CWI and Università di Pisa*

Abstract

In this paper we study the *replacement* transformation for Constraint Logic Programming modules. We define new applicability conditions which guarantee the correctness of the operation also wrt module's *composition*: under this conditions, the original and the transformed modules have the same *observable properties* also when they are composed with other modules. Furthermore, the applicability conditions are not bound to a specific notion of observable. Here we consider three distinct such notions: two of them are operational and are based on the computed constraints; the third one is the algebraic one based on the least model. We show that our transformation method can be applied in any of these distinct contexts, thus providing a parametric approach.

1 Introduction

Constraint Logic Programming (CLP for short) is a powerful declarative programming paradigm in which constraints are primitive elements and the computation is specified by a logical inference rule. CLP has already been successfully employed in many diverse fields such as financial analysis [19], circuit synthesis [14] and combinatorial search problems [30]. Its success is partially due to the fact that the declarative nature of CLP allows to solve complex problems by simple and concise programs. CLP's flexibility can be further enhanced by the adoption of constructs for structuring programs. This is an important step forward as the incremental and modular design is by now a well established software-engineering methodology used to design, verify and maintain large applications. Indeed, splitting a program into several smaller *modules* reduces the complexity of the design and of the validation phases, moreover, it also helps to develop adaptable software, as changes in program's specification can affect only some modules rather that the whole program. For these reasons, modularity has been receiving a considerable attention and, as the recent survey [7] shows, in the last few years several different proposals were introduced for integrating module constructs into logic languages. Here we adhere to the original approach of R. O'Keefe [25], and we consider a constraint logic program to be a combination of several separate modules, where different modules are combined together by a simple composition operator $\oplus$.

Central to the development of large and efficient applications is now the study of optimization techniques for programs and modules. Concerning the CLP paradigm, the literature on this subject can be divided into two main branches. On one hand we find methods which focus exclusively on the manipulation of the constraint for compile-time [24] and for low-level local optimization (in which the constraint solving may be partially compiled into imperative statements) [18]. Compile time optimizations based on static analysis have also been investigated [23]. On the other hand there are techniques such as the unfold/fold transformation systems, which were developed initially for Logic Programs [29] and then applied to CLP [22, 1, 10], these ones focus primarily on the declarative side of the program.

Replacement is a program transformation technique flexible enough to encompass both the above kind of optimization: it can be profitably used to manipulate both the constraint and the "declarative" side of a CLP program. In fact the replacement operation, which was introduced in the field of Logic Programming by Tamaki and Sato [29] and later applied to CLP in [22, 1], syntactically consists in replacing a conjunction of atoms in the body of a program clause by another conjunction. It is therefore a very general operation and it is able to mimic many other transformations, such as thinning, fattening [3] and folding (see [26] for a survey on transformation techniques for logic languages).

Clearly, a primary requirement a transformation operation should satisfy is *correctness*: the original and the transformed program should be equivalent wrt to some (operational or declarative) reference semantics. In the logic programming area, a lot of research [29, 20, 13, 28, 4, 5, 22, 1, 9, 27] has been devoted to the definition of *applicability conditions* sufficient to guarantee the correctness of replacement wrt several different semantics. Unfortunately, apart from [22], none of these transformation systems can be correctly applied to *modules*. In fact, since they all refer to semantics

*This paper has been prepared during both author's stay at CWI - Center for Mathematics and Computer Science - Amsterdam. Permanent address of authors: Sandro Etalle, D.I.S.I, Università di Genova Viale Benedetto XV 3, 16132 Genova, Italy. sandro@disi.unige.it. Maurizio Gabbrielli, Dipartimento di Informatica, Università di Pisa Corso Italia 40, 56125 Pisa, Italy gabbri@di.unipi.it.. The research of the first author has been partially supported by the ERCIM Fellowship Program The research of the second author has been supported by the EC/HCM network EUROFOCS under grant n. ERBCHBGCT930496

PEPM '95 La Jolla, CA USA
© 1995 ACM 0-89791-720-0/95/0006...$3.50

which are not compositional wrt $\oplus$, they provide correctness results which are adequate only if programs are seen as stand alone units. When we transform a *module* M into M' we don't just want M and M' to have the same behavior: we want them semantically equivalent *whatever is the context in which we use them.* In other words we need some further applicability conditions which guarantee that, given any other module Q, $M \oplus Q$ and $M' \oplus Q$ will be equivalent to each other. When this condition is satisfied we say that M and M' are *compositionally* equivalent or *congruent*[1].

Furthermore, even when restricting to the non modular setting, the applicability conditions so far provided for the replacement transformations suffer from drawbacks which, in our opinion, prevented a wider diffusion of the operation. On one hand, some of them [13, 28, 20, 22] do not allow replacement to introduce recursion, which, as we will shortly see, is an important feature for optimizing Constraint Logic Programs. On the other hand, other approaches [29, 4, 5, 27] do exploit the full potentiality of replacement, but at the price of applicability conditions which are discouragingly complicated.

In this paper we study optimizations based on the replacement operation for CLP modules. We provide some natural and relatively simple applicability conditions which ensure us that the transformed program is compositionally equivalent to the original one. Our approach is based on the following two requirements:

(i) *The replacing conjunction must be equivalent to the replaced one (in a sense which enforces compositional equivalence).* This is already the point where we depart from previous approaches: the equivalences used so far to relate the replacing and the replaced part are not sufficient to guarantee the preservation of compositional equivalence.

(ii) *The replacement must not introduce (fatal) loops.*

Here, we call a loop *fatal* if it prevents the computation from ending successfully. Indeed, the equivalence of the replacing and the replaced part alone is not sufficient to guarantee that the replacement is correct. We individuate two situations in which the operation certainly does not introduce any fatal loop:

(a) *When the replacing conjunction is at least as efficient as the replaced one.*
Referring to the operational semantics this means that each time we can compute an "answer" constraint c for the replaced conjunction (in the given program) in n steps, we can also compute the answer c for the replacing one in m steps with $m \leq n$. This is undoubtedly a desirable situation which fits well in the natural context in which the transformation is performed in order to increase program's execution speed. Moreover, this condition is flexible enough to allow us to introduce recursion (which can be seen as an example of *non-fatal* loop) in the definition of the predicates.

(b) *When the replacing conjunction is independent from the clause that is going to be transformed.*
This clearly guarantees that no loops are introduced.

The advantages of this approach to the replacement operation are twofold.

[1] Of course, depending on which observable property of computation we consider, different instances of congruence can be obtained.

Firstly, our method is parametric wrt the semantic properties of the program we want to maintain along the transformation. We consider here three such observable properties: two of them are *operational*, as they are based on the result of the the computations (the computed answer constraints), while the third one is a logical notion (the least model on the relevant algebraic structure). Depending on which property we refer to, we can naturally instantiate the generic notion of equivalence relative to the requirement (i) above and obtain applicability conditions which guarantee the preservation of the desired properties.

Secondly, as we said, our approach allows us to obtain *compositionally* equivalent programs. We can then transform independently the components of an application and successively combine together the results while preserving the original meaning of the program. This is also useful when a program is not completely specified in all its parts, as it allows us to optimize on the available modules. Moreover, the equivalence mentioned in (i) can be simply modified to match the "degree" of modularity we desire. Results for the non-modular cases are then obtained as easy corollaries.

The remainder of this paper is organized as follows. Next subsection contains some preliminaries on CLP programs. In Section 2 we introduce the optimization technique based on replacement through a simple example. In Section 3 we define formally CLP modules and the composition operation. In Section 4 we give the details of the applicability conditions we provide in order to obtain compositionally equivalent programs, wrt the answer constraints notion of observable, and we state the main correctness result. Section 5 shows how these conditions can be weakened when considering other (operational and logic) properties of modules. Section 6 concludes by comparing our results to those contained in some related papers.

1.1 Preliminaries: CLP programs

The reader is assumed to be familiar with the terminology and the main results on the semantics of constraint logic programs. The original paper [15] by Jaffar and Lassez and the recent survey [16] by Jaffar and Maher provide the necessary background material. We introduce now the notation we'll use in the sequel.

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. When no ambiguity arise, we will use $\tilde{x}$ also to denote a set of variables. The connectives "," and $\square$ will often be used instead of $\wedge$ to denote conjunction. We find also convenient to use the notation $\exists_{-\tilde{x}}\ \phi$ from [16] to denote the existential closure of the formula ϕ *except* for the variables $\tilde{x}$ which remain unquantified.

A *constraint* c is a first order formula built using primitive constraints, which are essentially predefined predicates over a computational domain $\mathcal{D}$. Formally, $\mathcal{D}$ is a *structure* which determines the interpretation of constraints. 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). A CLP rule is denoted by $H \leftarrow c \square B_1, \ldots, B_n$ where c is a constraint, H (the head) is an atom and $B_1, \ldots, B_n$ (the body) is a sequence of atoms. Analogously a *goal* (or query) is denoted by $c \square B_1, \ldots, B_n$. We recall that there exists ([15]) the least $\mathcal{D}$-model of a program P which is the natural CLP

counterpart of the least Herbrand model for logic programs. Here and in the following, given the atoms A, H, we write $A = H$ as a shorthand for:

- *false*, if H and A have different relation symbols,
- $a_1 = t_1 \wedge \ldots \wedge a_n = t_n$, if $A = p(a_1, \ldots, a_n)$ and $H = p(t_1, \ldots, t_n)$.

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 $G: c_0 \square B_1, \ldots, B_n$ in the program P results in a goal of the form $c_1 \square B_1, \ldots, B_{i-1}, \tilde{B}, B_{i+1}, \ldots, B_n$ 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) $H \leftarrow c \square \tilde{B}$ such that $c_1 : (c_0 \wedge (B_i = H) \wedge c)$ is $\mathcal{D}$-satisfiable, that is, $\mathcal{D} \models \exists\, c_1$. A derivation of length i for a goal G_0 in the program P is a sequence of goals $G_0, G_1, \ldots, G_i$ such that G_j is obtained from G_{j-1} in one derivation step in P, for $j \in [1, i]$. In the following a derivation $\xi: G_0, G_1, \ldots, G_i$ in P will be denoted by $G_0 \overset{P}{\rightsquigarrow} G_i$ and its length by $|\xi|$. Note that we denote by $G_0 \overset{P}{\rightsquigarrow} G_0$ a derivation of length 0 for G_0. A *successful* derivation (*refutation*) is a finite derivation whose last element is a goal of the form $(c \square)$. In this case, $\exists_{-Var(G)}\, c$ is called the *answer constraint* and is considered the result of the computation. Finally, the following relation $\simeq$ generalizes to CLP the notion of variance on clauses (viewing bodies as multisets). Consider two clauses $cl_1 : A_1 \leftarrow c_1 \square \tilde{B}_1$ and $cl_2 : A_2 \leftarrow c_2 \square \tilde{B}_2$. We write $cl_1 \simeq cl_2$ iff, for $i, j \in [1, 2]$, for any $\mathcal{D}$-solution ϑ of c_i there exists a $\mathcal{D}$-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. For the sake of simplicity, we will denote the $\simeq$ equivalence class of a clause c by c itself.

2 An Example

In this section we show what kind of optimizations can be achieved via replacement through a worked example. In particular we'll show how replacement allows us to introduce recursion in the definition of predicates. For this we employ a transformation strategy which is typically used in unfold/fold systems such as the one in [29]. Indeed, the applicability conditions we will give are general enough to let replacement mimic most of the transformations feasible with the tools of [29]. One advantage of replacement over folding is that the applicability conditions for the former refer solely to the (semantic) properties of the program we are working on, while for folding these depend also on the history of the transformation (that is, on the transformation steps previously performed). Naturally, to the replacement operation there is much more than just mimicking the folding one, since it allows optimizations which cannot be obtained by unfold/fold: elimination of redundant atoms in the bodies of the clauses is a simple typical example.

In order to provide the example we need the usual definition of *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. In order to simplify the notation, the definition is given modulo reordering of the atoms in the bodies and we assume that the clauses of a program are variable disjoint.

Definition 2.1 (Unfolding) Let $cl : A \leftarrow c \square H, \tilde{K}$ be a clause in a 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 $A \leftarrow c \wedge c_i \wedge (H = H_i) \square \tilde{B}_i, \tilde{K}$. Then *unfolding* H *in* cl *in* P consists of replacing cl by $\{cl'_1, \ldots, cl'_n\}$ in P. $\square$

In this situation $\{H_1 \leftarrow c_1 \square \tilde{B}_1, \ldots, H_n \leftarrow c_n \square \tilde{B}_n\}$ are referred to as the *unfolding* clauses.

Example 2.2 (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 ⟨`Currency,Amount`⟩. The applicable exchange rates may be found by calling the predicate `exchange_rates`, which will return a list containing terms (pairs) of the form ⟨`Currency,Exchange_Rate`⟩, where `Exchange_Rate` is the exchange rate relative to `Currency`. Despite its simplicity, this is a typical program that 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* (or imported) predicate, which may refer to some external information server to access always the most up-to-date information.

```
    average(List, Av) ←
        Av is the average of the list List

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

    sum(List, Rates, Sum) ←
        Sum is the sum of the values in the list List
        where each value is multiplied by the exchange
        rate corresponding to its currency

    sum([], 0).
    sum([ ⟨Curr, Amount⟩ | Ts], Rates, Sum) ←
        Sum = Amount*Value + Sum' □
        member(⟨Curr, Value⟩, Rates),
        sum(Ts, Rates, Sum').

    len(List, Len) ←
        Len is the length of list List

    len([], 0 ).
    len([H|Ts], Len) ← Len = Len'+1 □
        len(Ts, 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 unfolding and replacement operations. The transformation strategy which we are going use use is often referred to as *tupling* ([26]). First, we introduce a *new* predicate `sum_len` defined by the following clause

```
c2: sum_len(Xs, Rates, Sum, Len) ← □
        exchange_rates(Rates),
        sum(Xs, Rates, Sum),
        len(Xs, Len).
```

[2] CLP($\Re$) [17] is the CLP language obtained by considering the constraint domain $\Re$ of arithmetic over the real numbers. The signature for $\Re$ contains the constant symbols 0 and 1, the binary function symbols $+$ and $*$, and the binary predicate symbols $+, <, \leq$ for constraints which are interpreted on the real numbers as usual.

sum_len reports the weighted sum of the values in Xs, together with the length of the Xs itself and the list of the exchange rates. Notice that sum_len, as it is now, needs to traverse the list Xs twice as well. We start to transform AVERAGE by unfolding both sum(Xs, Rates, Sum) and len(Xs, Len) in the body of c2. This operations yield the module AV_1 which contains the following two clauses:

```
c3: sum_len([], Rates, 0, 0) ← □
        exchange_rates(Rates).
c4: sum_len([⟨Curr,Amount⟩|Rest], Rates, Sum, Len) ←
        Len = Len'+1  ∧ Sum = Amount*Value+Sum' □
        exchange_rates(Rates),
        member(⟨Curr, Value⟩, Rates),
        sum(Rest, Rates, Sum'),
        len(Rest, Len').
```

Now, we can *replace* exchange_rates(Rates), sum(Rest, Rates, Sum'), len(Rest, Len') by sum_len(Rest, Rates, Sum', Len') in the body of c4. In the resulting module AV_2, after cleaning up the constraints[3], the predicate sum_len is defined by the following clauses:

```
c3: sum_len([], Rates, 0, 0) ← □
        exchange_rates(Rates).
c5: sum_len([⟨Curr,Amount⟩|Rest],Rates,Sum,Len) ←
        Len = Len'+1  ∧ Sum = Amount*Value+Sum' □
        sum_len(Rest, Rates, Sum', Len'),
        member(⟨Curr, Value⟩, Rates).
```

Notice that, because of this last operation, the definition of sum_len is now recursive and it needs to traverse the list only once. Finally, in order to let also the definition of average enjoy of these improvements, we simply replace exchange_rates(Rates),sum(Xs,Rates,Sum),len(Xs,Len) by sum_len(Xs, Rates, Sum, Len) in the body of c1. After the cleaning-up the resulting clause is

```
c6: average(Xs, Av)  ← Len>0  ∧ Av * Len = Sum □
        sum_len(Xs, Rates, Sum, Len).
```

So, we have obtained the module AV_3, consisting of the clauses c6, c3 and c5, where we find a definition of average which needs to scan the list only once. The *correctness* of the transformations, will be discussed (and proved) in Section 4. □

3 Modular CLP Programs

We now provide a formal background to the usual software engineering techniques for the incremental development of programs. Following the original paper of R. O'Keefe [25], the approach to modular programming we consider here is based on a *meta-linguistic* programs composition mechanism.

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, in order to keep the presentation simple, we follow [6] and say that a module is a CLP program P together with a set π of predicate symbols specifying the *open* predicates. We call *open atom* an atom whose predicate symbol is in π.

Definition 3.1 (Module) A CLP module is a pair $\langle P, \pi \rangle$ where P is a CLP program and π is a set of predicate symbols. □

The underlying idea is that the *open* predicates, specified in π, behave as an interface for composing a module M with other modules, and they are allowed to be (further) specified by adjoining other modules. On one hand, the definition of open predicates could be partially given in M and further specified by *importing* it from other modules. Symmetrically, the definitions of open predicates may be *exported* and used by other modules. For instance, a deductive database can be seen as the composition of two (or more) modules. The first one $\mathcal{I}$ contains the *intensional part* in the form of some *rules* which refer to an unspecified *extensional part*. This is specified by another module $\mathcal{E}$ which contains some facts (unit clauses) describing the basic relations. So, the extensional predicates which are defined in $\mathcal{E}$ are exported to $\mathcal{I}$ when composing the two parts. Further definitions for the extensional predicates can be incrementally added to the database by adjoining new modules.

To compose CLP modules we again follow [6] and use a simple program union operator which takes into account the interface π. Here denote by $Pred(E)$ set of predicate symbols which appear in the expression E.

Definition 3.2 (Module Composition) Let $M_1 : \langle P_1, \pi_1 \rangle$ and $M_2 : \langle P_2, \pi_2 \rangle$ be modules. We define

$$M_1 \oplus M_2 = \langle P_1 \cup P_2, \pi_1 \cup \pi_2 \rangle$$

provided that $Pred(P_1) \cap Pred(P_2) \subseteq \pi_1 \cap \pi_2$ holds. Otherwise $M_1 \oplus M_2$ is undefined. □

So, when composing M_1 and M_2, 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).

4 Operational correctness of Replacement

As previously discussed, the replacement operations consists simply in replacing a conjunction of atoms in the body of a program clause by another conjunction. Clearly, some applicability conditions are necessary in order to ensure the correctness of the operation.

In this section we first define an *operational* notion of correctness based on the *answer constraints*. Then we provide some applicability conditions for replacement in form of a natural formalization of the requirements (i) and (ii) discussed in the introduction. Then we show that, whenever

[3] Since all the semantic properties we refer to are invariant under $\simeq$, we can always replace any clause cl in a program P by a clause cl', provided that $cl' \simeq cl$ (typically, we can rename the variables in cl). This operation is often referred to as a *"clean up"* since it is mainly used to present a clause in a more readable form.

these conditions are satisfied, the replacement operation is *operationally* correct. Later, in Section 5, we will also show how these conditions can be modified (weakened) when considering correctness based on different operational and logical notions.

Operational congruence

To define formally the notion of operational correctness we first provide the definition of module's *operational congruence*. This concept allows us to identify those modules which have the same operational behavior in any $\oplus$-context, (this is why it is actually a congruence relation, wrt the $\oplus$ operator).

First, we extend the equivalence $\simeq$ to derivations.

Definition 4.1 Let P, P' be two programs, $\xi : c \Box \tilde{C} \overset{P}{\rightsquigarrow} b \Box \tilde{B}$ and $\xi' : c \Box \tilde{C} \overset{P'}{\rightsquigarrow} b' \Box \tilde{B}'$ be two derivations starting in the same goal. Let also $\tilde{x} = Var(c \Box \tilde{C})$. We say that

$$\xi \text{ is } \textit{similar to } \xi',\ \xi \simeq \xi',$$

iff $q(\tilde{x}) \leftarrow b \Box \tilde{B} \simeq q(\tilde{x}) \leftarrow b' \Box \tilde{B}'$, where q is any (dummy) predicate symbol[4]. □

This concept allows us to give the definition of operational congruence. Recall that a refutation is a derivation that ends in a goal with an empty body.

Definition 4.2 (Operational Congruence) Let M_1 and M_2 be CLP modules that have the same set of open predicates. We say that

$$M_1 \text{ and } M_2 \text{ are } \textit{operationally congruent}, M_1 \approx_{\mathcal{O}} M_2,$$

iff, for every module N such that $M_1 \oplus N$ and $M_2 \oplus N$ are defined, we have that for each refutation in $M_1 \oplus N$ there exists a similar refutation in $M_2 \oplus N$ and vice-versa. □

Accordingly, we say that a transformation is *operationally* (totally) *correct* iff it maps modules into operationally congruent ones.

In order to give the applicability conditions for the replacement operation, we start with requirement (i): we want the replacing conjunction to be equivalent to the replaced one. To this end, we provide the following definition of query's equivalence. Here and in the following we say that a derivation ξ is *renamed apart wrt a set of variable* $\tilde{x}$ if all the clauses used in ξ are variable disjoint with $\tilde{x}$.

Definition 4.3 (Query's operational equivalence) Let $M = \langle P, \pi \rangle$ be a module, $c_1 \Box \tilde{C}_1$ and $c_2 \Box \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

$$c_1 \Box \tilde{C}_1 \text{ is } \textit{$\mathcal{O}$-equivalent} \text{ to } c_2 \Box \tilde{C}_2 \textit{ under } \tilde{x} \text{ in } M$$

iff for each π-derivation $c_i \Box \tilde{C}_i \overset{P}{\rightsquigarrow} b_i \Box \tilde{B}_i$, renamed apart wrt $\tilde{x}$, there exists a derivation $c_j \Box \tilde{C}_j \overset{P}{\rightsquigarrow} b_j \Box \tilde{B}_j$, renamed apart wrt $\tilde{x}$ such that $q(\tilde{x}) \leftarrow b_i \Box \tilde{B}_i \simeq q(\tilde{x}) \leftarrow b_j \Box \tilde{B}_j$, where $i, j \in [1,2]$, $i \neq j$ and q is any (dummy) predicate symbol[5]. □

[4] We use the notation based on q as a shorthand: indeed, according to the definition of $\simeq$, this means that for for any $\mathcal{D}$-solution ϑ of b there exists a $\mathcal{D}$-solution ϑ' of b' such that ϑ and ϑ' coincide on the set $\tilde{x}$ and the multisets $\tilde{B}\vartheta$ and $\tilde{B}'\vartheta'$ are equal, and vice-versa.

[5] The condition on clauses used in the derivation is needed to avoid variable name clashes.

The idea behind the above definition, and which distinguishes it from all the previous approaches, is that in a modular context we cannot just refer to refutations, 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. Notice that the larger is the set of open predicates we consider, the stronger becomes the definition of equivalence. Indeed, having more open predicates implies that the derivations we consider are more likely to be influenced by the adjoining of external definitions.

As we informally mentioned in the introduction, when we replace $c \Box \tilde{C}$ by $d \Box \tilde{D}$ in the clause $cl : A \leftarrow c \Box \tilde{C}, \tilde{E}$, our first requirement will be the equivalence of $c \Box \tilde{C}$ and $d \Box \tilde{D}$ under $Var(A, \tilde{E})$ in M. It can be shown that this requirement alone is sufficient to guarantee the *partial* operational correctness of the operation, i.e. that (in any context) each computation that can be performed in the transformed module can also be performed in the initial one. However, this may not be enough to obtain total correctness, as there may be computations which can be done in the original, but not in the transformed program. In fact, when the replacing conjunction depends on the modified clause the replacement can introduce a loop thus affecting the total correctness. This is shown by the following classical counter-example.

Example 4.4 Let $\langle P, \emptyset \rangle$ be the module consisting of the following clauses.

```
cl: q ←r.
    r.
```

In this case both `q` and `r` succeed with empty computed answer, so they they are actually equivalent to each other (under any set of variables). However, if we replace `r` with `q` in the body of *cl* we obtain

```
cl':q ←q.
    r.
```

which is by no means congruent to the previous module. In fact we have introduced a loop and `p` and `q` do not succeed any longer. □

Now we propose two methods for guaranteeing that no "fatal" loops are introduced. These methods formalize the requirement (ii) we mentioned in the introduction. The first one is the most complex but in our opinion is also the most useful for program's optimization. It is based on the following definition. Recall that if ξ is a derivation, then $|\xi|$ denotes its length (i.e. the number of resolution steps in it).

Definition 4.5 (Not Slower) Let $M = \langle P, \pi \rangle$ be a module, $c_1 \Box \tilde{C}_1$ and $c_2 \Box \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

$$c_2 \Box \tilde{C}_2 \text{ is } \textit{$\mathcal{O}$-not-slower} \text{ than } c_1 \Box \tilde{C}_1 \textit{ under } \tilde{x} \text{ in } M$$

iff for each π-derivation $\xi_1 : c_1 \Box \tilde{C}_1 \overset{P}{\rightsquigarrow} b_1 \Box \tilde{B}_1$, renamed apart wrt $\tilde{x}$, there exists a derivation $\xi_2 : c_2 \Box \tilde{C}_2 \overset{P}{\rightsquigarrow} b_2 \Box \tilde{B}_2$, renamed apart wrt $\tilde{x}$ such that $|\xi_2| \leq |\xi_1|$ and that $q(\tilde{x}) \leftarrow b_1 \Box \tilde{B}_1 \simeq q(\tilde{x}) \leftarrow b_2 \Box \tilde{B}_2$, where q is any (dummy) predicate symbol[6]. □

[6] Again, the condition on clauses used in the derivation is needed to avoid variable name clashes.

We are now ready to state our first result on total correctness.

Theorem 4.6 (Correctness I) Let $cl: A \leftarrow c \,\square\, \tilde{C}, \tilde{E}$ be a clause in the module $M: \langle P, \pi \rangle$ and $M': \langle P', \pi \rangle$ be the result of replacing $c \,\square\, \tilde{C}$ by $d \,\square\, \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl': A \leftarrow d \,\square\, \tilde{D}, \tilde{E}\}$. If

- $d \,\square\, \tilde{D}$ is *$\mathcal{O}$-equivalent to* $c \,\square\, \tilde{C}$ and
- $d \,\square\, \tilde{D}$ is *$\mathcal{O}$-not-slower than* $c \,\square\, \tilde{C}$ under $Var(A, \tilde{E})$ in M

then $M \approx_{\mathcal{O}} M'$. □

Notice that in the above Theorem we assume that when we perform the replacement, then we always substitute the whole constraint of the clause with a new one. This is obviously no restriction: if in the clause $A \leftarrow b \wedge c \,\square\, \tilde{C}, \tilde{E}$ we want to replace $c \,\square\, \tilde{C}$ with $d \,\square\, \tilde{D}$, then we can always say that we are actually replacing $b \wedge c \,\square\, \tilde{C}$ with $b \wedge d \,\square\, \tilde{D}$, in fact if the conditions of the above Theorem are satisfied in the first case, they are also satisfied in the latter.

Notice also that $d \,\square\, \tilde{D}$ is (operationally) not-slower than $c \,\square\, \tilde{C}$ in M if computing an answer for $d \,\square\, \tilde{D}$ in M, under any $\oplus$-context, never requires more iterations that computing the corresponding answer for $c \,\square\, \tilde{C}$. Clearly, this means that the definition of $d \,\square\, \tilde{D}$ is at least as efficient as the one of $c \,\square\, \tilde{C}$. Therefore, the requirement of the above theorem, namely that the replacing conjunction has to be not-slower than the replaced one, fits well in a context where transformation operations are intended to increase the performances of programs.

Since in our example we used also the unfolding operation, in order to state the correctness of the overall transformation we need also the following.

Proposition 4.7 [10] Let $M: \langle P, \pi \rangle$ be a module, cl be a clause in P and let P' be the result of unfolding the atom H in cl in P. If $Pred(H) \notin \pi$ then $M \approx_{\mathcal{O}} M'$, where $M': \langle P', \pi \rangle$. □

So unfolding is correct also in the modular setting, as long as this operation is not applied to open atoms.

Let us now go back to Example 2.2 and consider the first replacement we performed. This operation was the one that allowed us to introduce recursion in the definition of `sum_len(Rest,Rates,Sum',Len')` and therefore it constituted the crucial optimization step. We show now that in that case the conditions of Theorem 4.6 were satisfied. For this we can use the following proposition.

Proposition 4.8 Let $cl: H \leftarrow b \,\square\, \tilde{B}$ be the unique clause which defines $Pred(H)$ in the module $M: \langle P, \pi \rangle$ and assume $Pred(H) \notin \pi$. Then $true \,\square\, H$ is operationally equivalent to $b \,\square\, \tilde{B}$ under $Var(H)$ in M. Moreover, if $M': \langle P', \pi \rangle$ is the module obtained by unfolding some atoms $A_1, \ldots, A_n$ in the body of cl such that $Pred(A_i) \notin \pi$ for all $i \in [1, m]$, then $true \,\square\, H$ is operationally not-slower than $b \,\square\, \tilde{B}$ under $Var(H)$ in M'. □

Previous proposition shows also that the applicability conditions given in Theorem 4.6 allow the replacement to mimic, to a large extent, the unfold/fold transformation as defined in [29].

Example 2.2 (Part 2) From the correctness of the unfolding operation it follows that `AVERAGE` $\approx$ $\mathtt{AV}_1$. Because of the above Proposition, denoting by c_4 the constraint which appear in the clause `c4`, we have that

$c_4 \,\square$ `sum_len(Rest,Rates,Sum',Len')`

is $\mathcal{O}$-equivalent to and $\mathcal{O}$-not-slower than

$c_4 \,\square$ `exchange_rates(Rates), sum(Rest, Rates, Sum'), len(Rest, Len')`

under { `Curr, Amount,Rest,Rates, Sum, Len` } in $\mathtt{AV}_1$. Therefore the conditions of Theorem 4.6 are satisfied and `AVERAGE` $\approx_{\mathcal{O}}$ $\mathtt{AV}_2$ holds.

The second and maybe easiest method we propose for ensuring that no fatal loops are introduced by the replacement, is to require that no predicate symbol in $\tilde{D}$ depends on the predicate symbol in the head of cl. In this case no loop can be introduced at all. For this we need the following formal notion of dependency.

Definition 4.9 (Dependency) Let P be a program, p and q be relations. We say that p *refers to* q in P iff there is a clause in P with p in the head and q in the body. We say that p *depends on* q in P iff (p, q) is in the reflexive and transitive closure of the relation *refers to*. □

We can now state our second result on (total) correctness.

Theorem 4.10 (Correctness II) Let $cl: A \leftarrow c \,\square\, \tilde{C}, \tilde{E}$ be a clause of the module $M: \langle P, \pi \rangle$, and $M': \langle P', \pi \rangle$ be the result of replacing $c \,\square\, \tilde{C}$ by $d \,\square\, \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl': A \leftarrow d \,\square\, \tilde{D}, \tilde{E}\}$. If

- $c \,\square\, \tilde{C}$ is *$\mathcal{O}$-equivalent to* $d \,\square\, \tilde{D}$ under $Var(A, \tilde{E})$ in M and
- no predicate in $\tilde{D}$ *depends* on $Pred(A)$ in M

then $M \approx_{\mathcal{O}} M'$. □

Example 2.2 (Part 3) Consider now the second replacement we performed and let us denote by c_1 the constraint in clause `c1`. As before, because of Proposition 4.8 we have that `exchange_rates(Rates), sum(Xs, Rates, Sum), len(Xs, Len)` is $\mathcal{O}$-equivalent to `sum_len(Xs, Rates, Sum, Len)` under { `Xs, Rates, Sum, Len` } in $\mathtt{AV}_1$. Since the correctness of the the first replacement implies that $\mathtt{AV}_1 \approx_{\mathcal{O}} \mathtt{AV}_2$, we have that the previous equivalence holds also in $\mathtt{AV}_2$. This implies that c_1 `□exchange_rates(Rates), sum(Xs, Rates, Sum), len(Xs, Len)` is $\mathcal{O}$-equivalent to c_1 `□sum_len(Xs, Rates, Sum, Len)` under {`List, Av`} in $\mathtt{AV}_2$.

Moreover, `sum_len` does not depend on clause `c1` in $\mathtt{AV}_2$. Therefore, from Theorem 4.10 it follows `AVERAGE` $\approx_{\mathcal{O}}$ $\mathtt{AV}_3$, that is the transformation is correct. □

5 Correctness wrt other congruences

In some cases one can be interested in preserving other kind of properties of modules rather than (all) their answer constraints. Indeed in the literature, together with the answer constraint semantics [12], we find two other semantics for CLP without negation. One is the so-called $\mathcal{C}$-semantics

which was defined for pure logic programs [8, 11] and then adapted to CLP (specifically for program's transformation) in [1] by using an operational definition. The $\mathcal{C}$-semantics characterizes the most general answer constraints of a CLP program. The second, and more notable one, is the least model semantics (on the relevant algebraic structure $\mathcal{D}$) [15]. This semantics is the CLP counterpart of the least Herbrand model and it is commonly considered the standard declarative semantics for CLP.

In this Section we consider the congruences induced by these two semantics. We show that we can easily adapt to both the contexts the applicability conditions used in Theorems 4.6 and 4.10. Moreover, since these semantics induce congruences which are weaker than the operational one, the resulting applicability conditions are weaker than the previous ones, thus allowing more optimizations on the modules.

In order to define formally the new congruences we first need the following.

Definition 5.1 Let P, P' be two programs, $\xi : \ c \Box \tilde{C} \overset{P}{\rightsquigarrow} b \Box \tilde{B}$ and $\xi' : \ c \Box \tilde{C} \overset{P'}{\rightsquigarrow} b' \Box \tilde{B}'$ be two derivations starting in the same goal. Let also $\tilde{x} = Var(c \Box \tilde{C})$. We say that

$$\xi' \text{ is } \textit{more general than } \xi,\ \xi \preceq \xi',$$

iff $\mathcal{D} \models \exists_{-\tilde{x}}\, b \Box \tilde{B} \rightarrow \exists_{-\tilde{x}}\, b' \Box \tilde{B}'$. □

Notice that $\mathcal{D} \models \exists_{-\tilde{x}}\, b \Box \tilde{B} \rightarrow \exists_{-\tilde{x}}\, b' \Box \tilde{B}'$ holds iff, for each solution θ of b, there exists a solution θ' of b' such that θ and θ' agree on the variables $\tilde{x}$ and each element in the conjunction $\tilde{B}'\theta'$ is also an element of the conjunction $\tilde{B}\theta$. It is also worth noticing that $\preceq$ does not represent "one side" of $\simeq$, since we can have that $\xi \preceq \xi'$, $\xi' \preceq \xi$ and still $\xi \not\simeq \xi'$. This is due to the fact that in the definition of $\simeq$ the goals have to be considered as multisets, while here considering them as sets is sufficient. For instance, this is the case when we consider the derivations $\xi : \ p(x) \rightsquigarrow x = y \Box q(y), q(y).$ and $\xi' : \ p(x) \rightsquigarrow x = y \Box q(y)$.

We can now define the $\mathcal{C}$- and the $\mathcal{M}$-congruence as follows.

Definition 5.2 ($\mathcal{C}$- and $\mathcal{M}$-congruence) Let M_1 and M_2 be CLP modules that have the same set of open predicates. We say that

$$M_1 \text{ and } M_2 \text{ are } \mathcal{C}\textit{-congruent},\ M_1 \approx_{\mathcal{C}} M_2,$$

iff, for every module N such that $M_1 \oplus N$ and $M_2 \oplus N$ are defined, we have that for each refutation in $M_1 \oplus N$ there exists a more general refutation in $M_2 \oplus N$ and vice-versa. Moreover, we say that

$$M_1 \text{ and } M_2 \text{ are } \mathcal{M}\textit{-congruent},\ M_1 \approx_{\mathcal{M}} M_2,$$

Iff for every module M such that $M_1 \oplus M$ and $M_2 \oplus M$ are defined, we have that $M_1 \oplus M$ and $M_2 \oplus M$ have the same least $\mathcal{D}$-model. □

It is not difficult to prove that the operational congruence is stronger than the $\mathcal{C}$-congruence, which in turn is stronger than the $\mathcal{M}$-congruence. To clarify the difference among the three kind of relations let us consider the following simple modules where we assume the set of open atoms to be empty.

M_1 :	M_2 :	M_3 :
`p(X).`	`p(X).`	`p(X) ← X = Y+1 □ p(Y).`
	`p(0).`	`p(0).`

It is easy to check that no one of these three modules is operationally congruent to another. On the other hand M_1 is $\mathcal{C}$-congruent (and therefore also $\mathcal{M}$-congruent) to M_2, while it is not $\mathcal{C}$-congruent to M_3. Finally, if the structure we refer to is the one whose domain contains only the set of natural numbers, then M_3 is $\mathcal{M}$-congruent to both M_1 and M_2.

Note 5.3 For the reader familiar with the original definition of the $\mathcal{C}$-semantics [8] some explanations are in order here. The $\mathcal{C}$-semantics of a pure logic program P is defined indifferently as

(a) the set of atomic logical consequences of P, or

(b) the set of *most general* answers computed by P.

It is also proven [21] that, it the underlying language is infinite, then two pure logic programs have the same $\mathcal{C}$ semantics iff they have the same least Herbrand model.

Now, the CLP counterpart of the $\mathcal{C}$-semantics is defined in [1] just as the counterpart of (b) above. The fact is that, for CLP programs the statements (a) and (b) are not equivalent to each other. This is shown for example by the programs

```
p(X) ← X = a  ∨ X = b.
```

and

```
p(X) ← X = a.
p(X) ← X = b.
```

Moreover, since in the CLP context we need the domain $\mathcal{D}$ for evaluating the constraint, it makes little sense talking about the *logical consequences of* P (which are the formulae ϕ such that $P \models \phi$). On the other hand, it is meaningful talk about the logical consequences of P "*under* $\mathcal{D}$", by this we mean the set of formulae ϕ such that $\mathcal{D} \models P \rightarrow \phi$. Now, since the domain of $\mathcal{D}$ determines the universe of our interpretations and models, we have that two CLP programs have the same "set of atomic[7] logical consequences under $\mathcal{D}$" iff they have the same least $\mathcal{D}$-model, but this does not imply that they have the same most general answers. Indeed, if we consider the programs in M_1 and M_3 above, we have that, if $\mathcal{D}$ is the usual additive structure on the set of natural numbers, M_1 and M_3 (seen as programs) have the same least $\mathcal{D}$ models, therefore the same set of logical consequences "*under* $\mathcal{D}$", but they do not have the same set of most general answers. Notice that this is the case even though our structure contains the infinite set of constants corresponding to the natural numbers. □

As before, we say that a transformation is (totally) $\mathcal{C}$-*correct* (resp. $\mathcal{M}$-correct) iff it maps modules into $\mathcal{C}$- (resp. $\mathcal{M}$-) congruent ones. Of course, the weaker the congruence we consider, the more operations we are going to be allowed on the modules, but also the less "faithful" will be the resulting module. For example, a typical operation which is $\mathcal{C}$-correct but possibly not operationally correct is the elimination of duplicated atoms in the body of the clause (see later).

[7] Here we can consider atomic also a formula of the form $p(\tilde{X}) \leftarrow c$ where c is a constraint.

5.1 Correctness wrt $\mathcal{C}$-congruence

In this Subsection we provide the applicability conditions for the replacement operation in the case we refer to the $\mathcal{C}$-congruence. More precisely, we are going to reformulate appropriately Theorems 4.6 and 4.10. This provides a generalization of the result on the correctness of the replacement operation given in [1].

First, we restate the Definitions 4.3 and 4.5 to adapt them to the new context.

Definition 5.4 Let $M = \langle P, \pi \rangle$ be a module, $c_1 \square \tilde{C}_1$ and $c_2 \square \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

$$c_2 \square \tilde{C}_2 \text{ is } \mathcal{C}\textit{-equivalent} \text{ to } c_1 \square \tilde{C}_1 \textit{ under } \tilde{x} \text{ in } M$$

iff for each π-derivation $\xi_i : c_i \square \tilde{C}_i \overset{P}{\rightsquigarrow} b_i \square \tilde{B}_i$, there exists a π-derivation $\xi_j : c_j \square \tilde{C}_j \overset{P}{\rightsquigarrow} b_j \square \tilde{B}_j$ such that $\mathcal{D} \models \exists_{-\tilde{x}}\, b_i \square \tilde{B}_i \rightarrow \exists_{-\tilde{x}}\, b_j \square \tilde{B}_j$ $(i \neq j,\ i,j \in [1,2])$. Moreover, we say that

$$c_2 \square \tilde{C}_2 \text{ is } \mathcal{C}\textit{-not-slower} \text{ than } c_1 \square \tilde{C}_1 \textit{ under } \tilde{x} \text{ in } M$$

iff for each π-derivation $\xi_1 : c_1 \square \tilde{C}_1 \overset{P}{\rightsquigarrow} b_1 \square \tilde{B}_1$ there exists a π-derivation $\xi_2 : c_2 \square \tilde{C}_2 \overset{P}{\rightsquigarrow} b_2 \square \tilde{B}_2$ such that $|\xi_2| \leq |\xi_1|$ and $\mathcal{D} \models \exists_{-\tilde{x}}\, b_1 \square \tilde{B}_1 \rightarrow \exists_{-\tilde{x}}\, b_2 \square \tilde{B}_2$.

In this definitions all the derivations are supposed to be renamed apart wrt $\tilde{x}$. □

It is easy to see that the concepts of $\mathcal{C}$-equivalence and of $\mathcal{C}$-not-slower are weaker than their operational counterparts given in Definitions 4.3 and 4.5. Intuitively, the difference in terms of derivations lies in the fact that for the former we want a one-to-one correspondence between all the partial derivations ending with open atoms, while the latter requires this one-to-one correspondence to hold only for the "most general" ones. Now when we refer to the $\mathcal{C}$-congruence we can weaken the hypothesis of Theorems 4.6 and 4.10 by replacing the concepts of *equivalent* and *not-slower* by their $\mathcal{C}$-counterparts. Namely, we have the following.

Theorem 5.5 ($\mathcal{C}$-correctness) Let $cl : A \leftarrow c \square \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi \rangle$, and $M' : \langle P', \pi \rangle$ be the result of replacing $c \square \tilde{C}$ by $d \square \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \square \tilde{D}, \tilde{E}\}$. If

- $d \square \tilde{D}$ is $\mathcal{C}$-*equivalent* to $c \square \tilde{C}$ under $Var(A, \tilde{E})$ in M and
  - either $d \square \tilde{D}$ is $\mathcal{C}$-*not slower* than $c \square \tilde{C}$ under $Var(A, \tilde{E})$ in M,
  - or no predicate in $\tilde{D}$ depends on $Pred(A)$ in M,

then $M \approx_{\mathcal{C}} M'$ □

This result generalizes Proposition 4.6 in [1]. In fact, it is easy to check that when the hypothesis of that proposition are satisfied then the replacing and the replaced conjunction are always $\mathcal{C}$-equivalent to each other and that the replacing conjunction is always not-slower than the replaced one (under an appropriate set of variables).

The applicability conditions in the previous Theorem are weaker than the ones in Theorems 4.6 and 4.10. This reflects the fact that some replacement operations which are correct wrt $\mathcal{C}$ congruence may not be so wrt the operational one. A typical example of a replacement operation which always satisfies the hypothesis of Theorem 5.5, but which is possibly not operationally correct, and therefore does not satisfy the hypothesis of Theorems 4.6 and 4.10, is the elimination of duplicate atoms in the body of a clause. Indeed, consider a program M consisting the following clause

```
c1: p(X,Y) ← q(X,Y), q(X,Y).
    q(a,W).
    q(W,b).
```

If we eliminate one of the atoms in the body of `c1` then we loose the answer { `X=a` ∧ `Y=b`} to the query `p(X,Y)`. For this reason the operation is not operationally correct. However it is $\mathcal{C}$-correct, in fact the most "general" answers to the query `p(X,Y)` (which are { `X=a`} and { `Y=b`}) are not lost.

5.2 Correctness wrt $\mathcal{M}$-congruence

In this subsection we give the $\mathcal{M}$-counterpart of the results stated in the previous one. We formulate the applicability conditions for the replacement operation for the case in which we want to preserve the $\mathcal{M}$-congruence. First, we need to adapt to the new context the concepts of equivalent and of not-slower query.

Definition 5.6 Let $M = \langle P, \pi \rangle$ be a module, $c_1 \square \tilde{C}_1$ and $c_2 \square \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

$$c_1 \square \tilde{C}_1 \text{ is } \mathcal{M}\textit{-equivalent} \text{ to } c_2 \square \tilde{C}_2 \textit{ under } \tilde{x} \text{ in } M$$

iff for each π-derivation $c_i \square \tilde{C}_i \overset{P}{\rightsquigarrow} b_i \square \tilde{B}_i$ and each solution ϑ_i of b_i, there exists a derivation $c_j \square \tilde{C}_j \overset{P}{\rightsquigarrow} b_j \square \tilde{B}_j$ $(i \neq j)$ and a solution ϑ_j of b_j such that $\mathcal{D} \models \tilde{B}_i\vartheta_i \rightarrow \tilde{B}_j\vartheta_j$ and $\tilde{x}\vartheta_1 = \tilde{x}\vartheta_2$. Moreover, we say that

$$c_2 \square \tilde{C}_2 \text{ is } \mathcal{M}\textit{-not-slower} \text{ than } c_1 \square \tilde{C}_1 \textit{ under } \tilde{x} \text{ in } M$$

iff for each π-derivation $\xi_1 : c_1 \square \tilde{C}_1 \overset{P}{\rightsquigarrow} b_1 \square \tilde{B}_1$ and for each solution ϑ_1 of b_1, there exists a derivation $\xi_2 : c_2 \square \tilde{C}_2 \overset{P}{\rightsquigarrow} b_2 \square \tilde{B}_2$ and a solution ϑ_2 of b_2 such that $|\xi_2| \leq |\xi_1|$, $\mathcal{D} \models \tilde{B}_1\vartheta_1 \rightarrow \tilde{B}_2\vartheta_2$ and $\tilde{x}\vartheta_1 = \tilde{x}\vartheta_2$.

Again, all the considered derivations here considered are supposed to be renamed apart wrt $\tilde{x}$. □

The $\mathcal{M}$-equivalence is the weakest of the three congruence we have introduced. This is due to the fact that it checks only the "ground" derivations.

Theorem 5.5 can be immediately restated for the case of the $\mathcal{M}$-congruence as follows.

Theorem 5.7 ($\mathcal{M}$-correctness) Let $cl : A \leftarrow c \square \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi \rangle$, and $M' : \langle P', \pi \rangle$ be the result of replacing $c \square \tilde{C}$ by $d \square \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \square \tilde{D}, \tilde{E}\}$. If

- If $d \square \tilde{D}$ is $\mathcal{M}$-*equivalent* $c \square \tilde{C}$ under $Var(A, \tilde{E})$ in M and
  - either $d \square \tilde{D}$ is $\mathcal{M}$-*not slower* than $c \square \tilde{C}$ under $Var(A, \tilde{E})$ in M,
  - or no predicate in $\tilde{D}$ depends on $Pred(A)$ in M,

then $M \approx_{\mathcal{M}} M'$. □

The non-modular case

From the definitions it is clear that the smaller is the set of open predicates, the weaker become the applicability conditions needed to ensure correctness of replacement, for all the three congruences considered. In particular, if we assume that the set of open predicates is empty we obtain (much) weaker conditions adequate for a non-modular setting in which programs are viewed as stand-alone units (or, equivalently, only compositions of predicate disjoint modules are allowed). Therefore, the non-modular case can be naturally regarded as a particular instance of the general one: the correctness results in this specific case can be obtained by just setting $\pi = \emptyset$ in Theorems 4.6, 4.10, 5.5, and 5.7.

6 Conclusions

We have investigated optimizations of CLP modules based on the replacement transformation.

Our results extend previous ones in the field of transformations for logic programs in that we have defined applicability conditions for replacement which guarantee that the original and the transformed module are semantically equivalent under any $\oplus$-context. These conditions have been instantiated to consider three different semantic notions. To the best of our knowledge, the only other papers which consider transformations for modular logic programs are [22, 10]. However, our notion of module composition is more general than the one considered in [22], since the latter assumes that each predicate is defined within a single module and does not allow mutual recursion among modules. On the other hand, [10] considers only fold/unfold systems which, as previously mentioned, are quite different from those based on replacement.

Also when restricting to the non-modular setting our results generalizes previous ones in the field. More specifically, we extend those in the original paper [29] (which introduced replacement) by considering CLP equipped with three different semantics (the method in [29] is devised for pure logic programs and the least Herbrand model semantics) and by providing a different applicability condition to avoid loops (the one relative to condition (b) above). We also believe that our setting is more suitable for practical applications, as it does not rely on bottom-up concepts as [29] does.

In the literature we find only two other papers which investigated replacement for CLP: one by Maher ([22]) which refers (when restricted to programs without negation) to the $\mathcal{M}$-congruence, and one by Bensaou and Guessarian ([1]) which considers the $\mathcal{C}$-semantics.

We extend the part of the results of Maher relative to definite programs[8] by allowing the replacement to introduce recursion in the predicate's definition and by weakening the requirements of the applicability conditions: each time that the requirements of [22] are satisfied also the hypothesis of of Theorem 5.7 are satisfied.

On the other hand, Theorem 5.5 provides us with a generalization of Proposition 4.6 in [1]: each time that the applicability conditions given in that paper are satisfied we can also apply the replacement. The converse is not true, even in the non-modular case. For instance the replacements performed in Example 2.2 are not feasible using the tools of [1]. As previously said, the paper by Bensaou and Guessarian does not considers modules, so their replacement operation in general does not transform a program into a congruent one.

Finally, we have proved correctness wrt the operational congruence which, we believe, is the natural notion to be considered for practical applications and which was not taken into account in [22, 1].

[8] Although Maher [22] considers also negated atoms in the body of the clauses, which are beyond the scope of this paper.

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. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[3] A. Bossi and N. Cocco. Basic Transformation Operations which preserve Computed Answer Substitutions of Logic Programs. *Journal of Logic Programming*, 16(1&2):47–87, 1993.

[4] A. Bossi, N. Cocco, and S. Etalle. On Safe Folding. In M. Bruynooghe and M. Wirsing, editors, *Programming Language Implementation and Logic Programming - Proceedings PLILP'92*, volume 631 of *Lecture Notes in Computer Science*, pages 172–186. Springer-Verlag, 1992.

[5] A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. Technical Report CS-R9357, CWI, Centre for Mathematics and Computer Science, Amsterdam, The Netherlands, August 1993. To appear in Journal of Logic and Computation, 1995. Available via ftp (or xmcsaic) at ftp.cwi.nl, in /pub/etalle.

[6] 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.

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

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

[9] J. Cook and J.P. Gallagher. A transformation system for definite programs based on termination analisys. In F. Turini, editor, *Proc. Fourth Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1994.

[10] S. Etalle and M. Gabbrielli. Modular Transformations of CLP Programs. In L. Sterling, editor, *Proc. Twelfth Int'l Conf. on Logic Programming*, 1995. Extended version available as CWI Technical Report, 1995.

[11] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. Declarative modelling of the operational behavior of logic languages. *Theoretical Computer Science*, 69(3):289–318, 1989.

[12] M. Gabbrielli and G. Levi. Modeling Answer Constraints in Constraint Logic Programs. In K. Furukawa, editor, *Proc. Eighth Int'l Conf. on Logic Programming*, pages 238– 252. The MIT Press, Cambridge, Mass., 1991.

[13] P.A. Gardner and J.C. Shepherdson. Unfold/fold transformations of logic programs. In J-L Lassez and G. Plotkin, editors, *Computational Logic: Essays in Honor of Alan Robinson*. MIT Press, 1991.

[14] 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.

[15] 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.

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

[17] Joxan Jaffar, Spiro Michayov, Peter J. Stuckey, and Roland H. C. Yap. The CLP($\mathcal{R}$) language and system. *TOPLAS: ACM Transactions on Programming Languages and Systems*, 14(3):339–395, July 1992.

[18] Niels Jørgensen, Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, editors, *ILPS'91: Proceedings of the International Logic Programming Symposium*, pages 420–434, San Diego, October 1991. MIT Press.

[19] C. Lassez, K. McAloon, and R. Yap. Constraint Logic Programming and Option Trading. *IEEE Expert*, 2(3), 1987.

[20] M.J. Maher. Correctness of a logic program transformation system. IBM Research Report RC13496, T.J. Watson Research Center, 1987.

[21] M.J. Maher. Equivalences of logic programs. In editor J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 627–658. Morgan Kaufmann, 1988.

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

[23] Kim Marriott and Harald Søndergaard. Analysis of constraint logic programs. In Saumya Debray and Manuel Hermenegildo, editors, *NACLP'90: Proceedings North American Conference on Logic Programming*, pages 531–547, Austin, 1990. MIT Press.

[24] Kim Marriott and Peter J. Stuckey. The 3 r's of optimizing constraint logic programs: Refinement, removal and reordering. In *POPL'93: Proceedings ACM SIGPLAN Symposium on Principles of Programming Languages*, Charleston, January 1993.

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

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

[27] M. Proietti and A. Pettorossi. Total correctness of a goal replacement rule based of the unfold/fold proof method. In M. Alpuente, editor, *Proc. 1994 Joint Conference on Declarative Progrmming GULP-PRODE'94*, 1994.

[28] T. Sato. Equivalence-preserving first-order unfold/fold transformation system. *Theoretical Computer Science*, 105(1):57–84, 1992.

[29] 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.

[30] Pascal Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, Cambridge, MA, 1989.