ELSEVIER

Theoretical Computer Science 166 (1996) 101–146

Theoretical Computer Science

Transformations of CLP modules ☆

Sandro Etalle [a], Maurizio Gabbrielli [b,*]

[a] *D.I.S.I, Università di Genova, Via Dodecaneso 35, 16146 Genova, Italy*
[b] *Dipartimento di Informatica, Università di Pisa, Corso Italia 40, 56125 Pisa, Italy*

Received January 1995; revised August 1995
Communicated by G. Levi

Abstract

We propose a transformation system for Constraint Logic Programming (CLP) programs and modules. The framework is inspired by the one of Tamaki and Sato (1984) for pure logic programs. However, 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 behaviour, 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. This result is proved by first introducing a new formulation, in terms of trees, of a resultants semantics for CLP. As corollaries we obtain the correctness of both the modular and the nonmodular 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 [9] for transforming declaratively clear functional programs into equivalent, more complex and efficient ones, and then adapted to logic programs both for program synthesis [10, 17], and for program specialization and optimization [25]. Soon later, Tamaki and Sato [37] proposed an elegant framework for the transformation of logic programs based

☆ This work has been carried out while both the authors were visiting the Centrum voor Wiskunde en Informatica (CWI), Amsterdam, The Netherlands. 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. A preliminary, shorter version of this paper appeared as [11].

* Corresponding author. E-mail: gabbri@di.unipi.it.

SSDI 0304-3975(95)00148-4

on unfold/fold rules. Their system was proven to be correct w.r.t. the least Herbrand model semantics [37] and the computed answer substitution semantics [24].

The system was then extended by Seki [34] to logic programs with negation, in particular he provided new, more restrictive applicability conditions which guarantee that the system preserves also the finite failure set and the perfect model semantics of stratified programs. Since then serious research effort has been devoted to proving its correctness w.r.t. the various semantics available for normal programs. For instance, the new system was then adapted by Sato to full first-order programs [33]. Related work has been done by Maher [29], Gardner and Shepherdson [16], Aravidan and Dung [2], Seki [35], Bossi and Cocco [5] and Bensaou and Guessarian [3]. Among these papers only [3, 29] treated the case of Constrain Logic Programming. We defer to Section 7 a comparison of these approaches with ours.

All the (unfold/fold) transformation systems proposed so far for logic programming and for CLP, with the only exception of [29], assume that the entire 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 [8] 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 which essentially consists of union of program clauses.

Now, a transformation system for modules requires ad-hoc applicability conditions: when we transform P into P' we do not just want P and P' to have the same observable behaviour (e.g. the same answer constraints); we want them to have the same observable behaviour *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 [37] 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. In this first part we also define new applicability conditions for the folding operation which avoid the use of substitutions and which are simpler than 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 behaviour in terms of answer constraints.

This result is proved by using a new formulation, in terms of trees, of a *resultants semantics* which models answer constraints and is compositional w.r.t. union of programs. From a particular case of the main theorem it follows that the transformation system for non-modular programs also preserves the computational behaviour of programs. Finally, since the least model (on the relevant algebraic structure) can be seen as an abstraction of the compositional semantics, we obtain as a corollary that the least model is also preserved.

The paper is organized as follows. The next section contains some preliminaries on CLP programs. In Section 3 we introduce the notion of module and we formalize the resultants semantics for CLP by using trees. Section 4 provides the definition of the transformation system. In Section 5 we add the applicability conditions needed to obtain a modular system and we state the main correctness result. In Section 6 we show that the system of Tamaki–Sato can be embedded into ours. As a consequence, the conditions given in Section 5 can also be added to those defined in [37] in order to obtain a modular unfold/fold system for pure logic programs. Section 7 concludes by comparing our results with those contained in two related works. The proof of the main technical result of the paper is deferred to the Appendix.

2. Preliminaries: CLP programs

The *Constraint Logic Programming* paradigm CLP(X) (CLP for short) has been proposed by Jaffar and Lassez [18, 19] in order to integrate a generic computational mechanism based on constraints with the logic programming framework. The advantages of such an integration are several. From a pragmatic point of view, CLP(X) allows one to use a specific constraints domain X and a related constraint solver within the declarative paradigm of logic programming. From the theoretical viewpoint, CLP provides a unified view of several extensions of pure logic programming (e.g. arithmetics, equational programming) within a framework which preserves the existence of equivalent operational, model-theoretic and fixpoint semantics [18]. Indeed, as discussed in [29], most of the results which hold for pure logic programs can be lifted to CLP in a quite straightforward way.

The reader is assumed to be familiar with the terminology and the main results on the semantics of (constraint) logic programs. In this subsection we introduce some notations we will use in the sequel and, for the reader's convenience, we recall some basic notions on constraint logic programs. Lloyd's book and the survey by Apt [1, 28] provide the necessary background material for logic programming theory. For constraint logic programs we refer to the original papers [18, 19] by Jaffar and Lassez and to the recent survey [20] by Jaffar and Maher.

The CLP framework was originally defined using a many-sorted first-order language. In this paper, to keep the notation simple, we consider a one sorted language (the extension of our results to the many sorted case is immediate). We assume programs defined on a signature with predicates Σ consisting of a pair of disjoint sets containing function symbols and predicate symbols. The set of predicate symbols, denoted by Π, is assumed to be partitioned into two disjoint sets: Π_c (containing predicate symbols used for constraints) which contains also the equality symbol "=", and Π_u (containing symbols for user definable predicates). All the following definitions will refer to some given Σ, Π_c and Π_u.

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 $\square$ will often be used instead of "$\wedge$" to denote conjunction.

A *primitive constraint* is an atomic formula $p(t_1,\ldots,t_n)$ where the t_i's are terms (built from Σ and a denumerable set of variables) and $p \in \Pi_c$. A *constraint* is a first order formula built using primitive constraints. A CLP rule is a formula of the form

$$H \leftarrow c \square B_1,\ldots,B_n.$$

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_u only. When the body is empty we will omit the connective $\square$. A *goal* (or query), denoted by $c \square B_1,\ldots,B_n$, is a conjunction of a constraint and atomic formulas as before. A CLP program is a finite set of CLP rules.

The semantics of CLP programs is based on the notion of *structure*. Given a signature with predicates Σ, a Σ-structure (structure for short) $\mathscr{D}$ consists of a set (the domain) D and an assignment that maps function symbols in Σ and predicate symbols in Π_c to functions and relations on D respecting arities.

A $\mathscr{D}$-interpretation is an assignment that maps each predicate symbol in Π_u to a relation on the domain of the structure. A $\mathscr{D}$-interpretation I is called a *$\mathscr{D}$-model* of a CLP program P if all the clauses of P evaluate to true under the assignment of relations and function provided by I and by $\mathscr{D}$. We recall that there exists [19] the least $\mathscr{D}$-model of a program P which is the natural CLP counterpart of the least Herbrand model for logic programs.

Given a structure $\mathscr{D}$ and a constraint c, $\mathscr{D} \models c$ denotes that c is true under the interpretation for constraints provided by $\mathscr{D}$. Moreover if ϑ is a valuation (i.e. a mapping of variables on the domain D), and $\mathscr{D} \models c\vartheta$ holds, then ϑ is called a *$\mathscr{D}$-solution* of c ($c\vartheta$ denotes the application of ϑ to the variables in c).

Here and in the sequel, given the atoms A, H, we write $A = H$ as a shorthand for:
- $a_1 = t_1 \wedge \cdots \wedge a_n = t_n$, if, for some predicate symbol p and natural n, $A \equiv p(a_1,\ldots,a_n)$ and $H \equiv p(t_1,\ldots,t_n)$ (where $\equiv$ denotes syntactic equality)
- *false*, otherwise.

This notation readily extends to conjunctions of atoms. We also find convenient to use the notation $\exists_{-\tilde{x}}\ \phi$ from [20] to denote the existential closure of the formula ϕ *except* for the variables $\tilde{x}$ which remain unquantified.

The operational model of CLP is obtained from SLD resolution by simply substituting $\mathscr{D}$-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 the goal

$$c_0 \wedge (B_i = H) \wedge c \square B_1, \ldots, B_{i-1}, \tilde{B}, B_{i+1}, \ldots, B_n$$

provided that 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 $(c_0 \wedge (B_i = H) \wedge c)$ is $\mathscr{D}$-satisfiable, that is, $\mathscr{D} \models \exists\ (c_0 \wedge (B_i = H) \wedge c)$.

A derivation via a selection rule R of a goal G in the program P is a finite or infinite sequence of goals, starting in G, such that every next goal is obtained from the previous one by means of a derivation step where the atom is selected according to R. A derivation is *successful* if it is finite and its last element is a goal of the form c, i.e. consisting only of a constraint. In this case, $\exists_{-Var(G)}\ c$ is called the *answer constraint*.[1] In what follows a derivation of a goal G whose last goal is G_i in the program P will be denoted by

$$G \overset{P}{\rightsquigarrow} G_i.$$

Finally, by naturally extending the usual notion used for pure logic programs, we say that a query $c \square \tilde{C}$ is *an instance* of the query $d \square \tilde{D}$ iff for any solution γ of c there exists a solution δ of d such that $\tilde{C}\gamma \equiv \tilde{D}\delta$.

3. Modular CLP programs

Following the original paper of O'Keefe [31], the approach to modular programming we consider here is based on a *metalinguistic* program composition mechanism. This provides a formal background to the usual software engineering techniques for the incremental development of programs.

Viewing modularity in terms of *metalinguistic* 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, *metalinguistic* 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 [4].

[1] We follow here the more recent terminology used in [20]. In the original papers [18, 19] a derivation step was defined by rewriting in parallel all the atoms of the goal. As far as successful derivation are concerned the two formulations are equivalent. Moreover in [18, 19] the answer constraint was considered c (without quantification).

Here, in order to keep the presentation simple, we follow [6] and say that a module M is a CLP program P together with a set $Op(M)$ of predicate symbols specifying the *open* predicates.

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

The idea underlying the previous definition is that the *open* predicates, specified in $Op(M)$, behave as an interface for composing M with other modules. 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. A typical practical example is a deductive database composed of two modules, in which the first one $\mathscr{I}$ contains the *intensional part* in the form of some *rules* which refer to an unspecified *extensional part*. This latter is defined in the second module $\mathscr{E}$ which contains facts (unit clauses) describing the basic relations. In this case the extensional predicates which are defined in $\mathscr{E}$ are exported to $\mathscr{I}$, which in turn imports them when composing the two parts. Further definitions for the extensional predicates can be incrementally added to the database by adjoining new modules.

To simplify the notation, when no ambiguity arises we will denote by M also the set of clauses P. To compose CLP modules we again follow [6] and use a simple program union operator. We denote by $Pred(E)$ set of predicate symbols which appear in the expression E.

Definition 3.2 (*Module composition*). Let $M = \langle P, Op(M)\rangle$ and $N = \langle Q, Op(N)\rangle$ be modules. We define

$$M \oplus N = \langle P \cup Q, Op(M) \cup Op(N)\rangle$$

provided that $Pred(P) \cap Pred(Q) \subseteq Op(M) \cap Op(N)$ holds. Otherwise $M \oplus N$ is undefined.

So, when composing M and N, 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 our transformation systems, we need to fix the kind of module's (and program's) equivalence that we want to establish between a program and its transformed version.

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 $\mathscr{D}$) for any query. This concept is formalized in the following Definition.

Definition 3.3 (*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 $Q \overset{P_i}{\leadsto} c_i$ then there exists a derivation $Q \overset{P_j}{\leadsto} c_j$ such that $\mathscr{D} \models \exists_{-Var(Q)}\, c_i \leftrightarrow \exists_{-Var(Q)}\, c_j$.

This notion is satisfactory when programs are seen as completely defined units. However, the relation $\approx$ is far too weak when considering modules. For instance, consider the following:

Example 3.4. Consider the modules $M_1 : \langle P_1, \{p\}\rangle$ and $M_2 : \langle P_2, \{p\}\rangle$ where P_1 is

```
q(X) ← true□p(X).
p(X) ← X=a.
```

While P_2 is

```
q(X) ← X=a□p(X).
p(X) ← X=a.
```

It is easy to see that $P_1 \approx P_2$. However, if we compose these two modules with $M : \langle P, \{p\}\rangle$ where P is the program

```
p(X) ← X=b.
```

we have that $M_1 \oplus M$ and $M_2 \oplus M$ have quite different behaviour, in particular $M_1 \oplus M \not\approx M_2 \oplus M$.

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 $\approx$ to obtain a congruence w.r.t. the $\oplus$ operator. Therefore the following.

Definition 3.5 (*Module's congruence*). Let M_1 and M_2 be CLP modules. We say that M_1 is (observationally) *congruent* to M_2,

$$M_1 \approx_c M_2$$

iff $Op(M_1) = Op(M_2)$ and for every module N such that $M_1 \oplus N$ and $M_2 \oplus N$ are defined, $M_1 \oplus N \approx M_2 \oplus N$ holds.

So $M_1 \approx_c M_2$ iff they have the same open predicates and, for any query, they produce the same answer constraints in any $\oplus$-context. By taking N as the empty module we immediately see that if $M_1 \approx_c M_2$ then $M_1 \approx M_2$.

This notion of equivalence and of congruence are used to define the correctness of our transformation system.

Definition 3.6 (*Correctness*). We say that a transformation for CLP programs (modules) is *correct* iff it maps a program (a module) into an $\approx$- ($\approx_c$-) equivalent one.

3.1. A compositional semantics for CLP modules

The correctness proofs for our transformation system will be carried out by showing that the system preserves a semantics (borrowed from [13]) which models answer constraints and is compositional w.r.t. $\oplus$. This implies that it is also *correct* w.r.t. $\approx_c$, in the sense that if two modules have the same semantics then they are $\approx_c$-equivalent. From this property it follows the desired correctness result. Basically, the semantics we are going to use is a straightforward lifting to the CLP case of the compositional semantics defined in [6] for logic programs. The aim of [6] was to obtain a semantics compositional w.r.t. union of programs. In this respect it is easy to see that the standard semantics, such as the least $\mathscr{D}$-model and the computed answer semantics, are not compositional w.r.t. $\oplus$; consider for instance the modules M_1 and M_2 in Example 3.4: they have the same least $\mathscr{D}$-model, where $M_1 \oplus M$ and $M_2 \oplus M$ do not (the same reasoning applies for the answer constraint semantics of [14]). Following an idea first introduced in [15], compositionality was then obtained by choosing a semantic domain based on clauses. As we discuss below the resulting semantics turns out to model the notion of "resultant", hence its name.

In order to define the semantic domain, we use the following equivalence relation, which, intuitively, is a generalization to the CLP case of the notion of variance.

Definition 3.7. Let $cl_1 : A_1 \leftarrow c_1 \square \tilde{B}_1$ and $cl_2 : A_2 \leftarrow c_2 \square \tilde{B}_2$ be two clauses. We write $cl_1 \simeq cl_2$ iff for any $i, j \in [1,2]$ and for any $\mathscr{D}$-solution ϑ of c_i there exists a $\mathscr{D}$-solution γ of c_j such that $A_i\vartheta \equiv A_j\gamma$ and $\tilde{B}_i\vartheta$ and $\tilde{B}_j\gamma$ are equal as multisets. Moreover, given two programs P and P' we say that $P \simeq P'$ iff P' is obtained by replacing some clauses in P for $\simeq$-equivalent ones.

Notice that, in the previous definition, the body of a clause is considered as a multiset. Considering bodies of clauses as sets instead of multisets would not allow us to model correctly answer constraints, since adding a duplicate atom to the body of a clause can augment the set of computed constraints. For instance, if we consider the programs Q_1 :

```
q(X,Y) ←true  □r(X,Y),r(X,Y).
r(X,Y) ←X=a.
r(X,Y) ←Y=b.
```

and Q_2 :

```
q(X,Y) ←true  □r(X,Y).
r(X,Y) ←X=a.
r(X,Y) ←Y=b.
```

The query `q(X,Y)` has the computed answer constraint $X = a \wedge Y = b$ in Q_1 and not in Q_2.

The following lemma shows that the equivalence relation $\simeq$ is correct w.r.t. the congruence relation $\approx_c$.

Lemma 3.8 (Gabbrielli [13]). *Let $M = \langle P, \pi \rangle$ and $M' = \langle P', \pi \rangle$ be two modules with the same set of open predicates. If $P \simeq P'$ then $M \approx_c M'$.*

We are now able to define the semantic domain. For the sake of simplicity, we will denote the $\simeq$-equivalence class of a clause c by c itself.

Definition 3.9 (*Denotation*). Let π be a set of predicate symbols and let $\mathscr{C}$ be the set of the $\simeq$-equivalence classes of the CLP clauses in the given language. The *interpretation base* $\mathscr{C}_\pi$ is the set $\{A \leftarrow c \,\square\, \tilde{B} \in \mathscr{C} \mid Pred(\tilde{B}) \subseteq \pi\}$. A *denotation* is any subset of $\mathscr{C}_\pi$.

The following is the definition of the resultant semantics as it was originally given in [6] for pure logic programs and applied to CLP in [13].

Definition 3.10 (*Resultants Semantics for CLP*). Let $M = \langle P, Op(M) \rangle$ be a module. Then we define

$$\mathcal{O}(M) = \{p(\tilde{x}) \leftarrow c \,\square\, \tilde{B} \in \mathscr{C}_{Op(M)} \mid \text{there exists a derivation } true \,\square\, p(\tilde{x}) \overset{P}{\leadsto} c \,\square\, \tilde{B}\ \}.$$

If there exists a derivation $c \,\square\, \tilde{A} \overset{P}{\leadsto} d \,\square\, \tilde{B}$, then the formula $c \,\square\, \tilde{A} \leftarrow d \,\square\, \tilde{B}$ is called a *computed resultant* for the query $c \,\square\, \tilde{A}$ in P. It can be shown that computed resultants for generic queries can be obtained by combining together resultants for simple queries of the form $true \,\square\, p(\tilde{x})$. Therefore $\mathcal{O}(M)$ is expressive enough to characterize all the resultants computable in P. In particular, $\mathcal{O}(M)$ models also the answer constraints computed in M, since these can be obtained from resultants of the form $c \,\square\, \tilde{A} \leftarrow d$. The compositionality of previous semantics w.r.t. $\oplus$ is proved in [13]. From such a result follows the correctness of $\mathcal{O}$ w.r.t. $\approx_c$, stated by the following proposition.

Proposition 3.11 (Correctness, Gabbrielli [13]). *Let $M = \langle P, Op(M) \rangle$ and $N = \langle Q, Op(N) \rangle$ be modules such that $Op(M) = Op(N)$.*
If $\mathcal{O}(M) = \mathcal{O}(N)$ then $M \approx_c N$.

In the particular case $Op(M) = \emptyset$, i.e. when all the predicates are completely defined, $\mathcal{O}(M)$ coincides with the answer constraint semantics which is correct and fully abstract w.r.t. $\approx$ (see [14]).

Example 3.12. Consider again the modules M_1 and M_2 of Example 3.4. Then

$$\mathcal{O}(M_1) = \{p(X) \leftarrow X = a,\ q(X) \leftarrow X = a,\ q(X) \leftarrow true \,\square\, p(X)\}.$$
$$\mathcal{O}(M_2) = \{p(X) \leftarrow X = a,\ q(X) \leftarrow X = a,\ q(X) \leftarrow X = a \,\square\, p(X)\}.$$

So the fact that M_1 and M_2 are not observationally congruent is reflected by the fact that $\mathcal{O}(M_1) \neq \mathcal{O}(M_2)$.

3.2. Resultants semantics via trees

We now provide a new, alternative formulation of the resultant semantics in terms of proof trees. This particular notation will be used to prove the correctness results.

We assume known the usual notion of finite labelled tree and the related terminology. Given a finite labelled tree rooted in the node N, we say that T' is an *immediate subtree* of T if T' is the subtree of T which is rooted in a son of N.

Definition 3.13 (*Partial proof tree*). Let A be an atom. A *partial proof tree* for A is any finite labelled tree T satisfying the following conditions:
1. The root node of T is labelled by a pair $\langle A = A_0\ ;\ A_0 \leftarrow c_A \square A_1, \ldots, A_n \rangle$ such that A_0 and A have the same predicate symbol.
2. Each immediate subtree T_j of T is a partial proof tree for a distinct A_j with $1 \leqslant j \leqslant n$.
3. All the clauses used in the labels of T do not share variables pairwise and have no variables in common with the atom in the l.h.s (left-hand side) of the label equation in the root node.

We call *label equation* and *label clause* of the node N, the left- and the right-hand side of the label of N, respectively. Moreover, if A_i is an atom in the body of the label clause of the root of T and T_i is an immediate subtree of T which is a partial proof tree for A_i, we say that T_i is *attached* to A_i. Using this notation, condition 2 can be restated as follows: "no two immediate subtrees of T are attached to the same atom of the label clause of the root (and therefore, of any) node". Finally, we say that T is a tree *in P*, if the label clauses of all its nodes are (variants of) clauses of the program P.

Notice that, according to previous definition, there might be some A_j in the bodies of label clauses with no subtrees attached to them. We call them the elements of the *residual* as specified below.

Definition 3.14. Let T be a partial proof tree.
- The *residual* of a node in T having the *clause label* $A_0 \leftarrow c_A \square A_1, \ldots, A_n$, is the multiset consisting of those A_j's, $1 \leqslant j \leqslant n$, that do not have an immediate subtree attached to.
- The *residual* of T is the multiset resulting from the (multiset) union of the residuals of its nodes.

In order to establish the connection between the resultants semantics and partial proof-trees, we introduce now in a natural way the notion of *resultant* of partial proof trees.

Definition 3.15. Let T be a partial proof tree. We call the *global constraint* of T the conjunction of all the label equations together with the constraints of all the label clauses of the nodes of T.

Definition 3.16. Let T be a partial proof tree of A. Let c be its global constraint and $F_1, \ldots, F_k$ be its residual. If c is satisfiable we call the clause $A \leftarrow c \square F_1, \ldots, F_k$ the *resultant* of T.

In the sequel we are interested in those partial trees whose residuals consist exclusively of only open atoms and whose global constraint is satisfiable. Therefore the following definition:

Definition 3.17. Let π be a set of predicate symbols. We call π-atom any atom A such that $Pred(A) \in \pi$. A *π-tree* is a partial proof tree T such that
1. the residual of T contains only π-atoms,
2. the global constraint of T is satisfiable.

We can now establish the relation between open trees and the resultant semantics.

Proposition 3.18 (Correspondence). *Let $M = \langle P, Op(M) \rangle$ be a module. Then $A \leftarrow c \square \tilde{F} \in \mathcal{O}(M)$ iff there exists a π-tree of A in P with $A \leftarrow c' \square \tilde{F}'$ as resultant such that $A \leftarrow c \square \tilde{F} \simeq A \leftarrow c' \square \tilde{F}'$ and $\pi = Op(M)$.*

Proof. Straightforward. $\square$

4. A transformation system for CLP

In this section we define a transformation system for optimizing constraint logic programs. The system is inspired by the unfold/fold method proposed by Tamaki and Sato [37] for pure logic programs. Here, the use of constraint logic programs allows us to introduce some new operations which broaden the possible optimizations and to simplify the applicability conditions for the folding operation in [37].

Before we begin to define the transformation method, it is important to notice that all the observable properties of computations we refer to are invariant under $\simeq$. Moreover, as we formally prove later, such a replacement does not affect the applicability and the results of the transformations. Therefore 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 4.1 (*Initial program*). We call a CLP program P_0 an *initial program* if the following two conditions are satisfied:
(I1) P_0 is partitioned into two disjoint sets P_{new} and P_{old},
(I2) the predicates defined in P_{new} do not 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.

Our transformation system consists of five distinct operations. In order to illustrate them throughout this section we will use the following working example. 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 $H \leftarrow true \,\Box\, \tilde{B}$.

Example 4.2 (*Computing an average*). Consider the following CLP($\Re$) program[2] `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 predicate `exchange_rates`, which will return a list containing terms of the form ⟨`Currency, Exchange_Rate`⟩, where `Exchange_Rate` is the exchange rate relative to `Currency`. `AVERAGE` consists of the following clauses:

```
    average(List, AV) ←
        Av is the average of the list List
c1: average(Xs, Av) ← Len > 0 ∧ Av*Len = Sum □
        exchange_rates(Rates),
        weighted_sum(Xs, Rates, Sum),
        len(Xs, Len).
    weighted_sum(List, Rates, Sum) ←
        Sum is the sum of the values in the list List
        and each amount is multiplied first by the exchange rate corresponding
                                                              to its currency
    weighted_sum([], 0).
    weighted_sum([⟨Currency, Amount⟩| Rest], Rates, Sum) ←
        Sum = Amount*Value + Sum' □
        member(⟨Currency, Value⟩,Rates),
```

[2] CLP($\Re$) [22] is the CLP language obtained by considering the constraint domain $\Re$ of arithmetic over the real numbers.

```
    weighted_sum(Rest, Rates, Sum').
len(List, Len) ←
    Len is the length of the list List
len([], 0 ).
len([H|Rest], Len) ← Len = Len'+1 □ len(Rest, Len').
```

together with the usual definition for `member`. 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 the *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. As previously mentioned, all the observable properties we consider are invariant under reordering of the atoms in the bodies of clauses. Therefore the definition of unfolding, as well as those of the other operations, is given modulo reordering of the bodies. To simplify the notation, in the following definition we also assume that the clauses of a program have been renamed so that they do not share variables pairwise.

Definition 4.3 (*Unfolding*). 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 $\mathscr{D}$-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.

In this situation we also say that $\{H_1 \leftarrow c_1 \,\square\, \tilde{B}_1, \ldots, H_n \leftarrow c_n \,\square\, \tilde{B}_n\}$ are the *unfolding* clauses.

Example 4.2 (*Part 2*). The transformation strategy which we use to optimize `AVERAGE` is often referred to as *tupling* [32] or as *procedural join* [26]. First, we introduce a *new* predicate `avl` defined by the following clause:

```
    avl (List, RATES, AV, LEN) ←
        AV is the average of the list List, and LEN is its length
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:

```
avl([], Rates, Average, Len) ← Len > 0 ∧ Average*Len = 0  □
    exchange_rates(Rates),
    len([], Len).
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).
```

Furthermore, in the above clauses we unfold the atoms `len([], Len)` and `len([⟨Currency,Amount⟩ |Rest], Len)`. This yields the following two clauses:

```
c3: avl([], Rates, Average, 0) ← 0 > 0 ∧ Average*0 = 0  □
        exchange_rates(Rates).
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').
```

Notice that the constraint in the body of clause `c3` is unsatisfiable. For this reason `c3` could be removed from the program; to do that we need the following operation.

Definition 4.4 (*Clause removal*). Let $cl : H \leftarrow c \,\square\, \tilde{B}$ be a clause in the program P. If

$$\mathscr{D} \models \neg\exists\ c$$

Then we can *remove* cl from the program P, obtaining the program $P' = P \backslash \{cl\}$.

Note 4.5. In [32] we find the definition of a *clause deletion* operation for pure logic programs which in CLP terms can be expressed as follows: if $cl : H \leftarrow c \,\square\, \tilde{B}$ is a clause in P such that query $c \square \tilde{B}$ has a finitely failed tree in P then we[3] can remove cl from P. Obviously, if $\mathscr{D} \models \neg\exists\, c$ then the goal $c \square A$ has a (trivial) finitely failed tree; therefore each time that we can apply the clause removal operation we can also apply the clause deletion of [32]. However, clause removal is only apparently more restrictive than clause deletion, since by combining it with the unfolding operation we can easily simulate the latter. Indeed, if $c \square \tilde{B}$ has a finitely failed tree in P then, by a suitable sequence of unfoldings we can always transform the clause $A \leftarrow c \square \tilde{B}$, in such a way that the set of resulting clauses is either empty or contains only clauses whose constraints are unsatisfiable. So using clause removal, we can then (indirectly) remove cl from the program. We prefer to use clause removal rather than clause deletion, because when we will move to the context of *modular* CLP programs the first operation will remain unchanged while the latter will require some specific applicability conditions.

We now introduce the *splitting* operation. Here, just like for the unfolding operation, the definition is given modulo reordering of the bodies of the clauses and it is assumed that program clauses do not share variables pairwise.

Definition 4.6 (*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 $\mathscr{D}$-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. Indeed, consider for instance the program Q

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

If we split `p(X, Y)` in the body of the first clause we obtain the program Q', which after cleaning up the constraints consists of the following clauses:

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

[3] The definition of finitely failed tree for CLP is the obvious generalization of the one for pure logic programs.

Now $Q \not\approx Q'$ since the query q(X, Y) has in Q' the computed answer $\{\mathtt{X}=\mathtt{a}, \mathtt{Y}=\mathtt{b}\}$, while such an answer is not obtainable in Q.

Note 4.7. We should mention that an operation called splitting has also been defined in a technical report of Tamaki and Sato [36]. However, the operation described here is substantially different from theirs. In CLP terms the splitting operation defined in [36] can be expressed as follows. If $cl: H \leftarrow c \,\Box\, \tilde{B}$ is a clause and d a constraint then splitting cl via d consists in replacing cl by the two clauses $\{H \leftarrow c \wedge d \,\Box\, \tilde{B},\ H \leftarrow c \wedge \neg d \,\Box\, \tilde{B}\}$. This operation preserves the minimal $\mathscr{D}$-model (which corresponds to semantics used in [36]) but is does not produce $\approx$ equivalent programs. Indeed, if we consider the program $P = \{$p(X).$\}$ then by splitting its only clause w.r.t. the constraint X=a we obtain the program $P' = \{$p(X) $\leftarrow$ X=a., p(X) $\leftarrow$ X $\neq$ a.$\}$. Clearly $P' \not\approx P$, since the query p(X) returns the answer constraint X=a in P' only.

Example 4.2 (*Part 3*). By applying the splitting operation to len(Rest, 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 (by replacing c6 for a $\simeq$-equivalent clause) Len' = Len''+1, and in c5 we can do some cleaning up and we can unfold both weighted_sum([],Rates,Sum') and len([],0). After these operations we end up with the following clauses:

```
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 Len'>0. Clearly, adding Len'>0 to the body of c8 cannot be done via a simple cleaning-up of the constraints, as it transforms c8 in a clause that is not $\simeq$-equivalent. However, notice that the variable Len' in the atom len([J|Rest], Len') (in the body of c8) represents the length of the list [J|Rest] which obviously contains at least one element. Indeed, every time that c8 is used in a refutation its internal variable Len' will eventually be bounded to a numeric value greater than zero. We can then safely add the redundant constraint Len'>0 to body of c8. This type of operation is formalized by the following definition of *constraint replacement*. Notice that this operation relies on the semantics of the program (in the previous specific case, on the fact that if len([J|Rest], Len') succeeds in the current program with answer constraint c then c is equivalent to $c \wedge Len' > 0$).

Definition 4.8 (*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 $true \Box \tilde{B} \overset{P}{\leadsto} d$,

$$\mathscr{D} \models \exists_{-Var(H)} c_1 \wedge d \leftrightarrow \exists_{-Var(H)} c_2 \wedge d$$

holds, 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 similarities with the refinement operation as defined by Marriott and Stuckey in [30]. Refinement allows us to add a constrain c to a program clause $H \leftarrow c_1 \Box \tilde{B}$, provided that (for a given set of initial queries of interest) for any answer constraint d of $c_1 \Box \tilde{B}$, $\mathscr{D} \models d \rightarrow c$ holds, i.e. c is redundant in d. Clearly this case is covered by our definition. However, the similarities between this paper and [30] end here. In [30], refinement, together with two other operations, is used to define an optimization strategy which manipulates exclusively the constraints of the clauses and which is devised to reduce the overhead of the constraint solver in presence of the fixed left-to-right selection rule, thus providing a kind of optimization technique totally different from the one here considered.

Example 4.2 (*Part 4*). By performing a constraint replacement of

`Len > 0` $\wedge$ `Len = Len'+1` $\wedge$ `Average*Len = Amount*Value+Sum'`

by

`Len > 0` $\wedge$ `Len = Len'+1` $\wedge$ `Average*Len = Amount*Value+Sum'` $\wedge$ `Len' > 0`

we can add the constraint `Len'>0` to the body of clause `c8`, thus 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 `len([J|Rest], Len')` succeeds in the current program the variable `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 us 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.

As in [37], the applicability conditions of the folding operations depend on the history of the transformation, that is, on some previous programs of the transformation sequence. Recall that a transformation sequence is a sequence of programs obtained by applying some operations of unfolding, clause removal, splitting, constraint replacement and folding, starting from an *initial* program P_0 which is partitioned into P_{new} and P_{old}.

As usual, in the following definition we assume that the folding (d) and the folded (cl) clause are renamed apart and, as a notational convenience, that the body of the folded clause has been reordered so that the atoms that are going to be folded are found in the leftmost positions.

Definition 4.9 (*Folding*). Let $P_0,\dots,P_i$, $i\geqslant 0$, be a transformation sequence. Let also

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

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

If $c_A \square \tilde{K}$ is an instance of $true \square \tilde{H}$ 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

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

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

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

i.e. there is no clause $b : B \leftarrow c_B \square \tilde{L}$ in P_{new} such that $b \neq d$ and $c_A \wedge e \wedge (D = B) \wedge c_B$ is $\mathscr{D}$-satisfiable.

(F3) "*No self-folding is allowed* ", i.e.

(a) either the predicate in A is an old predicate;

(b) or cl is the result of at least one unfolding in the sequence $P_0, \ldots, P_i$.

Here, the constraint e acts as a bridge between the variables of d and cl. For this reason in the sequel we will often refer to it as *bridge* constraint. Moreover d and cl will be referred to as the folding and folded clause, respectively.

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. In this context condition **(F2)** ensures that in P_{new} there exists no clause other than d that can be used as *unfolding clause.*

We now show that (**F1**(i)) and (**F1**(ii)) are equivalent to each other. First notice that the folding and the folded clause are assumed to be standardized apart, so $\tilde{H}$ has no variables in common with A, c_A, $\tilde{K}$ and $\tilde{J}$. From this and the fact that $c_A \square \tilde{K}$ is an instance of $true \square \tilde{H}$, it follows that each solution of c_A can be extended to a solution of $c_A \wedge (\tilde{H} = \tilde{K})$. Hence

$$cl : A \leftarrow c_A \square \tilde{K}, \tilde{J} \ \simeq \ A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \square \tilde{K}, \tilde{J}.$$

Now, because of the constraint $\tilde{H} = \tilde{K}$, in the r.h.s. of the above formula, we also have that

$$cl \ \simeq \ A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \square \tilde{H}, \tilde{J}. \tag{1}$$

On the other hand, if we unfold cl' using d as unfolding clause, as a result we get the following clause:

$$cl'' : A \leftarrow c_A \wedge e \wedge (D = D') \wedge c'_D \square \tilde{H}', \tilde{J}$$

where $d' : D' \leftarrow c'_D \square \tilde{H}'$ is an appropriate renaming of d. Here, by the standardization apart and the fact that $Var(e) \subseteq Var(D) \cup Var(cl)$, the variables of c_D, $\tilde{H}$ which do not occur in D, do not occur anywhere else in this clause, so, by making explicit $(D = D')$, we can identify c'_D with c_D and $\tilde{H}'$ with $\tilde{H}$. Therefore we have that

$$cl'' \;\simeq\; A \leftarrow c_A \wedge e \wedge c_D \Box \tilde{H}, \tilde{J}. \tag{2}$$

From (1) and (2) it follows immediately that

$$cl'' \simeq cl \quad \text{iff} \quad \exists_{-Var(A,\tilde{J},\tilde{H})} \; c_A \wedge e \wedge c_D \quad \leftrightarrow \quad \exists_{-Var(A,\tilde{J},\tilde{H})} \; c_A \wedge (\tilde{H} = \tilde{K}).$$

This proves that condition (**F1**(i)) is equivalent to (**F1**(ii)). Of course, the former is more useful when we are transforming programs "by hand", while the latter is more suitable for an automatic implementation of the folding operation.

Here it is worth noticing 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.

Finally, we should mention that the purpose of (**F3**) is to avoid the introduction of loops which can occur if a clause is folded by itself. This condition is the same one that is found in Tamaki–Sato's definition of folding for logic programs.

Example 4.2 (*Part 5*). We can now fold

```
exchange_rates(Rates), weighted_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] ∧ RATES = Rates ∧ LEN = Len' ∧ AV = Sum'/Len'
```

In the resulting program, after cleaning up the constraints, the predicate avl is defined by the following clauses:

```
 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

$$\texttt{Xs = XS} \wedge \texttt{RATES = Rates} \wedge \texttt{AV = Av} \wedge \texttt{LEN = Len}$$

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, the transformed version of the program `AVERAGE`, consisting of the clauses `c11, c7, c10` together with the definition of `member`, contains a definition of `average` which needs to scan the list only once.

The transformation system given by the previous five operations is correct w.r.t. $\approx$, i.e. 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 result, which follows from the more general one contained in Section 5.

Theorem 4.10 (**Correctness**). *If $P_0, \ldots, P_n$ is a transformation sequence then*
(a) $P_0 \approx P_n$.
(b) *The least $\mathscr{D}$-models of P_0 and P_n coincide.*

Proof. Statement (a) is proven in Section 5 as a Corollary of Theorem 5.4. The fact that (a) implies (b) is proven in [13]. □

4.1. Invariance of the applicability conditions

As previously mentioned, we often substitute a clause in a program by an $\simeq$- equivalent one in order to clean up the constraints. The correctness of this operation w.r.t. the $\approx_c$ congruence is stated in Lemma 3.8. We now show that this operation is correct also in the sense that it does not affect the applicability and the result (up to $\simeq$) of the previously defined operations. This is the content of the following proposition.

Proposition 4.11. *Let $P_0, \ldots, P_n$ and $P_0^*, \ldots, P_n^*$ be two transformation sequences, such that, for $i \in [0 \ldots n]$, $P_i \simeq P_i^*$. If P_{n+1} is a program obtained from P_n via a transformation operation, then there exists a program P_{n+1}^* which can be obtained from P_n^* via the same transformation operation and such that*

$$P_{n+1} \simeq P_{n+1}^*.$$

Proof. In case that the operation used to obtain P_{n+1} from P_n was either an unfolding, a clause removal, a splitting, or a constraint replacement, this result follows immediately from the operation's definitions, so we only have to take care of the folding operation. We adopt the same notation used in Definition 4.9, so we let

– $cl : A \leftarrow c_A \Box \tilde{K}, \tilde{J}$ be the folded clause, in P_n,
– $d : D \leftarrow c_D \Box \tilde{H}$ be the folding clause, in $P_{\text{new}}(\subset P_0)$.
– e be the bridge constraint, $Var(e) \subseteq Var(D) \cup Var(cl)$,
– $cl' :\ A \leftarrow c_A \wedge e \Box D, \tilde{J}$ be the result of the folding operation.

Moreover, let

– $cl^* : A^* \leftarrow c_A^* \Box \tilde{K}^*, \tilde{J}^*$ be the clause of P_n^* corresponding to cl in P_n,
– $d^* : D^* \leftarrow c_D^* \Box \tilde{H}^*$ be the clause of P_0^* corresponding to d in P_0.

Now let e^* be a constraint such that $Var(e^*) \subseteq Var(D^*) \cup Var(cl^*)$ such that

– $cl^{*'} :\ A^* \leftarrow c_A^* \wedge e^* \Box D^*, \tilde{J}^* \simeq cl' :\ A \leftarrow c_A \wedge e \Box D, \tilde{J}$

We now only have to show that if the applicability conditions of the folding operation are satisfied (by cl, d and e) in P_n, then they are also satisfied (by cl^*, d^* and e^*) in P_n^*. To this end, the only delicate step is taken care of by the following observation.

Observation 1. *Referring to the program P_n, the clauses cl and d, and the constraint e, $c_A \Box \tilde{K}$ is an instance of $true \Box \tilde{H}$ and* (**F1**) *holds iff $c_A \Box \tilde{K}$ is an instance of $c_D \Box \tilde{H}$ and* (**F1**) *holds.*

Proof. (If) This is trivial, as if $c_A \Box \tilde{K}$ is an instance of $c_D \Box \tilde{H}$ then it is also an instance of $true \Box \tilde{H}$.

(Only if) The discussion after Definition 4.9 shows that, if $c_A \Box \tilde{K}$ is an instance of $true \Box \tilde{H}$ and (**F1**) holds, then we have the following equivalences:

$$\begin{aligned} cl :\ & A \leftarrow c_A \Box \tilde{K}, \tilde{J} \\ & \simeq A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \Box \tilde{K}, \tilde{J} \\ & \simeq A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \Box \tilde{H}, \tilde{J} \\ & \simeq A \leftarrow c_A \wedge e \wedge c_D \Box \tilde{H}, \tilde{J}. \end{aligned}$$

This implies that $c_A \Box \tilde{K}$ is an instance of $c_A \wedge e \wedge c_D \Box \tilde{H}$, which in turn is by definition an instance of $c_D \Box \tilde{H}$. This concludes the proof of the Observation. □

This Observation shows that there is no loss of generality in modifying the applicability conditions of the folding operation Definition 4.9 by replacing the condition "$c_A \Box \tilde{K}$ is an instance of $true \Box \tilde{H}$" for "$c_A \Box \tilde{K}$ is an instance of $c_D \Box \tilde{H}$". Now, from the definitions of instance and of $\simeq$ it is immediate to verify that the following facts hold:

(1) If $c_A \Box \tilde{K}$ is an instance of $c_D \Box \tilde{H}$ then $c_A^* \Box \tilde{K}^*$ is an instance of $c_D^* \Box \tilde{H}^*$.
(2) if (**F1**)$\wedge$(**F2**)$\wedge$(**F3**) are satisfied (by cl, d and e) in P_n, then they are also satisfied (by cl^*, d^* and e^*) in P_n^*.

This concludes the proof of the proposition. □

5. A transformation system for CLP modules

Theorem 4.10 shows 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 4.10 falls short from the minimal requirement since it does not guarantee that a module P will be transformed into a congruent one P'.

Transforming CLP modules requires then a strengthening of (some of) the applicability conditions given in the previous section. In what follows, we discuss such modifications considering the various operations one by one. Recall that the *open* predicates of a module M are the ones specified on $Op(M)$. Similarly, in the sequel we call *open* atoms those atoms whose predicate symbol belongs to $Op(M)$. Moreover, we assume that the transformed version of a module has the same open predicates as the original one.

Unfolding. In order to preserve the compositional equivalence, for the unfolding operation we need the following additional applicability condition:
(O1) The unfolding cannot be applied to an open atom.

This condition is clearly needed, for instance, consider the module M_0 consisting of the single clause $\{$`c1: p` $\leftarrow$ `q.`$\}$ and where $Op(M_0) = \{$`q`$\}$. Since M_0 contains no clause whose head unifies with `q`, unfolding `q` in `c1` will return an empty module $M_1 = \emptyset$. Obviously M_0 and M_1 are not observationally congruent.

Clause Removal. This operation may be safely applied to modules withoutthe need of any additional condition.

Splitting. Being closely connected to the unfolding operation, the splitting one 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)**.

Constraint replacement. This operation is the most delicate one: in order to apply it to modules we need to restate completely its applicability conditions. As a simple example showing the need of such a change, let us consider the following module M_0:

```
c1: p(X) ← true□q(X).
    q(a).
```

where $Op(M_0) = \{$`q`$\}$. The only answer constraint to the query `q(X)` in M_0 is `X` $=$ `a`. Therefore, if we refer to the applicability conditions of Definition 4.8, we could add the constraint X = a to the body of c1 thus obtaining M_1:

```
c2: p(X) ← X=a □ q(X).
    q(a).
```

Once again M_0 and M_1 are not congruent. In fact, for $N = \langle\{\texttt{q(b).}\}, \{\texttt{q}\}\rangle$, the query p(b) succeeds in $M_0 \oplus N$ and fails in $M_1 \oplus N$.

Definition 5.1 (*Constraint replacement for modules*). Let $cl : H \leftarrow c_1 \,\square\, \tilde{B}$ be a clause of a module M and let c_2 be a constraint. If

(O3) For each derivation $true \,\square\, \tilde{B} \overset{M}{\leadsto} d \,\square\, \tilde{D}$ such that $\tilde{D}$ is either empty or contains only open atoms, we have that

$$H \leftarrow c_1 \wedge d \,\square\, \tilde{D} \;\simeq\; H \leftarrow c_2 \wedge d \,\square\, \tilde{D}$$

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

In order to compare this definition with the corresponding one for nonmodular programs notice that the applicability conditions of Definition 4.8 can be restated as follows. We can replace c_1 with c_2 in the body of $cl : H \leftarrow c_1 \,\square\, \tilde{B}$ if, for each successful derivation $true \,\square\, \tilde{B} \overset{P}{\leadsto} d$ we have that

$$H \leftarrow c_1 \wedge d \;\simeq\; H \leftarrow c_2 \wedge d.$$

Now it is clear that the difference lies in the fact that here we cannot just refer to the successful derivations $true \,\square\, \tilde{B} \overset{P}{\leadsto} d$, 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 follows immediately that when the set of open atoms is empty, Definitions 4.8 and 5.1 coincide, while if $Op(M) \neq \emptyset$ then this definition is more restrictive than the previous one.

Folding. Finally, we consider the folding operation. In order to preserve the compositional equivalence the head of the folding clause cannot be an open atom. This is shown by the following simple example. Consider the initial module M_0:

```
c1: p ← q.
c2: r ← q.
```

where we assume $Op(M_0) = \{\texttt{p}\}$ and $M_{\text{new}} = \{\texttt{p} \leftarrow \texttt{q}\}$. Since r is an old atom, we can fold q in c2 using c1 as folding clause. The resulting module M_1 is

```
c3: p ← q.
c4: r ← p.
```

Again M_0 and M_1 are not observationally congruent. Indeed, if we compose them with the module $N = \langle\{\texttt{p.}\}, \{\texttt{p}\}\rangle$, we have that the query r succeeds in $M_1 \oplus N$, but fails in $M_0 \oplus N$. 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 4.2, part 6), we report an example of such a case.

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

Definition 5.2 (*Modular transformation sequence*). Let $M_0 = \langle P_0, Op(M_0)\rangle$ be a module and $P_0, \ldots, P_n$ be a transformation sequence. We say that $M_0, \ldots, M_n$ is a *modular* transformation sequence iff $M_i = \langle P_i, Op(M_0)\rangle$ for $i \in [0,n]$ 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 result stronger than the one contained in Theorem 4.10. Indeed, the system transforms a module into a congruent one.

This result is based on the following theorem which contains the main technical result of the paper and shows that any modular transformation sequence preserves the resultants semantics.

Theorem 5.3. *Let $M_0, \ldots, M_n$ be a modular transformation sequence. Then* $\mathcal{O}(M_0) = \mathcal{O}(M_n)$.

Proof. See the Appendix. □

From the previous theorem and the correctness result for the resultants semantics we can now easily derive the correctness of a modular transformation sequence.

Theorem 5.4 (Correctness of the modular transformation sequence). *Let $M_0, \ldots, M_n$ be a modular transformation sequence, then*

$$M_0 \approx_c M_n$$

Proof. Immediate from Theorem 5.3 and Proposition 3.11. □

In other words, for any module N such that $M_0 \oplus N$ is defined, $M_n \oplus N$ is also defined[4] and a generic query has the same answer constraints in $M_0 \oplus N$ and $M_n \oplus N$. From previous result we also obtain Theorem 4.10 of previous section.

Theorem 4.10. *If $P_0, \ldots, P_n$ is a transformation sequence, then,*

$$P_0 \approx P_n.$$

[4] The fact that $M_n \oplus N$ is also defined follows immediately from the fact that M_0 and M_n contain definitions for the same predicate symbols.

Proof. Note that when $Op(P_0)$ is empty, conditions **(O1)**,...,**(O4)** are trivially satisfied by any transformation sequence. Since $\approx$ can be seen as the particular case of $\approx_c$ applied to modules with an empty set of open predicates, the thesis follows from Theorem 5.4. □

Example 4.2 (*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 check that all the transformations we performed satisfied **(O1)**,...,**(O4)**. Therefore Theorem 5.4 guarantees that the final program will behave exactly as the initial one, even in this modular setting.

6. From LP to CLP

It is well-known that pure logic programming (LP for short) can be seen as a particular instance of the CLP scheme obtained by considering the Herbrand constraint system. This is defined by taking as structure the Herbrand universe and interpreting as identity the only predicate symbol for constraints "=". So it is natural to expect that an unfold/fold transformation for LP can be embedded into one for CLP. Indeed, in this section we show that the transformation system we propose is a generalization to the CLP (and modular) case of the unfold/fold system designed by Tamaki and Sato [37] for LP. As a consequence, conditions **(O1)** and **(O4)** can be used also in the LP case to transform a module into a congruent one.

We introduce the system of Tamaki and Sato by first considering the unfold operation for LP. Again, we assume that the clauses are standardized apart and we give the following definition modulo reordering of the bodies.

Definition 6.1 (*Unfolding for LP*). Let cl: $A \leftarrow H, \tilde{K}$ be a clause of a logic program P, and let $\{H_1 \leftarrow \tilde{B}_1, \ldots, H_n \leftarrow \tilde{B}_n\}$ be the set of clauses of P whose heads unify with H, by mgu's $\{\theta_1, \ldots, \theta_n\}$. For $i \in [1,n]$ let cl'_i be the clause

$$(A \leftarrow \tilde{B}_i, \tilde{K})\theta_i$$

Then *unfolding H in cl in P* consists of replacing cl by $\{cl'_1, \ldots, cl'_n\}$ in P.

Also in the LP case the notions of folding operation and of transformation sequence are defined in a mutually recursive way. So, in the sequel we use the same definition of *initial program* as before. However, since clause removal, splitting and constraint replacement are new operations which were not in [37], we call now *LP transformation sequence* a sequence of LP programs $P_0, \ldots, P_n$, in which P_0 is an initial program and each P_{i+1}, is obtained from P_i either via an unfolding or via a folding operation[5].

[5] However, we should mention that in [37] also a more general replacement operation is taken into consideration, but this operation is beyond the scope of this paper.

Now we also need some extra preliminary notions. Given a substitution $\theta = \{x_1/t_1,...,x_n/t_n\}$ we denote by $Dom(\theta)$ the set of variables $\{x_1,\ldots,x_n\}$, and by $Ran(\theta)$ the set of variables appearing in $\{t_1,\ldots,t_n\}$, if $Ran(\theta) = \emptyset$ we say that θ is *grounding*. Finally we denote by $Var(\theta)$ the set $Dom(\theta) \cup Ran(\theta)$.

We are now ready to give the definition of the folding operation for LP. Again, here we assume that the folding and the folded clause are renamed apart and that the body of the folded clause has been reordered (as in Definition 4.9).

Definition 6.2 (*Folding for LP, Tamaki and Sato* [37]). Let $P_0,\ldots,P_i$, $i \geqslant 0$, be an LP transformation sequence and

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

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

Let also $\tilde{v} = Var(\tilde{H}) \setminus Var(D)$ be the set of local variables of d. If there exists a substitution τ such that $Dom(\tau) = Var(d)$, then *folding* $\tilde{K}$ *in* cl *via* τ consists of replacing cl by $cl' :\ A \leftarrow D\tau,\tilde{J}$, provided that the following conditions hold:

(LP1) $\tilde{H}\tau = \tilde{K}$;

(LP2) For any $x, y \in \tilde{v}$

- $x\tau$ is a variable;
- $x\tau$ does not appear in A, $\tilde{J}$, $D\tau$;
- if $x \not\equiv y$ then $x\tau \not\equiv y\tau$;

(LP3) d is the only clause in P_{new} whose head is unifiable with $D\tau$;

(LP4) one of the following two conditions holds:

1. the predicate in A is an old predicate;
2. cl is the result of at least one unfolding in the sequence $P_0,\ldots,P_i$.

Concerning the unfolding operation, it is easy to see that Definition 6.1 is the LP counterpart of Definition 4.3. In fact, an LP clause is itself a CLP rule (with an empty constraint) and well-known results [27] imply that two terms s and t have an mgu iff the equation $s = t$ is satisfiable in the Herbrand constraint system. Therefore, given a logic program P, we can unfold P according to Definition 6.1 iff we can unfold P according to Definition 4.3. Clearly, the results of the two operations are syntactically different, since substitutions are used in the first case whereas constraints are employed in the second one. However, again by using standard results of unification theory, it is easy to check that the different results are $\simeq$ equivalent.

On the other hand, when considering the folding operation, the similarities between Definitions 6.2 and 4.9 are less immediate. Therefore we now formally prove that, whenever the folding operation for LP programs is applicable also the folding operation for CLP programs is, and the result of this latter operation is $\simeq$-equivalent to the result of the operation in LP. This is summarized in the following.

Theorem 6.3. *If P_0 is a logic program and $P_0, \ldots, P_n$ is an LP transformation sequence then there exists a CLP transformation sequence $P_0^*, \ldots, P_n^*$ such that, for $i \in [0,n]$, $P_i \simeq P_i^*$.*

Proof. In order to simplify the notation, we now define a simple mapping from LP clauses to clauses in *pure* CLP.[6] Let $cl : p_0(\tilde{t}_0) \leftarrow p_1(\tilde{t}_1), \ldots, p_n(\tilde{t}_n)$ be a clause in LP. Then $\mu(cl)$ is the CLP clause

$$p_0(\tilde{x}_0) \leftarrow \tilde{x}_0 = \tilde{t}_0 \wedge \tilde{x}_1 = \tilde{t}_1 \wedge \cdots \wedge \tilde{x}_n = \tilde{t}_n \,\square\, p_1(\tilde{x}_1), \ldots, p_n(\tilde{x}_n),$$

where $\tilde{x}_0, \ldots, \tilde{x}_n$ are tuple of new and distinct variables. Obviously $\mu(cl) \simeq cl$ for any clause cl. Therefore it suffices to prove that if $P_0, \ldots, P_n$ is a transformation sequence of logic programs, then $\mu(P_0), \ldots, \mu(P_n)$ is a transformation sequence in CLP. The proof proceeds by induction on the length of the sequence. For the the base case ($n = 0$) the result holds trivially, so we go immediately to the induction step: we assume that $P_0, \ldots, P_{n+1}$ is a transformation sequence in LP, that $\mu(P_0), \ldots, \mu(P_n)$ is a transformation sequence in CLP, and we now prove that $\mu(P_0), \ldots, \mu(P_{n+1})$ is a transformation sequence in CLP as well.

If P_{n+1} is the result of unfolding a clause cl of P_i, then it is straightforward to check that by unfolding $\mu(cl)$ in $\mu(P_i)$ we obtain $\mu(P_{i+1})$ (modulo $\simeq$).

Now we consider the case in which P_{n+1} is the result of a folding operation (applied to P_n). We prove the thesis for the simplified situation where $\tilde{H}$, $\tilde{K}$ and $\tilde{J}$ consist each of a single atom. The extension to the general case is straightforward. Let

$d : a(\tilde{s}) \leftarrow b(\tilde{t})$ be the folding clause, in P_{new}.

Since we are assuming that the applicability conditions of Definition 6.2 are satisfied, by (**LP1**) the folded clause (in P_n) can be written as follows:

$$cl : c(\tilde{u}) \leftarrow b(\tilde{t}\tau), d(\tilde{v}).$$

The result of the folding operation is then

$$cl' : c(\tilde{u}) \leftarrow a(\tilde{s}\tau), d(\tilde{v}).$$

which is a clause in P_{n+1}.

By translating the folding and the folded clause in CLP, we obtain

$$\begin{aligned} \mu(d) &\equiv d^* : a(\tilde{x}) \leftarrow \tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t} \,\square\, b(\tilde{y}), \\ \mu(cl) &\equiv cl^* : c(\tilde{z}) \leftarrow \tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \,\square\, b(\tilde{w}), d(\tilde{k}). \end{aligned}$$

Where $\tilde{x}$, $\tilde{y}$, $\tilde{z}$, $\tilde{w}$ and $\tilde{k}$ are tuples of new and distinct variables. Now, let e be the following constraint:

$$e \equiv \tilde{x} = \tilde{s}\tau$$

[6] *Pure* CLP programs are CLP programs in which the atoms in the clauses, apart from constraints, are always of the form $p(\tilde{x})$, where $\tilde{x}$ is a tuple of distinct variables.

the result of the folding operation in CLP is then

$$cl'^{*} : c(\tilde{z}) \leftarrow \tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \wedge \tilde{x} = \tilde{s}\tau \,\square\, a(\tilde{x}), d(\tilde{k}).$$

It is straightforward to check that $\mu(cl') \simeq cl'^{*}$. Now, it is also clear that $\tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \square b(\tilde{w})$ is an instance of $true \square b(\tilde{y})$, so in order to prove the thesis we now need to verify that if d, cl and τ satisfy **(LP1)**, **(LP2)** in P_n then d^*, cl^* and e satisfy **(F1)** in $\mu(P_n)$. Here the structure $\mathscr{D}$ is the Herbrand structure, whose domain is the Herbrand universe and where "=" is interpreted as the identity.

Now the condition **(F1)** is $\mathscr{D} \models \exists_{-\tilde{z},\tilde{y}}\, c_{\text{left}} \leftrightarrow \exists_{-\tilde{z},\tilde{y}}\, c_{\text{right}}$ where c_{left} is

$$\tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \wedge \tilde{x} = \tilde{s}\tau \wedge \tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t}$$

and c_{right} is

$$\tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \wedge \tilde{y} = \tilde{w}.$$

In both sides of the formula we find the equations $\tilde{w} = \tilde{t}\tau$, $\tilde{k} = \tilde{v}$, $\tilde{x} = \tilde{s}\tau$, where $\tilde{w}, \tilde{k}, \tilde{x}$ are tuple of fresh variable and are existentially quantified, hence we can simplify **(F1)** to

$$\mathscr{D} \models \exists_{-\tilde{z},\tilde{y}}\, \tilde{z} = \tilde{u} \wedge \tilde{s} = \tilde{s}\tau \wedge \tilde{y} = \tilde{t} \;\leftrightarrow\; \exists_{-\tilde{z},\tilde{y}}\, \tilde{z} = \tilde{u} \wedge \tilde{y} = \tilde{t}\tau. \tag{3}$$

Recall that, when considering the Herbrand structure, ϑ is a *solution* of a constraint c if ϑ is a grounding substitution such that $Dom(\vartheta) = Var(c)$ and $\mathscr{D} \models c\vartheta$.

We now show that for each solution η of one side of (3) there exists a solution η' of the other side of (3) such that $\eta|_{\tilde{z},\tilde{y}} = \eta'|_{\tilde{z},\tilde{y}}$; this will imply the thesis.

We now prove the two implications separately:

($\leftarrow$) Let η be a solution of $\tilde{z} = \tilde{u} \wedge \tilde{y} = \tilde{t}\tau$. We assume that η is minimal, in the sense that if l is a variable not occurring in $\tilde{z} = \tilde{u} \wedge \tilde{y} = \tilde{t}\tau$, then $l \notin Dom(\eta)$. Since, by standardization apart, $Dom(\tau) \cap Ran(\tau) = \emptyset$, we have that $Dom(\eta) \cap Dom(\tau) = \emptyset$. We can extend η to η' where $Dom(\eta') = Dom(\eta) \cup Dom(\tau)$: for each $l \in Dom(\tau)$, we let

$$l\eta' \text{ be equal to } l\tau\eta. \tag{4}$$

η' is now also a solution of the left-hand side of (3). In fact

$$\begin{aligned} \tilde{s}\eta' &= \tilde{s}\tau\eta \quad \text{(by (4))} \\ &= \tilde{s}\tau\eta' \quad \text{(because } \eta' \text{ is an extension of } \eta). \end{aligned}$$

Moreover

$$\begin{aligned} \tilde{y}\eta' &= \tilde{t}\tau\eta' \quad \text{(because } \eta' \text{ is an extension of } \eta, \text{ and } \eta \text{ is a solution of } y = \tilde{t}\tau) \\ &= t\eta' \quad \text{(by(4))}. \end{aligned}$$

Since η' is an extension of η, we have that $\eta|_{\tilde{z},\tilde{y}} = \eta'|_{\tilde{z},\tilde{y}}$.

($\rightarrow$) Let η be a solution of $\tilde{z} = \tilde{u} \wedge \tilde{s} = \tilde{s}\tau \wedge \tilde{y} = \tilde{t}$. Again, we assume η to be minimal (in the sense above, i.e. $Dom(\eta)$ =Var($\tilde{z} = \tilde{u} \wedge \tilde{s} = \tilde{s}\tau \wedge \tilde{y} = \tilde{t}$)). Observe that $Dom(\eta) \cap Ran(\tau) = Var(s\tau)$. We now extend η to η' in such a way that $Dom(\eta)$ encompasses the whole $Ran(\tau) = Var(t\tau) \cup Var(s\tau)$. Let $\tilde{l}$ be the tuple of variables given by $Var(\tilde{t}) \backslash Var(\tilde{s})$, by **(LP2)** we have that $\tilde{l}\tau$ is a tuple of distinct variables. Moreover, the variables in $\tilde{l}\tau$ do not occur anywhere else in the above formulas. So, for each $l_i \in \tilde{l}$, we can let

$$l_i\tau\eta' \text{ be equal to } l_i\eta. \tag{5}$$

Since η is already a solution of $\tilde{s} = \tilde{s}\tau$ and η' is an extension of η, by (5) we have that

$$\tilde{t}\tau\eta' = \tilde{t}\eta.$$

Since η is a solution of $\tilde{y} = \tilde{t}$, η' is then a solution of $\tilde{y} = \tilde{t}\tau$, and hence of the whole LHS of (3), which concludes the proof. □

Theorem 6.3 allows us to apply the results of the previous section also to the Tamaki–Sato schema, thus obtaining a transformation system for LP modules. The following corollary show the correctness result for this case. Here we consider as LP module a logic program P together with a set of predicate symbols π. Module composition and the related notions are the same as in the previous sections. Given two logic programs P_1 and P_2, the concept of observational equivalence $\approx^{\mathrm{LP}}$ is defined as follows:

- $P_1 \approx^{\mathrm{LP}} P_2$ iff, for any query Q and for any $i, j \in [1,2]$, if Q has a computed answer ϑ_i in the program P_i then Q has a computed answer ϑ_j in the program P_j such that $Q\vartheta_i \equiv Q\vartheta_j$.[7]

Therefore, in the LP context, the concept of module congruence is defined as follows. Given two modules M_1 and M_2,

- $M_1 \approx_c^{\mathrm{LP}} M_2$ iff $Op(M_1) = Op(M_2)$ and for every module N such that $M_1 \oplus N$ and $M_2 \oplus N$ are defined, $M_1 \oplus N \approx^{\mathrm{LP}} M_2 \oplus N$ holds.

Corollary 6.4. *Let $M_0 : \langle P_0, \pi\rangle$ be a logic programming module, $P_0, \ldots, P_n$ be an LP transformation sequence and for $i \in [1,n]$ let M_i be the module $\langle P_i, \pi\rangle$. If conditions* **(O1)** *and* **(O4)** *are satisfied then $M_0 \approx_c^{LP} M_n$.*

Proof. Immediate from Theorems 6.3 and 5.4. □

7. Conclusions

Among the works on program's transformations, the most closely related to this paper are Maher's [29] and the one of Bensaou and Guessarian [3].

[7] We assume here that generic mgu's are used in the SLD derivations. If only relevant mgu's were allowed, then the syntactic equality should be replaced by variance.

Maher considers several kinds of transformations for deductive database modules with constraints (allowing negation in the bodies of the clauses) and refers to the perfect model semantics. However, the folding operation proposed in [29] is quite restrictive, in particular it lacks the possibility of introducing recursion. Indeed, for positive programs, it is a particular case of the one defined here. Moreover, our notion of module composition is more general than the one considered in [29], since the latter does not allow mutual recursion among modules.

Recently, an extension of the Tamaki–Sato method to CLP programs has also been proposed by Bensaou and Guessarian [3], yet there are some substantial differences between [3] and our proposal.

Firstly, just as in the case of the operation defined in [29], also the folding defined in [3] is very restrictive in that it lacks the possibility of introducing recursion.

Secondly, since in an unfold/fold transformation sequence we allow more operations (namely splitting and constraint replacement), we obtain a more powerful system. For instance, the transformation performed in Example 4.2 is not feasible with the tools of [3]. On the other hand, since in [3] the authors define also a goal replacement operation, there exist also some transformation which can be done with the tools of [3] and not with ours. However, such a replacement operation does not fit in an unfold/fold transformation sequence, in particular no folding is allowed when the transformation sequence contains a goal replacement. For this reason a goal replacement operation as defined in [3] has to be regarded as an issue which is orthogonal to the one of the unfold/fold transformations, and which is also beyond the scope of this paper: We have studied replacement operations for CLP modules in [12].

A third relevant difference is due to the fact that since modularity is not taken into account in [3], the system introduced in that paper does not produce observationally congruent programs. As pointed out in the introduction, this issue is particularly relevant for practical applications.

Finally, one last improvement over [3] is that of the applicability conditions we propose are invariant under $\simeq$-equivalence (Proposition 4.11), while the ones in [3] are not: this means that in some cases the folding conditions of [3] may not be satisfiable unless we appropriately modify the constraints of the clauses (maintaining $\simeq$-equivalence). Moreover, since the reference semantics in [3] is an abstraction (upward closure) of the answer constraint semantics, the result on the correctness of the unfold/fold system of [3] can be seen as a particular case of our Theorem 4.10.

To conclude, the contributions of this paper 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 [37]. 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. Moreover, 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 $\mathscr{D}$-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 module's congruence. This means that the transformed version of a CLP module can replace the original one in any context, yet preserving the computational behaviour of the whole system in terms of answer constraints. As previously argued, this provides a useful tool for the development of real software since it allows incremental and modular optimizations of large programs.

Finally, the relations between transformation sequences for CLP and LP have been discussed. By mapping logic programs into CLP programs we have shown that our transformation system is a generalization to CLP (and to modules) of the one proposed by Tamaki and Sato [37]. This relation allows us to prove that, under conditions **(O1)** and **(O4)**, the system by Tamaki and Sato transforms an LP module into a congruent one.

In the literature we also find less related papers presenting methods which focus exclusively on the manipulation of the constraint for compile-time [30] and for low-level local optimization (in which the constraint solving is partially compiled into imperative statements) [23, 21]. These techniques are totally orthogonal to the one discussed here, and can therefore be integrated with our method. On the other hand, some strategies which use transformation rules for composing complex (pure) logic programs starting from simpler pieces have been presented in [26] and further discussed in [32]. Also these strategies could easily be extended to CLP and integrated with our transformation rules. Transformations based on partial evaluation for structured logic programs have been studied in [7]. These results however are quite different from ours, since they are not concerned with CLP, use a completely different kind of program transformation and refer to a different notion of module.

Acknowledgements

The authors want to thank K.R. Apt, A. Bossi and the referees for their helpful comments.

Appendix A

In this appendix we first give the proof of Theorem 5.3 which shows that any modular transformation sequence preserves the resultants semantics. The proof, quite long and tedious, is split in two parts (partial and total correctness) and is inspired by the one given in [24].

Throughout the Appendix we will adopt the following.

Notation. We refer to a fixed module

$$M_0 = \langle P_0, Op(M_0)\rangle$$

and to a fixed transformation sequence

$$M_0 \dots M_n.$$

Moreover, for notational convenience, we set

$$\pi = Op(M_0)$$

A.1. Partial correctness

Intuitively, a transformation is called partially correct if it does not introduce new semantic information. In our case, partial correctness corresponds to the inclusion $\mathcal{O}(M_0) \supseteq \mathcal{O}(M_n)$ of Theorem 5.3. Before proving such an inclusion we need to establish some further notation.

Definition A.1. We say that two trees T and T' are *similar* if they are partial proof trees for the same atom, and they have the same resultant, modulo $\simeq$.

This is (obviously) an equivalence relation, so we can also say that two trees belong to the same *equivalence class* iff they are trees of the same atom, and their resultants are equal, modulo $\simeq$.

The next two lemmata outline some simple properties of proof trees which will be useful in the sequel. The first one states that, given a tree T, we can replace a subtree S with a similar subtree S', without altering the main properties of T.

Lemma A.2. *Let T be a π-tree, S be a subtree of T, and S' be a partial proof tree similar to S and such that the clauses of S' do not share variables with T. Then the tree T' obtained from T by replacing S for S' is a π-tree and is similar to T.*

Proof. Straightforward.

Lemma A.3. *Let T be a partial proof tree of A; let also T' be the tree obtained from T by replacing A with A' in the l.h.s. of the label equation of the root node. If A' and A have the same predicate symbol, and A' does not share variables with T, then T' is a partial proof tree of A'.*

Proof. Obvious. □

In other words, a partial proof tree for A is basically also a partial proof tree for any A' that has the same relation symbol of A. Of course this lemma gives no guarantee that after the substitution of A with A', the global constraint of the tree will still be satisfiable.

We need a couple of final, preliminary results.

Remark A.4. Let P be a program and $A \leftarrow d \,\Box\, \tilde{D}$ be a resultant. Equivalent are

- There exists a derivation $true \,\Box\, A \overset{P}{\rightsquigarrow} d' \,\Box\, \tilde{D}'$ such that $A \leftarrow d \,\Box\, \tilde{D} \simeq A \leftarrow d' \,\Box\, \tilde{D}'$;

- There exists a partial proof tree of A in P whose whose resultant is $A \leftarrow d'' \square \tilde{D}''$ and such that $A \leftarrow d \square \tilde{D} \simeq A \leftarrow d'' \square \tilde{D}''$.

Proof. Straightforward. $\square$

Lemma A.5 (Gabbrielli et al. [13]). *Let P be a program, if, for distinct $i,j \in [1,k]$, there exists a derivation*

$$true \square A_i \overset{P}{\rightsquigarrow} c_i \square \tilde{F}_i$$

and $Var(c_i \square \tilde{F}_i) \cap Var(c_j \square \tilde{F}_j) \subseteq Var(A_i) \cap Var(A_j)$ then there also exists a derivation

$$true \square A_1,\dots,A_k \overset{P}{\rightsquigarrow} c_1 \wedge \cdots \wedge c_k \square \tilde{F}_1,\dots,\tilde{F}_k.$$

We can now state the partial correctness result for the transformation system.

Proposition A.6 (Partial correctness). *If $\mathcal{O}(M_0) = \mathcal{O}(M_i)$ then $\mathcal{O}(M_i) \supseteq \mathcal{O}(M_{i+1})$*

Proof. To simplify the notation, here and in the sequel we refer to $P_1,\dots,P_n$ rather than to $M_1,\dots,M_n$.

In case P_{i+1} was obtained from P_i by unfolding or by a clause removal operation then the result is straightforward, therefore we need only to consider the remaining operations.

We now show that if there exists a π-tree T_A of an atom A with resultant R in P_{i+1}, then there exists also π-tree of A with resultant R in P_i (modulo $\simeq$). By Proposition 3.18, this will imply the thesis. The proof is by induction on the *size* of a proof tree, which corresponds to the number of nodes it contains. Let cl' be the label clause of the root node of T_A, and let us distinguish various cases.

Case 1: $cl' \in P_i$. This is the case in which clause cl' was not affected by the passage from P_i to P_{i+1}. The result follows then from the inductive hypothesis: For each subtree S of T_A (in P_{i+1}) there exists a similar subtree S' in P_i, so the tree obtained by replacing each S with S' in T_A is a π-tree in P_i similar to T_A.

Case 2: *cl' is the result of splitting.* Let cl be the corresponding clause in P_i, i.e., the clause that was split. There is no loss in generality in assuming that the atom that was split was the leftmost one. Therefore the situation is the following:
– $cl: \; A_0 \leftarrow c_A \square A_1,\dots,A_n$
– $cl': \; A_0 \leftarrow c_A \wedge (A_1 = B) \wedge c_B \square A_1,\dots,A_n$
where $B \leftarrow c_B \square \tilde{D}$ is one of the splitting clauses, and has no variable in common with cl. Since by condition (**O2**) no open atom can be split, we have that A_1 may not belong to the residual of T_A, therefore there exist a subtree T_{A_1} of T_A which is attached to A_1. Let $C \leftarrow c_C \square \tilde{E}$ be the label clause of the root node of T_{A_1}. With this notation the global constraint of T_A has the form

$$(A = A_0) \wedge c_A \wedge (A_1 = B) \wedge c_B \wedge (A_1 = C) \wedge c_C \wedge \cdots \tag{A.1}$$

Now $C \leftarrow c_C \Box \tilde{E}$ is also one of the clauses used to split A_1; by the applicability conditions of the splitting operation either C and B are heads (of renamings) of the same clause, or $C = B \wedge c_C \wedge c_B$ is unsatisfiable. Since (A.1) is satisfiable, we have that C and B must be renamings of the heads of the same clause. Since by standardization apart, the variables in c_B and in B may not occur anywhere else in T_A, as far as global constraint of T_A is concerned, the expression $(A_1 = B) \wedge c_B$ is already implied by the expression $(A_1 = C) \wedge c_C$, therefore we can eliminate $(A_1 = B) \wedge c_B$ from the global constraint of T_A, and obtain a tree which is similar to it; in other words, by replacing the clause cl' with cl in the label of the root of T_A, we obtain a tree T_A^1 which is similar to T_A.

By inductive hypothesis, for each subtree T_{A_i} of T_A (and T_A^1) there exists a tree $T_{A_i}^2$ in P_{i+1} which is similar to T_{A_1}. We can assume without loss of generality that the clauses in each $T_{A_i}^2$ do not share variables with those in T_A^1.

Finally, let T_A^2 be the tree obtained from T_A^1 by substituting each subtree T_{A_i} with $T_{A_i}^2$, by Lemma A.2 we have that T_A^2 is similar to T_A^1, and therefore to T_A. Since T_A^2 is a π-tree of A in P_i, the result follows.

Case 3: *cl' is the result of a constraint replacement*. From now on, let us call *internal constraint* of a tree T, the conjunction of all the constraints in the label clauses of T, together with the label equations of the subtrees of T. So the *internal constraint* is obtained from the global constraint by removing from it the label equation of the root node of T.

Now, let

– $cl' :\ A \leftarrow c' \Box A_1, \dots, A_n$, and

– $cl :\ A \leftarrow c \Box A_1, \dots, A_n$. where cl is the clause to which the replacement was applied.

Let also $T_{A_1}, \dots, T_{A_{n'}}$ be the subtrees of T_A (which we suppose attached to $A_1, \dots, A_{n'}$), $c_{A_1}, \dots, c_{A_{n'}}$ be their internal constraints and $\tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}$ be their residuals. With this notation, the resultant of T_A is

$$A \leftarrow (A = A_0) \wedge c' \wedge c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n.$$

By Lemma A.4, the existence of $T_{A_1}, \dots, T_{A_{n'}}$ implies that for $i \in [1, n']$ there exists a derivation $true \Box A_i \overset{P_{i+1}}{\rightsquigarrow} c_{A_i} \Box \tilde{F}_{A_i}$ (modulo $\simeq$). Since by inductive hypothesis each subtree of T_A has a similar subtree in P_i, Remark A.4 also implies that, for $i \in [1, n']$ there exists a derivation which is equal (modulo $\simeq$) to

$$true \Box A_i \overset{P_i}{\rightsquigarrow} c_{A_i} \Box \tilde{F}_{A_i}.$$

By combining these derivations together (Remark A.5) we have that there exists a derivation

$$true \Box A_1, \dots, A_n \overset{P_i}{\rightsquigarrow} c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n. \tag{A.2}$$

Now, since $cl \in P_i$ it follows that there exists a derivation

$$true \Box A \overset{P_i}{\rightsquigarrow} (A = A_0) \wedge c \wedge c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n.$$

From Remark A.4 it follows that there exists a π-tree S_A of A in P_i whose resultant is

$$A \leftarrow (A = A_0) \wedge c \wedge c_{A_1} \wedge \ldots \wedge c_{A_{n'}} \square \tilde{F}_{A_1}, \ldots, \tilde{F}_{A_{n'}}, A_{n'+1}, \ldots, A_n.$$

From (A.2) and the applicability conditions for the replacement operation it follows that the resultant of S_A is $\simeq$-similar to the one of T_A. Hence the thesis.

Case 4: *cl' is the result of folding.* Let

- $cl: \ A_0 \leftarrow c_A \square B_1^-, \ldots B_m^-, A_1, \ldots, A_n$ be the folded clause (in P_i)
- $d: B_0 \leftarrow c_B \square B_1, \ldots, B_m$ be the folding clause (in P_{new}), so we have that
- $cl': \ A_0 \leftarrow c_A \wedge e \square B_0, A_1, \ldots, A_n$ is the label clause of the root node of T_A; Let also
- $B_0, A_1, \ldots, A_{n'}$ be the atoms of cl' that have an immediate subtree (in P_{i+1}) attached to in T_A; this choice causes no loss of generality, in fact, by **(O4)**, B_0 cannot be a π-atom, and hence it cannot be part of the residual of the root node of T_A.
- $A_{n'+1}, \ldots, A_n$ is then the residual of the root node.

So let

- $T_{B_0}, T_{A_1}, \ldots, T_{A_{n'}}$ be the immediate π-subtrees of T_A.

By the inductive hypothesis, there exist π-trees

- $T'_{B_0}, T'_{A_1}, \ldots, T'_{A_{n'}}$ in P_i which are similar to $T_{B_0}, T_{A_1}, \ldots, T_{A_{n'}}$.

Since $\mathcal{O}(P_0) = \mathcal{O}(P_i)$, from Proposition 3.18 it follows that there exists a π-tree S_{B_0} of B_0 in P_0 which is similar to T'_{B_0} (in P_i). Because of the condition **(F2)**, the label clause of the root of S_{B_0} is an appropriate renaming of d. Let

- $d^*: B_0^* \leftarrow c_B^* \square B_1^*, \ldots, B_m^*$ be the label clause of the root node of S_{B_0}, and
- $B_0 = B_0^*$ is then the label equation of the root of S_{B_0}.

Moreover, let

- $S_{B_1^*}, \ldots, S_{B_{m'}^*}$ be its immediate subtrees (in P_0), which we suppose to be attached to $B_1^*, \ldots, B_{m'}^*$
- $B_{m'+1}^*, \ldots, B_m^*$ is then the residual of its root node.

Let T_A^2 be the π-tree in $P_{i+1} \cup P_i \cup P_0$ obtained from T_A by replacing its subtrees $T_{B_0}, T_{A_1}, \ldots, T_{A_{n'}}$ with $S_{B_0}, T'_{A_1}, \ldots, T'_{A_{n'}}$ and let R^2 be its resultant. Since we can assume without loss of generality that the clauses in the subtrees $S_{B_0}, T'_{A_1}, \ldots, T'_{A_{n'}}$ do not share variables with each other and with the clauses in T_A, by Lemma A.2 we have that

$$R \simeq R^2. \tag{A.3}$$

Now let us write out explicitly the resultant of R^2, so let

- c_{rest} be the constraint given by the conjunction of all the global expressions of $T'_{A_1}, \ldots, T'_{A_{n'}}$, together with the internal constraint of $S_{B_1^*}, \ldots, S_{B_{m'}^*}$;
- $\tilde{F}$ be the (multiset) union of the residuals of $T'_{A_1}, \ldots, T'_{A_{n'}}, S_{B_1^*}, \ldots, S_{B_{m'}^*}$;
- $B_1^* = C_1, \ldots, B_{m'}^* = C_{m'}$ be the label equations of the root nodes of $S_{B_1^*}, \ldots, S_{B_{m'}^*}$;

We have that $R^2 = A \leftarrow c_{\text{tot}} \square \tilde{F}, B_{m'+1}^*, \ldots, B_m^*, A_{n'+1}, \ldots, A_n$, where c_{tot} is

$$(A = A_0) \wedge c_A \wedge e \wedge (B_0 = B_0^*) \wedge c_B^* \wedge \left(\bigwedge_{j=1}^{m'} B_j^* = C_j \right) \wedge c_{\text{rest}}.$$

By **(F1)**, this reduces to

$$(A = A_0) \wedge c_A \wedge (B_0^* = B_0) \wedge \left(\bigwedge_{j=1}^{m} B_j^* = B_j \right) \wedge \left(\bigwedge_{j=1}^{m'} B_j^* = C_j \right) \wedge c_{\text{rest}}. \tag{A.4}$$

Now we show that we can drop the constraint $B_0^* = B_0$. First notice that since B_0^* is a renaming of B_0, then $B_0^* = B_0$ can be reduced to a conjunction of equations of the form $x = y$, where x and y are distinct variables. In the case that for some x, y, $B_0^* = B_0$ implies $x = y$, then we have that either $x = y$ is already implied by the constraint $(\bigwedge_{j=1}^{m} B_j^* = B_j)$ or the variables x and y do not occur anywhere else in (A.4), nor in R^2. So (A.4) becomes

$$(A = A_0) \wedge c_A \wedge \left(\bigwedge_{j=1}^{m} B_j^* = B_j \right) \wedge \left(\bigwedge_{j=1}^{m'} B_j^* = C_j \right) \wedge c_{\text{rest}}. \tag{A.5}$$

On the other hand, by replacing B_j^* with B_j^- in the l.h.s. of the label equations of the root nodes of the trees $S_{B_1^*}, \ldots, S_{B_{m'}^*}$, we obtain the trees $S_{B_1^-}, \ldots, S_{B_{m'}^-}$, which, by Lemma A.3, are π-trees of $B_1^-, \ldots, B_{m'}^-$. Now let T_A^3 be the π-tree of A in $P_i \cup P_0$ which is constructed as follows:
– cl is the label clause of its root,
– its immediate subtrees are $S_{B_1^-}, \ldots, S_{B_{m'}^-}$ (in P_0) and $T'_{A_1}, \ldots, T'_{A_{n'}}$ (in P_i). Then the residual of T_A^3 is precisely $A \leftarrow c_{\text{tot}}^3 \,\square\, \tilde{F}, B_{m'+1}^-, \ldots, B_m^-, A_{n'+1}, \ldots, A_n$, where c_{tot}^3 is

$$c_A \wedge \left(\bigwedge_{j=1}^{m} B_j^- = B_j \right) \wedge \left(\bigwedge_{j=1}^{m'} B_j^- = C_j \right) \wedge c_{\text{rest}}.$$

By this, (A.5) and (A.3), we have that T_A^3 is similar to T_A.

Finally, since $\mathcal{O}(P_0) = \mathcal{O}(P_i)$, each of the trees $S_{B_j^-}$ (in P_0) has a similar tree in P_i, S_{O^-} by replacing each $S_{B_j^-}$ with it in T_A^3, we obtain T_A^4; by Lemma A.2 and the usual assumption on the variables of the clauses in the $S_{B_j^-}$'s, T_A^4 is similar to T_A^3, and hence to T_A. Since T_A^4 is a tree in P_i, this proves the thesis. □

A.1.1. Total correctness

We say that a transformation sequence is *complete*, if no information is lost during it, that is $\mathcal{O}(M_0) \subseteq \mathcal{O}(M_i)$. When a transformation sequence is partially correct and complete we say that it is *totally correct*. Before entering in the details of the proof of total correctness, we need the following simple observation.

Remark A.7. If cl is a clause of P_i that does not satisfy condition **(F3)** then the predicate in the head of cl is a *new* predicate, while the predicates in the atoms in the body are *old* predicates.

The proof of the completeness is basically done by induction on the weight of a tree, which is defined by the following.

Definition A.8. (*weight*)
- The weight of a π-tree T, $w(T)$, is defined as follows:
  - $w(T) = size(T) - 1$ if the predicate of A is a *new* predicate;
  - $w(T) = size(T)$ if the predicate of A is an *old* predicate.
- The weight of a pair (*atom, resultant*), (A,R), $w(A,R)$, is the minimum of the weights of the π-trees of A in P_0, that have R as resultant. (modulo $\simeq$).

In the proof we also make use of trees which have for label clause of their root a clause of P_i but that for the rest are trees of P_0. In particular we need the following.

Definition A.9. We call a tree T of atom A, *descent tree* in $P_i \cup P_0$ if
- the clause label of its root node cl, is in P_i;
- its immediate subtrees $T_1,\dots,T_k$ are trees in P_0;
- if $T_1,\dots,T_k$ are trees of $A_1,\dots,A_k$ and $R_1,\dots,R_k$ are their resultants, then
  (a) $w(A,R) \geqslant w(A_1,R_1) + \cdots + w(A_k,R_k)$;
  (b) $w(A,R) > w(A_1,R_1) + \cdots + w(A_k,R_k)$ if cl satisfies **(F3)**.

The above definition is a generalization of the definition of *descent clause* of [24].

Definition A.10. We call P_i *weight complete* iff for each atom A and resultant R, if there is a π-tree of A in P_0 with resultant R, then there is a descent tree of A with resultant $\simeq$-equivalent to R in $P_i \cup P_0$.

So P_i is weight complete if we can actually reconstruct the resultants semantics of P_0 by using only descent trees in $P_i \cup P_0$.

We can now state the first part of the completeness result.

Proposition A.11. *If P_i is weight complete, then $\mathcal{O}(M_0) \subseteq \mathcal{O}(M_i)$.*

Proof. We now proceed by induction on atom-resultant pairs ordered by the following well-founded ordering $\succ$: $(A,R) \succ (A',R')$ iff
- $w(A,R) > w(A',R')$; or
- $w(A,R) = w(A',R')$, and the predicate of A is a new predicate, while the one of A' is an old one.

Let A, R, be an atom and a resultant such that there exist a π-tree of A in P_0 with resultant R. Since P_i is weight complete, there exists descent tree T_A of A in $P_i \cup P_0$ with resultant R. Let also
– $cl : A_0 \leftarrow c_A \square A_1, \dots A_n$ (in P_i) be the label clause of its root,
– $A_1,\dots,A_{n'}$ be those atoms of cl that have an immediate subtree attached to
– $T_{A_1},\dots,T_{A_{n'}}$ be the immediate subtrees of T_A (in P_0) and $R_{A_1},\dots,R_{A_{n'}}$ be their resultants.

Then, since T_A is a descent tree,

$$w(A,R) \geqslant w(A_1,R_{A_1}) + \ldots + w(A_{n'},R_{A_{n'}}).$$

Now if $w(A,R) > w(A_1,R_{A_1}) + \ldots + w(A_{n'},R_{A_{n'}})$, then $(A,R) \succ (A_j,R_{A_j})$. Otherwise, if $w(A,R) = w(A_1,R_{A_1}) + \ldots + w(A_{n'},R_{A_{n'}})$, by condition (b) on the descent tree, we have that cl does not satisfy (**F3**), by Remark A.7, this implies that the predicate of A is a *new* predicate, while the predicates in $A_1,\ldots,A_{n'}$ are *old* predicates. By the definition of $\succ$, this implies that $(A,R) \succ (A_j,R_{A_j})$.

Hence, by the inductive hypothesis, there exist π-trees $T''_{A_1},\ldots,T''_{A_{n'}}$ of $A_1,\ldots,A_{n'}$ in P_i whose resultants are $R_{A_1},\ldots,R_{A_{n'}}$ (modulo $\simeq$). As usual we assume that the clauses in the T''_{A_i}'s do not share variables with each other and with those in T_A. By Lemma A.2 the tree T''_A, obtained from T_A by replacing each subtree T_{A_j} with T''_{A_j}, is a π-tree of A in P_i with resultant R. This proves the proposition. $\square$

We are now ready to prove our total correctness theorem.

Theorem 5.3 (Total correctness). *Let $M_0 = \langle P_0, Op(M_0)\rangle$ be a module and $M_0,\ldots,M_n$ be a modular transformation sequence. Then*
- $\mathcal{O}(M_0) = \mathcal{O}(M_n)$.

Proof. We will now prove, by induction on i, that for $i \in [0,n]$,
- $\mathcal{O}(M_0) = \mathcal{O}(M_i)$,
- P_i is weight complete.

Base case. We just need to prove that P_0 is weight complete.

Let A be an atom, and R be a resultant such that there is a π-tree of A in P_0 with resultant R. Let T be a minimal π-tree of A in P_0 having R as resultant. T obviously satisfies the condition (a) of Definition A.9. Let cl be the label clause of the root of T, notice that cl satisfies (**F3**) iff its head is an *old* atom, just like the elements of its body. From the definition of weight A.8 and the minimality of T, it follows that condition (b) in Definition A.9 is satisfied as well.

Induction step. We now assume that $\mathcal{O}(P_0) = \mathcal{O}(P_i)$, and that P_i is weight complete.

From Propositions A.6 and A.11 it follows that if P_{i+1} is weight complete then $\mathcal{O}(P_0) = \mathcal{O}(P_{i+1})$. So we just need to prove that P_{i+1} is weight complete.

Let A be an atom, and R be a resultant such that there is a π-tree of A in P_0 with resultant R. Since P_i is weight complete, there exists a descent tree T_A of A in $P_i \cup P_0$ with resultant R.

Let $cl : A_0 \leftarrow c_A \Box A_1, \ldots A_n$ be the label clause of its root. Let us assume that $A_1,\ldots,A_{n'}$ are the atoms of cl that have an immediate π-subtree attached to in T_A, let $T_{A_1},\ldots,T_{A_{n'}}$ be the immediate subtrees of T_A and let $R_{A_1},\ldots,R_{A_{n'}}$ be their resultants. By Lemma A.2 there is no loss in generality in assuming that $T_{A_1},\ldots,T_{A_{n'}}$ are the minimal π-trees of $A_1,\ldots,A_{n'}$ in P_0 that have $R_{A_1},\ldots,R_{A_{n'}}$ as resultants.

We now show that there exists a descent tree of A with resultant R (modulo $\simeq$) in $P_{i+1} \cup P_0$. We have to distinguish various cases, according to what happens to the clause cl when we move from P_i to P_{i+1}.

Case 1: $cl \in P_{i+1}$. That is, cl is not affected by the transformation step. Then T_A is a descent tree of A with resultant R in $P_{i+1} \cup P_0$.

Case 2: *cl is unfolded.* There is no loss in generality in assuming that A_1 is the unfolded atom. In fact, by **(O1)**, the unfolded atom cannot be a π-atom, so it cannot belong to the residual of T_A.

Now, since P_i is weight complete, there exist a descent tree T_{B_0} of A_1 in $P_i \cup P_0$, with clause $d:\ B_0 \leftarrow c_B \square B_1,\ldots,B_m$ (in P_i) as label clause of the root, that has the same resultant (modulo $\simeq$) of T_{A_1}.

Let T'_A be the partial tree obtained from T_A by replacing T_{A_1} with T_{B_0}. T'_A is a π-tree of A in $P_i \cup P_0$; let R'_A be its resultant, by Lemma A.2 and the usual assumption on the variables in the clauses of the subtrees, we have that

$$R \simeq R'_A. \tag{A.6}$$

Let $T_{B_1},\ldots,T_{B_{m'}}$ be the immediate subtrees of T_{B_0}, which we suppose attached to $B_1,\ldots,B_{m'}$, let also $R_{B_1}\ldots R_{B_{m'}}$ be their resultants. By Lemma A.2 there is no loss in generality in assuming that $T_{B_1},\ldots,T_{B_{m'}}$ are the smallest trees of P_0 in their equivalence class.

Let c_{rest} be the conjunction of the global constraints of $T_{B_1},\ldots,T_{B_{m'}},T_{A_1},\ldots,T_{A_{n'}}$, and $\tilde{F}$ be the multiset union of their residuals; we have that

$$R'_A \simeq A \leftarrow (A = A_0) \wedge c_A \wedge (A_1 = B_0) \wedge c_B \wedge c_{\mathrm{rest}} \square \tilde{F}, B_{m'+1},\ldots,B_m, A_{n'+1},\ldots,A_n. \tag{A.7}$$

Since A_1 is the unfolded atom, d is one of the unfolding clauses, it follows that one of the clauses of P_{i+1} resulting from the unfold operation is the following clause:

$$cl':\ A_0 \leftarrow c_A \wedge (A_1 = B_0) \wedge c_B \square B_1,\ldots,B_m, A_2,\ldots,A_n.$$

Now consider the π-tree T''_A of A which is built as follows:
- cl' is the label clause of the root.
- $T_{B_1},\ldots,T_{B_{m'}},T_{A_2},\ldots,T_{A_{n'}}$ are its immediate subtrees.

Its resultant is then

$$R'' = A \leftarrow (A = A_0) \wedge c_A \wedge (A_1 = B_0) \wedge c_B \wedge c_{\mathrm{rest}} \square \tilde{F}, B_{m'+1},\ldots,B_m, A_{n'+1},\ldots,A_n.$$

By (A.6) and (A.7) we have that the resultant of T''_A is R (modulo $\simeq$). Now, in order to prove that T''_A is a descent tree, we have to prove that conditions (a) and (b) in Definition A.9 are satisfied. Now

$$\begin{aligned} w(A,R_A) &\geqslant w(A_1,R_{A_1}) + \cdots + w(A_{n'},R_{A_{n'}}) \quad \text{(since } T_A \text{ is a descent tree)},\\ &\geqslant w(B_1,R_{B_1}) + \cdots + w(B_{m'},R_{B_{m'}}) + w(A_2,R_{A_2}) + \cdots + w(A_{n'},R_{A_{n'}})\\ &\qquad \text{(since } (T_{A_1}) \text{ is a descent tree)} \end{aligned}$$

Moreover, if d satisfies **(F3)** then, by condition (b) in Definition A.9.

$$w(A_1,R_{A_1}) > w(B_1,R_{B_1}) + \cdots + w(B_{m'},R_{B_{m'}}).$$

On the other hand if d does not satisfy (**F3**), then by Remark A.7 the predicate of B_0 and A_1 must be a new predicate; again, by Remark A.7 we have that cl must satisfy (**F3**). It follows that

$$w(A,R_A) > w(A_1,R_{A_1}) + \cdots + w(A_{n'},R_{A_{n'}}).$$

So, in any case, we have that

$$w(A,R_A) > w(T_{B_1}) + \cdots + w(T_{B_{m'}}) + w(T_{A_2}) + \cdots + w(T_{A_{n'}})$$

This proves that T''_A is a descent tree.

Case 3: *cl is removed from P_i via a clause removal operation.* This simply cannot happen: the constraint of cl is a component of the global constraint of T_A and since the latter is satisfiable, so is the first one. Therefore cl cannot be removed from P_i.

Case 4: *cl is split.* Since no π-atom can be split, the split atom may not belong to the residual of T_A, therefore there is no loss in generality in assuming that A_1 is the split atom and that $n' \geqslant 1$.

Since $\mathcal{O}(P_0) = \mathcal{O}(P_i)$, we have that for $i \in [1,n']$ there exist a π-tree S_{A_i} of A_i in P_i, which is similar to T_{A_i}. Let S_A be the π-tree obtained from T_A by substituting its subtrees $T_{A_1},\dots,T_{A_{n'}}$ with $S_{A_1},\dots,S_{A_{n'}}$. From Lemma A.2 and the usual standardization apart of the clauses in the subtrees, it follows that S_A is a π-tree of A in P_i and that S_A is similar to T_A.

Now let $\langle A_1 = B_0 \ ; \ d : \ B_0 \leftarrow c_B \Box B_1,\dots,B_m\rangle$ be the label of the root of S_{A_1}. With this notation, the resultant of T_A (and S_A) has the form

$$A \leftarrow (A = A_0) \wedge c_A \wedge (A_1 = B_0) \wedge c_B \wedge c_{\text{rest}} \Box \mathit{Residual}. \tag{A.8}$$

Since d is a clause of P_i it was certainly used to split A_1 in P_i. Therefore in P_{i+1} we find the clause

– $cl' : \ A_0 \leftarrow c_A \wedge (A_1 = B_0^*) \wedge c_B^* \Box A_1,\dots,A_n$

where $d^*{:}B_0^* \leftarrow c_B^* \Box B_1^*,\dots,B_m^*$ is a renaming of d. Here there in no loss in generality in assuming that the variables of d^* do not occur anywhere else in the trees considered so far. Now, let T'_A be the π-tree of A in $P_{i+1} \cup P_0$ obtained by substituting cl with cl' as label clause of the root of T_A. From (A.8) it follows that the resultant of T'_A is ($\simeq$ equivalent to)

$$A \leftarrow (A = A_0) \wedge c_A \wedge (A_1 = B_0) \wedge c_B \wedge (A_1 = B_0^*) \wedge c_B^* \wedge c_{\text{rest}} \Box \mathit{Residual}.$$

Since d^* is a renaming of d, and since its variables do not occur anywhere else in T'_A, in the above formula the subexpression $(A_1 = B_0^*) \wedge c_B^*$ is already implied by the fact that the expression contains $(A_1 = B_0) \wedge c_B$, and therefore it may be removed from the constraint. So, from (A.8) it follows that T'_A is similar to T_A. Now, in order to prove the thesis we only need to prove that T'_A is a descent tree, i.e. it satisfies conditions (a) and (b) of Definition A.9. This follows immediately from the fact that the subtrees of T_A and T'_A are the same ones (and T_A is a descent tree) and the fact that cl' satisfies (**F3**) iff cl does.

Case 5: *The constraint of cl is replaced.* The first part of this proof is similar to the one of the previous case. Since $\mathcal{O}(P_0) = \mathcal{O}(P_i)$, we have that for $i \in [1, n']$ there exist a π-tree S_{A_i} of A_i in P_i, which is similar to T_{A_i}. Let S_A be the π-tree obtained from T_A by substituting its subtrees $T_{A_1}, \dots, T_{A_{n'}}$ with $S_{A_1}, \dots, S_{A_{n'}}$. From Lemma A.2 and the usual standardization apart of the subtrees it follows that S_A is a π-tree of A in P_i and that S_A is similar to T_A.

Let $c_{A_1}, \dots, c_{A_{n'}}$ be the internal constraints of $S_{A_1}, \dots, S_{A_{n'}}$ and $\tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}$ be their residuals. With this notation, the resultant of T_A (and S_A) is

$$A \leftarrow (A = A_0) \wedge c_A \wedge c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n.$$

Recall that by the assumption that the trees are standardized apart, for distinct $i, j \in [1, n]$, we have that $Var(c_{A_i} \Box \tilde{F}_{A_i}) \cap Var(c_{A_j} \Box \tilde{F}_{A_j}) \subseteq Var(A_i) \cap Var(A_j)$. Then, from the existence of $S_{A_1}, \dots, S_{A_{n'}}$ and from Remarks A.4 and A.5 it follows that there exist a derivation

$$A_1, \dots, A_n \overset{P_i}{\leadsto} c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n.$$

Now, let the result of the constraint replacement operation be the clause
– cl' : $A_0 \leftarrow c'_A \Box A_1, \dots, A_n$.
From the applicability conditions of the constraint replacement operation it follows that

$$\begin{aligned} &A_0 \leftarrow (A = A_0) \wedge c_A \wedge c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n, \\ &\quad \simeq A_0 \leftarrow (A = A_0) \wedge c'_A \wedge c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n. \end{aligned} \tag{A.9}$$

Now, let T'_A be the tree obtained from T_A by replacing the clause label if its root, cl, with cl'. Its resultant is

$$A \leftarrow (A = A_0) \wedge c'_A \wedge c_{A_1} \wedge \dots \wedge c_{A_{n'}} \Box \tilde{F}_{A_1}, \dots, \tilde{F}_{A_{n'}}, A_{n'+1}, \dots, A_n$$

and from (A.9) it follows that T'_A is similar to T_A.

Now, in order to prove the thesis we only need to prove that T'_A is a descent tree, i.e., that it satisfies conditions (a) and (b) of Definition A.9; but this follows immediately from the fact that the subtrees of T_A and T'_A are the same ones (and T_A is a descent tree) and the fact that cl' satisfies (**F3**) iff cl does.

Case 6: *cl is folded.* Let $\{A_1 = C_1, \dots, A_{n'} = C_{n'}\}$ be the label equations of the root nodes of $T_{A_1}, \dots, T_{A_{n'}}$, let also c_{rest} be the conjunction of the remaining internal equations (label equations + clause constraints) of $T_{A_1}, \dots, T_{A_{n'}}$; finally, let $\tilde{F}$ be the residual of $T_{A_1}, \dots, T_{A_{n'}}$. We have that

$$R \simeq A \leftarrow (A = A_0) \wedge c_A \wedge \left(\bigwedge_{j=1}^{n'} A_j = C_j \right) \wedge c_{\text{rest}} \Box \tilde{F}, A_{n'+1}, \dots, A_n. \tag{A.10}$$

Now let the folding clause (in P_{new}) be

d : $B_0 \leftarrow B_1, \dots, B_m$.

There is no loss in generality in assuming that there exists an index k such that $A_k,\dots,A_{k+m}$ are the folded atoms, so for $j \in [1,m]$, A_{k+j} and B_j are unifiable atoms. The result of the folding operation is then

$$cl' : A_0 \leftarrow c_A \wedge e \square A_1,\dots A_k, B_0, A_{k+m+1},\dots A_{n'}.$$

Now notice that of the atoms of cl that are going to be folded, $A_{k+1},\dots,A_{n'}$ are the ones that have an immediate subtree attached to in T_A; these atoms correspond to $B_1,\dots,B_{n'-k}$ in d (we should also consider explicitly the cases when they all have or have not a subtree attached to, i.e., the cases in which $n' < k$ or $n' \geqslant m+k$. However these are easy corollaries of the general case, so we now assume that $k \leqslant n' < m+k$). Now let T_{B_0} be the π-tree of B_0 in P_0 built as follows:

– $d' : B'_0 \leftarrow c'_B \square B'_1,\dots,B'_m$. (an appropriate renaming of d) is the label clause of its root node,
– $B_0 = B'_0$ is then the label equation of its root node,
– $T_{B'_1},\dots,T_{B'_{n'-k}}$ are its immediate subtrees, which are obtained, as explained in Lemma A.3, from the trees $T_{A_{k+1}},\dots,T_{A_{n'}}$ by replacing A_{k+j} with B'_j in the l.h.s. of the label equations of their root nodes.
– $B'_{n'-k+1},\dots,B'_m$ is consequently the residual of its root node.

Finally, let T''_A be the π-tree of A in $P_{i+1} \cup P_0$ which is built as follows:

– cl' is the label clause if its root (and this is a clause in P_{i+1}).
– $T_{A_1},\dots,T_{A_{k-1}},T_{B_0}$ are its immediate subtrees (in P_0).

Let R'' be its resultant, we have that

$$R'' = A \leftarrow c_{\text{tot}} \square \tilde{F}, B'_{n'-k+1},\dots,B'_m, A_{k+m+1},\dots,A_n \tag{A.11}$$

where $\tilde{F}$ is the (multiset) union of the residuals of $T_{A_1},\dots,T_{A_{k-1}},T_{B_0}$ and c_{tot} is

$$(A = A_0) \wedge c_A \wedge e \wedge (B_0 = B'_0) \wedge c'_B \wedge \left(\bigwedge_{j=1}^{k} A_j = C_j\right) \wedge \left(\bigwedge_{j=k+1}^{n'} B'_{j-k} = C_j\right) \wedge c_{\text{rest}}$$

By **(F1)** this becomes:

$$(A = A_0) \wedge c_A \wedge (B_0 = B'_0) \wedge \left(\bigwedge_{j=1}^{m} B_j = B'_j\right) \wedge \left(\bigwedge_{j=1}^{k} A_j = C_j\right)$$
$$\wedge \left(\bigwedge_{j=k+1}^{n'} B'_{j-k} = C_j\right) \wedge c_{\text{rest}}. \tag{A.12}$$

As we did in Proposition A.6, we now show that we can drop the constraint $B_0 = B'_0$. First notice that since B'_0 is a renaming of B_0, then $B_0 = B'_0$ can be reduced to a conjunction of equations of the form $x = y$, where x and y are distinct variables. So suppose that for some x, y, $B_0 = B'_0$ implies that $x = y$, then either $x = y$ is already implied by the constraint $(\bigwedge_{j=1}^{m} B_j = B'_j)$, or the variables x and y do not occur anywhere else in (A.12), nor in R''.

Thus c_{tot} can be rewritten as follows:

$$(A = A_0) \wedge c_A \wedge \left(\bigwedge_{j=1}^{m} B_j = B'_j \right) \wedge \left(\bigwedge_{j=1}^{k} A_j = C_j \right) \wedge \left(\bigwedge_{j=k+1}^{n'} B'_{j-k} = C_j \right) \wedge c_{\text{rest}}$$

By making explicit the constraint $(\bigwedge_{j=1}^{m} B_j = B'_j)$ and comparing the result with (A.10) we see that T''_A is a π-tree of A in $P_{i+1} \cup P_0$ with resultant R (modulo $\simeq$). We now need only to prove that T''_A is a descent tree, i.e. it satisfies the conditions (a), (b) of the Definition A.9.

Let R_{B_0} be the resultant of T_{B_0}. Since d is the folding clause, the predicate of B_0 must be a *new* predicate, while the predicates of $B_1, \ldots, B_m$ have to be *old* predicates. Moreover, by condition **(F2)**, any proof tree of B_0 in P_0 whose global constraint is consistent with $c_a \wedge e$ must have (a renaming of) d as label clause of the root. By Definition A.8 we then have that

$$w(B_0, R_{B_0}) \leqslant w(T_{B_1}) + \cdots + w(T_{B_{n'-k}}). \tag{A.13}$$

Moreover, for $j \in [1, n'-k]$, $w(T_{A_{k+j}}) = w(T_{B_j})$, and, since T_A is a descent tree and the clause of its root node satisfies **(F3)**, by Definition A.8 we have that

$$\begin{aligned}
w(A,R) &> w(A_1, R_{A_1}) + \cdots + w(A_{n'}, T_{R_{n'}}) \\
&= w(A_1, R_{A_1}) + \cdots + w(A_k, R_{A_k}) + w(A_{k+1}, R_{A_{k+1}}) + \cdots + w(A_{n'}, R_{A_{n'}}) \\
&= w(A_1, R_{A_1}) + \cdots + w(A_k, R_{A_k}) + w(T_{A_{k+1}}) + \cdots + w(T_{A_{n'}}) \\
&\quad \text{(by the minimality of the } T_{A_j}\text{)} \\
&= w(A_1, R_{A_1}) + \cdots + w(A_k, R_{A_k}) + w(T_{B_1}) + \cdots + w(T_{B_{n'-k}}) \\
&\quad \text{(by the definition of } T_{B_j}\text{)} \\
&\geqslant w(A_1, R_{A_1}) + \cdots + w(A_k, R_{A_k}) + w(B_0, R_{B_0}) \quad \text{(by (18)).}
\end{aligned}$$

Thus T''_A satisfies conditions (a) and (b) of Definition A.9. □

References

[1] K.R. Apt., Introduction to logic programming, in: J. van Leeuwen, ed., *Handbook of Theoretical Computer Science, Vol. B: Formal Models and Semantics* (Elsevier, Amsterdam and The MIT Press, Cambridge, 1990) 495–574.

[2] C. Aravidan and P.M. Dung, On the correctness of Unfold/Fold transformation of normal and extended logic programs, Tech. report, Division of Computer Science, Asian Institute of Technology, Bangkok, Thailand, April 1993.

[3] N. Bensaou and I. Guessarian, Transforming constraint logic programs, in: F. Turini, ed., *Proc. 4th Workshop on Logic Program Synthesis and Transformation* (1994).

[4] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi and M.C. Meo, Differential logic programming, in: *Proc. 20th Ann. ACM Symp. on Principles of Programming Languages* (ACM, New York, 1993) 359–370.

[5] A. Bossi and N. Cocco, Basic transformation operations which preserve computed answer substitutions of logic programs, *J. Logic Programming* **16** (1 and 2) (1993) 47–87.

[6] A. Bossi, M. Gabbrielli, G. Levi and M.C. Meo, A compositional semantics for logic programs, *Theoret. Comput. Sci.* **122** (1–2) (1994) 3–47.

[7] M. Bugliesi, E. Lamma and P. Mello, Partial deduction for structured logic programs, *J. Logic Programming* **16** (1–2) (1993) 89–122.

[8] M. Bugliesi, E. Lamma and P. Mello, Modularity in logic programming, *J. Logic Programming* **19–20** (1994) 443–502.

[9] R.M. Burstall and J. Darlington, A transformation system for developing recursive programs, *J. Assoc. Comput. Math.* **24**(1) (1977) 44–67.

[10] K.L. Clark and S. Sickel, Predicate logic: a calculus for deriving programs, in: *Proc. IJCAI'77*, (1977) 419–420.

[11] S. Etalle and M. Gabbrielli, A transformation system for modular CLP programs, in: *Proc. 20th Internat. Conf. on Logic Programming*, ICLP 95 (MIT Press, Cambridge, MA, 1995).

[12] S. Etalle and M. Gabbrielli, The replacement operation for CLP modules, in: *Proc. ACM SIGPLAN Symp. Partial Evaluation and Semantics-Based Program Manipulation*, PEPM '95 (ACM Press, 1995) 168–177.

[13] M. Gabbrielli, G.M. Dore and G. Levi, Observable semantics for constraint logic programs, *J. Logic Comput.* **5**(2) (1995) 133–171.

[14] M. Gabbrielli and G. Levi, Modeling answer constraints in constraint logic programs, in: K. Furukawa, ed., *Proc. 8th Internat. Conf. on Logic Programming* (MIT Press, Cambridge, MA, 1991) 238–252.

[15] H. Gaifman and E. Shapiro, Fully abstract compositional semantics for logic programs, in: *Proc. 16th Ann. ACM Symp. on Principles of Programming Languages* (ACM, New York, 1989) 134–142.

[16] P.A. Gardner and J.C. Shepherdson, Unfold/fold transformations of logic programs, in: J-L. Lassez and G. Plotkin, eds., *Computational Logic: Essays in Honor of Alan Robinson* (MIT Press, Cambridge, MA, 1991).

[17] C.J. Hogger, Derivation of logic programs, *J. Assoc. Comput. Mach.* **28**(2) (1981) 372–392.

[18] J. Jaffar and J.-L. Lassez, Constraint logic programming, Tech. Report, Department of Computer Science, Monash University, June 1986.

[19] J. Jaffar and J.-L. Lassez, Constraint logic programming, in: *Proc. 14th Ann. ACM Symp. on Principles of Programming Languages* (ACM, New York, 1987) 111–119.

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

[21] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap, An abstract machine for CLP($\mathscr{R}$), in: *Proc. ACM SIGPLAN Symp. on Programming Language Design and Implementation (PLDI)*, (ACM, New York, 1992) 128–139.

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

[23] N. Jørgensen, K. Marriott and S. Michaylov, Some global compile-time optimizations for CLP($\mathscr{R}$), in: V. Saraswat and K. Ueda, eds., *ILPS'91: Proc. the Internat. Logic Programming Symposium*, (San Diego, October 1991) (MIT Press, Cambridge, MA, 1991) 420–434.

[24] T. Kawamura and T. Kanamori, Preservation of stronger equivalence in unfold/fold logic programming transformation, in: *Proc. Internat. Conf. on 5th Generation Computer Systems* (Institute for New Generation Computer Technology, Tokyo, 1988) 413–422.

[25] H.J. Komorowski, Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog, in: *9th ACM Symp. on Principles of Programming Languages* (Albuquerque, NM 1982) 255–267.

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

[27] J.-L. Lassez, M.J. Maher and K. Marriott, Unification revisited, in: J. Minker, ed., *Foundations of Deductive Databases and Logic Programming* (Morgan Kaufmann, Los Altos, CA., 1988) 587–625.

[28] J.W. Lloyd, *Foundations of Logic Programming* (Springer, Verlag, Berlin, 2nd ed., 1987).

[29] M.J. Maher, A transformation system for deductive databases with perfect model semantics, *Theoret. Comput. Sci.* **110** (1993) 377–403.

[30] K. Marriott and P.J. Stuckey, The 3 r's of optimizing constraint logic programs: Refinement, removal and reordering, in: *POPL'93*: *Proc. ACM SIGPLAN Symp. on Principles of Programming Languages*, (Charleston, January 1993).

[31] R.A. O'Keefe, Towards an algebra for constructing logic programs, in: *Proc. IEEE Symp. on Logic Programming* (1985) 152–160.

[32] A. Pettorossi and M. Proietti, Transformation of logic programs: Foundations and techniques, *J. Logic Programming* **19**(20) (1994) 261–320.

[33] T. Sato, Equivalence-preserving first-order unfold/fold transformation system, *Theoret. Comput. Sci.* **105**(1) (1992) 57–84.

[34] H. Seki, Unfold/fold transformation of stratified programs, *Theoret. Comput. Sci.* **86**(1) (1991) 107–139.

[35] H. Seki, Unfold/fold transformation of general logic programs for the well-founded semantics, *J. Logic Programming* **16** (1 and 2) (1993) 5–23.

[36] H. Tamaki and T. Sato, A transformation system for logic programs which preserves equivalence, Tech. Report ICOT TR-018, ICOT, Tokyo, Japan, August 1983.

[37] H. Tamaki and T. Sato, Unfold/fold transformations of logic programs, in: Sten-Åke Tärnlund, ed., *Proc. 2nd Internat. Conf. on Logic Programming* (1984) 127–139.