Transformation and Analysis of (Constraint) Logic Programs

Sandro Etalle

Transformation and Analysis of (Constraint) Logic Programs

ILLC Dissertation Series 1995-7

For further information about ILLC-publications, please contact

Institute for Logic, Language and Computation
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
phone: +31-20-5256090
fax: +31-20-5255101
e-mail: illc@fwi.uva.nl

Transformation and Analysis of (Constraint) Logic Programs

Academisch Proefschrift

ter verkrijging van de graad van doctor aan de
Universiteit van Amsterdam,
op gezag van de Rector Magnificus
Prof.dr P.W.M. de Meijer
in het openbaar te verdedigen in de
Aula der Universiteit
(Oude Lutherse Kerk, ingang Singel 411, hoek Spui)
op woensdag 7 juni 1995 te 9.00 uur

door

Sandro Etalle

geboren te Milaan.

1^e Promotor: Prof.dr. K. R. Apt
Faculteit Wiskunde en Informatica
Universiteit van Amsterdam
Plantage Muidergracht 24
1018 TV Amsterdam
en
CWI - Centrum voor Wiskunde en Informatica
Kruislaan 413
1098 SJ Amsterdam

2^e Promotor: Prof. A. Bossi
Università della Calabria
Rende
Italïe

This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 89.00026.69.

Dipartimento di Informatica
Universtià di Genova
Italïe
sandro@disi.unige.it

Maybe some additional info on the production of the dissertation.
Printed and bound by Angassini Arti Grafiche, Genova.

ISBN: 90-74795-27-7

Contents

Acknowledgments

I guess it is now appropriate to thank those (many) people that made possible such an unlikely event as my promotion. In the known universe there's hardly enough space for doing this the way it is supposed to, therefore I'll just mention those that had a *direct* role in my writing of this thesis. Some people say that one year spent looking for the right master is not a wasted year. Personally, I had more luck, as I found myself with a team of three exceptional advisors. I'm talking about Krzysztof Apt, Annalisa Bossi and Nicoletta Cocco. They took me, they taught me, they guided me, they lectured me, they helped me, they stimulated me, they supported me, they scared me, they cheered me up, and (mainly in the ceffeebreaks), they explained me how to write a paper on Logic programming. They had enough of me.

Nicoletta is also coauthor of chapters 2 and 4, while Maurizio "H.B." Gabbrielli has been the coauthor of chapters 6 and 7.

I want to thanks a couple of people who, despite everything that happened, in fortune and in misery, in the (few) lucky periods and in the (endless) unlucky ones, always remained true friends. Max and Giuseppe are friends and, alas, competitors. Nevertheless, they resisted everything and helped me out in doing all the massive bureaucratic stuff that the average Italian PhD student is required to do, but that I couldn't do myself because I was In Amsterdam. This included job and fellowship applications, and a whole bunch of stuff for which we were in direct competition with each other. Some people were killed for much less.

Remaining a bit longer in Padova, I want to thank all the members of the apartments in "via Pio X". Girls, you've been hosting me, always, even when I had no whiskey with me. You've rebuilt me from scratch in many occasions (you've also smashed me to pieces more than once, but that's a different story) and you've been making me feel at home in situations in which I really had none. A big kiss to everyone.

Moving on to Amsterdam, where I've spent most of my last few years, I want to thank Saar and Floor, who, despite everything, always, always, always remained two great friends. I really have no words for saying how important you've been for me. Then I want to thank Maurizio who, in music, conversations, and logic programming, has been the colleague I always dreamed and never dared to ask. Aino has been a fantastic house-mate and a great buddy in the endless nights at the *Bourbon Street*, the real research center of Amsterdam.

Last but not least I would like to hug again the whole troop of the *Sweelinck Orkest*. They gave a meaning to things that had none, and that's not easy. I'd really like to thank them one by one: Ingrid, Feico, Hester, Diederick, Anne-Marije, Steven, Geesje, Norbert, Marije, Maarten, Simonka, and there's another 70 of them so I'd better stop here.

Genova, Sandro Etalle
juni, 1995.

Chapter 1

Program's transformation

It is well-known that a good program has to be both *correct* (wrt a given specification) and *efficient*. A better program is also *inexpensive*. These three aspects are often in contrast with each other. On one hand, it is often the case that efficient programs (and algorithms) are so complicated that they're difficult to prove correct. On the other hand, the ones which are easy to prove correct are those that are simple and clear, which are often outperformed by more complex ones. Finally, because of the increase in program's size that the modern architectures allow (and require), and the decrease in the hardware's cost, the impact that cost of software has in the overall (software+hardware) expenses is more and more increasing. Of course the more complicated a program the more likely it is to be expensive.

Source-to-source program's transformation provide a methodology for deriving correct and possibly efficient and inexpensive programs starting from a specification. The underlying idea is to separate the problem of correctness from the issue of efficiency. To this end, the process of developing a (large) application is divided into two phases. First the programmer writes an *initial* program which may be simple and inefficient, but whose correctness is easily checkable. Secondly, this program is transformed into a more performing one. This latter is actually an *optimization* phase. This may take several steps, may return a program which is written in the same language of the original one and has to fulfill the following three important requirements:

First, It must be *effective*. In principle the optimization phase has to make up for the efficiency we have lost by writing a program which is (inexpensive and) easy to prove correct. In the logic programming area several strategies have been devised in order to achieve such an optimization. Among them we should mention program's *specialization* and *partial evaluation* [60]. The techniques program's specialization allow to obtain a more efficient program by exploiting the fact that the program itself will always be employed in a certain context, that is, together with an input that satisfies certain preconditions. In the Logic programming area, these techniques have been studied by Bossi et al. [19] and by Gallagher et al. [46, 45, 33]. An important special case of program's specialization is the technique of *partial evaluation* (also referred to as *partial deduction*). This methodology can be applied when a part of the input is known in advance (say, at compile-time), and can be regarded as an application of Kleene's s-m-n Theorem.

Secondly, the optimization phase must be at least *semi-automatizable*. Indeed, the task of transforming a program must be much more affordable than the one of writing one from scratch, and therefore it cannot be done "by hand".. To achieve this, the optimization phase is usually broken into several steps, in each step a *basic transformation* operation is applied. In the field of Logic programming, the most prominent basic operations are unfolding, folding and replacement which are the operations studied in this thesis. The applicability of each transformation step is usually automatically checkable; however, in order to achieve effectiveness, the sequence of steps to follow is determined by a *strategy* which may need human supervision.

It must be *correct*. This is the issue we'll mostly address in this thesis. Technically, we say that a transformation is observationally correct if the resulting program has the same behavior of the initial one, i.e. if the two programs are observationally equivalent. In this way, assuming that the initial program is correct, the problem of the correctness of the resulting program is reduced to the problem of the correctness of the transformation sequence, and, ultimately, to the problem of the correctness of each basic transformation operation. Being available a formal definition of semantics of, we say that the transformation is correct if the semantics of the resulting program is equal to the semantics of the initial one. Indeed, one reason why program's transformation (at the source-code level) are so popular in field such as logic and functional programming is that in these areas there exists elegant and mathematical methodologies for determining the semantics of a program. These *declarative* semantics have been (often) proven equivalent to the operational ones, and, being defined in mathematical terms, are much more suitable to be used for verifying a transformation's correctness.

In this thesis we'll focus on source-to-source program's transformation, specifically in the field of logic programming. Therefore, when we talk of transformation we'll actually refer to this more restrictive kind. Other forms of program's transformation which we won't cover here are the *compilation* of a program into machine code and the *synthesis* of programs from a given specification language. However, for this latter case, it should be mentioned that the techniques and the basic operations used for program's synthesis are often the same used and addressed in this thesis .

Unfold/Fold Transformations

Program's transformation techniques began to be studied in the early 70's. However, the first well-known formalization appeared in 1977, with the work of Burstall and Darlington [25]. [25] introduced for the first time the operations of unfolding and folding, which allowed the development of recursive programs. Since then a large body of literature has been produced on the subject. The transformation system was then adapted to logic programs both for program synthesis [30, 50], and for program specialization and optimization [60]. Soon later, Tamaki and Sato [96] proposed an elegant framework for the transformation of logic programs based on unfold/fold rules. Tamaki-Sato's system also included a replacement operation, which is a topic we'll address in the sequel. The operation of unfolding, consists in applying in all possible ways a resolution step to an atom in the body of a clause. Unfolding is the fundamental operation for partial evaluation [66] and is usually applied only to positive literals (an exception is [11]). Being such a "natural" operation, unfolding is correct wrt practically all the semantics available for logic programs.

Folding, can be regarded as the inverse of unfolding, as long as one single unfolding is possible. The main feature of this operation is that it can introduce recursion in the body of a clause, therefore allowing optimizations which are certainly non-trivial. On the other hand, if applied indiscriminately, this operation may well introduce infinite loops in the program, and therefore its applicability has to be restricted by suitable applicability conditions. Tamaki and Sato provided conditions which ensure the preservation of the least Herbrand model semantics (as proven in [96] itself) and of the computed answer substitution semantics (as proven by Kawamura and Kanamori in [58]). However, Seki showed that the system does not preserve the finite failure set of the initial program, this problem is particularly relevant when we transform *normal* logic programs, that is, programs which use the negation operator in the bodies of the clauses. In [91], Seki provides 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 correctness for the unfold/fold system w.r.t. the various semantics available for normal programs. Just to cite the most relevant works, we should mention Sato's [88] (in which he adapts the technique to full first-order programs), Maher's [67, 69], and the works of Gardner and Shepherdson [47], Aravidan and Dung [12], Seki [92], Bossi and Cocco [18] and Bensaou and Guessarian [14].

The replacement operation

Replacement is possibly the most general transformation operation for logic programs. Syntactically, it consist in substituting a conjunction of literals $\tilde{C}$ with another conjunction $\tilde{D}$ in the body of a clause. Clearly, for the syntactic point of view, this operation is able to imitate most of the other transformation operation. For instance, it can imitate the folding operation, and it can introduce recursion in the bodies of the clauses. On the other hand, being so general, if we want it to be also somehow correct, we have to restrict its use by suitable applicability conditions. These applicability conditions may vary according to the semantic properties that we are interested in preserving along the transformation. In the field of logic programs, the *replacement* operation has been studied for the first time in the context of *definite* programs by Tamaki and Sato in [96]. Later, developments were provided by the works of Sato himself [88], Gardner and Shepherdson [47], Bossi, Cocco and Etalle [20], Proietti and Pettorossi [79, 80] Maher [67, 69] Cook and Gallagher [32] and Bensaou and Guessarian [14]. For the technical details of each of these approach we refer to In section 7.5.

The applicability conditions for the replacement operations are usually undecidable. Indeed this operation is to be regarded as a more abstract operation than, for instance, unfolding and folding. We could say that while unfolding and folding are syntactic-driven operation, replacement in semantics-driven. The interest in the study of the applicability conditions of replacement is due to the fact that (a) it is an extremely powerful operation, and allows optimizations which have been proven impossible with unfold-fold transformations, and (b) it can be regarded as the operation that lies behind the folding one: i.e. as we'll show in this thesis folding can be often seen as a particular case of replacement in which the applicability conditions are syntactically checkable.

A basic applicability condition for the replacement operation, which is common to all the approaches mentioned above, is that the replacing conjunction has to be semantically equivalent to the replaced one. Unfortunately, this requirement alone is not sufficient to guarantee the correctness of the operation. The main problem is that the operation may still introduce an infinite loop, in which case the final program is likely not to have the same expressiveness of the initial one. The approaches in the literature differ a lot in the method for avoiding the introduction of a loop. In this thesis, in chapters 4 and 7 we'll propose new applicability conditions for it.
****

The system was then extended by Seki [91] 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 [88]. Related work has been done by Maher [69], Gardner and Shepherdson [47], Aravidan and Dung [12], Seki [92], Bossi and Cocco [18] and Bensaou and Guessarian [14].

The replacement operation

Replacement is possibly the most general transformation operation for logic programs. Syntactically, it consist in substituting a conjunction of literals $\tilde{C}$ with another conjunction $\tilde{D}$ in the body of a clause. Clearly, for the syntactic point of view, this operation is able to imitate most of the other transformation operation. To start with, it can imitate the folding operation. On the other hand, being so general, if we want it to be also somehow correct, we have to restrict its use by suitable applicability conditions. These applicability conditions may vary according to the semantic properties that we are interested in preserving along the transformation. In the field of logic programs, the *replacement* operation has been studied for the first time in the context of *definite* programs by Tamaki and Sato in [96]. Later, developments were provided by the works of Sato himself [88], Gardner and Shepherdson [47], Bossi, Cocco and Etalle [20], Proietti and Pettorossi [79, 80] Maher [67, 69] Cook and Gallagher [32] and Bensaou and Guessarian [14]. For the technical details of each of these approach we refer to In section 7.5.

********************************** and after that it has been rather neglected by people working on program transformations apart from Sato himself [88], Maher [67] and Gardner and Shepherdson [47]. Replacement consists in substituting a conjunction of literals, in the body of a clause, with another conjunction. It is a very general transformation able to mimic many other operations, such as thinning, fattening [18] and folding.

Some *applicability conditions* are necessary in order to ensure the preservation of the semantics through the transformation. Such conditions depend on the semantics we associate to the program. In the literature we find different proposal. In [96] definite programs are considered; the applicability condition requires the replaced atom C and the replacing atom D to be logically equivalent in P and that the size of the smallest proof tree for C is greater or equal to the size of the smallest proof tree for D. Gardner and Shepherdson, in [47], give different conditions for preserving procedural (SLDNF) semantics and the declarative one. Such conditions are based on Clark's (two valued) completion of the program. Also Maher, in [67, 69], studies replacement wrt Success set, Finite Failure Set, Ground Finite Failure Set and Perfect Model semantics. Sato, in [88], considers also replacement of formulas whose equivalence can be proved in first order logic and does not depend on the program. Bossi et al. have studied the correctness of this operation wrt the S-semantics for definite programs [20], and the Well-Founded semantics for normal programs [38].

Origin of the chapters

Chapter 2 and 4 will appear in A. Bossi, N. Cocco, and S. Etalle. Simultaneous replacement in normal programs. *Journal of Logic and Computation*, 1995. A preliminary version appeared in Transforming Normal Programs by Replacement. In A. Pettorossi, editor, *Meta Programming in Logic - Proceedings META'92*, volume 649 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, Berlin, 1992. Chapter 3 appears in A. Bossi and S. Etalle. Transforming Acyclic Programs. ACM Transactions on Programming Languages and Systems, Vol 16, n. 4, July 1994, pages 1081-1096. Chapter 5 appears in A. Bossi and S. Etalle. More on Unfold/Fold Transformations of Normal Programs: Preservation of Fitting's Semantics. In F. Turini, editor, *Proc. Fourth International Workshop on Meta Programming in Logic*. Springer-Verlag, Berlin, 1994. An extended abstract of chapter 6 appears in S. Etalle and M. Gabbrielli. Modular Transformations of CLP Programs. In L. Sterling, editor, *Proc. Twelfth Int'l Conf. on Logic Programming*, 1995. An extended abstract of chapter 7 appears in S. Etalle and M. Gabbrielli. The Replacement Operation for CLP Modules. In N. Jones, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM '95)*, 1995. Chapter 8 appears in S. Etalle. More (on) Unification-Free Prolog Programs. CWI Technical Report CS-R9454, September 1994, Amsterdam.

Chapter 2

The semantics of normal logic programs

In this chapter, we define the notation and we give the definitions of the basic declarative semantics for normal programs, that is, programs which may employ the negation operations in the bodies of the clauses. In particular we'll introduce Kunen's, and Fitting's semantics. We'll also provide a new result, which characterizes program's equivalence wrt Kunen's semantics.

2.1 Preliminaries

We assume that the reader is familiar with the basic concepts of logic programming; throughout the chapter we use the standard terminology of [65] and [3]. We consider *normal programs*, that is finite collections of *normal rules*, $A \leftarrow L_1, \ldots, L_m$. where A is an atom and $L_1, \ldots, L_m$ are literals. Symbols with a $\sim$ on top denote tuples of objects, for instance $\tilde{x}$ denotes a tuple of variables $x_1, \ldots, x_n$, and $\tilde{x} = \tilde{y}$ stands for $x_1 = y_1 \wedge \ldots \wedge x_n = y_n$. We also adopt the usual logic programming notation that uses "," instead of $\wedge$, hence a conjunction of literals $L_1 \wedge \ldots \wedge L_n$ will be denoted by $L_1, \ldots, L_n$ or by $\tilde{L}$.

In this chapter (and every time we'll deal with normal programs) we'll always work with three valued logic: the truth values are then *true*, *false* and *undefined*. We adopt the truth tables of [59], which can be summarized as follows: the usual logical connectives have value *true* (or *false*) when they have that value in ordinary two valued logic for all possible replacements of *undefined* by *true* or *false*, otherwise they have the value *undefined*.

Three valued logic allows us to define connectives that do not exist in two valued logic. In particular in the sequel we use the symbol $\Leftrightarrow$ corresponding to Lukasiewicz's operator of "having the same truth value": $a \Leftrightarrow b$ is *true* if a and b are both *true*, both *false* or both *undefined*; in any other case $a \Leftrightarrow b$ is *false*. As opposed to it, the usual $\leftrightarrow$ is *undefined* when one of its arguments is *undefined*.

In some cases we restrict our attention to formulas which we consider "well-behaving" in the three valued semantics. Next definition is intended for characterizing such formulas.

Definition 2.1.1

- A logic connective $\diamond$ is *allowed* iff the following property holds: when $a \diamond b$ is *true* or *false* then its truth value does not change if the interpretation of one of its argument is changed from *undefined* to *true* or *false*.
- A first order formula is *allowed* iff it contains only allowed connectives. □

Note that any formula containing the connective $\Leftrightarrow$ is not allowed, while formulas built with the usual logic connectives are allowed.

Allowed formulas can be seen as monotonic functions over the lattice on the set $\{$*undefined*, *true*, *false*$\}$ which has *undefined* as bottom element and *true* and *false* are not comparable.

Completion for Normal Programs

In this chapter we consider as semantics for a normal logic program P the set of all logical consequences of its completion $Comp(P)$, [28]; the problem of the consistency of $Comp(P)$ is here avoided by using three valued logic instead of the classical two valued.

The usual Clark's completion definition is extended to three valued logic by replacing $\leftrightarrow$, in the completed definitions of the predicates, with $\Leftrightarrow$. This saves $Comp(P)$ from the inconsistencies that it can have in two valued logic. For example the program $P = \{p \leftarrow \neg p.\}$ has $Comp(P) = \{p \Leftrightarrow \neg p\}$ which has a model with p *undefined*.

Definition 2.1.2 Let P be a program and $p(\tilde{t}_1) \leftarrow \tilde{B}_1, \ldots, p(\tilde{t}_r) \leftarrow \tilde{B}_r$ be all the clauses which define predicate symbol p in P. The *completed definition* of p is

- $p(\tilde{x}) \Leftrightarrow \bigvee_{i=1}^{r} \exists\, \tilde{y}_i\ (\tilde{x} = \tilde{t}_i) \wedge \tilde{B}_i.$

where $\tilde{x}$ are new variables and $\tilde{y}_i$ are the variables in $p(\tilde{t}_i) \leftarrow \tilde{B}_i$. If P contains no clause defining p, then the completed definition of p is

- $p(\tilde{x}) \Leftrightarrow false.$ □

The completed definition of a predicate is a first order formula that contains the equality symbol; hence, in order to interpret "=" correctly, we also need an equality theory. First recall that a *language* $\mathcal{L}$ is determined by a set of function and predicate symbols of fixed arities. Constants are treated as 0-ary function symbols.

Definition 2.1.3 $\mathrm{CET}_{\mathcal{L}}$, *Clark's Equality Theory for the language* $\mathcal{L}$, consists of the axioms:

- $f(x_1, \ldots, x_n) \neq g(y_1, \ldots, y_m)$ for all distinct f, g in $\mathcal{L}$;
- $f(x_1, \ldots, x_n) = f(y_1, \ldots, y_n) \rightarrow (x_1 = y_1) \wedge \ldots \wedge (x_n = y_n)$ for all f in $\mathcal{L}$;
- $x \neq t(x)$ for all terms $t(x)$ distinct from x in which x occurs;

together with the usual *equality axioms*, that are needed in order to interpret correctly "=" , which are *reflexivity, symmetry, transitivity,* and $(\tilde{x} = \tilde{y}) \rightarrow (f(\tilde{x}) = f(\tilde{y}))$ for all functions and predicate symbols f in $\mathcal{L}$. □

Note that "=" is always interpreted as two valued, since an expression of the form $t = s$, with t, s ground terms cannot be *undefined.*

Definition 2.1.4 The *Clark's completion of* P wrt the language $\mathcal{L}$, $Comp_{\mathcal{L}}(P)$ consists in the conjunction of the completed definition of all the predicates in P together with $\mathrm{CET}_{\mathcal{L}}$. □

The Language Problem

The semantics determined by $Comp(P)$ depends on the underlying language $\mathcal{L}$, and when $\mathcal{L}$ is finite (that is, when it contains only a finite number of functions symbols) the equality theory which is incorporated in $Comp(P)$ is not complete. This problem can be solved by adding to $Comp(P)$ some domain closure axioms which are intended to restrict the interpretation of the quantification to $\mathcal{L}$-terms. The situation is further complicated by the fact that in the literature we find two different kind of such axioms: the strong (DCA) and the weak (WDCA) ones. In total there exist three different "main" approaches, namely we may:

a) Consider an infinite language, with no domain closure axioms. This is the approach followed by Kunen [61].

b) Consider a finite language and adopt the weak domain closure axioms (WDCA). This has been studied by Shepherdson [93], and the results are similar to the ones found for the case of an infinite language (case (a) above).

c) Consider a finite language and adopt the strong domain closure axioms (DCA). This was studied by Fitting in the case that $\mathcal{L}$ coincides with the language of the program $\mathcal{L}(P)$; this semantics is commonly known as Fitting's Model semantics. His results can also be applied in the case in which $\mathcal{L}$ is larger than $\mathcal{L}(P)$.

In this chapter we consider the three cases separately: first we analyze the case in which the language is infinite, then in Section 4.3 we discuss how the results have to be modified when we drop the infiniteness assumption.

Fitting's operator

Fitting's operator can be considered the three-valued counterpart of the usual (two-valued) immediate consequence operator T_P, and it is extremely useful for characterizing the semantics we are going to refer to in the sequel. We begin with the following Definition.

Definition 2.1.5 Let $\mathcal{L}$ be a language. *A three valued (or partial) $\mathcal{L}$-interpretation*, I, is a mapping from the ground atoms of $\mathcal{L}$ into the set $\{$*true*, *false*, *undefined*$\}$. □

A partial interpretation I is represented by an ordered couple, (T, F), of disjoint sets of ground atoms. The atoms in T (resp. F) are considered to be *true* (resp. *false*) in I. T is the positive part of I and is denoted by I^+; equivalently F is denoted by I^-. Atoms which do not appear in either set are considered to be *undefined.* If I and J are two partial $\mathcal{L}$-interpretations, then $I \cap J$ is the three valued $\mathcal{L}$-interpretation given by $(I^+ \cap J^+, I^- \cap J^-)$, $I \cup J$ is the three valued $\mathcal{L}$-interpretation given by $(I^+ \cup J^+, I^- \cup J^-)$ and we say that $I \subseteq J$ iff $I = I \cap J$, that is iff $I^+ \subseteq J^+$ and $I^- \subseteq J^-$. The underlying universe of an $\mathcal{L}$-interpretation is the universe of $\mathcal{L}$-terms, consequently when we say that a first order formula ϕ is *true* in I, $I \models \phi$, we mean that the quantifiers of ϕ are ranging over the Herbrand Universe of $\mathcal{L}$.

We now give a definition of Fitting's operator [41]. In the sequel of the chapter we write $\exists\, y\ B\theta$ as a shorthand for $(\exists\, y\ B)\theta$, that is, unless explicitly stated, the quantification applies always before the substitution. We denote by $Var(E)$ the set of all the variables in an expression E and by $\mathcal{L}(P)$ the (finite) language consisting of the functions and predicate symbols actually occurring in the program P.

Definition 2.1.6 Let P be a normal program, $\mathcal{L}$ a language that contains $\mathcal{L}(P)$, and I a three valued $\mathcal{L}$-interpretation. $\Phi_P(I)$ is the three valued $\mathcal{L}$-interpretation defined as follows:

- A ground atom A is *true* in $\Phi_P(I)$, $(A \in \Phi_P(I)^+)$
  iff there exists a clause $c:\ \ B \leftarrow \tilde{L}.$ in P whose head unifies with A, $\theta = mgu(A, B)$, and $\exists\, \tilde{w}\ \tilde{L}\theta$ is *true* in I
  where $\tilde{w}$ is the set of local variables of c, $\tilde{w} = Var(\tilde{L}) \backslash Var(B)$.
- A ground atom A is *false* in $\Phi_P(I)$, $(A \in \Phi_P(I)^-)$
  iff for all clauses $c:\ B \leftarrow \tilde{L}$ in P for which there exists $\theta = mgu(A, B)$ we have that $\exists\, \tilde{w}\ \tilde{L}\theta$ is *false* in I
  where $\tilde{w}$ is the set of local variables of c, $\tilde{w} = Var(\tilde{L}) \backslash Var(B)$. □

Note that Φ_P depends on the language $\mathcal{L}$. It would actually be more appropriate to write $\Phi_P^{\mathcal{L}}$ instead of Φ_P, but then the notation would become more cumbersome. We adopt the standard notation:

- $\Phi_P^{\uparrow 0}(I) = I$;
- $\Phi_P^{\uparrow \alpha+1}(I) = \Phi_P(\Phi_P^{\uparrow \alpha}(I))$;
- $\Phi_P^{\uparrow \alpha}(I) = \cup_{\delta<\alpha} \Phi_P^{\uparrow \delta}(I)$, when α is a limit ordinal.

When the argument is omitted, we assume it to be the empty interpretation $(\emptyset, \emptyset)$: $\Phi_P^{\alpha} = \Phi_P^{\uparrow \alpha}(\emptyset, \emptyset)$.

Φ_P is a monotonic operator, that is $I \subseteq J$ implies $\Phi_P(I) \subseteq \Phi_P(J)$; it follows that the Kleene's sequence $\Phi_P^{\uparrow 0}$, $\Phi_P^{\uparrow 1}$, ..., $\Phi_P^{\uparrow k}$, ..., $\Phi_P^{\uparrow \omega}$, ... is monotonically increasing and it converges to the least fixpoint of Φ_P. Hence there always exists an ordinal α such that $lfp(\Phi_P) = \Phi_P^{\uparrow \alpha}$. Since Φ_P is monotone but not continuous, α could be greater than ω.

The Φ_P operator characterizes the three valued model semantics of $Comp_{\mathcal{L}}(P)$, in fact Fitting in [41] shows that the three-valued Herbrand models of $Comp_{\mathcal{L}}(P)$ are exactly the fixpoints of Φ_P; it follows that any program has a *least* (wrt. $\subseteq$) three-valued Herbrand model, which coincides with the least fixed point of Φ_P. This model is usually referred to as Fitting's model.

Example 2.1.7 Let P be the following program:

$$
\begin{array}{lll}
P = \{ & n(0). & \\
 & n(s(X)) & \leftarrow n(X). \\
 & q & \leftarrow \neg n(X). \qquad \}
\end{array}
$$

And let $\mathcal{L} = \mathcal{L}(P)$. We have that

$$
\begin{array}{lll}
 & \Phi_P^{\uparrow 0} & = (\emptyset, \emptyset). \\
 & \Phi_P^{\uparrow 1} & = (\{n(0)\}, \emptyset). \\
 & \Phi_P^{\uparrow 2} & = (\{n(0), n(s(0))\}, \emptyset). \\
 & \ldots & \\
 & \Phi_P^{\uparrow \omega} & = (\{n(0), \ldots, n(s^k(0)), \ldots\}, \emptyset). \\
\mathit{lfp}(\Phi_P) \ = & \Phi_P^{\uparrow \omega+1} & = (\{n(0), \ldots, n(s^k(0)), \ldots\}, \{q\}).
\end{array}
$$

□

2.2 Kunen's semantics

In this Section we will always refer to a fixed but unspecified *infinite* language $\mathcal{L}$, that we assume contains all the function symbols of the programs we are considering. Here by *infinite* language, we mean a language that contains infinitely many functions symbols (including those of arity 0). Later, in Section 2.3, we discuss the problems that arise when the language is finite and we show how the results we give here have to be modified in order to be applied in this other context.

Three valued program's completion semantics in the case of an infinite language has been studied by Kunen [61] and successively by Shepherdson [93]. For this reason, following the literature, we refer to it as *Kunen's semantics*. The main result is the following.

Theorem 2.2.1 ([61]) Let P be a normal program and ϕ an allowed formula.

- $Comp_{\mathcal{L}}(P) \models \phi$ iff for some integer n, $\Phi_P^{\uparrow n} \models \phi$

Proof. This is basically Theorem 6.3 in [61], however, in [61] it is assumed that the language contains a countably infinite number of symbols of each arity. Later, Shepherdson noticed that the result holds for any infinite language [93, Theorem 5b]. □

The aim of this Section is to define and characterize program's equivalence, this will provide the theoretical background for the analysis of the correctness of the transformation. The result we prove here is partially a strengthening of [88, Proposition 3.4] (however, in [88] the more general setting of first order programs under any base theory is considered). We start with the following basic definition.

Definition 2.2.2 We say that P and P' are *equivalent* (wrt Kunen's semantics) iff for each allowed formula ϕ

- $Comp_{\mathcal{L}}(P) \models \phi$ iff $Comp_{\mathcal{L}}(P') \models \phi$. □

Equivalence of two programs can be inferred by comparing the Kleene's sequences of the Φ_P operator. The following result has also been proved by Sato in [88] for the more general setting of first order programs under any base theory.

Theorem 2.2.3 Let P_1 and P_2 be two normal programs.
If

$$\forall n \; \exists m \; \Phi_{P_1}^{\uparrow n} \subseteq \Phi_{P_2}^{\uparrow m}$$

then for all ϕ,

$$Comp_{\mathcal{L}}(P_1) \;\models \phi \text{ implies } Comp_{\mathcal{L}}(P_2) \;\models \phi$$

where ϕ ranges over the set of allowed formulas and n and m are quantified over natural numbers.

Proof. Let us assume $\forall n \; \exists m \; \Phi_{P_1}^{\uparrow n} \subseteq \Phi_{P_2}^{\uparrow m}$, and let ϕ be any allowed formula such that $Comp_{\mathcal{L}}(P_1) \;\models \phi$. By Theorem 2.2.1, there exists an integer n such that $\Phi_{P_1}^{\uparrow n} \models \phi$; by the hypothesis there exists an m such that $\Phi_{P_1}^{\uparrow n} \subseteq \Phi_{P_2}^{\uparrow m}$, hence $\Phi_{P_2}^{\uparrow m} \models \phi$.
Again, by Theorem 2.2.1, this implies that $Comp_{\mathcal{L}}(P_2) \models \phi$. □

Interestingly, also the inverse implication holds. The following is the main original result of this chapter. Since the proof is quite long, it is deferred to the Appendix.

Theorem 2.2.4 Let P_1 and P_2 be two normal programs.
If for all ϕ,

$$Comp_{\mathcal{L}}(P_1) \;\models \phi \text{ implies } Comp_{\mathcal{L}}(P_2) \;\models \phi$$

then

$$\forall n \; \exists m \; \Phi_{P_1}^{\uparrow n} \subseteq \Phi_{P_2}^{\uparrow m}$$

where ϕ ranges over the set of allowed formulas and n and m are quantified over natural numbers.

These results allow us to characterize program's equivalence: Following Sato [88], we say that two programs P_1, P_2 are *chain equivalent* iff $\forall n \; \exists m \quad \Phi_{P_1}^{\uparrow n} \subseteq \Phi_{P_2}^{\uparrow m} \quad and \quad \Phi_{P_1}^{\uparrow m} \supseteq \Phi_{P_2}^{\uparrow n}$. Using this notation, from the previous Theorems, we immediately have the following.

Corollary 2.2.5 Let P_1 and P_2 be normal programs, then

- P_1 and P_2 are equivalent iff they are chain equivalent. □

Notice that, given two programs P_1, P_2, the fact that $\Phi_{P_1}^{\uparrow\omega} = \Phi_{P_2}^{\uparrow\omega}$ is necessary but not sufficient to ensure that P_1 is equivalent to P_2. This is due to the fact that the set of ground atomic logical consequences of $Comp_{\mathcal{L}}(P)$ (which coincide with $\Phi_P^{\uparrow\omega}$) is not sufficient to fully characterize Kunen's semantics of a program P. Consider for instance the following two programs ([61]): $P_1 = \{void(s(X)) \leftarrow void(X).\}$ and $P_2 = \{void(X) \leftarrow f.\}$ where the predicate f has no clause defining it in either programs, and consequently it is always *false*. For any term t, the predicate $void(t)$ is *false* before $\Phi_{P_1}^{\uparrow\omega}$, and indeed we have that $\Phi_{P_1}^{\uparrow\omega} = \Phi_{P_2}^{\uparrow\omega}$, however P_1 is not equivalent to P_2, in fact we have that $Comp_{\mathcal{L}}(P_2) \models \forall X \neg void(X)$ while $Comp_{\mathcal{L}}(P_1) \not\models \forall X \neg void(X)$. This is reflected by the fact that $\Phi_{P_2}^{\uparrow 2} \models \forall X \neg void(X)$ while there is no integer n such that $\Phi_{P_1}^{\uparrow n} \models \forall X \neg void(X)$. Indeed, P_1 has a model which contains, besides the (representation of) natural numbers, also an infinite chain of terms t_i such that for each i, $void(t_i)$ is *true*.

2.3 Adopting a (possibly) finite language

Our aim now is to analyze how the results given in the previous two Sections have to be modified when the language adopted is no longer infinite (or at least not necessarily infinite). Therefore in the sequel we still refer to a fixed but unspecified language $\mathcal{L}$, but we no longer assume it to be infinite. As we mentioned in section 2.1 the main problem we have to face when adopting a finite language is that $\mathrm{CET}_{\mathcal{L}}$ becomes an incomplete theory. The consequences of this are best shown by the following Example, which is borrowed from [93]. Let P be the program:

$$P = \{ \quad p \leftarrow \neg q(X). \quad q(a). \quad \}$$

The completed definition of P is

$$p \Leftrightarrow \exists X \neg q(X) \quad \wedge \quad q(X) \Leftrightarrow X = a.$$

That is, $Comp_{\mathcal{L}}(P) \models p \Leftrightarrow \exists X\ X \neq a$. If $\mathcal{L} = \{a\}$ then neither p nor $\neg p$ is a logical consequence of $Comp_{\mathcal{L}}(P)$. The problem here is that neither we have a "witness" that allows us to say that $\exists X\ X \neq a$ holds, nor we can formally infer that such a witness does not exists. The two main approaches used in logic programming in order to obtain a complete theory out of $\mathrm{CET}_{\mathcal{L}}$ are the following:

- adopting an infinite language (that is a language with infinitely many functions symbols, and that consequently contains infinitely many "witnesses");
- adopting a finite language together with some domain closure axioms, which are axioms that commit us to a specific universe.

For a extended discussion of the subject, we refer to [93].

As we mentioned before, in the literature we find two different kind of domain closure axioms.

Definition 2.3.1 Let $\mathcal{L}$ be a finite language.

- The *Domain Closure Axiom*, $\mathrm{DCA}_{\mathcal{L}}$, is

  $$x = t_1 \vee x = t_2 \vee \ldots$$

  where $t_1, t_2, \ldots$ is the sequence of all the ground $\mathcal{L}$-terms.
- The *Weak Domain Closure Axiom*, $\mathrm{WDCA}_{\mathcal{L}}$, is

  $$\exists \tilde{y}_1 \ (x = f_1(\tilde{y}_1)) \vee \ldots \vee \exists \tilde{y}_r \ (x = f_r(\tilde{y}_r)).$$

  where $f_1, \ldots, f_r$ are all the function symbols in $\mathcal{L}$ and $\tilde{y}_i$ are tuples of variables of the appropriate arity. □

Note that when $\mathcal{L}$ contains a function of arity greater than zero, $\mathrm{DCA}_{\mathcal{L}}$ is an infinite disjunction and hence it is not a first-order formula. For this reason, the notation $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$, that we are going to use often in the sequel is actually overloaded, nevertheless we shall use it for uniformity with the rest of the chapter. As opposed to $\mathrm{DCA}_{\mathcal{L}}$, $\mathrm{WDCA}_{\mathcal{L}}$ is a first-order formula.

The following simple example shows how the semantics of a program changes depending on the kind of closure axioms adopted. Let P be the same program we used in Example 2.1.7.

$$
\begin{array}{lll}
P = \{ & n(0). & \\
 & n(s(X)) & \leftarrow n(X). \\
 & q & \leftarrow \neg n(X). \quad \}
\end{array}
$$

and let $\mathcal{L} = \mathcal{L}(P)$. The completion of P is

$$n(x) \Leftrightarrow (x = 0) \vee (\exists\, y\ (x = s(y)) \wedge n(y)) \quad \wedge \quad q \Leftrightarrow \exists\, y\ \neg n(y)$$

together with $\mathrm{CET}_{\mathcal{L}}$. On one hand, when we use $\mathrm{DCA}_{\mathcal{L}}$ we have

$$Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \forall\, x\ n(x).$$

In fact assuming $\mathrm{DCA}_{\mathcal{L}}$ is equivalent to restrict ourselves to $\mathcal{L}$-Herbrand interpretations and models, and the formula $\forall\, x\ n(x)$ is *true* in the unique Herbrand model of P. From this it follows that:

$$Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \neg q.$$

On the other hand, if we use $\mathrm{WDCA}_{\mathcal{L}}$ we have

$$Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \not\models \forall\, x\ n(x).$$

In fact $\mathrm{WDCA}_{\mathcal{L}}$ allows a model which contains, besides the natural numbers, also an infinite chain of terms t_i such that for each i, $t_i = s(t_{i+1})$. In such a model each $n(t_i)$ can be *false*. It follows that:

$$Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \not\models \neg q.$$

By assuming $\mathrm{WDCA}_{\mathcal{L}}$ we obtain a semantics which is stronger than the one adopting $\mathrm{DCA}_{\mathcal{L}}$. In fact $\mathrm{DCA}_{\mathcal{L}} \models \mathrm{WDCA}_{\mathcal{L}}$, and hence if $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \models \phi$, then also $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \phi$.

It is important to observe that when we adopt some domain closure axioms, we have to modify in the obvious way, the Definitions of programs equivalence (2.2.2).

Let us now give another Example showing how program's equivalence may be affected by the choices of the language and of the closure axioms.

Example 2.3.2 Consider the three programs:

$$
\begin{array}{lll}
P_1 = \{ & n(0). & \\
 & n(s(X)) & \leftarrow n(X). \quad \}
\end{array}
$$

$$
\begin{array}{ll}
P_2 = \{ & n(0). \\
 & n(s(X)). \quad \}
\end{array}
$$

$$P_3 = \{ \quad n(X). \quad \}$$

Let $\mathcal{L} = \mathcal{L}(P_1)$.

If we assume $\mathrm{DCA}_{\mathcal{L}}$, for all three the programs we have

$$Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \forall x\; n(x), \quad P \in \{P_1, P_2, P_3\}.$$

Actually, all the programs are pairwise *equivalent* wrt this semantics.

If we assume $\mathrm{WDCA}_{\mathcal{L}}$,

$$Comp_{\mathcal{L}}(P_1) \cup \mathrm{WDCA}_{\mathcal{L}} \not\models \forall x\; n(x),$$

while for $P \in \{P_2, P_3\}$

$$Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \models \forall x\; n(x), \tag{2.1}$$

then only P_2 and P_3 are *equivalent* wrt this semantics.

Finally if we assume that $\mathcal{L}$ strictly contains $\mathcal{L}(P_1)$, then P_3 is the only program for which (2.1) holds. In this case no program is equivalent to any of the other ones, no matter which are the axioms we adopt. □

This Example shows that two programs may be equivalent wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$ and not equivalent wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$. But there are also cases in which the converse of this statement is true. So even though the semantics obtained by assuming $\mathrm{WDCA}_{\mathcal{L}}$ is stronger than the one obtained by assuming $\mathrm{DCA}_{\mathcal{L}}$, no program's equivalence is stronger than the other one.

2.3.1 The semantics given by $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$

As far as we are concerned the semantics given by $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ (with $\mathcal{L}$ possibly finite) behaves exactly as Kunen's semantics. This fact is due to the following result.

Theorem 2.3.3 ([93]) Let P be a normal program, $\mathcal{L}$ a finite language and ϕ an allowed formula

- $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \models \phi$ iff for some integer n, $\Phi_P^{\uparrow n} \models \phi$. □

Here $\mathcal{L}$ is required to be finite uniquely because otherwise $\mathrm{WDCA}_{\mathcal{L}}$ is not a first-order formula. Notice that Theorem 2.3.3 is identical to Theorem 2.2.1, which was the only result on the semantics that we used in Section 4.1. Consequently, the results that we can prove on program's and formula's equivalence and on the replacement operation are identical to the ones given in the previous Section. In particular, Theorems 2.2.3 and 2.2.4 and Corollary 2.2.5 hold also for $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$.

2.3.2 Fitting's Model Semantics

We now introduce the semantics given by $Comp_{\mathcal{L}}(P)_{\mathcal{L}} \cup \mathrm{DCA}_{\mathcal{L}}$. As opposed to what happened in the previous Section, there is no point in requiring $\mathcal{L}$ to be a finite language. Since $\mathrm{DCA}_{\mathcal{L}}$ is (usually) already a non first-order axiom, we have to leave the first-order context anyhow, and there is no reason here in restricting the domain. As we said before, adopting $\mathrm{DCA}_{\mathcal{L}}$ is equivalent to restricting our attention to Herbrand interpretations and models (on the language $\mathcal{L}$). This particular semantics enjoys a remarkable property: namely that there always exists a *minimal* Herbrand model (wrt $\subseteq$), this model is usually referred to as Fitting's model.

Definition 2.3.4 Let P be a program, *Fitting's model* of P, $Fit(P)$, is the least three valued Herbrand model of $Comp(P)$. □

In order to check if an allowed formula is a logical consequence of $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$ it is sufficient to check if it is true in $Fit(P)$. Indeed, we have the following.

Theorem 2.3.5 ([41]) Let P be a normal program and ϕ an allowed formula

- $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \phi$ iff $Fit(P) \models \phi$. □

A remarkable property of $Fit(P)$ is that it coincides with the interpretation given by the least fixpoint of the operator Φ_P, $\mathit{lfp}(\Phi_P)$. Now, from the monotonicity of Φ_P, it follows that the Kleene's sequence $\{\ldots \Phi_P^{\uparrow\alpha}, \ldots\}$ is monotonically increasing and it converges to its least fixpoint. Hence there always exists an ordinal α such that $\mathit{lfp}(\Phi_P) = \Phi_P^{\uparrow\alpha}$. Since Φ_P is monotone but not continuous, α could be greater than ω. Summarizing we have that.

Theorem 2.3.6 ([41]) Let P be a normal program, then, for some ordinal α,

- $Fit(P) = \mathit{lfp}(\Phi_P) = \Phi_P^{\uparrow\alpha}$ □

2.4 Appendix. Proof of Theorem 2.2.4

We need a Lemma first.

Lemma 2.4.1 Let P be a normal program and χ an allowed formula with free variables $\tilde{x}$. For each integer n, there exist two formulas in the language of equality, T_χ^n and F_χ^n, with free variables $\tilde{x}$ such that, for any tuple $\tilde{t}$ of ground terms,

- $T_\chi^n(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff $\chi(\tilde{t}/\tilde{x})$ is;
  in any other case $T_\chi^n(\tilde{t}/\tilde{x})$ is *false* in $\Phi_P^{\uparrow n}$.
- $F_\chi^n(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff $\chi(\tilde{t}/\tilde{x})$ is *false* in $\Phi_P^{\uparrow n}$.
  in any other case $F_\chi^n(\tilde{t}/\tilde{x})$ is *false* in $\Phi_P^{\uparrow n}$.

Proof. From Lemma 4.1 in [93] it follows that $T_\chi^n(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff $\chi(\tilde{t}/\tilde{x})$ is, and that $F_\chi^n(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff $\chi(\tilde{t}/\tilde{x})$ is *false* in $\Phi_P^{\uparrow n}$. From the completeness of $\mathrm{CET}_{\mathcal{L}}$ in the case that the underlying universe is the Herbrand universe, we have that when $T_\chi^n(\tilde{t}/\tilde{x})$ (resp. $F_\chi^n(\tilde{t}/\tilde{x})$) is not *true* in $\Phi_P^{\uparrow n}$, it has to be *false* in $\Phi_P^{\uparrow n}$. □

Actually, this result holds for any choice of $\mathcal{L}$. To give the intuitive idea of how such formulas are built, let us consider the simple case in which $\chi = n(x)$, and P is the program

$$P = \{ \quad n(0). \\ \qquad n(s(x)) \quad \leftarrow n(x) \quad \}.$$

We have that

$T_n^1(x) \equiv x = 0,$

$T_n^2(x) \equiv x = 0 \vee x = 1,$

$\ldots$

On the other hand,

$F_n^1(x) \equiv x \neq 0 \wedge \neg\exists\, y\ x = s(y),$

$F_n^2(x) \equiv (x \neq 0 \wedge \neg\exists\, y\ x = s(y)) \vee (\exists\, y\ x = s(y) \vee (y \neq 0 \wedge \neg\exists\, z\ y = s(z))), \ldots$

We can now prove the result we were aiming at.

Theorem 2.2.4 Let P_1 and P_2 be two normal programs.
If for all ϕ,

$$Comp_{\mathcal{L}}(P_1) \ \models \phi \text{ implies } Comp_{\mathcal{L}}(P_2) \ \models \phi$$

then

$$\forall\, n\ \exists\, m\ \Phi_{P_1}^{\uparrow n} \subseteq \Phi_{P_2}^{\uparrow m}$$

where ϕ ranges over the set of allowed formulas and n and m are quantified over natural numbers.

Proof.

The proof is by contradiction. Assume that for all ϕ, $Comp_{\mathcal{L}}(P_1) \ \models \phi$ implies $Comp_{\mathcal{L}}(P_2) \ \models \phi$ and that there exists a fixed n such that

$$\text{for all } m, \quad \Phi_{P_1}^{\uparrow n} \not\subseteq \Phi_{P_2}^{\uparrow m}. \tag{2.2}$$

For each predicate symbol p let $T^n_{p(\tilde{x})}$ and $F^n_{p(\tilde{x})}$ be the equality formulas described in Lemma 2.4.1. Hence $T^n_{p(\tilde{x})}(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff $p(\tilde{t}/\tilde{x})$ is, and $F^n_{p(\tilde{x})}(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff $p(\tilde{t}/\tilde{x})$ is *false* in $\Phi_P^{\uparrow n}$. Let also

$$\chi \equiv \bigwedge_{p \in pred(P_1)} \forall\, \tilde{x}\ (T^n_{p(\tilde{x})} \rightarrow p(\tilde{x}) \wedge F^n_{p(\tilde{x})} \rightarrow \neg p(\tilde{x}))$$

where p ranges over the finite set of predicate symbols occurring in P_1. From Lemma 2.4.1 it follows that $\Phi_{P_1}^{\uparrow n} \models \chi$, and, by Theorem 2.2.1

$$Comp_{\mathcal{L}}(P_1) \models \chi.$$

By hypothesis we have that $Comp_{\mathcal{L}}(P_2) \models \chi$, and, by Theorem 2.2.1 there exists an integer r such that

$$\Phi_{P_2}^{\uparrow r} \models \chi.$$

By (2.2) $\Phi_{P_1}^{\uparrow n} \not\subseteq \Phi_{P_2}^{\uparrow r}$, hence there exists a ground atom $q(\tilde{t})$ such that

$$\text{either } \Phi_{P_1}^{\uparrow n} \models q(\tilde{t}) \text{ and } \Phi_{P_2}^{\uparrow r} \not\models q(\tilde{t}) \quad \text{or} \quad \Phi_{P_1}^{\uparrow n} \models \neg q(\tilde{t}) \text{ and } \Phi_{P_2}^{\uparrow r} \not\models \neg q(\tilde{t}).$$

We consider only the first possibility, the other case is perfectly symmetrical. So we assume that

$$\Phi_{P_1}^{\uparrow n} \models q(\tilde{t}) \quad \text{and} \quad \Phi_{P_2}^{\uparrow r} \not\models q(\tilde{t}) \tag{2.3}$$

By the left hand side of 2.3 and the definition of $T^n_{q(\tilde{x})}$ in Lemma 2.4.1,

$$\Phi_{P_1}^{\uparrow n} \models T^n_{q(\tilde{x})}(\tilde{t}/\tilde{x}).$$

$T^n_{q(\tilde{x})}(\tilde{t}/\tilde{x})$ is a formula of the equality language and contains no predicate symbols other than "=", so if it is *true* in $\Phi_{P_1}^{\uparrow n}$ it must be *true* also in $\Phi_{P_1}^{\uparrow 0}$, i.e. $\Phi_{P_1}^{\uparrow 0} \models T^n_{q(\tilde{x})}(\tilde{t}/\tilde{x})$. But $\Phi_{P_1}^{\uparrow 0} = (\emptyset, \emptyset) \subseteq \Phi_{P_2}^{\uparrow r}$, hence

$$\Phi_{P_2}^{\uparrow r} \models \; T^n_{q(\tilde{x})}(\tilde{t}/\tilde{x}).$$

Since $\Phi_{P_2}^{\uparrow r} \models \chi$, from the definition of χ, it follows that also $\Phi_{P_2}^{\uparrow r} \models \forall\, \tilde{x}\, (T^n_{q(\tilde{x})}(\tilde{x}) \rightarrow q(\tilde{x}))$, hence $\Phi_{P_2}^{\uparrow r} \models \; T^n_{q(\tilde{x})}(\tilde{t}/\tilde{x}) \rightarrow q(\tilde{t})$; and, from the above statement,

$$\Phi_{P_2}^{\uparrow r} \models \; q(\tilde{t})$$

which contradicts the right hand side of (2.3). $\square$

Chapter 3

Transforming Acyclic Programs

1

An Unfold/Fold transformation system is a source-to-source rewriting methodology devised to improve the efficiency of a program. Any such transformation should preserve the main properties of the initial program: among them, termination. In the field of logic programming, the class of acyclic programs plays an important role in this respect, as it is closely related to the one of terminating programs. The two classes coincide when negation is not allowed in the bodies of the clauses.

In this chapter it is proven that the Unfold/Fold transformation system defined by Tamaki and Sato preserves the acyclicity of the initial program. As corollaries, it follows that when the transformation is applied to an acyclic program, then finite failure set for definite programs is preserved; in the case of normal programs, all major declarative and operational semantics are preserved as well. These results cannot be extended to the class of left terminating programs without modifying the definition of the transformation.

3.1 Introduction

Motivation

In this chapter we focus on the unfold/fold transformation system proposed by Tamaki and Sato [96].

As the large literature shows [96, 58, 90, 91, 92, 12], a lot of research has been devoted to proving the correctness of the system wrt the various semantics proposed for logic programs. However the question of the consequences of the transformation on the (universal) termination of the program has not yet been tackled.

Recall that a program is called *terminating* if all its SLDNF derivations starting in a ground goal are finite.

Here we follow the approach to termination of Apt and Bezem [5]. They investigate the class of *acyclic* programs (introduced by Cavedon [26]) and prove that it is closely related to the one of terminating programs. In fact we have that every acyclic program is terminating [5] and that every definite, terminating program is acyclic [15]; however, when negation is allowed in the bodies of the clauses, there are programs which are terminating but not acyclic. This is caused either by the presence of floundering derivations or by the fact that since nonground negative literals might not be selected, some infinite branches of the search tree cannot be explored, see [5] for examples.

In this chapter we prove that when the initial program of an unfold/fold transformation sequence is acyclic, then the resulting program is acyclic as well.

This has some obvious consequences on the preservation of termination and some semantic repercussions. For definite programs, the transformation preserves the Finite Failure Set. In fact, since acyclic programs are terminating, and since definite programs cannot flounder, their Finite Failure Set coincides with the complement of their Success Set. For programs with negation, the transformation preserves all the major formalisms, namely Fitting's model, 2 and 3 valued ground logical consequence of the completion, and, in the non-floundering cases, the operational semantics based on the SLDNF-resolution: when the program is acyclic they all coincide and thus they are preserved by the transformation.

Structure of the chapter

Section 3.3 contains the preliminaries on terminating and acyclic programs and on the Tamaki-Sato's unfold/fold transformation system. In section 3.4 we prove that the transformation preserves the acyclicity of the initial program; we also discuss the case in which the initial program is left terminating. In Section 3.5 we give a brief summary of the semantic properties of acyclic programs and we show that they are preserved through the transformation.

3.2 Unfold/Fold Transformation Systems

We now give the formal definitions of the two unfold/fold transformation systems that we are going to refer to in the rest of the thesis. We start with the method proposed by Tamaki and Sato [96] for definite programs and then used by Seki [90, 92] for normal programs. Here we present it as it is in [92]. Later in this section we'll also report the more restrictive *modified* folding operation introduced by Seki [91] which guarantees the correctness of the operation also wrt the finitefailure set.

OCHO

We start with the requirements on the initial program. All definitions are given modulo reordering of the bodies of the clauses, and standardization apart is always assumed.

Definition 3.2.1 (initial program) We call a normal program P_0 an *initial program* if the following two conditions are satisfied:

(I1) P_0 is divided into two disjoint sets $P_0 = P_{new} \cup P_{old}$;

(I2) All the predicates which are defined in P_{new} occur neither in P_{old} nor in the bodies of the clauses in P_{new}. □

The predicates defined in P_{new} are called *new* predicates, while those defined in P_{old} are the *old* predicates. For the porpose of this chapter, clauses in P_{new} will also be referred to as *defining* clauses.

Example 3.2.2 **[92]** Let P_0 be the following program

$$
\begin{array}{rlll}
P_0 = DB \cup \{ & c_1: & path(X,[X]) & \leftarrow node(X). \\
& c_2: & path(X,[X|Xs]) & \leftarrow arc(X,Y), path(Y,Xs). \\
& c_3: & goodlist([]). & \\
& c_4: & goodlist([X|Xs]) & \leftarrow \neg bad(X), goodlist(Xs). \\
& c_5: & goodpath(X,Xs) & \leftarrow path(X,Xs), goodlist(Xs). \quad \}
\end{array}
$$

where predicates *node, arc* and *bad* are defined in DB by a set of unit clauses. Predicate $goodpath(X,Xs)$ can be employed for finding a path Xs starting from the node X which doesn't contain "bad" nodes. Let $P_{old} = \{c_1,\ldots,c_4\} \cup DB$ and $P_{new} = \{c_5\}$, thus *goodpath* is the only new predicate. □

Unfolding is the fundamental operation for partial evaluation [66] and consists in applying a resolution step to the considered atom in all possible ways.

Definition 3.2.3 (Unfolding) Let $cl : A \leftarrow H, \tilde{K}.$ be a clause of a normal program P, where H is an atom. 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 $\{\theta_1,\ldots,\theta_n\}$.

- *Unfolding H in cl* consists of substituting cl with $\{cl'_1,\ldots,cl'_n\}$, where, for each i, $cl'_i = (A \leftarrow \tilde{B}_i, \tilde{K})\theta_i$.

$unfold\ (P, cl, H) \stackrel{\text{def}}{=} P \backslash \{cl\} \cup \{cl'_1,\ldots,cl'_n\}$. □

Example 3.2.2 (part 2) By unfolding the atom $path(X,Xs)$ in the body of c_5, we obtain

$$
\begin{array}{lll}
c_6: & goodpath(X,[X]) & \leftarrow node(X), goodlist([X]). \\
c_7: & goodpath(X,[X|Xs]) & \leftarrow arc(X,Y), path(Y,Xs), goodlist([X|Xs]).
\end{array}
$$

Both clauses can be further unfolded (c_6 twice), the resulting clauses are

$$
\begin{array}{lll}
c_8: & goodpath(X,[X]) & \leftarrow node(X), \neg bad(X). \\
c_9: & goodpath(X,[X|Xs]) & \leftarrow arc(X,Y), path(Y,Xs), \neg bad(X), goodlist(Xs).
\end{array}
$$

Let $P_1 = \{c_1,\ldots,c_4,c_8,c_9\} \cup DB$. □

Folding is the inverse of unfolding when one single unfolding is possible. It consists in substituting an atom A for an equivalent conjunction of literals $\tilde{K}$ in the body of a clause c. This operation is used in all the transformation systems in order to pack back unfolded clauses and to detect implicit recursive definitions. In the literature we find different definitions for this operation. This is due to the fact that it does not always preserve the declarative semantics and thus its use must be restricted by some applicability conditions. Depending on the approach, such conditions can be either a constraint on how to sequentialize the operations while transforming the program [96, 58], or can be expressed in terms of semantic properties of the program, independently from its transformation history [18, 67].

In the method proposed by Tamaki and Sato [96], the transformation sequence and the folding operation are defined in terms of each other.

Definition 3.2.4 (transformation sequence) A *transformation sequence* is a sequence of programs $P_0, \dots, P_n$, $n \geq 0$, such that each program P_{i+1}, $0 \leq i < n$, is obtained from P_i by unfolding or folding a clause of P_i. □

Definition 3.2.5 (folding) Let $P_0, \dots, P_i$, $i \geq 0$, be a transformation sequence, $c : A \leftarrow \tilde{K}', \tilde{J}.$ a clause in P_i and $d : D \leftarrow \tilde{K}.$ a clause in P_{new}. Let $X = Var(d)$ be the set of all the variables occurring in the clause d, and $Y = Var(\tilde{K}') \backslash Var(A, \tilde{J})$ be the set of variables in $\tilde{K}'$ not in $A, \tilde{J}$. If there exists a substitution τ whose domain is the set X, such that the following conditions hold:

(F1) $\tilde{K}\tau = \tilde{K}'$;
(F2) τ renames with variables in Y the variables in $\tilde{K}$ not in D;
(F3) d is the only clause in P_{new} whose head is unifiable with $D\tau$;
(F4) one of the following two conditions holds

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

then *folding* $D\tau$ *in* c *in* P_i consists of substituting c' for c in P_i, where

$head(c') \stackrel{\text{def}}{=} A$

$body(c') \stackrel{\text{def}}{=} D\tau, \tilde{J}.$

$\text{fold}(P_i, D\tau, c) \stackrel{\text{def}}{=} (P_i \backslash \{c\}) \cup \{c'\}.$ □

Example 3.2.2 (part 3) We can now fold the body of c_9, using c_5 as folding clause, the resulting program is $P_2 = DB \cup \{c_1, \dots, c_4, c_{10}\}$, where c_{10} is the following clause:

$c_{10}: \; goodpath(X, [X|Xs]) \leftarrow arc(X, Y), \neg bad(X), goodpath(Y, Xs).$

Notice that because this operation the definition of *goodpath* is now recursive. □

The transformation enjoys the following important properties.

Theorem 3.2.6 Let $P_0, \dots, P_n$ be a transformation sequence.

- If P_0 is a definite program then
  - [96] The least Herbrand models of the initial and final programs coincide.
  - [58] The computed answers substitution semantics of the initial and final programs coincide.
- If P_0 is a normal program, then
  - [90] The Stable models of the initial and final programs coincide.
  - [92] The Well-Founded models of the initial and final programs coincide.
  - [89] Under a further mild assumption on the initial program; if the initial program is stratified then the final program is stratified and their Perfect models coincide.

– [12] The *semantic kernels* of the initial and final program coincide; this implies also that the Stable model semantics, the preferred extension semantics, the stationary semantics and the stable theory semantics of the initial and the final programs coincide. □

Modified folding

We have to mention that the above transformation does not preserve the Finite Failure set of the initial (definite) program. More precisely we have that the Finite Failure set of the final program is contained in the one of the initial program, but, in general, not vice-versa. This is shown by the following example.

Example 3.2.7 Let P_0 be the following program:

$$P_0 = \{ \begin{array}{lll} c_1: & p & \leftarrow q, h(X). \\ c_2: & h(s(X)) & \leftarrow h(X). \quad \} \end{array}$$

Here we use the following partition: $P_{new} = \{c_1\}$, $P_{old} = \{c_2\}$; notice that there is no definition for predicate q, so the queries $P \cup \{\leftarrow q\}$ and $P \cup \{\leftarrow p\}$ will always fail. Now if we unfold atom $h(X)$ in the body of the first clause, we obtain a renaming of the clause itself, namely:

$$P_1 = \{c_2\} \cup \{c_3: \quad p \leftarrow q, h(Y).\}$$

c_3 satisfies condition (F4.2), so it can be folded, using c_1 as folding clause. The resulting program is:

$$P_2 = \{c_2\} \cup \{c_4: \quad p \leftarrow p.\}$$

Now the query $P_2 \cup \{\leftarrow p\}$ does not terminate. □

The problem of the correctness of the operation wrt the Finite Failure Set was pointed out by Seki, who modified the applicability conditions of the folding operation as follows.

Definition 3.2.8 (modified folding) [91] The *modified* folding operation is defined exactly as in Definition 3.2.5, with the exception of condition (F4.2), which is replaced by the following

(F4.2') all the atoms in $\tilde{K}'$ are the result of some previous unfold operation. □

This Definition first appeared in [89]. It is easy to see that when (F4.2') holds, then (F4.2) holds as well, hence that the *modified folding* operation enjoys all the properties that were proven for the folding operation. Seki proved that modified folding preserves the Finite Failure set of a definite program [89, 91]; later on Sato, on a work that extends this definition to full first order programs [87], proved the correctness of the system wrt Kunen's semantics.

3.3 Termination

The following notion is crucial.

Definition 3.3.1 A program is called *terminating* iff all its SLDNF-derivations starting from a ground goal are finite. □

Hence terminating programs are the ones whose SLDNF-trees starting in a ground goal are finite. We now present the approach to the issue of termination followed by Apt and Bezem [5].

Acyclic programs

Acyclic programs form a natural subclass of the locally stratified ones; they were introduced by Cavedon [26] and have been further studied by Apt and Bezem [5]. To give their definition, first we need the following notion.

Definition 3.3.2 Let P be a program, a *level mapping* for P is a function $|\ |: B_P \rightarrow \mathbf{N}$ from ground atoms to natural numbers. □

For an atom A, $|A|$ denotes the level of A. Following [5], we extend this definition to ground literals by letting $|\neg A| = |A|$.

Definition 3.3.3 Let $|\ |$ be a level mapping.

- A clause is *acyclic wrt* $|\ |$ iff for every ground instance $A \leftarrow L_1, \ldots, L_k$ of it, and for each i, $|A| > |L_i|$;
- A program P is *acyclic wrt* $|\ |$ iff all its clauses are. P is called *acyclic* if it is acyclic wrt some level mapping. □

Following Bezem [15], we introduce the concept of boundedness, which applies also to nonground atoms.

Definition 3.3.4 Let $|\ |$ be a level mapping. A literal L is called *bounded* wrt $|\ |$ if $|\ |$ is bounded on the set $[L]$ of ground instances of L. A goal is called *bounded* wrt $|\ |$ iff all its literals are. □

Example 3.3.5 [8] Consider the program *member*.

$$P = \{ \quad member(X, [Y|Xs]) \quad \leftarrow member(X, Xs). \\ member(X, [X|Xs]). \quad \}$$

We adopt the standard list notation and define the function $|\ |_l$, called *listsize* which assigns natural numbers to ground terms as follows:

$|t|_l = 1$ if t is not of the form $[x_1|x_s]$ (this takes also care of the case $t = [\,]$).

$|[x_1|x_s]|_l = 1 + |x_s|_l$.

We can now define the level mapping $|\ |$ for the *member* program: $|member(t, s)| = |s|_l$. It is easy to see that program *member* is acyclic wrt $|\ |$ and that if l is a list (by this we mean $l = [x_1, \ldots, x_n]$, where the x_i's need not be ground), then $member(t, l)$ is a bounded atom. □

We can now relate acyclic and terminating programs.

Theorem 3.3.6 [5] Let P be a program and G be a goal. If there exists a level mapping $|\ |$ such that P is acyclic wrt $|\ |$ and G is bounded wrt $|\ |$ then all SLDNF derivations of $P \cup \{G\}$ are finite. □

Since ground goals are bounded, this implies the following.

Theorem 3.3.7 [5] If P is an acyclic program then P is terminating. □

In [5] is stated that the converse of Theorem 3.3.7 holds in the case that no SLDNF-derivation starting in a ground goal contains a goal with a nonground negative literal in it, and that since that condition is quite constraining, the result itself is too weak to be formalized. However it is significant at least for the case that we restrict our attention to definite programs; in fact in [15] we find the following.

Theorem 3.3.8 [15] Let P be a definite program, then P is terminating iff P is acyclic. □

From the procedural point of view, acyclic programs enjoy the following important property: the two most prominent approaches, namely the SLDNF resolution (see Lloyd [65] and Apt [3]) and the SLS resolution from Przymusinski [82], coincide when applied to acyclic programs. For the semantic properties of acyclic programs we refer to section 3.5.

3.4 Transforming Acyclic Programs

We now show that if the initial program of a transformation sequence is acyclic then the resulting program is acyclic as well. We do this by showing that there exists a level mapping with respect to which every program in the transformation sequence is acyclic.

Notation

Let $P_0, \ldots, P_n$ be the transformation sequence we are considering. Since P_0 is acyclic, then it is acyclic wrt some level mapping, say $\|\ \|$, moreover, there in no loss of generality in assuming that $\|\ \|$ does not take value zero on any atom. Let nf be the number of foldings that are going to be performed in the sequence (which we assume greater than zero), and let $maxbody$ be the maximum number of literals that a body of a clause of P_0 contains, augmented by one. We also suppose that $maxbody > 1$, as it is not possible to perform any unfold or fold operations on a program consisting solely of unit clauses.

We now define a new level mapping $|\ |$ for P_0.

Definition 3.4.1 Let P_0 be acyclic wrt the level mapping $\|\ \|$. The level mapping $|\ |$ is defined as follows. Let A be a ground atom.

- If A is an *old* atom then we let $|A| = nf \cdot maxbody^{\|A\|}$.
- If A is an *new* atom then we distinguish two subcases:
  (a) If A unifies with the head of only one clause of P_{new}, $N \leftarrow B_1, \ldots, B_n$, suppose that $A = N\theta$, since $B_1, \ldots, B_n$ are *old* atoms, we have that $|\ |$ is already defined on their ground instances, so we set
  $|A| = |N\theta| = sup\{\sum_{i=1}^{n} |B_i\theta\gamma| \ \mid\ Dom(\gamma) = Var(B_1\theta, \ldots, B_n\theta)\} + 1.$

(b) (This case is of no relevance for the proof, as, because of condition (F3), we are interested in computing the level mapping of atoms that unify with the head of only one clause of P_{new}; but we do have to extend $|\ |$ in a consistent way). If A unifies with the head of a (non-unit) set of clauses $\{N_1 \leftarrow B_{1,1}, \ldots, B_{1,n(1)} \quad \ldots \quad N_j \leftarrow B_{j,1}, \ldots, B_{j,n(j)}\} \subseteq P_{new}$, suppose that $A = N_i\theta_i$, we define
$|A| = sup\{\sum_{k=1}^{n(i)} |B_{i,k}\theta_i\gamma|\} + 1$
where i ranges in $[1, \ldots, j]$ and γ ranges over the ground substitutions whose domain is $Var(B_{i,1}\theta_i, \ldots, B_{i,n(i)}\theta_i)$ □

Here the *sup* of an empty set is assumed to be 0. $|\ |$ is obviously a level mapping, as it is defined and finite on each ground atom.

In order to prove that each of the programs in the transformation sequence is acyclic wrt $|\ |$ we need the following simple but technical lemma.

Lemma 3.4.2 For nonzero integers $nf, n, n_1, \ldots, n_k$, if $1 < k < maxbody$ then

- if $n > sup\{n_1, \ldots, n_k\}$, then $nf \cdot maxbody^n > nf + \sum_{j=1}^{k} nf \cdot maxbody^{n_j}$

Proof.
$nf + \sum_{j=1}^{k} nf \cdot maxbody^{n_j} \leq nf + nf \cdot k \cdot maxbody^{sup\{n_j\}}$
Since $k < maxbody$
$\leq nf + nf \cdot (maxbody - 1) \cdot maxbody^{sup\{n_j\}}$
$= nf + nf \cdot maxbody^{sup\{n_j\}+1} - nf \cdot maxbody^{sup\{n_j\}}$
Since $maxbody > 0$ and $n > sup\{n_j\}$,
$\leq nf \cdot maxbody^n + nf - nf \cdot maxbody^{sup\{n_j\}}$
$= nf \cdot maxbody^n + nf \cdot (1 - maxbody^{sup\{n_j\}})$.
Since all integers are nonzero and $maxbody > 1$, $1 - maxbody^{sup\{n_j\}} < 0$. This proves the Lemma. □

Lemma 3.4.3 For each P_i in the transformation sequence the level mapping $|\ |$ of Definition 3.4.1 satisfies the following.

(a) for each ground instance of a *defining* clause $H \leftarrow B_1, \ldots, B_k.$,
$|H| > |B_1| + \ldots + |B_k|$;
(b) for any other clause $H \leftarrow B_1, \ldots, B_k.$ in $Ground(P_i)$,
$|H| > |B_1| + \ldots + |B_k| + nf_i$.

Where for each i, nf_i is the number of folding operations that will be performed in the sequence from P_i to P_n.

Proof. The proof proceeds by induction on the index i.

Base Case: P_0.

Let $c : H \leftarrow B_1, \ldots, B_k.$ be a clause of $Ground(P_0)$. If $k = 0$ then the result holds trivially. So we assume $k > 0$. We have to distinguish two cases:

If H is a *new* predicate, then c is an instance of a *defining* clause, and condition (a) is then trivially satisfied by the definition of $|\ |$.

If H is an *old* predicate, then, since $||H|| > sup\{||B_j||\}$ and since $1 < k <maxbody$, the result follows from Lemma 3.4.2.

Induction Step: P_{i+1}.

For those clauses that P_i and P_{i+1} have in common, the result follows from the inductive hypothesis and the fact that $nf_{i+1} \leq nf_i$. Hence we can focus on those clauses that were introduced or modified in the last transformation step (from P_i to P_{i+1}). We distinguish upon the operation that has been used for going from P_i to P_{i+1}

Unfolding

Let
$d : H \leftarrow B', L_1, \ldots, L_h$. be the unfolded clause, and
$c : B \leftarrow B_1, \ldots, B_k$. be one of the unfolding ones.
Let also $\theta = mgu(B, B')$, then the resulting clause is
$H\theta \leftarrow B_1\theta, \ldots, B_k\theta, L_1\theta, \ldots, L_h\theta$.
Since $nf_{i+1} = nf_i$, in order to prove the thesis, we have to prove that, for each γ

$$|H\theta\gamma| > |B_1\theta\gamma| + \ldots + |B_k\theta\gamma| + |L_1\theta\gamma| + \ldots + |L_h\theta\gamma| + nf_i. \tag{3.1}$$

We have to distinguish two cases:
First we suppose that d is a *defining* clause. Then B is an old predicate and clause c satisfies condition (b), hence
$|B\theta\gamma| > |B_1\theta\gamma| + \ldots + |B_k\theta\gamma| + nf_i$.
On the other hand, clause d satisfies condition (a), hence
$|H\theta\gamma| > |B'\theta\gamma| + |L_1\theta\gamma| + \ldots + |L_h\theta\gamma|$.
Since $B'\theta\gamma = B\theta\gamma$ this proves (3.1).
Secondly we consider the case in which d is not a *defining* clause. Hence d satisfies condition (b), and we have that
$|H\theta\gamma| > |B'\theta\gamma| + |L_1\theta\gamma| + \ldots + |L_h\theta\gamma| + nf_i$.
Since clause c must satisfy either (a) or (b), we also have that
$|B\theta\gamma| > |B_1\theta\gamma| + \ldots + |B_k\theta\gamma|$.
Since $B'\theta\gamma = B\theta\gamma$ this proves again (3.1).

Folding

Suppose that:
$c:\; H \leftarrow B'_1, \ldots, B'_k, L_1, \ldots, L_h$. is the folded clause of P_i,
$d:\; N \leftarrow B_1, \ldots, B_k$ is the folding clause of P_{new}.
Hence $(B'_1, \ldots, B'_k) = (B_1, \ldots, B_k)\tau$, and $H \leftarrow N\tau, L_1, \ldots, L_h$. is the clause we add to P_{i+1}.
By (F4), c is not a *defining* clause, hence its ground instances have to satisfy condition (b), that is, for each γ, $|H\gamma| > |B'_1\gamma| + \ldots + |B'_k\gamma| + |L_1\gamma| \ldots + |L_h\gamma| + nf_i$.
Since $(B'_1, \ldots, B'_k) = (B_1, \ldots, B_k)\tau$, this implies that, for each γ,
$|H\gamma| > |B_1\tau\gamma| + \ldots + |B_k\tau\gamma| + |L_1\gamma| \ldots + |L_h\gamma| + nf_i$,
where τ is a renaming on the variables in $\tilde{w} = Var(B_1, \ldots, B_k)\backslash Var(N)$. Let $\tilde{z} = \tilde{w}\tau$, by the assumptions in (F2), $Var(H, L_1, \ldots, L_h) \cap \tilde{z} = \emptyset$. Hence we can split γ into

two independent orthogonal substitutions: $\gamma = \gamma_{|\tilde{z}}\gamma_{|\bar{\tilde{z}}}$, where $\gamma_{|\tilde{z}}$ is γ restricted to $\tilde{z}$, and $\gamma_{|\bar{\tilde{z}}}$ is γ restricted to the complement of $\tilde{z}$. And we have that, for each γ,

$|H\gamma_{|\bar{\tilde{z}}}| > |B_1\tau\gamma_{|\bar{\tilde{z}}}\gamma_{|\tilde{z}}| + \ldots + |B_k\tau\gamma_{|\bar{\tilde{z}}}\gamma_{|\tilde{z}}| + |L_1\gamma_{|\bar{\tilde{z}}}| + \ldots + |L_h\gamma_{|\bar{\tilde{z}}}| \ + nf_i.$

Since this holds for any choice of $\gamma_{|\tilde{z}}$, for each γ

$|H\gamma_{|\bar{\tilde{z}}}| > sup\{\sum_{i=1}^{k} |B_i\tau\gamma_{|\bar{\tilde{z}}}\eta| \mid Dom(\eta) = \tilde{z}\} + |L_1\gamma_{|\bar{\tilde{z}}}| + \ldots + |L_h\gamma_{|\bar{\tilde{z}}}| + nf_i.$

Now by (F3) d is the only clause whose head unifies with $N\tau$; it follows that, by the definition of $|\ |$, $|N\tau\gamma_{|\bar{\tilde{z}}}| = sup\{\sum_{i=1}^{k} |B_i\tau\eta|\} + 1$, hence we have that, for each γ,

$|H\gamma_{|\bar{\tilde{z}}}| > |N\tau\gamma_{|\bar{\tilde{z}}}| + |L_1\gamma_{|\bar{\tilde{z}}}| + \ldots + |L_h\gamma_{|\bar{\tilde{z}}}| + nf_i - 1.$

Now the variables of $\tilde{z}$ do not occur in any atom of this clause we have that, for each γ

$|H\gamma| > |N\tau\gamma| + |L_1\gamma| + \ldots + |L_h\gamma| + nf_i - 1$

Since this is a folding step, $nf_{i+1} < nf_i$ and hence we have that (b) is satisfied in P_{i+1}.
□

This implies immediately the desired conclusion

Corollary 3.4.4 Let $P_0, \ldots, P_n$ be a transformation sequence, then

(a) if P_0 is acyclic then P_n is.

In the case that P_0 is a definite program, this can be restated as follows

(b) if P_0 is definite and terminating, then P_n is.

Proof. It follows at once from Lemma 3.4.3 □

Transforming left-terminating programs

One would like Corollary 3.4.4b to hold also in the case of *left terminating* programs, which are those programs whose LDNF (SLDNF with leftmost selection rule) derivations starting in a ground goal are finite. *Left terminating* programs form an important superclass of the terminating programs and, as pointed out by Apt and Pedreschi [8], there are natural left terminating programs that are not terminating. However, left-termination is not preserved by the transformation system. In fact, if we consider the three programs P_0, P_1, P_2 of Example 3.2.7, we have that P_0 and P_1 are left terminating, while P_2 is not.

In general left termination is not preserved even when Seki's (more restrictive) *modified* folding operation is used. This is shown by the following example.

Example 3.4.5 Let P_0, be the following program:

$$
\begin{array}{llll}
P_0 = \{ & c_1: & d(X) & \leftarrow h(X), q(X). \\
 & c_2: & p & \leftarrow q(X), h(X). \\
 & c_3: & q(s(0)). & \\
 & c_4: & h(s(X)) & \leftarrow h(X). \qquad \}
\end{array}
$$

Where we adopt the following partition: $P_{new} = \{c_1\}$, $P_{old} = \{c_2, c_3, c_4\}$. It is easy to verify that the program is left-terminating. Since the head of c_2 is an old predicate (and then (F4.1) is satisfied), we can fold $q(X), h(X)$ in the body of c_2. the resulting program is

$P_1 = \{c_1, c_3, c_4\} \cup \{c_5 : \quad p \leftarrow d(X)\}$

Now the goal $P_1 \cup \{\leftarrow p\}$ originates an infinite LDNF-derivation. □

In this case the problem is due to the fact that the definition of transformation sequence is given modulo reordering of the bodies of the clauses, and the operation of reordering itself does not preserve left-termination.

It can be argued that then what we have to do is to start by adopting the *modified* folding instead of the one of Tamaki-Sato and by restating the definition of unfolding and folding so that the order of the literals in the bodies of the clauses is taken into account. That is indeed a possible approach, however a fold operation so defined would be of far more limited applicability than the present one; this holds not only because the *modified folding* is more restrictive than the ordinary one, but mainly because we would have to require that the literals that are going to be folded are all found next to each other in the exact same sequence as in the body of the folding clause. This is often not the case, in particular when the folded clause is the result of some previous unfold operation; notice that this is what happens in Example 3.2.2.

Nevertheless, we can relax the requirement of the acyclicity of the initial program, by exploiting the result in a modular way. First we need the following definition.

Definition 3.4.6 Let $P_0, \ldots, P_n$ be a transformation sequence and let $P_0 = Q_0 \cup R$. We say that the transformation is performed *within* Q_0 if there exist programs $Q_1, \ldots, Q_n$ such that, for each i,

- $P_i = Q_i \cup R$;
- No clause of R is used as folding or unfolding clause. □

Now we have to use the concept of *acceptable* programs, introduced by Apt and Pedreschi in [8]. Here the notation becomes more cumbersome as the notion of acceptability is bound both to a level mapping and to a (not necessarily Herbrand) model. For the definition we refer to [8]. Informally, acceptable are to left terminating programs what acyclic are to terminating ones, in fact in [8] is proven that, in cases of non-floundering programs, the classes of acceptable and of left terminating programs coincide.

Corollary 3.4.4a can then be restated as follows.

Proposition 3.4.7 Let $P_0, \ldots, P_n$ be a transformation sequence. Suppose that P_0 is acceptable wrt the level mapping $|\ |$ and the model M. If there exists a program $Q_0 \subseteq P_0$ such that Q_0 is acyclic wrt $|\ |$ and the transformation is performed within Q_0, then each P_i is acceptable.

Proof. It is a standard extension of the proof of Lemma 3.4.3. □

That is, if the initial program is acceptable (wrt some model and some level mapping) and if the transformation is performed *within* a subset of P_0 which is also acyclic (wrt the same level mapping), then the resulting program is acceptable (hence left-terminating) as well.

3.5 Semantic consequences

From the point of view of declarative semantics, acyclic programs enjoy the following relevant properties. Here, for the definition and the properties of the Well-Founded model semantics we refer to [48].

Theorem 3.5.1 Let P be an acyclic program, and let $M = \Phi_P^{\uparrow\omega}$. Then M is total, that is, no atom is undefined in it, moreover

(i) M is the unique fixpoint of Φ_P; hence it is the unique three-valued (and also two-valued) Herbrand model of $Comp(P)$ and coincides with Fitting's model of P.
(ii) M coincides with the Well-Founded model of P;
(iii) M coincides with the set of ground atomic logical consequences of $Comp(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ in 2 and 3 valued logic;
(iv) for all ground atoms A such that no SLDNF-derivation of $P \cup \{\leftarrow A\}$ flounders,
  - A is *true* in M iff there exists a SLDNF-refutation for $P \cup \{\leftarrow A\}$;
  - A is *false* in M iff $P \cup \{\leftarrow A\}$ has a finitely failed SLDNF tree.

Proof. The fact that M is total and statement (i) are consequences of Lemma 2.6 and Theorem 4.4 in [5]; more general statements are also proven in [8], where the case of *acceptable* programs is considered; (ii) is a consequence if (i) and the fact that the Well-Founded model is also a three-valued model of $Comp(P)$ [48]; (iii) and (iv) are consequences of Theorem 4.4 in [5]. □

Semantics of transformed programs

An immediate consequence of Theorem 3.5.1 is the following.

Lemma 3.5.2 Let $P_0, \ldots, P_n$ be a transformation sequence, suppose that P_0 is acyclic, then $\Phi_{P_0}^{\uparrow\omega} = \Phi_{P_n}^{\uparrow\omega}$.

Proof. By Theorem 3.5.1, for each i, the Well-Founded model of P_i coincides with $\Phi_{P_i}^{\uparrow\omega}$ and by Proposition 4.1 in [92], the Well-Founded models of P_0 and P_n coincide. □

Because of Theorem 3.5.1, Corollary 3.4.4 has also some semantic consequences, the most relevant of which are:

Corollary 3.5.3 Let $P_0, \ldots, P_n$ be a transformation sequence, suppose that P_0 is acyclic, then

(a) the Fitting's models of P_0 and of P_n coincide;
(b) the set of ground logical consequences of $Comp(P_0) \cup \mathrm{WDCA}_{\mathcal{L}}$ and of $Comp(P_n) \cup \mathrm{WDCA}_{\mathcal{L}}$ coincide;
(c) for all ground atoms A such that no SLDNF-derivation of $P_0 \cup \{\leftarrow A\}$ and of $P_n \cup \{\leftarrow A\}$ flounders,
  - there exists a SLDNF-refutation for $P_0 \cup \{\leftarrow A\}$ iff there exists one for $P_n \cup \{\leftarrow A\}$,

- all SLDNF trees for $P_0 \cup \{ \leftarrow A\}$ are finitely failed iff all SLDNF trees for $P_n \cup \{ \leftarrow A\}$ are;

in particular we have that

(d) If P_0 is definite, then its Finite Failure Set coincides with the one of P_n. $\square$

This shows that if the initial program is acyclic, then the transformation enjoys most of the properties that were proven for Seki's more restrictive modified folding. In some situations this can be useful for relaxing the applicability of the folding operation.

Chapter 4

Transforming Normal Logic Programs by Replacement

In this chapter we study *simultaneous replacement* which consists in performing many replacements all at the same time, and define *applicability conditions* able to guarantee the correct application of the operation in normal programs with respect to the semantics of the logical consequences of the program's completion (Kunen's semantics). We also take into consideration the case in which we adopt some domain closure axioms, this will allow us to draw conclusions for Fitting's semantics as well. As we mentioned in chapter 1, a basic requirement for the applicability of replacement is that the replaced and replacing parts are equivalent with respect to the considered semantics. But this alone is not sufficient to avoid the risk of introducing a loop. For this reason we introduce two new concepts: the *semantic delay between two conjunctions of literals* and the *dependency degree of a conjunction of literals wrt a clause*: the applicability conditions for replacement we propose compare the semantic delay between the two conjunctions of literals and the dependency degree of the replaced part with the clause to be transformed. In this way it is possible to characterize some situation in which "there is no space to introduce a loop". Such applicability conditions are undecidable in general, but decidable syntactic conditions can be derived for special cases. For instance in chapter 5 these results will be used for proving the correctness of an unfold/fold transformation sequence wrt Fitting's semantics.

Structure of the Chapter

In Section 4.1 we study the correctness of the replacement operation wrt Kunen's semantics. In section 4.2 we reformulate the results for the cases in which we adopt some domain closure axioms. In Section 4.3 some examples are provided and it is shown also how thinning and fattening can be seen as special cases of replacement, thus yielding, as a consequence, conditions for a safe application of these operations to normal programs. A short conclusion follows. Part of the proofs are given in the Appendices.

The simultaneous replacement operation

The replacement operation has been introduced by Tamaki and Sato in [96] for definite programs. Syntactically it consists in substituting a conjunction, $\tilde{C}$, of literals with another one, $\tilde{D}$, in the body of a clause. Similarly, *simultaneous* replacement consists in substituting a set of conjunctions of literals $\{\tilde{C}_1, \ldots, \tilde{C}_n\}$, with another corresponding set of conjunctions $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ in the bodies of some clauses $\{cl_1, \ldots, cl_p\}$ of a program P. We assume that if $i \neq j$ then $\tilde{C}_i$ and $\tilde{C}_j$ do not overlap, even if they may actually represent identical literals, that is, they are either in different clauses or in disjoint subsets of the same clause.

Note that, because of the semantics we consider, the order of literals in the bodies of the clauses is irrelevant.

4.1 Correctness wrt Kunen's semantics

OCHO

In this Section we will always refer to a fixed but unspecified *infinite* language $\mathcal{L}$, that we assume contains all the function symbols of the programs we are considering. Again, by *infinite* language, we mean a language that contains infinitely many functions symbols (including those of arity 0). As we explained in section 2.2, three valued program's completion semantics in the case of an infinite language is commonly referred to as *Kunen's semantics.*

Assume P' is obtained by transforming P, then Definition 2.2.2 (program's equivalence) is used to define the *correctness* of a transformation operation as follows.

Definition 4.1.1 Let P, P' be normal programs. Suppose that P' is obtained by applying a transformation operation to P. We say that the transformation is

- *Partially Correct* when for each allowed formula ϕ, if $Comp_{\mathcal{L}}(P') \models \phi$ then also $Comp_{\mathcal{L}}(P) \models \phi$.
- *Complete* when for each allowed formula ϕ, if $Comp_{\mathcal{L}}(P) \models \phi$ then also $Comp_{\mathcal{L}}(P') \models \phi$.
- *Totally Correct* or *Safe* when it is both partially correct and complete. This is the case in which P and P' are *equivalent* . □

Note that the transformation is partially correct if all the information contained in (the semantics of) P' was already present in (the semantics of) P, that is if no new knowledge was added to the program during the transformation. On the other hand the transformation is complete if no information is lost during the transformation.

Partial correctness

When we replace the conjunction $\tilde{C}$ with $\tilde{D}$ in the body of a clause, we are actually replacing a subformula inside a formula, the clause itself. Clearly, some conditions are needed to guarantee the safeness of the operation. When we abstract from the particular context, that is from the specific clause where the replacement occurs, a natural condition for replacing a (possibly open) formula χ by a (possibly open) formula ϕ is their *equivalence* in the sense of the following definition.

Before stating it we need to establish some further notation: given the formulas ζ, χ and ϕ, we denote by $\zeta[\phi/\chi]$ the formula obtained from ζ by replacing all occurrences of the subformula χ by ϕ.

Definition 4.1.2 (equivalence of formulas) Let χ, ϕ be first order formulas. We say that

- χ *is less specific or equal to* ϕ *(wrt* $Comp_{\mathcal{L}}(P)$*)*, $\chi \preceq_{Comp_{\mathcal{L}}(P)} \phi$, iff for each allowed formula ζ and each substitution σ,
$$Comp_{\mathcal{L}}(P) \models \zeta\sigma \quad \text{implies} \quad Comp_{\mathcal{L}}(P) \models \zeta[\phi/\chi]\sigma;$$
- χ *is equivalent to* ϕ *wrt* $Comp_{\mathcal{L}}(P)$, $\chi \cong_{Comp_{\mathcal{L}}(P)} \phi$, iff $\chi \preceq_{Comp_{\mathcal{L}}(P)} \phi$ and $\phi \preceq_{Comp_{\mathcal{L}}(P)} \chi$. □

The following Example shows how the problem of the equivalence of formulas naturally arises when using the replacement operation.

Example 4.1.3 Let us consider the following program:

$$\begin{array}{lll}
 & m1(El, [El \mid Tail], s(0)). & \\
 & m1(El, [X \mid Tail], s(N)) & \leftarrow m1(El, Tail, N). \\
 & m2(El, [El \mid Tail]). & \\
 & m2(El, [X \mid Tail]) & \leftarrow m2(El, Tail). \\
d: & common_element(L1, L2) & \leftarrow m1(El, L1, N1), m1(El, L2, N2).
\end{array}$$

Both predicates $m1$ and $m2$ behave like "member" predicates. The only difference between the two is that $m1$ "reports", as third argument, the location where element El has been found. As far as the definition of $common\_element$ goes, this is totally unnecessary, and we can replace the conjunction $m1(El, L1, N1), m1(El, L2, N2)$ with the conjunction $m2(El, L1), m2(El, L2)$ in the body of d, without affecting the semantics of the program. In practice we want to replace clause d with
$d' : common\_element(L1, L2) \leftarrow m2(El, L1), m2(El, L2)$.
Now observe that the completed definition of $common\_element$ before the transformation is

$$common\_element(L1, L2) \Leftrightarrow \exists\, N,\ M.\ m1(El, L1, N), m1(El, L2, M), \tag{4.1}$$

while after the transformation it is

$$common\_element(L1, L2) \Leftrightarrow m2(El, L1), m2(El, L2). \tag{4.2}$$

When applying a replacement we want the replacing conjunction to be semantically equivalent to the replaced one. In this particular case we can formalize this statement by requiring the equivalence of the two "bodies", (4.1) and (4.2), of the completed definition of $common\_element$, that is, we require that

$$\exists\, N,\ M.\ m1(El, L1, N), m1(El, L2, M) \cong_{Comp_{\mathcal{L}}(P)} m2(El, L1), m2(El, L2). \tag{4.3}$$

Which is easy to prove true. □

In (4.3) we have specified two existentially quantified variables: N and M which are *local* to the replaced conjunct. If we didn't do so, (4.3) would not hold, as $m1(El, L1, N), m1(El, L2, M) \ \not\cong_{Comp_{\mathcal{L}}(P)} \ m2(El, L1), m2(El, L2)$. In the sequel, when replacing, say, $\tilde{C}$ with $\tilde{D}$, we always specify a set $\tilde{x}$ of "local" variables, which are variables that can appear in either $\tilde{C}$ or $\tilde{D}$ (or both) but cannot occur in the rest of the clause where $\tilde{C}$ is found. Consequently, our first requirement is the equivalence of $\exists\, \tilde{x}\, \tilde{C}$ and $\exists\, \tilde{x}\, \tilde{D}$. Such an equivalence is weaker than the equivalence between $\tilde{C}$ and $\tilde{D}$, but still sufficient for our purposes.

We now formalize this concept of local variables for simultaneous replacement. First let us establish the notation we'll use throughout the chapter.

Notation 4.1.4
P is the normal program we want to transform.
$\tilde{C}_1, \ldots, \tilde{C}_n$ are the conjunctions of literals we want to replace with $\tilde{D}_1, \ldots, \tilde{D}_n$.
$\{cl_1, \ldots, cl_p\}$ is the subset of P consisting of the clauses that are going to be affected by the transformation.
P' is the result of the transformation. □

Definition 4.1.5 (locality property) Referring to Notation 4.1.4, we say that a set of variables $\tilde{x}_i$ satisfies the *locality property* with respect to $\tilde{C}_i$ and $\tilde{D}_i$ if the following holds:

- $\tilde{x}_i \subseteq Var(\tilde{C}_i) \cup Var(\tilde{D}_i)$ and the variables in $\tilde{x}_i$ do not occur anywhere else neither in the clause cl_j, where $\tilde{C}_i$ is found, nor, after replacement, in cl'_j, where $\tilde{D}_i$ is found. □

Note that the locality property is trivially satisfied when $\tilde{x}_i$ is empty. Note also that the locality property implies that if $\tilde{C}_h$ and $\tilde{C}_k$ occur in the same clause then the corresponding $\tilde{x}_h$ and $\tilde{x}_k$ are disjoint.

Before we state the result on partial correctness, we have to give a characterization of the equivalence of formulas wrt Kunen's semantics, which refers solely to the Kleene sequence of the operator Φ_P. Here we denote by $FV(\chi)$ the set of free variables in a formula χ.

Lemma 4.1.6 Let P be a normal program, χ, ϕ be first order allowed formulas and $\tilde{x} = \{x_1, \ldots, x_k\} = FV(\chi) \cup FV(\phi)$, The following statements are equivalent

(a) $\chi \ \preceq_{Comp_{\mathcal{L}}(P)} \ \phi$;
(b) $\forall\, n\, \exists\, m\, \forall\, \tilde{t} \quad \Phi_P^{\uparrow n} \models (\neg)\chi(\tilde{t}/\tilde{x}) \quad implies \quad \Phi_P^{\uparrow m} \models (\neg)\phi(\tilde{t}/\tilde{x})$;

where n, m are quantified over natural numbers and $\tilde{t}$ is quantified over k-tuples of $\mathcal{L}$-terms.

Proof. The proof is given in the Appendix A. □

We can finally state the result on partial correctness of the replacement operation we were aiming at. As we anticipated at the beginning of this Section, when replacing $\tilde{C}$ with $\tilde{D}$, our first requirement is the equivalence of $\exists\, \tilde{x}\, \tilde{C}$ and $\exists\, \tilde{x}\, \tilde{D}$, where x is a set of variables satisfying the locality property. However, if we are only interested in proving the *partial* correctness of the operation, a partial equivalence (namely, that $\exists\, \tilde{x}\, \tilde{D} \ \preceq_{Comp_{\mathcal{L}}(P)} \ \exists\, \tilde{x}\, \tilde{C}$) is perfectly sufficient. This is shown by the following Theorem. Again we adopt Notation 4.1.4.

Theorem 4.1.7 (partial correctness) If for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{D}_i \ \preceq_{Comp_{\mathcal{L}}(P)} \ \exists\, \tilde{x}_i\, \tilde{C}_i$$

then the simultaneous replacement operation is partially correct.

Proof. First let us make the following observation. With the exception of clauses $\{cl_1, \ldots, cl_p\}$, P is just like P'. Hence if for each i, $\exists\, \tilde{x}_i\, \tilde{C}_i$ and $\exists\, \tilde{x}_i\, \tilde{D}_i$ had the same meaning in a given interpretation I, (that is, if $I \models \exists\, \tilde{x}_i\, \tilde{C}_i \Leftrightarrow \exists\, \tilde{x}_i\, \tilde{D}_i$), then we would have that $\Phi_P(I) = \Phi_{P'}(I)$. It follows that whenever $\Phi_P(I) \neq \Phi_{P'}(I)$, there has to be an index j such that $\exists\, \tilde{x}_j\, \tilde{C}_j$ and $\exists\, \tilde{x}_j\, \tilde{D}_j$ have different meanings in I. This idea is formalized and extended in the following Lemma, whose proof is given in the Appendix A.

Lemma 4.1.8 Let I, I' be two partial interpretations. If $I' \subseteq I$ but $\Phi_{P'}(I') \not\subseteq \Phi_P(I)$, then there exist a conjunction $\tilde{C}_j \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$ and a ground substitution θ such that:

- either $I' \models \exists\, \tilde{x}_j\, \tilde{D}_j\theta$, while $I \not\models \exists\, \tilde{x}_j\, \tilde{C}_j\theta$;
- or $I' \models \neg\exists\, \tilde{x}_j\, \tilde{D}_j\theta$, while $I \not\models \neg\exists\, \tilde{x}_j\, \tilde{C}_j\theta$. □

Now we proceed with the proof, which is by contradiction. By Theorems 2.2.3 and 2.2.4 the operation is partially correct iff $\forall\, n\ \exists\, m \quad \Phi_P^{\uparrow m} \supseteq \Phi_{P'}^{\uparrow n}$, so let us suppose there exist two integers i and j such that:

$$\Phi_P^{\uparrow i} \supseteq \Phi_{P'}^{\uparrow j} \quad \text{and} \quad \text{for all integers } l,\ \Phi_P^{\uparrow l} \not\supseteq \Phi_{P'}^{\uparrow j+1}.$$

Clearly it also follows that

$$\text{for all integers } l,\ \Phi_P^{\uparrow l+i+1} \not\supseteq \Phi_{P'}^{\uparrow j+1}.$$

Since $\Phi_{P'}^{\uparrow j+1} = \Phi_{P'}(\Phi_{P'}^{\uparrow j})$, $\Phi_P^{\uparrow i} \supseteq \Phi_{P'}^{\uparrow j}$ and $\Phi_{P'}$ is monotone, we have that $\Phi_{P'}(\Phi_P^{\uparrow i}) \supseteq \Phi_{P'}^{\uparrow j+1}$, hence

$$\text{for all integers } l,\ \Phi_P(\Phi_P^{\uparrow l+i}) \not\supseteq \Phi_{P'}(\Phi_P^{\uparrow i}).$$

Since $\Phi_P^{\uparrow l+i} \supseteq \Phi_P^{\uparrow i}$, from Lemma 4.1.8, it follows that for each integer l there exist an integer $j(l) \in \{1, \ldots, n\}$ and a ground substitution θ_l such that:

$$\exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ (or } \mathit{false}\text{) in } \Phi_P^{\uparrow i}, \text{ while } \exists\, \tilde{x}_{j(l)}\, \tilde{C}_{j(l)}\theta_l \text{ is not } \mathit{true} \text{ (resp. } \mathit{false}\text{) in } \Phi_P^{\uparrow l+i}. \tag{4.4}$$

By hypothesis $\exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)} \ \preceq_{Comp_{\mathcal{L}}(P)} \ \exists\, \tilde{x}_{j(l)}\, \tilde{C}_{j(l)}$, we can then apply Lemma 4.1.6 to the left hand side of (4.4). It follows that there has to be an integer r such that for each l,

$$\exists\, \tilde{x}_{j(l)}\, \tilde{C}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ (resp } \mathit{false}\text{) } \mathit{in}\ \Phi_P^{\uparrow r};$$

but when l satisfies $l+i > r$, we have that $\Phi_P^{\uparrow l+i} \supseteq \Phi_P^{\uparrow r}$ and hence

$$\text{for each } l \text{ such that } l+i > r, \quad \exists\, \tilde{x}_{j(l)}\, \tilde{C}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ (resp } \mathit{false}\text{) in } \Phi_P^{\uparrow l+i}.$$

This contradicts (4.4). □

An immediate consequence of previous Theorem 4.1.7 is the following simple Corollary on total correctness.

Corollary 4.1.9 Using Notation 4.1.4, if for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{D}_i \;\cong_{Comp_{\mathcal{L}}(P)}\; \exists\, \tilde{x}_i\, \tilde{C}_i$$

then P is equivalent to P' iff, for each i, $\exists\, \tilde{x}_i\, \tilde{D}_i \;\cong_{Comp_{\mathcal{L}}(P')}\; \exists\, \tilde{x}_i\, \tilde{C}_i$.

Proof.
"if". From the assumption that $\exists\, \tilde{x}_i\, \tilde{D}_i \;\cong_{Comp_{\mathcal{L}}(P)}\; \exists\, \tilde{x}_i\, \tilde{C}_i$ and Theorem 4.1.7 it follows that for each allowed formula ϕ, if $Comp_{\mathcal{L}}(P') \models \phi$ then $Comp_{\mathcal{L}}(P) \models \phi$. Now P can be re-obtained from P' by replacing back each $\tilde{D}_i$ with $\tilde{C}_i$, moreover each set of variables $\tilde{x}_i$ satisfies the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ also in P'. Since by hypothesis $\exists\, \tilde{x}_i\, \tilde{D}_i \;\cong_{Comp_{\mathcal{L}}(P')}\; \exists\, \tilde{x}_i\, \tilde{C}_i$, from Theorem 4.1.7 it also follows that , if $Comp_{\mathcal{L}}(P) \models \phi$, then $Comp_{\mathcal{L}}(P') \models \phi$.
"only if". It is easy to see that if $\exists\, \tilde{x}_i\, \tilde{D}_i \;\cong_{Comp_{\mathcal{L}}(P)}\; \exists\, \tilde{x}_i\, \tilde{C}_i$ and P is equivalent to P' then $\exists\, \tilde{x}_i\, \tilde{D}_i \;\cong_{Comp_{\mathcal{L}}(P')}\; \exists\, \tilde{x}_i\, \tilde{C}_i$. □

Roughly speaking, this Corollary states that if the replacing and the replaced conjunctions are equivalent both in the initial and the resulting program, then the transformation is safe.

Of course this result requires some knowledge of the the semantics of the resulting program and therefore it is not quite satisfactory: what we want are applicability conditions for the replacement operation which are based solely on the semantic properties of the *initial* program. To this is devoted the rest of this Section.

Semantic Delay and Dependency Degree

As we proved in the previous Section, if $\tilde{x}$ is a set of variables that satisfies the locality property, the equivalence of $\exists\, \tilde{x}\, \tilde{C}$ and $\exists\, \tilde{x}\, \tilde{D}$ wrt $Comp_{\mathcal{L}}(P)$ is sufficient to guarantee the partial correctness of the replacement. Unfortunately this is not enough to ensure total correctness.

This is shown by the next Example.

Example 4.1.10 Let P be the following definite program:

$$\begin{array}{lrl} P = \{ & & p \leftarrow q. \\ & cl: & q \leftarrow r. \\ & & r. \qquad\qquad \} \end{array}$$

Let also $\mathcal{L} = \mathcal{L}(P)$. In this case p, q and r are all *true* in all the models of $Comp_{\mathcal{L}}(P)$, they are actually *equivalent* wrt $Comp_{\mathcal{L}}(P)$. However, if we replace r with p in the body of cl we obtain

$$\begin{array}{lll} P' = \{ & & p \leftarrow q. \\ & cl': & q \leftarrow p. \\ & & r. \qquad\qquad \} \end{array}$$

which is by no means equivalent to the previous program. In fact we have introduced a loop and p and q are no more *true* in all the models of $Comp_{\mathcal{L}}(P)$. $\square$

In order to obtain the desired completeness results we introduce two more concepts: the *semantic delay* and the *dependency degree.* They are meant to express relations between first order formulas, such as conjunctions of literals, in terms of their semantic properties.

Consider the following definite program:

$$\begin{array}{llll} P = \{ & m(X) & \leftarrow & n(s(X)). \\ & n(0). & & \\ & n(s(X)) & \leftarrow & n(X). \quad \} \end{array}$$

The predicates m and n have exactly the same meaning, but in order to refute the goal $\leftarrow m(s(0))$. we need four resolution steps, while for refuting $\leftarrow n(s(0))$. two steps are sufficient. Each time $\leftarrow n(t)$. has a refutation (or finitely fails) with j resolution steps, $\leftarrow m(t)$. has a refutation (or fails) with k resolution steps, where $k \leq j+2$. By transposing this idea into the three valued semantics we are adopting, we have that each time $n(t)$ is *true* (or *false*) in $\Phi_P^{\uparrow j}$, $m(t)$ is *true* (resp. *false*) in $\Phi_P^{\uparrow j+2}$. We can formalize this intuitive idea by saying that *the semantic delay of m wrt n is 2.*

Definition 4.1.11 (semantic delay in $\Phi_P^{\uparrow\omega}$) Let P be a normal program, χ and ϕ be first order formulas, and $\tilde{x} = \{x_1, \dots, x_k\} = FV(\chi) \cup FV(\phi)$. Suppose that $\phi \preceq_{Comp_{\mathcal{L}}(P)} \chi$.

- *The semantic delay of χ wrt ϕ in $\Phi_P^{\uparrow\omega}$* is the least integer k such that, for each integer n and each k-uple of $\mathcal{L}$-terms $\tilde{t}$: if $\Phi_P^{\uparrow n} \models (\neg)\phi(\tilde{t}/\tilde{x})$, then $\Phi_P^{\uparrow n+k} \models (\neg)\chi(\tilde{t}/\tilde{x})$. $\square$

Notice that since we are assuming that $\phi \preceq_{Comp_{\mathcal{L}}(P)} \chi$, if $\phi(\tilde{t}/\tilde{x})$ is *true* in some $\Phi_P^{\uparrow n}$, then there has to exists an integer m such that $\chi(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow m}$.
Intuitively, $\phi(\tilde{t}/\tilde{x})$ is *true* in $\Phi_P^{\uparrow n}$ iff its truth has been proved from scratch in at most n steps. The semantic delay of χ wrt ϕ shows how many steps later than $\phi(\tilde{t}/\tilde{x})$, we determine the truth value of $\chi(\tilde{t}/\tilde{x})$ (at worse).

Example 4.1.12 Let P be the following program:

$$\begin{array}{lllll} P = \{ & p(0). & & q(0). & \\ & p(s(0)). & & q(s(X)) & \leftarrow q(X). \\ & p(s(s(X))) & \leftarrow p(X). & & \qquad \} \end{array}$$

p and q both compute natural numbers, and $p(X) \cong_{Comp_{\mathcal{L}}(P)} q(X)$, but while

$q(s^k(0))$ is *true* starting from $\Phi_P^{\uparrow k+1}$, $p(s^k(0))$ is *true* starting from $\Phi_P^{\uparrow (k/2)+1}$. The delay of $p(X)$ wrt $q(X)$ in $\Phi_P^{\uparrow\omega}$ is zero, in fact if for some ground term t and integer n, $q(t)$ is *true* (resp. *false*) in $\Phi_P^{\uparrow n}$, then $p(t)$ is also *true* (resp. *false*) in $\Phi_P^{\uparrow n}$. Vice versa, the delay of $q(X)$ wrt $p(X)$ is not definable, in fact there exists no integer $m < \omega$ such that if, for some ground term t and integer n, $p(t)$ is *true* (resp. *false*) in $\Phi_P^{\uparrow n}$, then $q(t)$ is *true* (resp. *false*) in $\Phi_P^{\uparrow n+m}$. □

A simple property of semantic delay which is used in the sequel is the following.

Lemma 4.1.13 If $d: \ A \leftarrow \tilde{L}.$ is the only clause in a program P whose head unifies with an atom A, and $\tilde{w}$ is the set of variables local to the body of d, $\tilde{w} = Var(\tilde{L}) \backslash Var(A)$, then

- $A \ \cong_{Comp_{\mathcal{L}}(P)} \exists\, \tilde{w}\, \tilde{L}$;
- the delay of A wrt $\exists\, \tilde{w}\, \tilde{L}$ in $\Phi_P^{\uparrow\omega}$ is one.

Proof. It is a straightforward application of the definition of Fitting's operator, since, by Definition 2.1.6, for all integers r and substitutions θ, $(\exists\, \tilde{w}\, \tilde{L})\theta$ is *true* (*false*) in Φ_P^r iff $A\theta$ is *true* (*false*) in Φ_P^{r+1}. □

Now we want to introduce one further concept: the *dependency degree*. Let us consider the following normal program:

$$
\begin{array}{lllll}
P = \{ & c1: & p & \leftarrow & \neg q, s. \\
 & c2: & q & \leftarrow & r. \\
 & c3: & r. & & \\
 & c4: & s & \leftarrow & q. \quad \}
\end{array}
$$

The definitions of the atoms p, q, s and r, all depend from clause $c3$. Informally we could say that *the dependency degree of the predicate p over clause $c3$ is two*, as the shortest derivation path from a clause having head p to $c3$ contains two arcs: the first from $c1$ to $c2$, through the negative literal $\neg q$; the second from $c2$, to $c3$, through the atom r. Similarly, the dependency degree of q and s on $c3$ are respectively one and two and the dependency degree of r on $c3$ is zero. The next definition formalizes this intuitive notion. The atom A and the clause cl are assumed to be standardized apart.

Definition 4.1.14 (dependency degree) Let P be a program, cl a clause of P and A an atom. *The dependency degree of A (and $\neg A$) on cl*, $depen_P(A, cl)$, is

0 if A unifies with the head of cl;

n+1 if A does not unify with the head of cl and n is the least integer such that there exists a clause $C \leftarrow C_1, \ldots, C_k.$ in P, whose head unifies with A via mgu, say, θ, and, for some i, $depen_P(C_i\theta, cl) = n$;

ω when there exists no such n. In this case we say that *A is independent from cl.*

Now let $\tilde{L} = L_1, \ldots, L_n$ be a conjunction of literals. *The dependency degree of $\tilde{L}$ on cl* is equal to the least dependency degree of one of its elements on cl, $depen_P(\tilde{L}, cl) = inf\{depen_P(L_i, cl),$ where $1 \leq i \leq n\}$. Similarly, *$\tilde{L}$ is independent from cl* iff all its components are independent from cl. □

The following Example shows how the concepts of dependency degree and semantic delay can be used to prove the safeness of the replacement operation.

Example 4.1.15 Consider the following normal program:

$$\begin{array}{llll} P = \{ & d: & p(X) & \leftarrow & \neg q(X). \\ & cl: & r & \leftarrow & \ldots, \neg q(t), \ldots \\ & & & \ldots & \} \end{array}$$

where d is the only clause defining the predicate symbol p. By Lemma 4.1.13 $p(X) \cong_{Comp_{\mathcal{L}}(P)} \neg q(X)$. Now, if we replace $\neg q(t)$ with $p(t)$ in cl, we obtain the following program:

$$\begin{array}{llll} P' = \{ & d: & p(X) & \leftarrow & \neg q(X). \\ & cl: & r & \leftarrow & \ldots, p(t), \ldots \\ & & & \ldots & \} \end{array}$$

which has the same Kunen's semantics of the previous one, that is the set of logical consequences of $Comp_{\mathcal{L}}(P)$ and of $Comp_{\mathcal{L}}(P')$ are identical. This holds even if the definition of p is not independent from cl; that is, even if we are exposed to the risk of introducing a loop, losing completeness. But in this case we can show that "there is no room for introducing a loop"; in fact

- *the dependency degree of p on cl* (this is how big the loop would be) *is greater or equal to the semantic delay of $p(X)$ wrt $\neg q(X)$* (this can be seen as the "space" where the loop would have to be introduced).

By Lemma 4.1.13 the *delay* of $p(X)$ wrt $\neg q(X)$ in $\Phi_P^{\uparrow\omega}$ is one; moreover, since d is the only clause defining the predicate p and $d \neq cl$, $depen_P(p(X), cl) > 0$, thus satisfying the above conditions. □

Completeness

The aim of this section is to provide a completeness result which formalizes the idea outlined in Example 4.1.15 and that matches with Theorem 4.1.7. Throughout this Section we adopt Notation 4.1.4.

Let us first state a few simple results.

The first Remark states that when a conjunction of literals $\tilde{L}$ is *independent* from clauses $\{cl_1, \ldots, cl_p\}$ then its meaning does not change when replacing $\{cl_1, \ldots, cl_p\}$ with $\{cl'_1, \ldots, cl'_p\}$.

Remark 4.1.16 Let $\tilde{L}$ be a conjunction of literals independent from the clauses $\{cl_1, \ldots, cl_p\}$ in P. Let $\tilde{w} = Var(\tilde{L})$, Then, for each ordinal α,

- $\Phi_P^{\uparrow\alpha} \models (\neg)\exists\, \tilde{w}\, \tilde{L} \quad$ iff $\quad \Phi_{P'}^{\uparrow\alpha} \models (\neg)\exists\, \tilde{w}\, \tilde{L}$. □

The following Lemma represents an important step in the proof of the completeness result.

Let I be an $\mathcal{L}$-interpretation and B a ground atom that can be proved *true* (or *false*), starting from I, in m steps, that is, B is *true* in $\Phi_P^{\uparrow m}(I)$. The Lemma states that if the dependency level of B on $\{cl_1, \ldots, cl_p\}$ is greater or equal to m, then the clauses $\{cl_1, \ldots, cl_p\}$ cannot have been used in the proof of B, hence B is *true* in $\Phi_{P'}^{\uparrow m}(I)$ too.

Lemma 4.1.17 Let B be a ground atom, m a natural number such that $depen_P(B, \{cl_1, \ldots, cl_p\}) \geq m$; then

- B is *true* (resp. *false*) in $\Phi_P^{\uparrow m}(I)$ iff B is *true* (resp. *false*) in $\Phi_{P'}^{\uparrow m}(I)$.

Proof. The proof is by induction on m.

The base of the induction ($m = 0$) is trivial, since $\Phi_{P'}^{\uparrow 0}(I) = \Phi_P^{\uparrow 0}(I) = I$.

Induction step: $m > 0$. We will now proceed as follows: in a) we show that if B is *true* (resp. not *false*) in $\Phi_P^{\uparrow m}(I)$, then it is also *true* (resp. not *false*) in $\Phi_{P'}^{\uparrow m}(I)$. That is, we show that if B is *true* in $\Phi_P^{\uparrow m}(I)$, then it is also *true* in $\Phi_{P'}^{\uparrow m}(I)$; and, by contradiction, that if B is *false* in $\Phi_{P'}^{\uparrow m}(I)$, then it is also *false* in $\Phi_P^{\uparrow m}(I)$. In b) we consider the converse implications. This will be sufficient to prove the thesis.

a) Let us assume B *true* (resp. not *false*) in $\Phi_P^{\uparrow m}(I)$. There has to be a clause $c \in P$ and a ground substitution γ such that $head(c)\gamma = B$ and $body(c)\gamma$ is *true* (resp. not *false*) in $\Phi_P^{\uparrow m-1}(I)$. It follows that, for each literal L belonging to $body(c)\gamma$:

- L is *true* (resp. not *false*) in $\Phi_P^{\uparrow m-1}(I)$;
- $depen_P(L, \{cl_1, \ldots, cl_p\}) \geq m - 1$.

Then, from the inductive hypothesis, each L is *true* (resp. not *false*) in $\Phi_{P'}^{\uparrow m-1}(I)$. Since $depen_P(B, \{cl_1, \ldots, cl_p\}) \geq m > 0$, B does not unify with the head of any clause in $\{cl_1, \ldots, cl_p\}$, that is $c \notin \{cl_1, \ldots, cl_p\}$. Hence $c \in P'$ and B is *true* (not *false*) in $\Phi_{P'}^{\uparrow m}(I)$.

b) Now we have to prove that if B is *true* (not *false*) in $\Phi_{P'}^{\uparrow m}(I)$, then it is also *true* (not *false*) in $\Phi_P^{\uparrow m}(I)$. This part is omitted as it is perfectly symmetrical to the previous one. □

The previous Lemma leads to the following generalization.

Lemma 4.1.18 Let $\tilde{L}$ be a conjunction of literals, $\tilde{w} = Var(\tilde{L})$ and I be an $\mathcal{L}$-interpretation. Suppose that, for some integer m, $depen_P(\tilde{L}, \{cl_1, \ldots, cl_p\}) \geq m$, then,

- $\Phi_P^{\uparrow m}(I) \models (\neg)\exists\, \tilde{w}\, \tilde{L}$ iff $\Phi_{P'}^{\uparrow m}(I) \models (\neg)\exists\, \tilde{w}\, \tilde{L}$.

Proof. Let $\tilde{L} = L_1, \ldots, L_j$. Observe that $depen_P(\tilde{L}, \{cl_1, \ldots, cl_p\}) \geq m$ implies that for $i \in [1, j]$, $depen_P(L_i, \{cl_1, \ldots, cl_p\}) \geq m$.

Suppose first that $\exists\, \tilde{w}\, \tilde{L}$ is *true* in $\Phi_P^{\uparrow m}(I)$. Then for some ground substitution θ, with $Dom(\theta) = \tilde{w}$, $\tilde{L}\theta$ is *true* in $\Phi_P^{\uparrow m}(I)$. Then for $i \in [1, j]$, $L_i\theta$ is *true* in $\Phi_P^{\uparrow m}(I)$, and by Lemma 4.1.17, it is true also in $\Phi_{P'}^{\uparrow m}(I)$. Hence the conjunction $\tilde{L}\theta$ is *true* in $\Phi_{P'}^{\uparrow m}(I)$. It follows that $\exists\, \tilde{w}\, \tilde{L}$ is *true* in $\Phi_{P'}^{\uparrow m}(I)$.

Now suppose that $\exists\, \tilde{w}\, \tilde{L}$ is *false* in $\Phi_P^{\uparrow m}(I)$. Then for each ground substitution θ, with $Dom(\theta) = \tilde{w}$, $\tilde{L}\theta$ is *false* in $\Phi_P^{\uparrow m}(I)$. That is, for each of the above θ, there exists an $i \in [1, j]$ such that $L_i\theta$ is *false* in $\Phi_P^{\uparrow m}(I)$. By Lemma 4.1.17 $L_i\theta$ is also *false* in $\Phi_{P'}^{\uparrow m}(I)$. Hence $\tilde{L}\theta$ is *false* in $\Phi_{P'}^{\uparrow m}(I)$. It follows that $\exists\, \tilde{w}\, \tilde{L}$ is *false* in $\Phi_{P'}^{\uparrow m}(I)$. □

We can now state the completeness result. As before, we refer to Notation 4.1.4.

Recall that, when replacing $\tilde{C}$ with $\tilde{D}$, in order to prove the partial correctness of the replacement operation, we required that $\exists\,\tilde{x}\,\tilde{D}\ \preceq_{Comp_{\mathcal{L}}(P)}\ \exists\,\tilde{x}\,\tilde{D}$, where x is a set of variables satisfying the locality property. It should be no surprise that to prove the completeness of the operation we have to require the opposite side of the equivalence, namely that $\exists\,\tilde{x}\,\tilde{C}\ \preceq_{Comp_{\mathcal{L}}(P)}\ \exists\,\tilde{x}\,\tilde{D}$.

Theorem 4.1.19 (completeness) If for each $\tilde{C}_i \in \{\tilde{C}_1,\dots,\tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\,\tilde{x}_i\,\tilde{C}_i\ \preceq_{Comp_{\mathcal{L}}(P)}\ \exists\,\tilde{x}_i\,\tilde{D}_i,$$

and if one of the following two conditions holds:

(a) $\{\tilde{D}_1,\dots,\tilde{D}_n\}$ are all independent from the clauses $\{cl_1,\dots,cl_p\}$; or

(b) there exists an integer m such that, for each $\tilde{C}_i \in \{\tilde{C}_1,\dots,\tilde{C}_n\}$, and each $cl_j \in \{cl_1,\dots,cl_p\}$:
- the delay of $\exists\,\tilde{x}_i\,\tilde{D}_i$ wrt $\exists\,\tilde{x}_i\,\tilde{C}_i$ in $\Phi_P^{\uparrow\omega}$ is less or equal to m, and
- $depen_P(\tilde{D}_i, cl_j) \geq m$;

then the simultaneous replacement operation is complete.

Proof. First we need to establish a Lemma similar to the one in the proof of Theorem 4.1.7.

Lemma 4.1.20 Let I, I' be two partial interpretations. If $I \subseteq I'$ but $\Phi_P(I) \not\subseteq \Phi_{P'}(I')$, then there exist a conjunction $\tilde{C}_j \in \{\tilde{C}_1,\dots,\tilde{C}_n\}$ and a ground substitution θ such that:

- either $I \models \exists\,\tilde{x}_j\,\tilde{C}_j\theta$, while $I' \not\models \exists\,\tilde{x}_j\,\tilde{D}_j\theta$;
- or $I \models \neg\exists\,\tilde{x}_j\,\tilde{C}_j\theta$, while $I' \not\models \neg\exists\,\tilde{x}_j\,\tilde{D}_j\theta$.

Proof. The proof is identical to the one given in the Appendix A for Lemma 4.1.8 in Theorem 4.1.7, and it is omitted. □

Again the proof of the Theorem is by contradiction. By Theorems 2.2.3 and 2.2.4 the operation is complete iff $\forall\, n\ \exists\ m\ \ \Phi_P^{\uparrow n} \subseteq \Phi_{P'}^{\uparrow m}$, so let us suppose that there exist two integers i and j such that:

$$\Phi_{P'}^{\uparrow i} \supseteq \Phi_P^{\uparrow j}\ \text{ and }\ \text{for all integers } l,\ \Phi_{P'}^{\uparrow i+l+1} \not\supseteq \Phi_P^{\uparrow j+1}.$$

Since $\Phi_P^{\uparrow j+1} = \Phi_P(\Phi_P^{\uparrow j})$, from Lemma 4.1.20 we have that:
for each integer l there exists an integer $j(l) \in \{1,\dots,n\}$ and a ground substitution θ_l such that:

$$\exists\,\tilde{x}_{j(l)}\,\tilde{C}_{j(l)}\theta_l \text{ is } true \text{ (or } false\text{) in } \Phi_P^{j},\ \text{ while } \exists\,\tilde{x}_{j(l)}\,\tilde{D}_{j(l)}\theta_l \text{ is not } true \text{ (resp. not } false\text{) in } \Phi_{P'}^{\uparrow i+l}. \tag{4.5}$$

Let us distinguish two cases.

1) Hypothesis (a) is satisfied and each conjunction in $\{\tilde{D}_1,\dots,\tilde{D}_n\}$ is independent from $\{cl_1,\dots,cl_p\}$. By hypothesis $\exists\,\tilde{x}_i\,\tilde{C}_i\ \preceq_{Comp_{\mathcal{L}}(P)}\ \exists\,\tilde{x}_i\,\tilde{D}_i$, we can then apply

Lemma 4.1.6 to the left hand side of (4.5), it follows that there has to be an integer r such that for each l,

$$\exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l \text{ is } \mathit{true} \ \ \text{(resp. } \mathit{false}\text{) in } \Phi_P^r.$$

From Remark 4.1.16, it follows that for each integer l, $\exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l$ is *true* (resp. *false*) in $\Phi_{P'}^r$.
This contradicts (4.5); in fact, when $i + l > r$, by the monotonicity of $\Phi_{P'}$, we have that $\Phi_{P'}^r \subseteq \Phi_{P'}^{i+l}$ and since $\exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l$ is *true* (resp. *false*) in $\Phi_{P'}^r$, it must be *true* (resp. *false*) in $\Phi_{P'}^{i+l}$.

2) Hypothesis (b) is satisfied. We know that for each integer l, the delay of $\exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}$ wrt $\exists\, \tilde{x}_{j(l)}\, \tilde{C}_{j(l)}$ is not greater than m, hence from the left hand side of (4.5) it follows that,

$$\text{for each } l, \ \ \exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ or } \mathit{false} \text{ in } \Phi_P^{j+m}.$$

Since $\Phi_P^{j+m} = \Phi_P^m(\Phi_P^j)$, it follows that,

$$\text{for each } l, \ \ \exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ (resp. } \mathit{false}\text{) in } \Phi_P^m(\Phi_P^j).$$

$\mathit{depen}_P(\tilde{D}_{j(l)}\theta_l, \{cl_1, \ldots, cl_p\}) \geq m$, then, from Lemma 4.1.18 it follows that,

$$\text{for each } l, \ \ \exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ (resp. } \mathit{false}\text{) in } \Phi_{P'}^m(\Phi_P^j).$$

Now $\Phi_P^j \subseteq \Phi_{P'}^i$ and $\Phi_{P'}$ is monotone, then,

$$\text{for each } l, \ \ \exists\, \tilde{x}_{j(l)}\, \tilde{D}_{j(l)}\theta_l \text{ is } \mathit{true} \text{ (resp. } \mathit{false}\text{) in } \Phi_{P'}^m(\Phi_{P'}^i) = \Phi_{P'}^{m+i},$$

this contradicts the right hand side of (4.5). □

Finally, from Theorems 4.1.7 and 4.1.19 we obtain the following safeness result for the replacement operation.

Corollary 4.1.21 (applicability conditions for the replacement operation) Using Notation 4.1.4, if for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{D}_i \ \cong_{Comp_{\mathcal{L}}(P)} \ \exists\, \tilde{x}_i\, \tilde{C}_i$$

and one of the following two conditions holds:

1. $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ are all independent from the clauses in $\{cl_1, \ldots, cl_p\}$; or
2. there exists an integer m such that, for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, and each $cl_j \in \{cl_1, \ldots, cl_p\}$:
   - the delay of $\exists\, \tilde{x}_i\, \tilde{D}_i$ wrt $\exists\, \tilde{x}_i\, \tilde{C}_i$ in $\Phi_P^{\uparrow\omega}$ is less or equal to m, and
   - $\mathit{depen}_P(\tilde{D}_i, cl_j) \geq m$;

then the simultaneous replacement operation is safe, that is P is *equivalent* to P'. □

Conditions 1 and 2 reflect two different ways in which we can guarantee that we are not introducing dangerous loops. Condition 2 is automatically satisfied when, for each i, the semantic delay of $\exists\,\tilde{x}_i\,\tilde{D}_i$ wrt $\exists\,\tilde{x}_i\,\tilde{C}_i$ in $\Phi_P^{\uparrow\omega}$ is zero. This is probably the most interesting situation in which it can be applied. Recall that the semantic delay of $\exists\,\tilde{x}_i\,\tilde{D}_i$ wrt $\exists\,\tilde{x}_i\,\tilde{C}_i$ shows (for each θ) how many steps later than $\exists\,\tilde{x}_i\,\tilde{C}_i\theta$, we determine the truth value of $\exists\,\tilde{x}_i\,\tilde{D}_i\theta$ (at worse). Therefore, when the delay is zero, we can determine the truth value of $\exists\,\tilde{x}_i\,\tilde{D}_i\theta$ "faster" than the truth value $\exists\,\tilde{x}_i\,\tilde{C}_i\theta$. By stretching the notation we could say that in this case $\exists\,\tilde{x}_i\,\tilde{D}_i$ is "more efficient" than $\exists\,\tilde{x}_i\,\tilde{C}_i$. By the above Corollary we have that if the replacing conjunctions are "equivalent to" and "more efficient than" the replaced ones, then the replacement is safe. This fits well in a context where transformation operations are intended for increasing the performances of programs. Of course here we are referring to a bottom-up way of determining truth values, while most resolutions methods employ a top-down search, hence what is considered "more efficient" here may not necessarily be "more efficient" when we actually run the program.

Other Semantics

Corollary 4.1.21 can easily be applied to other declarative semantics. Basically what we need is a definition of equivalence and semantic delay: any model theoretic semantics which can be defined in terms of the Kleene sequence of some operator is potentially suitable. For example the Well-Founded semantics is appropriate, while the 2-valued completion semantics (considered in [47]) is not, as it lacks a constructive definition. Of course, when we change the semantics we refer to, the concept of equivalence of programs and formulas can differ significantly.

Let us for example consider the S-semantics [39], a model theoretic reconstruction of the computed answer semantics[1]. The S-semantics does not take into consideration the negative information that can be inferred from (the completion of) a program. This influences significantly the applicability conditions of replacement. Consider for instance the following program:

$$P = \{cl:\ p \leftarrow q, p.\}$$

q has no definition and therefore it fails. If we eliminate q from the body of cl, we obtain

$$P' = \{cl:\ p \leftarrow p.\}$$

The S-semantics (as well as the least Herbrand model semantics) of P and P' coincide (they are both empty as both p and q do not succeed in either program), so this transformation is (S-)safe. Now let us show how the S-correspondent of Corollary 4.1.21 can be applied to this situation: the transformation of P into P' can be seen as a replacement of q, p with p in the body of cl, and we have that

- q, p is equivalent to p in the S-semantics of P (neither succeeds),
- the delay of p wrt q, p in $T_S^{\omega}(P)$[2] is zero,

[1] A result similar to Corollary 4.1.21 for the S-semantics is given in [20]

[2] $T_S(P)$ is the S-semantics counterpart of Φ_P

- $depen_P(p, cl) = 0$,

Hence the applicability conditions for the S-version of Corollary 4.1.21 are satisfied.

Now, if we switch back to Kunen's semantics, P is no longer equivalent to P', in fact, $Comp_{\mathcal{L}}(P) \models \neg p$ while $Comp_{\mathcal{L}}(P') \not\models \neg p$. In the transformation we have lost some negative information, the replacement is therefore not (Kunen-)safe. Indeed, the applicability conditions of Corollary 4.1.21 are not satisfied as

- $q, p \cong_{Comp_{\mathcal{L}}(P)} p$,
- the delay of p wrt q, p in $\Phi_P^{\uparrow\omega}$ is **one**. ($\Phi_P^{\uparrow 1} \models \neg(q, p)$, while $\Phi_P^{\uparrow 2} \models \neg p$),
- $depen_P(p, cl) = 0$,

Here the delay of p wrt q, p is greater than $depen_P(p, cl)$ and consequently Corollary 4.1.21 is no longer applicable. This is due to the fact that, since we are now taking into account also the negative information, the delay of p wrt q, p is no longer zero.

However, there exists a semantics, the Well-Founded semantics, that does take into consideration negative information, but for which the above programs P and P' are nevertheless equivalent. Loosely speaking, the Well-Founded semantics does not distinguish *finite* from *infinite* failure. So the query $\leftarrow p$ fails both in P (finitely) and in P' (infinitely). The authors have also stated a counterpart of Corollary 4.1.21 for this semantics [38]. It can be applied to the transformation performed above: we have that q, p is equivalent to p and that the delay of p wrt q, p is zero. The applicability conditions for the replacement operation are then, in this context, satisfied.

Checking applicability conditions

Determining whether two conjunctions of literals are equivalent is in general an undecidable problem, moreover, the *semantic delay* is not a computable function, and for this reason Corollary 4.1.21 must be regarded as a theoretical result. It is therefore important to single out some situations in which its hypothesis can be guaranteed either by a syntactic check or, when the replacement belongs to a transformation sequence, by the previous *history* of the transformation. This Section shows some of these situations. Later, in Section 4.3 we also show an example of a transformation sequence in which the conditions of Corollary 4.1.21 are checked by hand. We hope that this provides a better understanding of the concepts we use.

Reversible folding

We now show how Corollary 4.1.21 can be used to prove the correctness of the reversible folding operation, which is the kind of folding operation studied in [67, 47]. First of all let us state its definition.

Definition 4.1.22 (reversible folding) Let $cl:\ A \leftarrow \tilde{B}', \tilde{S}.$ and $d:\ H \leftarrow \tilde{B}$ be distinct clauses in a program P; let also $\tilde{w}$ be the set of local variables of d, $\tilde{w} = Var(\tilde{B}) \backslash Var(H)$. If there exists a substitution θ, $Dom(\theta) = Var(d)$ such that

(i) $\tilde{B}' = \tilde{B}\theta$;

(ii) θ does not bind the local variables of d, that is for any $x, y \in \tilde{w}$ the following three conditions hold

- $x\theta$ is a variable;
- $x\theta$ does not appear in A, $\tilde{S}$, $H\theta$;
- if $x \neq y$ then $x\theta \neq y\theta$;

(iii) d is the only clause of P whose head unifies with $H\theta$;

then we can fold $H\theta$ in cl, obtaining $cl' :\ A \leftarrow H\theta, \tilde{S}$. □

Example 4.1.23 Let us consider the following program:

$$
\begin{array}{llll}
P = \{ & cl: & p(X) & \leftarrow q(X,b), \neg s(X), r(a,X). \\
 & d: & r(Z,Y) & \leftarrow q(Y,Z), \neg s(Y). \\
 & & r(a,Y) & \leftarrow p(Y). \\
 & & q(X,a). & \\
 & & q(X,b). & \qquad\qquad\qquad \}
\end{array}
$$

With $\theta = \{b/Z, X/Y\}$, we have $body(d)\theta = (q(X,b), \neg s(X))$ and that d is the only clause of P whose head unifies with $r(Z,Y)\theta$. Hence we can fold clause cl, thus obtaining the program:

$$
\begin{array}{llll}
P = \{ & cl: & p(X) & \leftarrow r(b,X), r(a,X). \\
 & d: & r(Z,Y) & \leftarrow q(Y,Z), \neg s(Y). \\
 & & r(a,Y) & \leftarrow p(Y). \\
 & & q(X,a). & \\
 & & q(X,b). & \qquad\qquad\qquad \}
\end{array}
$$

□

This operation can be seen as a special case of replacement in which the conditions of Corollaries 4.1.21 are always satisfied. First of all notice that, by using the notation of Definition 4.1.22, the operation reduces to a replacement of $\tilde{B}'$ with $H\theta$. Now by the conditions on folding (i)...(iii) and Lemma 4.1.13, we have that

- $\tilde{w}$ satisfies the locality property wrt $\tilde{B}'$ and H, (recall that $\tilde{w}$ is the set of local variables of d);

- $H\theta$ is equivalent to $\exists\, \tilde{w}\theta\, \tilde{B}'$, (Lemma 4.1.13);

- the delay of $H\theta$ wrt $\exists\, \tilde{w}\theta\, \tilde{B}'$ in $\Phi_P^{\uparrow\omega}$ is one, (Lemma 4.1.13).

Finally, from (iii) we also have that the dependency degree of $depen_P(H\theta, cl) > 0$. Hence, the applicability conditions of Corollary 4.1.21 are satisfied and the operation is safe.

Recursive folding

The reversible folding operation is a rather restrictive kind of folding, in particular it lacks the possibility of introducing recursion in the definition of predicates. This can be done via an *unfold/fold transformation sequence*. Unfold/fold transformation sequences were introduced in the area of logic programming by Tamaki and Sato [96] and, as a large literature shows, proved to be an effective methodology for program's development and optimization.

The following Example shows how this kind of folding can be used for introducing recursion in definitions.

Example 4.1.24 We start with the following program where *initial* defines the property of being a prefix of a list.

$$\begin{array}{llll} P_0 = \{ & d: & initial(Xs, Zs) & \leftarrow append(Xs, Ys, Zs). \\ & & append([A|Xs], Ys, [A|Zs]) & \leftarrow append(Xs, Ys, Zs). \\ & & append([\,], Ys, Ys). & \quad \} \end{array}$$

We now unfold the body of the first clause, obtaining the two clauses

$$\begin{array}{llll} P_1 = \{ & cl: & initial([A|Xs], [A|Zs]) & \leftarrow append(Xs, Ys, Zs). \\ & & initial([\,], Zs). & \\ & & \ldots & \text{together with the clauses defining } append \quad \} \end{array}$$

The safeness of the unfolding operation is proven in Appendix C. Now we can fold *append*(Xs, Ys, Zs) in the body of the first clause, using d as folding clause. We obtain

$$\begin{array}{llll} P_2 = \{ & cl': & initial([A|Xs], [A|Zs]) & \leftarrow \text{initial(Xs, Zs)} \\ & & initial([\,], Zs). & \\ & & \ldots & \text{together with the clauses defining } append \quad \} \end{array}$$

The predicate *initial* has now a recursive definition.

Notice that the folding operation of the above example can be seen as a replacement of *append*(Xs, Ys, Zs) with *initial*(Xs, Zs), and also in this case the applicability conditions of Corollary 4.1.21 are satisfied, in fact we have that:

- Ys satisfies the locality property wrt *append*(Xs, Ys, Zs) and *initial*(Xs, Zs) in P_1;

- $initial(Xs, Zs) \cong_{Comp_{\mathcal{L}}(P_1)} \exists Ys\, append(Xs, Ys, Zs)$;

- the delay of $initial(Xs, Zs)$ wrt $\exists Ys\, append(Xs, Ys, Zs)$ in P_1 is zero.

The last two statements are also consequences of the following more general result which will be proven in chapter 5 (it follows directly from Lemma 5.3.2).

Observation 4.1.25 Let $H \leftarrow \tilde{B}$ be a non-recursive clause in a program P and, $\tilde{w}$ be its set of local variables $\tilde{w} = Var(\tilde{B}) \backslash Var(H)$. If P' is a program obtained from P by unfolding all the atoms in $\tilde{B}$ then $H \cong_{Comp_{\mathcal{L}}(P')} \exists\, \tilde{w}\, B$, and the delay of H wrt $\exists\, \tilde{w}\, \tilde{B}$ in P' is zero. □

This provides a further example of the kind of situations to which Corollary 4.1.21 can be applied. Actually, chapter 5 we'll prove a correctness result over the correctness of unfold/fold transformation sequence by using the above observation and Fitting's counterpart of Corollary 4.1.21, Corollary 4.2.7.

4.2 Correctness wrt other semantics

The results we've just proved can be adapted to the cases in which we adopt some *domain closure axioms*. As we have seen in chapter 2 the adoption of such axioms is important when the underlying language $\mathcal{L}$ is finite. Recall that the two kind of domain closure axioms we'll adopt are the *weak domain closure axioms* ($\mathrm{WDCA}_{\mathcal{L}}$) and the *strong domain closure axioms* ($\mathrm{DCA}_{\mathcal{L}}$), both reported in definition 2.3.1.

It is important to observe that when we adopt some domain closure axioms, we have to modify in the obvious way the Definitions of programs equivalence (2.2.2), of formulas equivalence (4.1.2) and of correctness of a transformation (4.1.1).

Correctness Results wrt $Comp_{\mathcal{L}}(P) \cup \mathbf{WDCA}_{\mathcal{L}}$

As we explained in Section 2.3.1, as far as we are concerned the semantics given by $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ (with $\mathcal{L}$ possibly finite) behaves exactly as Kunen's semantics. Consequently, the results that we can prove on formula's equivalence and on the replacement operation are identical to the ones given in the previous Section. In particular Corollary 4.1.9, Lemma 4.1.6 on the equivalence of formulas, Theorems 4.1.7, 4.1.19 and Corollary 4.1.21 hold also for this semantics. Let us now restate this Corollary.

Corollary 4.2.1 (applicability conditions wrt $Comp_{\mathcal{L}} \cup \mathbf{WDCA}_{\mathcal{L}}$) Using Notation 4.1.4, if for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{D}_i \text{ is equivalent to } \exists\, \tilde{x}_i\, \tilde{C}_i \text{ wrt } Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}},$$

and one of the following two conditions holds:

1. $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ are all independent from the clauses in $\{cl_1, \ldots, cl_p\}$; or
2. there exists an integer m such that, for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, and each $cl_j \in \{cl_1, \ldots, cl_p\}$:
   - the delay of $\exists\, \tilde{x}_i\, \tilde{D}_i$ wrt $\exists\, \tilde{x}_i\, \tilde{C}_i$ in $\Phi_P^{\uparrow\omega}$ is less or equal to m, and
   - $depen_P(D_i, cl_j) \geq m$;

then the simultaneous replacement operation is safe, that is P is equivalent to P' (wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$). □

Correctness Results wrt Fitting's Semantics

In this section we refer to the semantics given by $Comp_{\mathcal{L}}(P)_{\mathcal{L}} \cup \mathrm{DCA}_{\mathcal{L}}$. As we have seen in Section 2.3.2, this semantics corresponds to Fitting's model semantics. Using Theorem 2.3.5 we can easily characterize the correctness of the transformation wrt to this semantics by referring to the least fixed point of the Φ_P operator.

Lemma 4.2.2 Let P, P' be normal programs and $\mathcal{L}$ be a finite language. Suppose that P' is obtained by applying a transformation operation to P. Then the operation is

- partially correct iff $Fit(P) \supseteq Fit(P')$;
- complete iff $Fit(P) \subseteq Fit(P')$;
- totally correct (safe) iff $Fit(P) = Fit(P')$. □

Partial Correctness

We now consider the problem of proving partial correctness of the replacement operation. When we replace the conjunction $\tilde{C}$ with $\tilde{D}$, the first natural requirement we ask for, is the equivalence of $\tilde{C}$ and $\tilde{D}$ wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$.

Here we need again Theorem 2.3.5 in order to give a characterization of the equivalence of formulas wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$. First we introduce the three valued operator $\Rightarrow$, which is "one side" of $\Leftrightarrow$ and it is defined as follows: $\phi \Rightarrow \chi$ is *true* iff ϕ is less specific than χ, that is if ϕ and χ are both *true* (or both *false*) or if ϕ is *undefined*. In any other case $\phi \Rightarrow \chi$ is *false*.

Lemma 4.2.3 Let χ, ϕ be first order allowed formulas and P be a normal program. The following statements are equivalent:

(a) $\chi \ \preceq_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}} \ \phi$;
(b) $Fit(P) \models \chi \Rightarrow \phi$.

Proof. The proof is given in Appendix A. □

Statement (b) differs from the corresponding one of Lemma 4.1.6. In Lemma 4.1.6 we were considering the completion with an infinite language, which as far as this Lemma is concerned, is equivalent to assuming a finite language and $\mathrm{WDCA}_{\mathcal{L}}$. In such cases the universe of a model of $Comp_{\mathcal{L}}(P)$ may contain non-standard elements, that is, elements which are not $\mathcal{L}$-terms. Hence the equivalence between all the closed instances of χ and ϕ alone is not sufficient to ensure the equivalence between χ and ϕ.

For example, if we consider the following program where, for simplicity, we refer to $\mathrm{WDCA}_{\mathcal{L}}$:

$$
\begin{array}{lll}
P = \{ & n(0). & \\
 & n(s(X)) \leftarrow n(X). & \\
 & m(X). & \}
\end{array}
$$

and we fix $\mathcal{L} = \mathcal{L}(P)$, we have that for each $\mathcal{L}$-term t, both $n(t)$ and $m(t)$ are *true* in all models of $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$, but $n(X) \ \not\cong_{Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}} \ m(X)$. In fact, let $\zeta \equiv \forall\, x \ m(x)$, then $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \models \zeta$, while $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}} \not\models \zeta[n(x)/m(x)]$ (see Example 2.3.2). Indeed $m(X)$ and $n(X)$ must not be considered equivalent wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$, in fact if we consider the following extension to program P:

$$
\begin{array}{llll}
P_1 = P \cup \{ & q_1 \leftarrow & \neg n(X). & \\
 & q_2 \leftarrow & \neg m(X). & \}
\end{array}
$$

and $\mathcal{L} = \mathcal{L}(P_1)$, $n(X)$ is equivalent to $m(X)$ while q_1 is not equivalent to q_2.

Next we give the theorem on partial correctness of the replacement operation we were aiming at. It still shows that a partial equivalence between the replacing and the replaced literals is sufficient to ensure the partial correctness of the replacement operation.

Theorem 4.2.4 (partial correctness) Let us adopt Notation 4.1.4, if for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{D}_i \ \preceq_{Comp_{\mathcal{L}}(P) \,\cup\, \mathrm{DCA}_{\mathcal{L}}}\ \exists\, \tilde{x}_i\, \tilde{C}_i$$

then the simultaneous replacement operation is partially correct.

Proof. The proof is by contradiction. By Lemma 4.2.2 and the fact that $Fit(P) = lfp(\Phi_P)$, we have that the operation is partially correct iff $lfp(\Phi_P) \supseteq lfp(\Phi_{P'})$, so let us suppose $lfp(\Phi_P) \not\supseteq lfp(\Phi_{P'})$. Since the sequence $\Phi_{P'}^{\uparrow 0}, \Phi_{P'}^{\uparrow 1}, \ldots$ is monotonically increasing and $\Phi_{P'}^{\uparrow 0} = (\emptyset, \emptyset) \subseteq lfp(\Phi_P)$, there has to be an ordinal α such that

$$lfp(\Phi_P) \supseteq \Phi_{P'}^{\uparrow \alpha} \ \text{ and } \ lfp(\Phi_P) \not\supseteq \Phi_{P'}^{\uparrow \alpha+1} = \Phi_{P'}(\Phi_{P'}^{\alpha}).$$

Hence $lfp(\Phi_P) \not\supseteq \Phi_{P'}(lfp(\Phi_P))$ and $\Phi_{P'}(lfp(\Phi_P)) \supseteq \Phi_{P'}(\Phi_{P'}^{\alpha})$, since Φ is monotone. Since $\Phi_P(lfp(\Phi_P)) = lfp(\Phi_P)$ we have that

$$\Phi_P(lfp(\Phi_P)) \not\supseteq \Phi_{P'}(lfp(\Phi_P)). \tag{4.6}$$

From Lemma 4.1.8 and (4.6) it follows that there exists an integer j and a ground substitution θ such that $\exists\, \tilde{x}_j\, \tilde{D}_j\theta$ is *true* (or *false*) in $lfp(\Phi_P)$, while $\exists\, \tilde{x}_j\, \tilde{C}_j\theta$ is not. This, by Lemma 4.2.3, contradicts the hypothesis. □

As it happened with Theorem 4.1.7, this result brings us to a first completeness result: with the notation of the previous Theorem, if for each i we also have that $\exists\, \tilde{x}_i\, \tilde{D}_i \ \cong_{Comp_{\mathcal{L}}(P) \,\cup\, \mathrm{DCA}_{\mathcal{L}}}\ \exists\, \tilde{x}_i\, \tilde{C}_i$, then the transformation is safe iff for each i, $\exists\, \tilde{x}_i\, \tilde{D}_i \ \cong_{Comp_{\mathcal{L}}(P') \,\cup\, \mathrm{DCA}_{\mathcal{L}}}\ \exists\, \tilde{x}_i\, \tilde{C}_i$. The proof is identical to the one given for Corollary 4.1.9.

Completeness

We want a completeness result which matches with Theorem 4.1.19. First of all we need a slightly stronger definition of semantic delay.

Definition 4.2.5 (semantic delay in $lfp(\Phi_P)$) Let P be a normal program, χ and ϕ be first order formulas, and $\tilde{x} = \{x_1, \ldots, x_k\} = FV(\chi) \cup FV(\phi)$. Suppose that $\phi \preceq_{Comp_{\mathcal{L}}(P) \,\cup\, \mathrm{DCA}_{\mathcal{L}}} \chi$.

- *The semantic delay of χ wrt ϕ in $lfp(\Phi_P)$* is the least integer k such that, for each ordinal α and each k-uple of $\mathcal{L}$-terms $\tilde{t}$: if $\Phi_P^{\uparrow \alpha} \models (\neg)\phi(\tilde{t}/\tilde{x})$, then $\Phi_P^{\uparrow \alpha+k} \models (\neg)\chi(\tilde{t}/\tilde{x})$. □

Unsurprisingly, the difference between this Definition and the one of semantic delay in $\Phi_P^{\uparrow\omega}$ (4.1.11) is that here we also have to consider ordinals which are greater that ω.

Now we can prove the completeness result in this case.

Theorem 4.2.6 (completeness) In the hypothesis of 4.1.4, if for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{C}_i \preceq_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}} \exists\, \tilde{x}_i\, \tilde{D}_i,$$

and if one of the following two conditions holds:

(a) $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ are all independent from the clauses $\{cl_1, \ldots, cl_p\}$; or
(b) there exists an integer m such that, for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, and each $cl_j \in \{cl_1, \ldots, cl_p\}$:
  - the delay of $\exists\, \tilde{x}_i\, \tilde{D}_i$ wrt $\exists\, \tilde{x}_i\, \tilde{C}_i$ in $lfp(\Phi_P)$ is less or equal to m, and
  - $depen_P(\tilde{D}_i, cl_j) \geq m$;

then the simultaneous replacement operation is complete.

Proof. The proof is by contradiction. By Lemma 4.2.2 and the fact that $Fit(P) = lfp(\Phi_P)$ we have that the operation is complete iff $lfp(\Phi_P) \subseteq lfp(\Phi_{P'})$, so let us suppose that $lfp(\Phi_P) \not\subseteq lfp(\Phi_{P'})$. By the same argument used in the proof of Theorem 4.2.4, it follows that there exists an ordinal α such that:

$$lfp(\Phi_{P'}) \supseteq \Phi_P^{\uparrow\alpha} \quad \text{and} \quad lfp(\Phi_{P'}) \not\supseteq \Phi_P^{\uparrow\alpha+1}.$$

Since $\Phi_{P'}(lfp(\Phi_{P'})) = lfp(\Phi_{P'})$, it follows that $\Phi_{P'}(lfp(\Phi_{P'})) \supseteq \Phi_P(\Phi_P^{\alpha})$.
From Lemma 4.1.20 there exists an integer j and a ground substitution θ such that:

$$\exists\, \tilde{x}_j\, \tilde{C}_j\theta \text{ is } true \text{ (or } false\text{) in } \Phi_P^{\alpha}, \quad \text{while} \quad \exists\, \tilde{x}_j\, \tilde{D}_j\theta \text{ is not } true \text{ (resp. not } false\text{) in } lfp(\Phi_{P'}). \tag{4.7}$$

Let us distinguish two cases.

1) Condition (a) of the hypothesis applies, and $\tilde{D}_j$ is independent from $\{cl_1, \ldots, cl_p\}$. Since $\Phi_P^{\alpha} \subseteq lfp(\Phi_P)$, from the left hand side of (4.7), $\exists\, \tilde{x}_j\, \tilde{C}_j\theta$ is also *true* (resp. *false*) in $lfp(\Phi_P)$.
Hence, by the hypothesis and Lemma 4.2.3, also $\exists\, \tilde{x}_j\, \tilde{D}_j\theta$ is *true* (resp. *false*) in $lfp(\Phi_P)$. Because of condition (a) and Remark 4.1.16, $\exists\, \tilde{x}_j\, \tilde{D}_j\theta$ is *true* (resp. *false*) in $lfp(\Phi_{P'})$. This contradicts the left hand side of (4.7).

2) Condition (b) of the hypothesis applies. The delay of $\exists\, \tilde{x}_j\, \tilde{D}_j$ wrt $\exists\, \tilde{x}_j\, \tilde{C}_j$ is not greater that m, hence from the left hand side of (4.7) it follows that $\exists\, \tilde{x}_j\, \tilde{D}_j\theta$ is *true* (or *false*) in $\Phi_P^{\alpha+m}$, that is, $\exists\, \tilde{x}_j\, \tilde{D}_j\theta$ is *true* (resp. *false*) in $\Phi_P^m(\Phi_P^{\alpha})$.
Since by (b), $depen_P(\tilde{D}_j\theta, \{cl_1, \ldots, cl_p\}) \geq m$, from Lemma 4.1.18 it follows that

$$\exists\, \tilde{x}_j\, \tilde{D}_j\theta \text{ is } true \text{ (resp. } false\text{) in } \Phi_{P'}^m(\Phi_P^{\alpha}).$$

Now $\Phi_P^{\alpha} \subseteq lfp(\Phi_{P'})$ and $\Phi_{P'}$ is monotone, then

$$\exists\, \tilde{x}_j\, \tilde{D}_j\theta \text{ is } true \text{ (resp. } false\text{) in } \Phi_{P'}^m(lfp(\Phi_{P'}))$$

But since $\Phi_{P'}^m(lfp(\Phi_{P'})) = lfp(\Phi_{P'})$, this contradicts the right hand side of (4.7) . □

Finally, from Theorems 4.2.4 and 4.2.6 we obtain the following result on the safeness of the replacement operation.

Corollary 4.2.7 (applicability conditions wrt $Comp_{\mathcal{L}} \cup \mathbf{DCA}_{\mathcal{L}}$ with $\mathcal{L}$ finite) In the hypothesis of 4.1.4, if for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, there exists a (possibly empty) set of variables $\tilde{x}_i$ satisfying the locality property wrt $\tilde{C}_i$ and $\tilde{D}_i$ such that

$$\exists\, \tilde{x}_i\, \tilde{D}_i \ \cong_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}} \ \exists\, \tilde{x}_i\, \tilde{C}_i$$

and one of the following two conditions holds:

1. $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ are all independent from the clauses in $\{cl_1, \ldots, cl_p\}$; or
2. there exists an integer m such that, for each $\tilde{C}_i \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$, and each $cl_j \in \{cl_1, \ldots, cl_p\}$:
   - the delay of $\exists\, \tilde{x}_i\, \tilde{D}_i$ wrt $\exists\, \tilde{x}_i\, \tilde{C}_i$ in $lfp(\Phi_P)$ is less or equal to m, and
   - $depen_P(D_i, cl_j) \geq m$;

then the simultaneous replacement operation is safe, that is, P is equivalent to P' (wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$). □

4.3 Replacement vs. other operations.

In this Section we consider the operations of thinning and fattening, and show how they can be seen as particular cases of replacement. We introduce them by means of an example of transformation sequence. This also give us the opportunity of illustrating how the applicability conditions for the replacement operation can be checked "by hand".

For the sake of simplicity, we consider the semantics given by $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$. The results hold also in the case we adopt $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ (and therefore also for Kunen's semantics) although the proofs are then more complicated.

Example 4.3.1 (sorting by permutation and check, part I) The following program is borrowed from [96]. The transformation process is intentionally redundant in order to be more explanatory.

Let P_0 be the following program:

$$
\begin{array}{llll}
P_0 = \{ & c1: & perm([\,],[\,]). & \\
& c2: & perm([A \mid Xs], Ys) & \leftarrow \quad perm(Xs, Zs), ins(A, Zs, Ys). \\
& c3: & ins(A, Xs, [A \mid Xs]). & \\
& c4: & ins(A, [B \mid Xs], [B \mid Ys]) & \leftarrow \quad ins(A, Xs, Ys). \\
& c5: & ord([\,]). & \\
& c6: & ord([A]). & \\
& c7: & ord([A, B \mid Xs]) & \leftarrow \quad A \leq B, ord([B \mid Xs]). \\
& c8: & sort(Xs, Ys) & \leftarrow \quad perm(Xs, Ys), ord(Ys). \\
& \ldots & & \qquad\qquad\qquad \}
\end{array}
$$

(1) If we unfold $perm(Xs, Ys)$ in the body of $c8$; the resulting program is:
$P_1 = \{c1, \ldots, c7\} \cup$

$$\begin{array}{llll} \{ & c9: & sort([\,],[\,]) & \leftarrow & ord([\,]). \\ & c10: & sort([A \mid Xs], Ys) & \leftarrow & perm(Xs, Zs), ins(A, Zs, Ys), ord(Ys).\} \end{array}$$

(2) By unfolding $ord([\,])$ in $c9$, we eliminate $ord([\,])$ from the body of that clause. $P_2 = \{c1, \ldots, c7\} \cup \{c10\} \cup \{\ c11:\ sort([\,],[\,]).\}$
By the safeness of the unfold operation (Corollary 4.7.2) P_0, P_1 and P_2 are equivalent programs both wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$ and $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$. □

Fattening

The *fatten* operation consists in introducing redundant literals in the body of a clause. It is generally used in order to make possible some other transformations such as folding.

Definition 4.3.2 (fatten) Let $cl : A \leftarrow \tilde{L}.$ be a clause in a program P and $\tilde{H}$ a conjunction of literals.

- *Fattening cl with $\tilde{H}$* consists of substituting cl' for cl, where $cl' : A \leftarrow \tilde{L}, \tilde{H}$.

$fatten\ (P, c, \tilde{H}) \stackrel{\text{def}}{=} P \backslash \{cl\} \cup \{cl'\}$. □

The fatten operation is a special case of replacement, and then its applicability conditions can be drawn directly from Corollaries 4.2.7 and 4.2.1.

The next Lemma shows that for fattening, part of the applicability conditions always hold.

Lemma 4.3.3 Let $cl = A \leftarrow \tilde{E}, \tilde{G}.$ be a clause in the normal program P, $\tilde{x}$ be a set of variables not occurring in $(A, \tilde{E})$ and $\tilde{H}$ be another conjunction of literals. Then

(a) If for each θ, $lfp(\Phi_P) \models \exists\, \tilde{x}\, \tilde{G}\theta$ implies $lfp(\Phi_P) \models (\exists\, \tilde{x}\, \tilde{G}, \tilde{H})\theta$,
then $\exists\, \tilde{x}\, \tilde{G} \ \preceq_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}}\ \exists\, \tilde{x}\, \tilde{G}, \tilde{H}$.

(b) If for each θ, $lfp(\Phi_P) \models \neg(\exists\, \tilde{x}\, \tilde{G}, \tilde{H})\theta$ implies $lfp(\Phi_P) \models \neg\exists\, \tilde{x}\, \tilde{G}\theta$
then $\exists\, \tilde{x}\, \tilde{G}, \tilde{H} \ \preceq_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}}\ \exists\, \tilde{x}\, \tilde{G}$.

(c) If m is an integer such that, for each α and θ, $\Phi_P^{\uparrow\alpha} \models \exists\, \tilde{x}\, \tilde{G}\theta$ implies $\Phi_P^{\uparrow\alpha+m} \models (\exists\, \tilde{x}\, \tilde{G}, \tilde{H})\theta$, then
- $\exists\, \tilde{x}\, \tilde{G} \ \preceq_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}}\ \exists\, \tilde{x}\, \tilde{G}, \tilde{H}$,
- the delay of $\exists\, \tilde{x}\, \tilde{G}, \tilde{H}$ wrt $\exists\, \tilde{x}\, \tilde{G}$ in $lfp(\Phi_P)$ is less or equal to m.
If m is the least of such integers, then the delay of $\exists\, \tilde{x}\, \tilde{G}, \tilde{H}$ wrt $\exists\, \tilde{x}\, \tilde{G}$ in $lfp(\Phi_P)$ is exactly m.

Proof. It is a straightforward application of Theorem 2.3.5 together with the fact that if $\tilde{G}\theta$ is *false* in some interpretation I, then also $(\tilde{G}, \tilde{H})\theta$ is *false* in I. □

This Lemma applies as well to the semantics given by $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$, as it is shown by Lemma 4.6.1 in the Appendix B.

Example 4.3.1 (sorting by permutation and check, part II)

(3) Now we can fatten clause $c10$ by adding $ord(Zs)$ to its body.
Let P_3 be the resulting program:
$P_3 = \{c1, \ldots, c7\} \cup$

$$\{ \quad c11: \; sort([\,],[\,]). \qquad c12: \; sort([A \mid Xs], Ys) \;\leftarrow\; perm(Xs, Zs), ord(Zs), ins(A, Zs, Ys), ord(Ys).\}$$

This operation corresponds to a replacement of $ins(A, Zs, Ys), ord(Ys)$ with $ord(Zs)$, $ins(A, Zs, Ys)$, $ord(Ys)$.

We now use Theorem 4.2.6 to prove that the operation is complete.

Observe that if $(ins(A, Zs, Ys), ord(Ys))\theta$ is *true* in $lfp(\Phi_{P_2})$ then $Ys\theta$ is an ordered list and $Zs\theta$ is a sublist of $Ys\theta$; hence also $Zs\theta$ is ordered and $(ord(Zs), ins(A, Zs, Ys), ord(Ys))$ is also *true* in $lfp(\Phi_{P_2})$. By Lemma 4.3.3, this is sufficient to state that:

$$ins(A, Zs, Ys), ord(Ys) \preceq_{Comp_{\mathcal{L}}(P_2) \,\cup\, \mathrm{DCA}_{\mathcal{L}}} ord(Zs), ins(A, Zs, Ys), ord(Ys)^3.$$

Moreover, the conjunction $ord(Zs), ins(A, Zs, Ys), ord(Ys)$ is independent from clause $c10$, hence, by Theorem 4.2.6, the operation is $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$-complete.

To show that the operation is safe we could use Corollary 4.2.7, but in this case it is easier to observe that $lfp(\Phi_{P_2})$ is also a *total* model[4], that is, no ground atom is *undefined* in it, and therefore that $lfp(\Phi_{P_2}) \subseteq lfp(\Phi_{P_3})$ implies that $lfp(\Phi_{P_2}) = lfp(\Phi_{P_3})$. By Lemma 4.2.2 this implies that the operation is also safe.

(4) We can now fatten $c12$ with $sort(Xs, Zs)$. The resulting program is:
$P_4 = \{c1, \ldots, c7\} \cup$

$$\{ \quad c11: \; sort([\,],[\,]). \qquad c13: \; sort([A \mid Xs], Ys) \;\leftarrow\; sort(Xs, Zs), perm(Xs, Zs), ord(Zs), ins(A, Zs, Ys), ord(Ys).\}$$

This operation corresponds to a replacement of $perm(Xs, Zs), ord(Zs)$ with $sort(Xs, Zs)$, $perm(Xs, Zs)$, $ord(Zs)$. Using Corollary 4.2.7 we can prove that the operation is safe, in order to do it we prove that:

(a) $sort(Xs, Zs), perm(Xs, Zs), ord(Zs) \cong_{Comp_{\mathcal{L}}(P_3) \,\cup\, \mathrm{DCA}_{\mathcal{L}}} perm(Xs, Zs), ord(Zs)$;
(b) the delay of $sort(Xs, Zs), perm(Xs, Zs), ord(Zs)$ wrt $perm(Xs, Zs), ord(Zs)$ in $lfp(\Phi_{P_3})$ is zero.

To prove (a) we proceed as follows: since $sort(Xs, Zs) \leftarrow perm(Xs, Zs), ord(Zs)$, is a clause of P_0, by Lemma 4.1.13, $sort(Xs, Zs) \cong_{Comp_{\mathcal{L}}(P_0) \,\cup\, \mathrm{DCA}_{\mathcal{L}}} perm(Xs, Zs), ord(Zs)$. This clearly implies that $sort(Xs, Zs), perm(Xs, Zs), ord(Zs) \quad \cong_{Comp_{\mathcal{L}}(P_0) \,\cup\, \mathrm{DCA}_{\mathcal{L}}}$

[3] When using WDCA instead of DCA, in order to establish the equivalence, computations are in general more complicated. In this Example it is sufficient to observe that $(ins(A, Zs, Ys), ord(Ys))\theta$ is *true* in $\Phi^n_{P_2}$ then also $ord(Zs)\theta$ is *true* in $\Phi^n_{P_2}$.

[4] This also follows from a result due to Apt and Bezem [5], that states that the Fitting's Model of an acyclic program is always a total model.

$perm(Xs, Zs), ord(Zs)$. Moreover, by the safeness of the previous transformation steps, P_0 is equivalent to P_3 and therefore, by a straightforward application of Lemma 4.2.3, we have that also (a) holds.

We now prove (b).

First, let us prove a few properties. In the following we denote the length of a list l by $|l|$.

(i) $ins(A, Zs, Ys)\theta$ becomes *true* at step $\Phi_{P_3}^{\uparrow n}$, where $n \leq |Ys\theta|$. In fact n is precisely the place where A ends up in Ys.
For example: $ins(a, [t, s, \ldots], [a, t, s, \ldots])$ is *true* in $\Phi_{P_3}^{\uparrow 1}$.
$ins(a, [t, s, \ldots], [t, a, s, \ldots])$ is *true* in $\Phi_{P_3}^{\uparrow 2}$.
$ins(a, [t, s, \ldots], [t, s, a, \ldots])$ is *true* in $\Phi_{P_3}^{\uparrow 3}$....
Moreover, when $ins(A, Zs, Ys)\theta$ is *true* in $lfp(\Phi_{P_3})$, we have that

$$|Ys\theta| = |Zs\theta| + 1. \tag{4.8}$$

(ii) $perm(Xs, Zs)\theta$ becomes *true* in $\Phi_{P_3}^{\uparrow |Zs\theta|+1}$.
This can be proven by induction on the length of $|Zs\theta|$.
$perm([\,], [\,])$ is *true* in $\Phi_{P_3}^{\uparrow 1}$;
if $|Zs\theta| > 0$ then $perm(Xs, Zs)\theta$ is *true* in $\Phi_{P_3}^{\uparrow \alpha}$ iff there exists an instance of $c2$,
$(perm([A'|Xs'], Ys') \leftarrow perm(Xs', Zs'), ins(A', Zs', Ys').)\theta'$,
such that
- $perm([A'|Xs'], Ys')\theta' = perm(Xs, Zs)\theta$ and
- $(perm(Xs', Zs'), ins(A', Zs', Ys'))\theta'$ is *true* in $\Phi_{P_3}^{\uparrow \alpha-1}$.
Now we can apply the inductive hypothesis and the previous results in order to determine $\alpha - 1$:
- $perm(Xs', Zs')\theta'$ is, by the inductive hypothesis, *true* in $\Phi_{P_3}^{\uparrow |Zs'\theta'|+1}$;
- $ins(A', Zs', Ys')\theta'$ becomes *true* at step $\Phi_{P_3}^{\uparrow n}$, where $n \leq |Ys'\theta'|$.
By (4.8), $|Ys'\theta'| = |Zs'\theta'|+1$, hence the conjunction $(perm(Xs', Zs'), ins(A', Zs', Ys'))\theta'$ becomes *true* exactly at step $\Phi_{P_3}^{\uparrow |Ys'\theta'|}$. But $|Ys'\theta'| = |Zs\theta|$, hence $perm(Xs, Zs)\theta$ becomes *true* at step $\Phi_{P_3}^{\uparrow |Zs\theta|+1}$.

(iii) $ord(Zs)\theta$ becomes *true* at step $\Phi_{P_3}^{\uparrow max(1,|Zs\theta|)}$.
This can be proven by induction on $|Zs\theta|$.

(iv) $sort(Xs, Zs)\theta$ becomes *true* at step $\Phi_{P_3}^{\uparrow |Zs\theta|+1}$.
This can also be proven by induction on $|Zs\theta|$.
$sort([\,], [\,])$ is *true* in $\Phi_{P_3}^{\uparrow 1}$.
When $|Zs\theta| > 0$, $sort(Xs, Zs)\theta$ is in $\Phi_{P_3}^{\uparrow \alpha}$ iff there exists an instance of $c12$:
$(sort([A \mid Xs'], Ys') \leftarrow perm(Xs', Zs'), ord(Zs'), ins(A, Zs', Ys'), ord(Ys').)\theta'$
such that
- $sort([A \mid Xs'], Ys')\theta' = sort(Xs, Zs)\theta$ and
- $(perm(Xs', Zs'), ord(Zs'), ins(A, Zs', Ys'), ord(Ys').)\theta'$ is *true* in $\Phi_{P_3}^{\uparrow \alpha-1}$.
Now to determine the value of $\alpha - 1$, we can use (i), (ii) and (iii):
- $perm(Xs', Zs')\theta'$ is *true* in $\Phi_{P_3}^{\uparrow |Zs'\theta'|+1}$;

- $ord(Zs')\theta'$ is $true$ in $\Phi_{P_3}^{\uparrow max(1,|Zs'\theta'|)}$;
- $ins(A, Zs', Ys')\theta'$ is $true$ in $\Phi_{P_3}^{\uparrow n}$, where $n \leq |Ys'\theta'|$;
- $ord(Ys')\theta'$ is $true$ in $\Phi_{P_3}^{\uparrow max(1,|Ys'\theta'|)}$.

Since $|Zs'\theta'|+1 = |Ys'\theta'| = |Zs\theta|$, $(perm(Xs', Zs'), ord(Zs'), ins(A, Zs', Ys'), ord(Ys'))\theta'$ becomes $true$ exactly at step $\Phi_{P_3}^{\uparrow|Ys'\theta|}$ and $sort(Xs, Zs)\theta$ becomes $true$ at step $\Phi_{P_3}^{\uparrow|Zs\theta|+1}$.

We can finally prove (b). By (iv), whenever $sort(Xs, Zs)\theta$ is $true$ in $lfp(\Phi_{P_3})$, it is $true$ in $\Phi_{P_3}^{\uparrow|Zs\theta|+1}$; but by (ii) and (iii), whenever $(perm(Xs, Zs), ord(Zs))\theta$ is $true$ in $lfp(\Phi_{P_3})$, it is also $true$ in $\Phi_{P_3}^{\uparrow|Zs\theta|+1}$.

This implies the following statement: for all θ, if $(perm(Xs, Zs), ord(Zs))\theta$ is $true$ in some $\Phi_{P_3}^{\uparrow k}$, then also $sort(Xs, Zs)\theta$ is $true$ in $\Phi_{P_3}^{\uparrow k}$.

Clearly, this can be restated as follows: for all θ, if $(perm(Xs, Zs), ord(Zs))\theta$ is $true$ in some $\Phi_{P_3}^{\uparrow k}$, then also $(sort(Xs, Zs), perm(Xs, Zs), ord(Zs))\theta$ is $true$ in $\Phi_{P_3}^{\uparrow k}$.

By Lemma 4.3.3 this implies (b). □

Thinning

The *thinning* operation is the converse of fattening, and allows one to eliminate superfluous literals from the body of a clause.

Definition 4.3.4 (thin) Let $cl : A \leftarrow \tilde{L}, \tilde{H}.$ be a clause in a program P.

- *Thinning cl of the literals* $\tilde{H}$ consists of substituting cl' for cl, where $cl' : A \leftarrow \tilde{L}$.

$thin(P, cl, \tilde{H}) \stackrel{\text{def}}{=} P \backslash \{cl\} \cup \{cl'\}$. □

As for fattening, thinning can be interpreted as a replacement and then its applicability conditions can be inferred from Corollaries 4.2.7 and 4.2.1. Moreover Lemma 4.3.3 applies in a natural way also to this operation; only statement (c) requires a symmetric formulation. We now restate only this last point.

Lemma 4.3.5 Let $cl = A \leftarrow \tilde{E}, \tilde{G}, \tilde{H}.$ be a clause in P and $\tilde{x}$ be a set of variables not occurring in $(A, \tilde{E})$. The following property holds:

- If m is an integer such that, for each α and θ, $\Phi_P^{\uparrow\alpha} \models \neg(\exists\, \tilde{x}\, \tilde{G}, \tilde{H})\theta$ implies $\Phi_P^{\uparrow\alpha+m} \models \neg\exists\, \tilde{x}\, \tilde{G}\theta$, then
  - $\exists\, \tilde{x}\, \tilde{G}, \tilde{H} \preceq_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}} \exists\, \tilde{x}\, \tilde{G}$,
  - the delay of $\exists\, \tilde{x}\, \tilde{G}$ wrt $\exists\, \tilde{x}\, \tilde{G}, \tilde{H}$ in $lfp(\Phi_P)$ is smaller or equal to m.

  If m is the least of such integers, then the delay of $\exists\, \tilde{x}\, \tilde{G}, \tilde{H}$ wrt $\exists\, \tilde{x}\, \tilde{G}$ in $lfp(\Phi_P)$ is exactly m.

Proof. It is a straightforward application of the fact that if $(\tilde{G}, \tilde{H})\theta$ is $true$ in some interpretation I, then also $\tilde{G}\theta$ is $true$ in I. □

In the Appendix B (Lemma 4.6.2) we state a corresponding Lemma for the case in which we adopt $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ instead of $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$.

Example 4.3.1 (sorting by permutation and check, part III)

(5) We can eliminate $ord(Zs)$ from the body of $c13$ by thinning it. The resulting program is:
$P_5 = \{c1, \ldots, c7\} \cup$

$$\begin{array}{lllll} \{ & c11: & sort([\,],[\,]). & & \\ & c14: & sort([A|Xs], Ys) & \leftarrow & sort(Xs, Zs), perm(Xs, Zs), ins(A, Zs, Ys), ord(Ys).\} \end{array}$$

This corresponds to replacing $ord(Zs), ins(A, Zs, Ys), ord(Ys)$ with $ins(A, Zs, Ys), ord(Ys)$. In order to prove that the operation is $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$-complete, we apply Theorem 4.2.6.
First we have to prove that

$$\text{if } ord(Zs)\theta \text{ is } \mathit{false} \text{ in } \mathit{lfp}(\Phi_{P_4}) \text{ then } (ins(A, Zs, Ys), ord(Ys))\theta \text{ is } \mathit{false} \text{ in } \mathit{lfp}(\Phi_{P_4})\ ^{5}. \tag{4.9}$$

This is easy to prove: if $ins(A, Zs, Ys)\theta$ is *false* in $\mathit{lfp}(\Phi_{P_4})$ then we have the thesis. Otherwise, since $\mathit{lfp}(\Phi_{P_4})$ is a total interpretation, $ins(A, Zs, Ys)\theta$ cannot be *undefined* in it, and $ins(A, Zs, Ys)\theta$ is *true* in $\mathit{lfp}(\Phi_{P_4})$, but in this case $Zs\theta$ is a sublist of $Ys\theta$, hence if $ord(Zs)\theta$ is *false* in $\mathit{lfp}(\Phi_{P_4})$, so is $ord(Ys)\theta$; and (4.9) follows. Now (4.9) implies that whenever $(ord(Zs), ins(A, Zs, Ys), ord(Ys))\theta$ is *false* in $\mathit{lfp}(\Phi_{P_4})$ then also $(ins(A, Zs, Ys), ord(Ys))\theta$ is *false* in $\mathit{lfp}(\Phi_{P_4})$, and, by Lemma 4.3.3, that

$$ord(Zs), ins(A, Zs, Ys), ord(Ys) \preceq_{Comp_{\mathcal{L}}(P_4) \cup \mathrm{DCA}_{\mathcal{L}}} ins(A, Zs, Ys), ord(Ys).$$

Since we also have that $ins(A, Zs, Ys), ord(Ys)$ is independent from $c13$, from Theorem 4.2.6 it follows that the operation is $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$-complete.

As in part **(3)**, since $\mathit{lfp}(\Phi_{P_4})$ is a total interpretation, $\mathit{lfp}(\Phi_{P_4}) \supseteq \mathit{lfp}(\Phi_{P_5})$ implies that $\mathit{lfp}(\Phi_{P_4}) = \mathit{lfp}(\Phi_{P_5})$. In other words, the completeness of the operation implies its safeness (wrt $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$).

(6) Finally we can eliminate $perm(Xs, Zs)$ from the body of $c14$ by a further thinning, thus obtaining:
$P_6 = \{c1, \ldots, c7\} \cup$

$$\begin{array}{lllll} \{ & c11: & sort([\,],[\,]). & & \\ & c15: & sort([A|Xs], Ys) & \leftarrow & sort(Xs, Zs), ins(A, Zs, Ys), ord(Ys).\} \end{array}$$

[5]When adopting WDCA instead of DCA, calculations are truly more complicated. In fact in order to ensure the equivalence, we have to show that for each j there is a k such that if $ord(Zs)\theta$ is *false* in $\Phi_{P_4}^{\uparrow j}$ then $(ins(A, Zs, Ys), ord(Ys))\theta$ is *false* in $\Phi_{P_4}^{\uparrow k}$.

This can be proved by the following schema: suppose that $ord(Zs)\theta$ is *false* in $\mathit{lfp}(\Phi_{P_4})$ and let $Ws\theta$ be the maximal ordered prefix of $Zs\theta$, then $ord(Zs)\theta$ becomes *false* at step $\Phi_{P_4}^{\uparrow |Ws\theta|}$. We have to distinguish two cases:
- if there is no $Xs\theta$ such that $Xs\theta$ is a prefix of $Ys\theta$ and $ins(A, Ws, Xs)\theta$ is *true* in some $\Phi_{P_4}^{\uparrow n}$, then $ins(A, Zs, Ys)\theta$ becomes *false* no later than $ord(Zs)\theta$ does, and we have the desired result.
- otherwise, either $X_s\theta$ is not ordered or it is the maximal ordered prefix of $Ys\theta$; in either cases, $ord(Ys)\theta$ becomes *false* no later than step $\Phi_{P_4}^{\uparrow |Xs\theta|}$.
In any case if $ord(Zs)\theta$ is *false* in $\Phi_{P_4}^{\uparrow j}$ then $(ins(A, Zs, Ys), ord(Ys))\theta$ is *false* in $\Phi_{P_4}^{\uparrow j+1}$.

This is an $O(n^3)$ sorting program, while P_0 runs in $O(n!)$.
To prove the $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$-completeness of this last step, we use Theorem 4.2.6. Let us distinguish two cases.

- If $Xs\theta = []$, then $perm(Xs, Zs)\theta$ is *false* in $\Phi_{P_5}^{\uparrow 1}$ iff $Zs\theta \neq []$, but in this case also $sort(Xs, Zs)\theta$ is *false* in $\Phi_{P_5}^{\uparrow 1}$;
- otherwise observe that the body of $c2$, which defines $perm$, is contained in the body of $c14$, defining $sort$. This implies that if some instance of $body(c2)$ is *false* in some interpretation I, then the corresponding instance of $body(c14)$ is *false* in I. Hence, if $perm([A|Xs], Zs)\theta$ is *false* in $\Phi_{P_5}(I)$ then $sort([A|Xs], Zs)\theta$ is *false* in $\Phi_{P_5}(I)$.

It follows that

$$\text{if } (sort(Xs, Zs), perm(Xs, Zs))\theta \text{ is } \textit{false} \text{ in } \Phi_{P_5}^{\uparrow j} \text{ then } sort(Xs, Zs)\theta \text{ is } \textit{false} \text{ in } \Phi_{P_5}^{\uparrow j}.$$

By Lemma 4.3.5, this is sufficient to show that $sort(Xs, Zs), perm(Xs, Zs) \preceq_{Comp_{\mathcal{L}}(P_5) \cup \mathrm{DCA}_{\mathcal{L}}} sort(Xs, Zs)$ and that the semantic delay of $sort(Xs, Zs), perm(Xs, Zs)$ wrt $sort(Xs, Zs)$ is zero, and hence, by Theorem 4.2.6, the operation is $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$-complete.

On the other hand, if $sort(Xs, Zs)\theta$ is *true* in some interpretation I, then $Zs\theta$ must be a reordering of $Xs\theta$, therefore $perm(Xs, Zs)\theta$ is also *true* in I. It follows that

$$\text{if } sort(Xs, Zs)\theta \text{ is } \textit{true} \text{ in } \textit{lfp}(\Phi_{P_5}) \text{ then also } (sort(Xs, Zs), perm(Xs, Zs))\theta \text{ is } \textit{true} \text{ in } \textit{lfp}(\Phi_{P_5}).$$

By Lemma 4.3.3, this implies that $sort(Xs, Zs) \preceq_{Comp_{\mathcal{L}}(P_5) \cup \mathrm{DCA}_{\mathcal{L}}} sort(Xs, Zs), perm(Xs, Zs)$, and hence, by Theorem 4.2.4, that the operation is also $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$-partially correct. □

4.4 Conclusions

In this chapter we study the simultaneous replacement operation for normal logic programs. Simultaneous replacement is a transformation operation which consists in substituting a set of conjunctions of literals $\{\tilde{C}_1, \ldots, \tilde{C}_n\}$ in the bodies of some clauses, with a set of equivalent conjunctions $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$. The set of logical consequences of the program's completion is considered as the semantics of the normal program. In this way we obtain three different semantics which depend on the domain closure axioms and on the finiteness properties of the language we choose. More precisely, the semantics we consider are:

- $Comp_{\mathcal{L}}(P)$,
  where $\mathcal{L}$ is an infinite language, this corresponds to Kunen's semantics.
- $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$,
  where $\mathcal{L}$ is a finite language, namely it has a finite number of function symbols, and WDCA is the set of Weak Domain Closure Axioms.
- $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$,
  where $\mathcal{L}$ is a finite language and DCA is the set of Domain Closure Axioms.

All these semantics can be characterized by means of the Kleene sequence of the three valued immediate consequence operator Φ_P.

For each of these semantics we define and characterize formulas equivalence, programs equivalence and safeness of program transformations, namely their correctness and completeness, and express them in terms of the Φ_P operator.

Furthermore, we propose applicability conditions for simultaneous replacement which guarantee safeness, that is the preservation of each semantics during the transformation. The equivalence between $\tilde{C}_i$ and $\tilde{D}_i$ is obviously necessary but it is generally not sufficient. In fact, as it is shown by Corollary 4.1.9, we also need the equivalence to hold after the transformation. Such equivalence can be destroyed when a $\tilde{D}_i$ depends on one of the clauses on which the replacement is performed. Hence we establish a relation between the level of dependency of $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ over the modified clauses and the difference in "semantic complexity" between each $\tilde{C}_i$ and $\tilde{D}_i$. Such semantic complexity is measured by counting the number of the applications of the immediate consequence operator which are necessary in order to determine the truth or falsity of a predicate.

By considering replacement as a generalization of other transformation operations such as thinning, fattening and reversible folding, we show how applicability conditions can be used also for them.

4.5 Appendix A.

Proof of Lemma 4.1.6

Lemma 4.1.6 Let P be a normal program, χ and ϕ be first order allowed formulas and $\tilde{x} = \{x_1, \ldots, x_k\} = FV(\chi) \cup FV(\phi)$. The following statements are equivalent

(a) $\chi \ \preceq_{Comp_{\mathcal{L}}(P)} \ \phi$;

(b) $\forall\, n\ \exists\, m\ \forall\, \tilde{t} \quad \Phi_P^{\uparrow n} \models (\neg)\chi(\tilde{t}/\tilde{x}) \quad implies \quad \Phi_P^{\uparrow m} \models (\neg)\phi(\tilde{t}/\tilde{x})$;

where n, m are quantified over natural numbers and $\tilde{t}$ is quantified over k-tuples of $\mathcal{L}$-terms.

Proof. (a) implies (b)
This part is by contradiction. Let us assume there exists a *fixed* n, such that for each integer m there exists a k-uple of $\mathcal{L}$-terms $\tilde{t}_m$ for which the following hold

(i) $\Phi_P^{\uparrow n} \models (\neg)\chi(\tilde{t}_m/\tilde{x})$;

(ii) $\Phi_P^{\uparrow m} \not\models (\neg)\phi(\tilde{t}_m/\tilde{x})$.

By Lemma 2.4.1 there exist two formulas T_χ^n and F_χ^n in the language of equality, such that $FV(T_\chi^n) = FV(F_\chi^n) = FV(\chi)$ and

$$\Phi_P^{\uparrow n} \models \forall\, \tilde{x}\ (T_\chi^n \rightarrow \chi \ \wedge \ F_\chi^n \rightarrow \neg\chi).$$

By Theorem 2.2.1

$$Comp_{\mathcal{L}}(P) \ \models \forall\, \tilde{x}\ (T_\chi^n \rightarrow \chi \ \wedge \ F_\chi^n \rightarrow \neg\chi).$$

By (a),

$$Comp_{\mathcal{L}}(P) \models \forall\, \tilde{x}\ (T^n_\chi \to \phi\ \wedge\ F^n_\chi \to \neg\phi).$$

This is an allowed formula, then by Theorem 2.2.1 there exists an r such that

$$\Phi_P^{\uparrow r} \models \forall\, \tilde{x}\ (T^n_\chi \to \phi\ \wedge\ F^n_\chi \to \neg\phi). \tag{4.10}$$

But by (i) $\chi(\tilde{t}_r/\tilde{x})$ is either *true* or *false* in $\Phi_P^{\uparrow n}$, let us now consider just the first possibility, that is

$$\Phi_P^{\uparrow n} \models \chi(\tilde{t}_r/\tilde{x})$$

the other case is perfectly symmetrical and omitted here.
From this and the definition of T^n_χ in Lemma 2.4.1, we have $\Phi_P^{\uparrow n} \models T^n_\chi(\tilde{t}_r/\tilde{x})$, and since $T^n_\chi(\tilde{t}_r)$ is a formula in the language of equality, if it is *true* in $\Phi_P^{\uparrow n}$ it must be *true* already at stage 0, that is $\Phi_P^{\uparrow 0} \models T^n_\chi(\tilde{t}_r/\tilde{x})$, but $\Phi_P^{\uparrow 0} \subseteq \Phi_P^{\uparrow r}$, hence

$$\Phi_P^{\uparrow r} \models T^n_\chi(\tilde{t}_r/\tilde{x}).$$

But then, by (4.10), $\Phi_P^{\uparrow r} \models \phi(\tilde{t_r}/\tilde{x})$, contradicting (ii).

(b) implies (a)
We prove that for each n there exists an m such that for any allowed formula ζ, and for any substitution σ,

$$\Phi_P^{\uparrow n} \models \zeta\sigma \quad \text{implies} \quad \Phi_P^{\uparrow m} \models \zeta[\phi/\chi]\sigma. \tag{4.11}$$

By Theorem 2.2.1 this implies (a).

Fix an n, and let m be an integer that satisfies hypothesis (b). It is not restrictive to assume that $m \geq n$. Let ζ be an allowed formula and σ a substitution such that

$$\Phi_P^{\uparrow n} \models \zeta\sigma.$$

If ζ does not contain χ as a subformula then (4.11) follows immediately from the assumption that $m \geq n$. In the case that ζ contains χ as a subformula we proceed by induction on the structure of ζ.

Base step: $\zeta = \chi$, then (4.11) follows immediately from (b).

Induction step: we consider three cases:

1) If $\zeta = \triangle\ \zeta_1$, where $\triangle$ is any allowed unary connective, or $\zeta = \zeta_1 \diamond \zeta_2$, where $\diamond$ is any allowed binary connective, then we have that either ζ_i does not contain χ as a subformula (and the result holds trivially) or the inductive hypothesis applies.

2) $\zeta = \forall\, w\ \zeta_1$.
For each $\mathcal{L}$-term t, let γ_t be the substitution $[t/w]$. Since $\Phi_P^{\uparrow n} \models \zeta\sigma$, we have that

$$\text{for each } \mathcal{L}\text{-term } t, \Phi_P^{\uparrow n} \models \zeta_1\gamma_t\sigma.$$

By the inductive hypothesis there exists an m such that

$$\text{for each } \mathcal{L}\text{-term } t, \Phi_P^{\uparrow m} \models \zeta_1[\phi/\chi]\gamma_t\sigma.$$

Since the underlying universe of $\Phi_P^{\uparrow m}$ is the Herbrand universe on $\mathcal{L}$, this implies that

$$\Phi_P^{\uparrow m} \models (\forall\, w\, \zeta_1[\phi/\chi])\sigma.$$

3) Finally, the case $\zeta = \exists\, w\, \zeta_1(w)$, is treated as $\neg\forall\, w\ \neg\zeta_1(w)$. $\square$

Proof of Lemma 4.1.8

Let us first state a simple property of existentially quantified formulas.

Remark 4.5.1 Let $\mathcal{L}$ be any language, $\tilde{w}$ and $\tilde{z}$ be sets of variables, $\tilde{L}$ be a conjunction of literals, I a three valued $\mathcal{L}$-interpretation and θ any ground substitution. Suppose that $\tilde{w} \supseteq \tilde{z} \cap Var(\tilde{L})$. The following properties hold:

- If $\exists\, \tilde{z}\, \tilde{L}\theta$ is *true* in I then $\exists\, \tilde{w}\, \tilde{L}\theta$ is *true* in I.
- If $\exists\, \tilde{z}\, \tilde{L}\theta$ is not *false* in I then $\exists\, \tilde{w}\, \tilde{L}\theta$ is not *false* in I.

This is true in particular when $\tilde{z}$ is empty and $\exists\, \tilde{z}\, \tilde{L}\theta = \tilde{L}\theta$. □

Lemma 4.1.8 Notation as in Theorem 4.1.7. Let I, I' be two partial interpretations. If $I' \subseteq I$ but $\Phi_{P'}(I') \not\subseteq \Phi_P(I)$, then there exist a conjunction $\tilde{C}_j \in \{\tilde{C}_1, \ldots, \tilde{C}_n\}$ and a ground substitution θ such that:

- either $I' \models \exists\, \tilde{x}_j\, \tilde{D}_j\theta$ while $I \not\models \exists\, \tilde{x}_j\, \tilde{C}_j\theta$;
- or $I' \models \neg\exists\, \tilde{x}_j\, \tilde{D}_j\theta$ while $I \not\models \neg\exists\, \tilde{x}_j\, \tilde{C}_j\theta$.

Proof. Recall that $\Phi_{P'}(I') \not\subseteq \Phi_P(I)$ iff either $\Phi_{P'}(I')^+ \not\subseteq \Phi_P(I)^+$ or $\Phi_{P'}(I')^- \not\subseteq \Phi_P(I)^-$ (or both). We have to distinguish the two cases.

Case 1) Let us suppose that $\Phi_{P'}(I')^+ \not\subseteq \Phi_P(I)^+$ and let us take an atom $B \in \Phi_{P'}(I')^+ \backslash \Phi_P(I)^+$. There has to be a clause $c \in P' \backslash P$, a ground substitution θ' such that: $head(c)\theta' = B$ and $body(c)\theta'$ is *true* in I'.
$P' \backslash P = \{cl'_1, \ldots, cl'_p\}$, then there is an integer j such that: $c = cl'_j$ and $body(cl'_j)\theta' = (\tilde{D}_{j_1}, \ldots, \tilde{D}_{j_{r(j)}}, \tilde{E}_j)\theta'$. is *true* in I'.
Hence the conjunctions $\tilde{D}_{j_1}\theta', \ldots, \tilde{D}_{j_{r(j)}}\theta'$ are all *true* in I'. From Remark 4.5.1 it follows that the formulas:

$$\exists\, \tilde{x}_{j_1}\, \tilde{D}_{j_1}\theta', \ldots, \exists\, \tilde{x}_{j_{r(j)}}\, \tilde{D}_{j_{r(j)}}\theta' \text{ are } \mathit{true} \text{ in } I', \tag{4.12}$$

where the $\tilde{x}_i$ are sets of variables that satisfy the locality property wrt to $\tilde{C}_i$ and $\tilde{D}_i$.

We know that $B = head(cl'_j)\theta' = head(cl_j)\theta'$, but since $B \notin \Phi_P(I)^+$, by definition 2.1.6 we have that $(\exists\, \tilde{w} body(cl_j))\theta'$ is not *true* in I, where $\tilde{w} = Var(body(cl_j)) \backslash Var(head(cl_j))$, that is, $(\exists\, \tilde{w}\, \tilde{C}_{j_1}, \ldots, \tilde{C}_{j_{r(j)}}, \tilde{E}_j)\theta'$ is not *true* in I.
For each k, $\tilde{w} \supseteq \tilde{x}_{j_k} \cap Var(body(cl_j))$, now let $\tilde{y} = \tilde{w} \backslash \tilde{x}_{j_1} \cup \ldots \cup \tilde{x}_{j_{r(j)}}$ and θ be a ground extension of θ' whose domain contains $\tilde{y}$. Then from Remark 4.5.1 it follows that

$$(\exists\, \tilde{x}_{j_1}, \ldots, \tilde{x}_{j_{r(j)}}\ \ \tilde{C}_{j_1}, \ldots, \tilde{C}_{j_{r(j)}}, \tilde{E}_j)\theta \text{ is not } \mathit{true} \text{ in } I.$$

Since $\tilde{E}_j\theta$ is *true* in I' and $I' \subseteq I$, then $\tilde{E}_j\theta$ is *true* in I, by the locality property, the sets $\tilde{x}_{j_k}$ are pairwise disjoint, hence one of the formulas in $\exists\, \tilde{x}_{j_1}\, \tilde{C}_{j_1}\theta, \ldots, \exists\, \tilde{x}_{j_{r(j)}}\, \tilde{C}_{j_{r(j)}}\theta$ is not *true* in I.
Since (4.12) holds also for θ, the thesis follows.

Case 2) It is perfectly symmetrical to case 1) except for the fact that it is proven by contradiction. Let us suppose that $\Phi_{P'}(I')^- \not\subseteq \Phi_P(I)^-$, and let us take an atom $B \in \Phi_{P'}(I')^- \backslash \Phi_P(I)^-$. There has to be a clause $c \in P \backslash P'$, a ground substitution θ' such that $head(c)\theta' = B$ and $body(c)\theta'$ is not *false* in I.

$P \backslash P' = \{cl_1, \ldots, cl_p\}$, then there is an integer j such that: $c = cl_j$, and then the conjunction $(\tilde{C}_{j_1}, \ldots, \tilde{C}_{j_{r(j)}}, \tilde{E}_j)\theta'$ is not *false* in I.
Hence the conjunctions $\tilde{C}_{j_1}\theta', \ldots, \tilde{C}_{j_{r(j)}}\theta'$ are all not *false* in I. From Remark 4.5.1 it follows that:

$$\exists\, \tilde{x}_{j_1}\ \tilde{C}_{j_1}\theta', \ldots, \exists\, \tilde{x}_{j_{r(j)}}\ \tilde{C}_{j_{r(j)}}\theta' \text{ are not } \mathit{false} \text{ in } I. \tag{4.13}$$

We know that $B = head(cl_j)\theta' = head(cl'_j)\theta'$, but since $B \in \Phi_{P'}(I')^-$, by definition 2.1.6 we have that $(\exists\, \tilde{w} body(cl'_j))\theta'$ is *false* in I', with $\tilde{w} = Var(body(cl'_j)) \backslash Var(head(cl'_j))$, that is, $(\exists\, \tilde{w}\, \tilde{D}_{j_1}, \ldots, \tilde{D}_{j_{r(j)}}, \tilde{E}_j)\theta'$ is *false* in I'. For each k, $\tilde{w} \supseteq \tilde{x}_{j_k} \cap Var(body(cl_j))$, now let $\tilde{y} = \tilde{w} \backslash \tilde{x}_{j_1} \cup \ldots \cup \tilde{x}_{j_{r(j)}}$ and θ be a ground extension of θ' whose domain contains $\tilde{y}$. From Remark 4.5.1 it follows that

$$(\exists\, \tilde{x}_{j_1}, \ldots, \tilde{x}_{j_{r(j)}}\quad \tilde{D}_{j_1}, \ldots, \tilde{D}_{j_{r(j)}}, \tilde{E}_j)\theta \text{ is } \mathit{false} \text{ in } I'.$$

Since $\tilde{E}_j\theta$ is not *false* in I and $I' \subseteq I$, $\tilde{E}_j\theta$ is not *false* in I'. By the locality property, the sets $\tilde{x}_{j_k}$ are pairwise disjoint, then one of the formulas in $\exists\, \tilde{x}_{j_1}\ \tilde{D}_{j_1}\theta \cdots \exists\, \tilde{x}_{j_{r(j)}}\ \tilde{D}_{j_{r(j)}}\theta$ is *false* in I'.
Since (4.13) holds also for θ, the thesis follows. □

Proof of Lemma 4.2.3

Lemma 4.2.3 Let χ, ϕ be first order allowed formulas and P be a normal program. The following statements are equivalent:

(a) $\chi \ \preceq_{Comp_{\mathcal{L}}(P)\, \cup\, \mathrm{DCA}_{\mathcal{L}}}\ \phi$;
(b) $\mathit{lfp}(\Phi_P) \models \chi \Rightarrow \phi$.

Proof.
(a) implies (b).
By the definition of the operator $\Rightarrow$, (b) is equivalent to
for each tuple of $\mathcal{L}$-terms $\tilde{t}$, $\mathit{lfp}(\Phi_P) \models (\neg)\chi(\tilde{t}/\tilde{x})$ implies $\mathit{lfp}(\Phi_{P'}) \models (\neg)\phi(\tilde{t}/\tilde{x})$.
By Theorem 2.3.5 this is equivalent to
for each tuple of $\mathcal{L}$-terms $\tilde{t}$, $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models (\neg)\chi(\tilde{t}/\tilde{x})$ implies $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models (\neg)\phi(\tilde{t}/\tilde{x})$.
This is immediate by Definition 4.1.2.

(b) implies (a).
Let ζ be any allowed formula such that $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \zeta$, σ be any ground substitution; we have to prove that $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \zeta[\phi\sigma/\chi\sigma]$.
If ζ does not contain $\chi\sigma$ as a subformula then the result holds trivially, so let us suppose that ζ contains $\chi\sigma$ as a subformula. The proof proceeds by induction on the structure of ζ.
Base step: $\zeta \equiv \chi\sigma$. By Theorem 2.3.5, $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \chi\sigma$ implies that $\mathit{lfp}(\Phi_P) \models \chi\sigma$.
By (b) this implies that $\mathit{lfp}(\Phi_P) \models \phi\sigma$, and, by Theorem 2.3.5, that $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \phi\sigma$.
Since $\phi\sigma \equiv \zeta[\phi\sigma/\chi\sigma]$, this implies the thesis.

Induction step: we have to consider four cases:

1) $\zeta \equiv \triangle\ \zeta_1$, where $\triangle$ is any allowed unary connective. The result holds trivially, since by the inductive hypothesis, $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models (\neg)\zeta_1$ implies $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models (\neg)\zeta_1[\phi\sigma/\chi\sigma]$.

2) $\zeta \equiv \zeta_1 \diamond \zeta_2$, where $\diamond$ is any allowed binary connective. For $i \in \{1,2\}$, either ζ_i does not contain an instance of χ as a subformula, in which case the result holds trivially, or the inductive hypothesis applies to ζ_i.

3) $\zeta \equiv \forall\, w\ \zeta_1(w)$.
Suppose that $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \forall\, w\ \zeta_1(w)$.
This is equivalent to: for any $\mathcal{L}$-term t, $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \zeta_1(t)$.
For each $\mathcal{L}$-term t, let γ_t be the substitution (t/w), by the inductive hypothesis, we have that for any $\mathcal{L}$-term t, $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \zeta_1(t)[\phi\sigma\gamma_t/\chi\sigma\gamma_t]$.
Since $\mathrm{DCA}_{\mathcal{L}}$ forces the quantification to be over $\mathcal{L}$-terms, and $\mathrm{DCA}_{\mathcal{L}}$ is included in $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}$, this implies that $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \forall\, w\ \zeta_1(w)[\phi\sigma/\chi\sigma]$.
On the other hand, for the case when $Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}} \models \neg\forall\, w\ \zeta_1(w)$, a similar reasoning applies.

4) $\zeta \equiv \exists\, w\ \zeta_1(w)$
This falls into the previous case, since $\exists\, w\ \zeta_1(w) \equiv \neg\forall\, w\ \neg\zeta_1(w)$. □

4.6 Appendix B

Now we state two Lemmata which are the counterpart of Lemmata 4.3.3 and 4.3.5, for the case in which the closure axioms adopted are $\mathrm{WDCA}_{\mathcal{L}}$ rather than $\mathrm{DCA}_{\mathcal{L}}$.

Lemma 4.6.1 Let $cl = A \leftarrow \tilde{E}, \tilde{G}.$ be a clause in the normal program P, $\tilde{x}$ be a set of variables not occurring in $(A, \tilde{E})$ and $\tilde{H}$ be another conjunction of literals. Then

(a) If for each j there exists a k such that, for each θ, $\Phi_P^{\uparrow j} \models \exists\, \tilde{x}\ \tilde{G}\theta$ implies $\Phi_P^{\uparrow k} \models (\exists\, \tilde{x}\ \tilde{G}, \tilde{H})\theta$, then $\exists\, \tilde{x}\ \tilde{G} \preceq_{Comp_{\mathcal{L}}(P)} \exists\, \tilde{x}\ \tilde{G}, \tilde{H}$.

(b) If for each j there exists a k such that, for each θ, $\Phi_P^{\uparrow j} \models \neg(\exists\, \tilde{x}\ \tilde{G}, \tilde{H})\theta$ implies $\Phi_P^{\uparrow k} \models \neg\exists\, \tilde{x}\ \tilde{G}\theta$, then $\exists\, \tilde{x}\ \tilde{G}, \tilde{H} \preceq_{Comp_{\mathcal{L}}(P)} \exists\, \tilde{x}\ \tilde{G}$.

(c) If m is an integer such that, for each n and θ, $\Phi_P^{\uparrow n} \models_{\mathcal{L}} \exists\, \tilde{x}\, \tilde{G}\theta$ implies $\Phi_P^{\uparrow n+m} \models_{\mathcal{L}} (\exists\, \tilde{x}\ \tilde{G}, \tilde{H})\theta$ then
- $\exists\, \tilde{x}\ \tilde{G} \preceq_{Comp_{\mathcal{L}}(P)} \exists\, \tilde{x}\ \tilde{G}, \tilde{H}$;
- the delay of $\exists\, \tilde{x}\ \tilde{G}, \tilde{H}$ wrt $\exists\, \tilde{x}\ \tilde{G}$ in $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ is smaller or equal to m.
If m is the least of such integers, then the delay of $\exists\, \tilde{x}\ \tilde{G}, \tilde{H}$ wrt $\exists\, \tilde{x}\ \tilde{G}$ in $Comp_{\mathcal{L}}(P) \cup \mathrm{WDCA}_{\mathcal{L}}$ is exactly m.

Proof. It is a straightforward application of Theorem 2.3.3 together with the fact that, if $\tilde{G}\theta$ is *false* in some interpretation I, then also $(\tilde{G}, \tilde{H})\theta$ is *false* in I. □

Lemma 4.6.2 Let $cl = A \leftarrow \tilde{E}, \tilde{G}, \tilde{H}.$ be a clause in P and $\tilde{x}$ be a set of variables not occurring in A, $\tilde{E}$. The following property holds:

- If m is an integer such that, for each integer n and substitution θ, $\exists\,\tilde{x}\,(\tilde{G},\tilde{H})\theta$ *false* in $\Phi_P^{\uparrow n}$ implies $\exists\,\tilde{x}\,\tilde{G}\theta$ *false* in $\Phi_P^{\uparrow n+m}$, then
  - $\exists\,\tilde{x}\,\tilde{G},\tilde{H}\ \preceq_{Comp_{\mathcal{L}}(P)}\ \exists\,\tilde{x}\,\tilde{G}$,
  - the delay of $\exists\,\tilde{x}\,\tilde{G}$ wrt $\exists\,\tilde{x}\,\tilde{G},\tilde{H}$ in $\Phi_P^{\uparrow\omega}$ is less or equal to m.

  If m is the least of such integers, then the delay of $\exists\,\tilde{x}\,\tilde{G},\tilde{H}$ wrt $\exists\,\tilde{x}\,\tilde{G}$ in $\Phi_P^{\uparrow\omega}$ is exactly m.

Proof. It is a straightforward application of the fact that if $(\tilde{G},\tilde{H})\theta$ is *true* in some interpretation I, then also $\tilde{G}\theta$ is *true* in I. □

4.7 Appendix C (Safeness of the Unfolding Operation)

First we need the following technical Lemma.

Lemma 4.7.1 Let P' be the program obtained by unfolding an atom in a clause of program P. Then for each integer i and limit ordinal β,

(a) $\Phi_P^{\uparrow i} \subseteq \Phi_{P'}^{\uparrow i}$ and $\Phi_{P'}^{\uparrow i} \subseteq \Phi_P^{\uparrow 2i}$;
(b) $\Phi_P^{\uparrow i}(\Phi_P^{\uparrow\beta}) \subseteq \Phi_{P'}^{\uparrow i}(\Phi_{P'}^{\uparrow\beta})$ and $\Phi_{P'}^{\uparrow i}(\Phi_{P'}^{\uparrow\beta}) \subseteq \Phi_P^{\uparrow 2i}(\Phi_P^{\uparrow\beta})$.

Proof. Here we adopt the same notation of definition 3.2.3, so $cl:\ A \leftarrow H,\tilde{K}.$ is the clause of P to which we apply the unfold operation, $\{H_1 \leftarrow \tilde{B}_1., \ldots, H_n \leftarrow \tilde{B}_n.\}$ are the clauses of P whose heads unify with H, $\{cl'_1,\ldots,cl'_n\}$ are the resulting clauses, where, for each i, $cl'_i : (A \leftarrow \tilde{B}_i,\tilde{K})\theta_i.$ and $\theta_i = mgu(H,H_i)$. We also suppose that all this clauses are disjoint.

The next Claim is crucial

Claim 4.1 Suppose that α is an ordinal such that, for each ground τ,

(i) $\Phi_P^{\uparrow\alpha} = \Phi_{P'}^{\uparrow\alpha}$;
(ii) if $H\tau \in \Phi_P^{\uparrow\alpha^+}$ then there exist a substitution ϕ and an integer i such that $H\tau = H_i\theta_i\phi$ and $\tilde{B}_i\theta_i\phi$ is *true* in $\Phi_P^{\uparrow\alpha}$;
(iii) if $H\tau \in \Phi_P^{\uparrow\alpha^-}$ then for each substitution ϕ and integer i if $H\tau = H_i\theta_i\phi$ then $\tilde{B}_i\theta_i\phi$ is *false* in $\Phi_P^{\uparrow\alpha}$.

Then, for each integer j,

- $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha}) \subseteq \Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$;
- $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha}) \subseteq \Phi_P^{\uparrow 2j}(\Phi_P^{\uparrow\alpha})$.

Proof. First we prove the first statement, and we show by induction that if a ground atom R is *true* or *false* in $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha})$ then it is also so in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$.
The base case $j=0$ is trivial, since $\Phi_P^{\uparrow 0}(\Phi_P^{\uparrow\alpha}) = \Phi_P^{\uparrow\alpha}$, and from (i) we have the thesis.
Induction step, $j>0$; we have to distinguish two cases:

1) Suppose R is *true* in $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha})$; then there exists a clause $d \in P$ and a substitution θ such that $R = head(d)\theta$ and $body(d)\theta$ is *true* in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$.
If $d \neq cl$ then d belongs both to P and P', by the inductive hypothesis $body(d)\theta$ is *true* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$, and the result follows.

Otherwise, $d = cl$, $R = A\theta$ and $(H, \tilde{K})\theta$ is *true* in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$. So $H\theta$ is *true* in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$.
If $j > 1$ this implies that for some integer i and substitution ϕ, $H\theta = H\theta_i\phi = H_i\theta_i\phi$ and $\tilde{B}_i\theta_i\phi$ is *true* in $\Phi_P^{\uparrow j-2}(\Phi_P^{\uparrow\alpha})$.
On the other hand, if $j = 1$ the fact that $H\theta$ is *true* in $\Phi_P^{\uparrow\alpha}$ implies, by (ii), that for some integer i and some substitution ϕ, $\tilde{B}_i\theta_i\phi$ is *true* in $\Phi_P^{\uparrow\alpha}$.
In any case, $(\tilde{B}_i, \tilde{K})\theta_i\phi$ is *true* in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$ and, by inductive hypothesis, in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$.
Then $body(cl'_i)\phi$ is *true* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$, it follows that, $head(cl'_i)\phi$ is *true* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$.
We can assume that $\theta|_{Var(d)} = \theta_i\phi|_{Var(d)}$, and hence that $A\theta = A\theta_i\phi$.
As $R = A\theta = A\theta_i\phi = head(cl'_i)\phi$, the result follows.

2) Suppose that R is *false* in $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha})$, we prove this part by contradiction.We assume that R is not *false* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$; then there exists a clause $d' \in P'$ and a substitution θ such that $R = head(d')\theta$ and $body(d')\theta$ is not *false* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$.
If $d' \notin \{cl'_1, \ldots, cl'_n\}$, then d' belongs both to P' and P, by the inductive hypothesis $body(d')\theta$ is not *false* in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$, and $R = head(d')\theta$ is not *false* in $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha})$, which is a contradiction.
Otherwise, for some integer i and substitution ϕ, $d' = cl'_i$, $R = head(cl'_i)\phi = A\theta_i\phi$, and $body(cl'_i)\phi$ is not *false* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$. Recall that $body(cl'_i)\phi = (\tilde{B}_i, \tilde{K})\theta_i\phi$.
If $j > 1$, the fact that $\tilde{B}_i\theta_i\phi$ is not *false* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$ implies that $\tilde{B}_i\theta_i\phi$ is not *false* in $\Phi_{P'}^{\uparrow j-2}(\Phi_{P'}^{\uparrow\alpha})$, and since $H_i \leftarrow \tilde{B}_i$. is a clause of P', $H\theta_i\phi = H_i\theta_i\phi$ is not *false* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$.
On the other hand, if $j = 1$, the fact that $\tilde{B}_i\theta_i\phi$ is not *false* in $\Phi_{P'}^{\uparrow\alpha}$ implies by (ii) that $H\theta_i\phi$ is not *false* in $\Phi_{P'}^{\uparrow\alpha}$.
In any case $(H, \tilde{K})\theta_i\phi$ is not *false* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$, and by the inductive hypothesis, in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$. Since $H, \tilde{K} = body(cl)$ it follows that $R = A\theta_i\phi = head(cl)\theta_i\phi$ is not *false* in $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha})$, which gives a contradiction.

Now we prove the second statement: we show by induction that if a ground atom R is *true* or *false* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$ then it is also so in $\Phi_P^{\uparrow 2j}(\Phi_P^{\uparrow\alpha})$.
As above, the base case $j = 0$ is trivial.
Induction step $j > 0$: we have to distinguish two cases.

1) Suppose that R is *true* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$, then there exists a clause $d' \in P'$ and a substitution θ such that $R = head(d')\theta$ and $body(d')\theta$ is *true* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$.
If $d' \notin \{cl'_1, \ldots, cl'_n\}$ then d' belongs both to P' and P, by the inductive hypothesis $body(d')\theta$ is *true* in $\Phi_P^{\uparrow j-1}(\Phi_P^{\uparrow\alpha})$, $R = head(d')\theta$ is *true* in $\Phi_P^{\uparrow j}(\Phi_P^{\uparrow\alpha})$ and the result follows.
Otherwise for some integer i and substitution ϕ, $d' = cl'_i$, $R = head(cl'_i)\phi = A\theta_i\phi$, and $body(cl'_i)\phi$ is *true* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$.
Recall that $body(cl'_i)\phi = (\tilde{B}_i, \tilde{K})\theta_i\phi$; by inductive hypothesis, $(\tilde{B}_i, \tilde{K})\theta_i\phi$ is also *true* in $\Phi_P^{\uparrow 2j-2}(\Phi_P^{\uparrow\alpha})$.
Since $\tilde{B}_i\theta_i\phi$ is *true* in $\Phi_P^{\uparrow 2j-2}(\Phi_P^{\uparrow\alpha})$ and $H_i \leftarrow \tilde{B}_i$. is a clause of P, $H_i\theta_i\phi$ is *true* in $\Phi_P^{\uparrow 2j-1}(\Phi_P^{\uparrow\alpha})$. But $H_i\theta_i\phi = H\theta_i\phi$, so $(H, \tilde{K})\theta_i\phi = body(cl)\theta_i\phi$ is *true* in $\Phi_P^{\uparrow 2j-1}(\Phi_P^{\uparrow\alpha})$, hence $R = A\theta_i\phi = head(cl)\theta_i\phi$ is *true* in $\Phi_P^{\uparrow 2j}(\Phi_P^{\uparrow\alpha})$.

2) Let R be *false* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$; we prove this part by contradiction, so we assume that R is not *false* in $\Phi_P^{\uparrow 2j}(\Phi_P^{\uparrow\alpha})$. Then there exists a clause $d \in P$ and a substitution

θ such that $R = head(d)\theta$ and $body(d)\theta$ is not *false* in $\Phi_P^{\uparrow 2j-1}(\Phi_P^{\uparrow\alpha})$.
If $d \neq cl$ then d belongs both to P and P', by the monotonicity of the Kleene sequence, $body(d)\theta$ is not *false* in $\Phi_P^{\uparrow 2j-2}(\Phi_P^{\uparrow\alpha})$ either, hence, by the inductive hypothesis $body(d)\theta$ is not *false* in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$. It follows that $head(d)\theta = R$ is not *false* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$ which gives a contradiction.
Otherwise, $d = cl$, $R = A\theta$ and $(H, \tilde{K})\theta$ is not *false* in $\Phi_P^{\uparrow 2j-1}(\Phi_P^{\uparrow\alpha})$. So $H\theta$ is not *false* in $\Phi_P^{\uparrow 2j-1}(\Phi_P^{\uparrow\alpha})$. This implies that for some integer i and substitution ϕ, $H\theta = H\theta_i\phi = H_i\theta_i\phi$ and $\tilde{B}_i\theta_i\phi$ is not *false* in $\Phi_P^{\uparrow 2j-2}(\Phi_P^{\uparrow\alpha})$.
Hence $(\tilde{B}_i, \tilde{K})\theta_i\phi$ is not *false* in $\Phi_P^{\uparrow 2j-2}(\Phi_P^{\uparrow\alpha})$, and by the inductive hypothesis, in $\Phi_{P'}^{\uparrow j-1}(\Phi_{P'}^{\uparrow\alpha})$. Since $\tilde{B}_i\theta_i\phi = body(cl'_i)\phi$, this implies that $head(cl'_i)\phi = A\theta_i\phi = R$ is not *false* in $\Phi_{P'}^{\uparrow j}(\Phi_{P'}^{\uparrow\alpha})$ which is a contradiction. □

Now, in order to prove (a) we observe that $\alpha = 0$ is an ordinal that trivially satisfies the hypothesis of Claim 4.1.

In order to prove (b) we have to show that Claim 4.1 also applies when α is any limit ordinal.
First consider the case $\alpha = \omega$. From (a) it follows that $\Phi_P^{\uparrow\omega} = \Phi_{P'}^{\uparrow\omega}$, moreover, if $H\tau$ is *true* (resp. *false*) in $\Phi_P^{\uparrow\omega}$, then, it is also *true* in some $\Phi_P^{\uparrow m}$, $(m < \omega)$. By applying the definition of Fitting's operator we have that condition (ii) (resp. (iii)) hold for $\alpha = \omega$. So $\alpha = \omega$ satisfies the requirements of Claim 4.1.
It follows that, for each i, $\Phi_P^{\uparrow\omega+i} \subseteq \Phi_{P'}^{\uparrow\omega+i}$ and that $\Phi_{P'}^{\uparrow\omega+i} \subseteq \Phi_P^{\uparrow\omega+2i}$. By the same reasoning it turns out that the ordinal 2ω, and iterating, all the other limit ordinals, satisfy the requirements of Claim 4.1. □

This brings us to the desired conclusions.

Corollary 4.7.2 (safeness of the unfolding operation) Let P' be the result of unfolding an atom of a clause in P. Then P is equivalent to P' wrt all three the semantics considered in this paper.

Proof. By Lemmata 4.7.1, 4.2.2 and Theorems 2.3.3 and 2.2.3. □

Chapter 5

Preservation of Fitting's Semantics in Unfold/Fold Transformations of Normal Programs

The unfold/fold transformation system defined by Tamaki and Sato was meant for definite programs. It transforms a program into an equivalent one in the sense of both the least Herbrand model semantics and the Computed Answer Substitution semantics. Seki extended the method to normal programs and specialized it in order to preserve also the finite failure set. The resulting system is correct wrt nearly all the declarative semantics for normal programs. An exception is Fitting's model semantics. In this chapter we consider a slight variation of Seki's method and we study its correctness wrt Fitting's semantics. We define an applicability condition for the fold operation and we show that it ensures the preservation of the considered semantics through the transformation.

5.1 Introduction

The unfold/fold transformation rules were introduced by Burstall and Darlington [25] for transforming clear, simple functional programs into equivalent, more efficient ones. The rules were early adapted to the field of logic programs both for program synthesis [30, 50] and for program specialization and optimization [1, 60]. Soon later, Tamaki and Sato [96] proposed an elegant framework for the transformation of logic programs based on unfold/fold rules.

The major requirement of a transformation system is its correctness: it should transform a program into an equivalent one. Tamaki and Sato's system was originally designed for definite programs and in this context a natural equivalence on programs is the one induced by the least Herbrand model semantics. In [96] it was shown that the system preserves such a semantics. Afterward, the system was proven to be correct wrt many other semantics: the computed answer substitution semantics [58], the Perfect model semantics [91], the Well-Founded semantics [92] and the Stable model semantics [90, 12].

In [91], Seki modified the method by restricting its applicability conditions. The system so defined enjoys all the semantic properties of Tamaki-Sato's, moreover, it preserves the finite failure set of the original program [89] and it is correct wrt Kunen's semantics [88].

However, neither Tamaki-Sato's, nor Seki's system preserve the Fitting model semantics.

In this chapter we consider a transformation schema which is similar yet slightly more restrictive to the one introduced by Seki [91] for normal programs and reported in definition 3.2.8. We study the effect of the transformation on the Fitting's semantics [41] and we individuate a sufficient condition for its preservation.

The difference between the method we propose and the one of Seki consists in the fact that here the operations have to be performed in a precise order. We believe that this order corresponds to the "natural" order in which the operations are usually carried out within a transformation sequence, and therefore that the restriction we impose is actually rather mild.

The structure of this chapter is the following. In Section 5.2 the transformation schema is defined and exemplified, and the applicability conditions for the fold operation are presented and discussed. Finally, in Section 5.3, we prove the correctness of the unfold/fold transformation wrt Fitting's semantics. For the notation and the preliminaries on Fitting's semantics we refer to section 2.3.2.

5.2 A four step transformation schema

In this section we introduce the unfold/fold transformation *schema*. All definitions are given modulo reordering of the bodies of the clauses and standardization apart is always assumed.

Let P be a normal program. A *four step transformation schema* starting in the program P consists of the following steps:

Step 1. Introduction of new definitions

We add to the program P the set of clauses $D_{\text{def}} = \{c_i : \ H_i \leftarrow \tilde{B}_i\}$, where the predicate symbol of each H_i is *new*, that is, it does not occur in P. On the other hand, we require that the predicate symbols found in each $\tilde{B}_i$ are defined in P, and therefore are not *new*. The result of this operation is then

- $P_1 = P \cup D_{\text{def}}$ □

Example 5.2.1 (min-max, part 1) Let P be the following program

$$
\begin{array}{llcl}
P = \{ & min([X], X). & & \\
& min([X|Xs], Y) & \leftarrow & min(Xs, Z), inf(X, Z, Y). \\
& max([X], X). & & \\
& max([X|Xs], Y) & \leftarrow & max(Xs, Z), sup(X, Z, Y). \\
& inf(X, Y, X) & \leftarrow & X \leq Y. \\
& inf(X, Y, Y) & \leftarrow & \neg(X \leq Y). \\
& sup(X, Y, Y) & \leftarrow & X \leq Y. \\
& sup(X, Y, X) & \leftarrow & \neg(X \leq Y). \\
c_1: & med(Xs, Med) & \leftarrow & min(Xs, Min), \\
& & & max(Xs, Max), \\
& & & Med\ is\ (Min + Max)/2. \qquad \}
\end{array}
$$

here $med(Xs, Med)$ reports in Med the average between the minimum and the maximum of the values in the list Xs.

We may notice that the definition of $med(Xs, Med)$ traverses the list Xs twice. This is obviously a source of inefficiency. In order to fix this problem via an unfold/fold transformation, we first have to introduce a new predicate *minmax*. Let us then add to program P the following new definition:

$D_{\text{def}} = \{c_2: \ minmax(Xs, Min, Max) \leftarrow min(Xs, Min), max(Xs, Max). \}$ □

Step 2. Unfolding in D_{def}

We transform D_{def} into D_{unf} by unfolding some of its clauses. The clauses of P are therefore used as unfolding clauses. This process can be iterated several times and usually ends when all the clauses that we want to fold have been obtained; the result of this operation is

- $P_2 = P \cup D_{\text{unf}}$ □

Example 5.2.1 (min-max, part 2). We can now unfold the atom $min(Xs, Min)$ in the body of c_2, the result is

$$
\begin{array}{llcl}
c_3: & minmax([X], X, Max) & \leftarrow & max([X], Max). \\
c_4: & minmax([X|Xs], Min, Max) & \leftarrow & min(Xs, Y), \\
& & & inf(X, Y, Min), \\
& & & max([X|Xs], Max).
\end{array}
$$

In the bodies of both clauses we can then unfold predicate *max*. Each clause generates two clauses.

$$\begin{array}{lllll}
c_5: & minmax([X], X, X). & & \\
c_6: & minmax([X], X, Max) & \leftarrow & max([\,], Z), sup(Z, X, Max). \\
c_7: & minmax([X], Min, X) & \leftarrow & min([\,], Y), inf(X, Y, Min). \\
c_8: & minmax([X|Xs], Min, Max) & \leftarrow & min(Xs, Y), \\
 & & & inf(X, Y, Min), \\
 & & & max(Xs, Z), \\
 & & & sup(X, Z, Max).
\end{array}$$

Clauses c_6 and c_7 can then be eliminated by unfolding respectively the atoms $max([\,], Z)$ and $min([\,], Y)$. D_{unf} consists then of the following clauses.

$$\begin{array}{lllll}
c_5: & minmax([X], X, X). & & \\
c_8: & minmax([X|Xs], Min, Max) & \leftarrow & min(Xs, Y), \\
 & & & inf(X, Y, Min), \\
 & & & max(Xs, Z), \\
 & & & sup(X, Z, Max).
\end{array}$$

Still, *minmax* traverses the list Xs twice; but now we can apply a *recursive folding* operation. □

Step 3. Recursive folding

Let $c_i: H_i \leftarrow \tilde{B}_i$ be one of the clauses of D_{def}, which was introduced in *Step 1*, and $cl: A \leftarrow \tilde{B}', \tilde{S}.$ be (a renaming of) a clause in D_{unf}. If there exists a substitution θ, $Dom(\theta) = Var(c_i)$ such that

(a) $\tilde{B}' = \tilde{B}_i\theta$;
(b) θ does not bind the local variables of c_i, that is for any $x, y \in Var(\tilde{B}_i)\backslash Var(\tilde{H}_i)$ the following three conditions hold
  - $x\theta$ is a variable;
  - $x\theta$ does not appear in A, $\tilde{S}$, $H_i\theta$;
  - if $x \neq y$ then $x\theta \neq y\theta$;

(c) c_i is the only clause of D_{def} whose head unifies with $H_i\theta$;
(d) all the literals of $\tilde{B}'$ are the result of a previous unfolding.

then we can fold $H_i\theta$ in cl, obtaining $cl': A \leftarrow H_i\theta, \tilde{S}$. This operation can be performed on several conjunctions simultaneously, even on the same clause. The result is that D_{unf} is transformed into D_{fold} and hence

- $P_3 = P \cup D_{\text{fold}}$ □

Example 5.2.1 (min-max, part 3). We can now fold $min(Xs, Y), max(Xs, Z)$ in the body of c_8. The resulting program D_{fold} consists of the following clauses

$$\begin{array}{lllll}
c_5: & minmax([X], X, X). & & \\
c_9: & minmax([X|Xs], Min, Max) & \leftarrow & minmax(Xs, Y, Z), \\
 & & & inf(X, Y, Min), \\
 & & & sup(X, Z, Max).
\end{array}$$

$minmax(Xs, Min, Max)$ has now a recursive definition and needs to traverse the list Xs only once. In order to let the definition of med enjoy of this improvement, we need to *propagate* predicate $minmax$ inside its body. □

Step 4. Propagation folding

Technically, the difference between this step and the previous one is that now the folded clause comes form the original program P. This allows us to drop condition (d) of the folding operation.

Let $c_i: \ H_i \leftarrow \tilde{B}_i$ be one of the clauses of D_{def}, which was introduced in *Step 1*, and $cl: \ A \leftarrow \tilde{B}', \tilde{S}$. be (a renaming of) a clause in the original program P. If there exists a substitution θ, $Dom(\theta) = Var(c_i)$ such that the conditions (a), (b) and (c) defined above are satisfied, then we can fold $H_i\theta$ in cl, obtaining $cl': \ A \leftarrow H_i\theta, \tilde{S}$. Also this operation can be performed on several conjunctions simultaneously, even on the same clause. The result is that P is transformed into P_{fold} and therefore

- $P_4 = P_{\text{fold}} \cup D_{\text{fold}}$ □

Example 5.2.1 (min-max, part 4). We can now fold $min(Xs, Y), max(Xs, Z)$ in the body of c_1, in the original program P. The resulting program is

$$P_{\text{fold}} = P \backslash \{c_1\} \cup \{c_{10} : med(Xs) \quad \leftarrow \quad \begin{array}{l} minmax(Xs, Min, Max), \\ Med\ is\ (Min + Max)/2. \quad \} \end{array}$$

And then the final program is $P_4 = P_{\text{fold}} \cup D_{\text{fold}} =$

$$= \{ \quad \begin{array}{lllll} c_5: & minmax([X], X, X). & & \\ c_9: & minmax([X|Xs], Min, Max) & \leftarrow & minmax(Xs, Y, Z), \\ & & & inf(X, Y, Min), \\ & & & sup(X, Z, Max). \\ c_{10}: & med(Xs) & \leftarrow & minmax(Xs, Min, Max), \\ & & & Med\ is\ (Min + Max)/2. \end{array}$$

$+$ definitions for predicates min, max, inf and $sup.\}$

Notice also that predicates min and max are no longer used by the program. □

Semantic considerations

The *schema* (that is, the method we propose) is similar but more restrictive than the *transformation sequence* with *modified* folding[1] proposed by Seki [91]. The (only) limitation consists in the fact that the schema requires the operations to be performed in fixed order: for instance it does not allow a *propagation folding* to take place before a *recursive folding*. We believe that in practice this is not a bothering restriction, as it corresponds to the "natural" procedure that is followed in the process of transforming

[1] here we are adopting Seki's notation, and we call *modified* folding the one presented in [89, 91], which preserves the finite failure set, as opposed to the one introduced by Tamaki and Sato in [96], which does not.

a program. In fact, in all the papers we cite, all the examples that can be reduced to a transformation sequence as in [91], can also be reduced to the given transformation schema.

Since the *schema* can be seen as a particular case of the transformation *sequence*, it enjoys all its properties, among them, it preserves the following semantics of the initial program: the success set [96], the computed answer substitution set [58], the finite failure set [91], the Perfect model semantics for stratified programs [91], the Well-Founded semantics [92], the Stable model semantics [90, 12].

However, as it is, the schema suffers of the same problems of the sequence, i.e., Fitting's Models is not preserved. This is shown by the following example.

Example 5.2.2 Let $P_1 = P \cup D_{\text{def}}$, where P and D_{def} are the following programs

$$\begin{array}{lllll} D_{\text{def}} & = \{ & p & \leftarrow & q(X). \quad \} \\ P & = \{ & q(s(X)) & \leftarrow & q(X), t(0). \\ & & t(0). & & \quad \} \end{array}$$

As we fix a language $\mathcal{L}$ that contains the constant 0 and the function $s/1$, we have that $\exists X\, q(X)$ is *false* in $Fit(P_1)$, consequently, p is also *false* in $Fit(P_1)$. Now let us unfold $q(X)$ in the body of the clause in D_{def}; the resulting program is the following. $P_2 = P \cup D_{\text{unf}}$, where

$$\begin{array}{lllll} D_{\text{unf}} & = \{ & p & \leftarrow & q(Y), t(0). \quad \} \\ P & = \{ & q(s(X)) & \leftarrow & q(X), t(0). \\ & & t(0). & & \quad \} \end{array}$$

We can now fold $q(Y)$ in the body of the clause of D_{unf}, the resulting program is $P_3 = P \cup D_{\text{fold}}$, where

$$\begin{array}{lllll} D_{\text{fold}} & = \{ & p & \leftarrow & p, t(0). \quad \} \\ P & = \{ & q(s(X)) & \leftarrow & q(X), t(0). \\ & & t(0). & & \quad \} \end{array}$$

Now we have that p is *undefined* in the Fitting model of P_3. □

So, in order for the transformation to preserve Fitting's model of the original program, we need some further applicability conditions. Therefore the following.

Theorem 5.2.3 (Correctness) Let $P_1, \ldots, P_4$ be a sequence of programs obtained applying the transformation *schema* to program P. Let also $D_{\text{def}} = \{H_i \leftarrow \tilde{B}_i\}$ be the set of clauses introduced in *Step 1*, and, for each i, $\tilde{w}_i$ be the set of local variables of c_i: $\tilde{w}_i = Var(\tilde{B}_i) \backslash Var(H_i)$. If each c_i in D_{def} satisfies the following condition:

A each time that $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is *false* in some $\Phi_{P_1}^{\uparrow\beta}$, then there exists a non-limit ordinal $\alpha \leq \beta$ such that $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow\alpha}$

Then $Fit(P_1) = Fit(P_2) = Fit(P_3) = Fit(P_4)$.

Proof. The proof is given in the subsequent Section 5.3. □

On condition A

Condition **A** is in general undecidable, it is therefore important to provide some other decidable sufficient conditions. For this, in the rest of this Section, we adopt the following notation:

- $D_{\rm def} = \{c_i : \ H_i \leftarrow \tilde{B}_i\}$ is the set of clauses introduced in *Step 1*,

and, for each i,

- $\tilde{w}_i = Var(\tilde{B}_i) \backslash Var(H_i)$ is the set of local variables of c_i.

First, it is easy to check that if c_i has no local variables, then it satisfies **A**.

Proposition 5.2.4 If $\tilde{w}_i = \emptyset$ then c_i satisfies **A**.

Proof. It follows at once from the definition of Fitting's operator. □

This condition, though simple, is met by most of the examples found in the literature; if we are allowed an informal "statistics", of all the papers cited in our bibliography, seven contain practical examples in clausal form which can be assimilated to our method ([21, 58, 78, 89, 91, 92, 96]), and of them, only two contain examples where the "introduced" clause contains local variables ([58, 78]). Our Example 5.2.1 satisfies the condition as well.

Nevertheless Proposition 5.2.4 can easily be improved. First let us consider the following Example[2].

Example 5.2.5 Let $P_1 = P \cup D_{\rm def}$, where P and $D_{\rm def}$ are the following programs

$$\begin{array}{llllll} D_{\rm def} & = \{ \ c_0 : & br(X,Y) & \leftarrow & reach(X,Z), reach(Y,Z). & \} \\ P & = \{ & reach(X,Y) & \leftarrow & arc(X,Y). & \\ & & reach(X,Y) & \leftarrow & arc(X,Z), reach(Z,Y). & \} \cup DB \end{array}$$

Where DB is any set of ground unit clauses defining predicate arc. $reach(X,Y)$ holds iff there exists a path starting from node X and ending in node Y, while $br(X,Y)$ holds iff there exists a node Z which is reachable both from node X and node Y. □

In this Example the definition of predicate br can be specialized and made recursive via an unfold/fold transformation. Despite the fact that clause c_0 contains the local variable Z, it is easy to see that **A** is satisfied. This is due to the fact that P is actually a DATALOG (function-free) program.

We now show that if (a part of) the original program P is function-free (or recursion-free) then **A** is always satisfied.

Let us first introduce the following notation. Let p, q be predicates, we say that p *refers to* q in program P if there is a clause of P with p in its head and q in its body. The *depends on* relation is the reflexive and transitive closure of *refers to.* Let $\tilde{L}$ be a conjunction of literals, by $P|_{\tilde{L}}$ we denote the set of clauses of P that define the predicates which the predicates in $\tilde{L}$ depend on. We say that a program is *recursion-free* if there is no chain $p_1, \ldots, p_k$ of predicate symbols such that p_i refers

[2] The example is actually a modification of Example 2.1.1 in [89]

to p_{i+1} and $p_k = p_1$. With an abuse of notation, we also call a program *function-free* if the only terms occurring in it are either ground or variables.

We can now state the following.

Proposition 5.2.6 For each index i, and each $w \in \tilde{w}_i$, let us denote by $\tilde{L}_w$ the subset of $\tilde{B}_i$ formed by those literals where w occurs. If for every $\tilde{L}_w$, one of the following two conditions holds:

(a) $P_1|_{\tilde{L}_w}$ is recursion-free, or
(b) $P_1|_{\tilde{L}_w}$ is function-free;

then each c_i satisfies **A**.

Proof. First we need the following Observation.

Observation 5.2.7 Let Q be a function-free or a recursion-free program, then for some integer k, $Fit(Q) = \Phi_Q^{\uparrow k}$

Proof. Straightforward □

Now fix an index i, and let $\tilde{w}_i = w_1, \ldots, w_m$, and let $\tilde{M}$ be the subset of $\tilde{B}_i$ consisting of those literals that do not contain any of the variables in $\tilde{w}_i$. It is immediate that, for any ordinal α, and for any substitution θ

$$\Phi_{P_1}^{\uparrow\alpha} \models \exists \tilde{w}_i\, \tilde{B}_i\theta \;\text{ iff }\; \Phi_{P_1}^{\uparrow\alpha} \models \exists w_1\, \tilde{L}_{w_1}\theta \wedge \ldots \wedge \exists w_m\, \tilde{L}_{w_m}\theta \wedge \tilde{M}\theta \tag{5.1}$$

Now suppose that, for some ordinal α, and substitution θ, $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow\alpha}$. By (5.1), either *(i)* $\tilde{M}\theta$ is *false* in $\Phi_{P_1}^{\uparrow\alpha}$, or *(ii)* there exists an i such that $\exists w_i\, \tilde{L}_{w_i}\theta$ is *false* in $\Phi_{P_1}^{\uparrow\alpha}$; we treat the two cases separately.

(i), $\tilde{M}\theta$ is *false* in $\Phi_{P_1}^{\uparrow\alpha}$, then, by the definition of Φ_{P_1}, there exists a non-limit ordinal $\beta \leq \alpha$ such that $\tilde{M}\theta$ is *false* in $\Phi_{P_1}^{\uparrow\beta}$, and, by (5.1), $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow\beta}$.

(ii), $\exists w_i\, \tilde{L}_{w_i}\theta$ is *false* in $\Phi_{P_1}^{\uparrow\alpha}$, since $P_1|_{\tilde{L}_{w_i}}$ is function or recursion-free, by Observation 5.2.7 there exists an integer k such that $\exists w_i\, \tilde{L}_{w_i}\theta$ is *false* in $\Phi_{P_1}^{\uparrow k}$; again, by (5.1), $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow k}$.

So, in any case, there exists a non-limit ordinal $\beta \leq \alpha$ such that $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow\beta}$. Since this holds for any index i, the thesis follows. □

Checking A "a posteriori"

We now show that condition **A** holds in P_0 iff it holds in any program of the unfold part of the transformation sequence. This gives us the opportunity of providing further sufficient conditions.

First let us restate **A** as follows:

A': For each substitution θ and non-limit ordinal β, if $H_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow\beta+1}$, then $H_i\theta$ is *false* in $\Phi_{P_1}^{\uparrow\beta}$ as well.

Now, let P_1' be a program which is obtained from P_1 by applying some unfolding transformation. It is easy to see[3] that H_i satisfies **A'** in P_1 iff H_i satisfies **A'** in P_1'. So the advantage of **A'** over **A** is that it can be checked *a posteriori* at any time during the unfolding part of the transformation. So Proposition 5.2.6 can be restated as follows.

Proposition 5.2.8 Let P_1' be a program obtained from P_1 by (repeatedly) applying the unfolding operation. Let D'_{def} be the subset of P' corresponding to D_{def} in P. If for each clause c of D'_{def}, and for every variable y, local to the body of c

- $P_1'|_{\tilde{L}_y}$ is recursion-free or function-free,
  where $\tilde{L}_y$ denotes the subset of the body of c consisting of those literals where y occurs;

then each c_i satisfies **A** in P_1.

Proof. It is a straightforward generalization of the proof of Proposition 5.2.6. □

5.3 Correctness of the transformation

The aim of this section is to prove the correctness of the transformation schema wrt Fitting's semantics, Theorem 5.2.3.

Correctness of the unfold operation

First we consider the unfold operation.

Corollary 5.3.1 (Correctness of the unfold operation) Let P' be the result of unfolding an atom of a clause in P. Then

- $Fit(P) = Fit(P')$

Proof. This is a subcase of Corollary 4.7.2, and the proof follows directly from Lemma 4.7.1. □

It should be mentioned that, because of the particular structure of the transformation sequence, here we never use self-unfoldings (that is, unfoldings in which the same clause is both the unfolded clause and one of the unfolding ones). Consequently the correctness of *Step 2* follows also from a result of Gardner and Shepherdson [47, Theorem 4.1] which states that if the program P' is obtained from P by unfolding (but not self-unfolding), then $Comp(P)$ and $Comp(P')$ are logically equivalent theories[4].

The following is a second, technical result on the consequences of an unfolding operation which will be needed in the sequel.

---

[3]This is a direct consequence of Lemma 4.7.1

[4]In [47] this result is stated for the usual two-valued program's completion. By looking at the proof it is straightforward to check that it holds also for the three-valued case

Lemma 5.3.2 Let P be a normal program, $cl : A \leftarrow \tilde{K}$. be a definite, clause of P. Suppose also that cl is the only clause of P whose head unifies with $A\theta$. If P' is the program obtained by unfolding at least once all the atoms in $\tilde{K}$, then, for each non-limit ordinal α

- if $A\theta$ is *true* (resp. *false*) in $\Phi_P^{\uparrow\alpha+1}$ then $A\theta$ is *true* (resp. *false*) in $\Phi_{P'}^{\uparrow\alpha}$

Proof. Let us first give a simplified proof by considering the case when $\tilde{K}$ consists of two atoms H, J and we perform a single unfolding on them; we will later consider the general case.

Let $\{H_1 \leftarrow \tilde{B}_1., \ldots, \; H_n \leftarrow \tilde{B}_n.\}$ be the set of clauses of P whose head unify with H via mgu's $\phi_1, \ldots, \phi_n$, and let $\{J_1 \leftarrow \tilde{C}_1., \ldots, \; J_m \leftarrow \tilde{C}_m\}$ be the set of clauses of P whose head unify with J. Unfolding H in cl and then J in the resulting clauses, will lead to the following program:

$P' = P \backslash \{cl\} \cup \{d_{i,j} : (A \leftarrow \tilde{B}_i, \tilde{C}_j.)\theta_{i,j})\}$

Where $\theta_{i,j} = mgu(J\phi_i, J_j)$. Here some of the clauses $d_{i,j}$ may be missing due to the fact that $J\phi_i$ and J_j may not unify, but this is of no relevance in the proof.

Note that the clauses $d_{i,j}$ are the only clauses of P' whose head could possibly unify with A.

Let $\tilde{y} = Var(H, J) \backslash Var(A)$ be the set of variables local to the body. We have to consider two cases.

a) $A\theta$ is *true* in $\Phi_P^{\uparrow\alpha+1}$. By the definition of Φ_P, $(\exists\tilde{y}\, H, J)\theta$ is *true* in $\Phi_P^{\uparrow\alpha}$. There has to be an extension σ of θ, $Dom(\sigma) = Dom(\theta) \cup \tilde{y} = Var(A, H, J)$ such that $(H, J)\sigma$ is *true* in $\Phi_P^{\uparrow\alpha}$. Let $H_i \leftarrow \tilde{B}_i$ and $J_j \leftarrow \tilde{C}_j$ be the clauses used to prove, respectively, $H\sigma$ and $J\sigma$. Hence there exists a τ such that $\theta_{i,j}\tau|_{Dom(\sigma)} = \sigma$, $H\sigma = H_i\theta_{i,j}\tau$, $J\sigma = J_j\theta_{i,j}\tau$, and $(\tilde{B}_i, \tilde{C}_j)\theta_{i,j}\tau$ is *true* in $\Phi_P^{\uparrow\alpha-1}$. By Lemma 4.7.1, $\Phi_P^{\uparrow\alpha-1} \subseteq \Phi_{P'}^{\uparrow\alpha-1}$, hence $(\tilde{B}_i, \tilde{C}_j)\theta_{i,j}\tau$ is *true* in $\Phi_{P'}^{\uparrow\alpha-1}$. It follows that $A\theta_{i,j}\tau = A\sigma = A\theta$ is *true* in $\Phi_{P'}^{\uparrow\alpha}$.

b) $A\theta$ is *false* in $\Phi_P^{\uparrow\alpha+1}$. By the definition of Φ_P, $(\exists\tilde{y}\, H, J)\theta$ is *false* in $\Phi_P^{\uparrow\alpha}$. Hence for all extensions σ of θ, such that $Dom(\sigma) = Dom(\theta) \cup \tilde{y} = Var(A, H, J)$, we have that $(H, J)\sigma$ is *false* in $\Phi_P^{\uparrow\alpha}$.

Hence for all such σ's, and for all i, j and τ such that $\theta_{i,j}\tau|_{Dom(\sigma)} = \sigma$, $H\sigma = H_i\theta_{i,j}\tau$, $J\sigma = J_j\theta_{i,j}\tau$, we have that $(\tilde{B}_i, \tilde{C}_j)\theta_{i,j}\tau$ is *false* in $\Phi_P^{\uparrow\alpha-1}$. By Lemma 4.7.1, $\Phi_P^{\uparrow\alpha-1} \subseteq \Phi_{P'}^{\uparrow\alpha-1}$, hence $(\tilde{B}_i, \tilde{C}_j)\theta_{i,j}\tau$ is *false* in $\Phi_{P'}^{\uparrow\alpha-1}$. Since the clauses $d_{i,j}$ are the only ones that define A in P', we have that $A\theta_{i,j}\tau = A\sigma = A\theta$ is *false* in $\Phi_{P'}^{\uparrow\alpha}$.

Now to complete the proof, we have to observe two facts:

- First, that if we perform some further unfoldings on the resulting clauses, then we can only "speed up" the process of finding the truth value of A. In fact, by the same kind of reasoning used above, if $A\theta$ is *true* in $\Phi_{P'}^{\uparrow\alpha}$, and P'' is obtained from P' by unfolding some atoms in the bodies of the clauses $d_{i,j}$, then, for some $\beta \leq \alpha$, $A\theta$ is *true* in $\Phi_{P''}^{\uparrow\beta}$.

- Second, that if cl contains just one atom, or more than two atoms, then the exact same reasoning applies. $\square$

The replacement operation

In order to prove the correctness of the unfold/fold transformation schema we will use (a simplified version of) the results in chapter 4 on the simultaneous replacement operation.

As we explained in section 2.3.2, Fitting's model semantics corresponds to the semantics given by $Comp_{\mathcal{L}}(P)_{\mathcal{L}} \cup \mathrm{DCA}_{\mathcal{L}}$. Here, for the sake of notation's simplicity, given two first-order formulas E and F and a normal program P, instead of writing, $E \cong_{Comp_{\mathcal{L}}(P) \cup \mathrm{DCA}_{\mathcal{L}}} F$ (See definition 4.1.2 and Lemma 4.2.3) we'll write $F \sim_P E$, or, equivalently, we'll say that F *is equivalent to* E *wrt* $Fit(P)$, Moreover, if the *delay* of F wrt E in *lfp*(Φ_P) is zero (see Definition 4.2.5) we'll say that F *is not-slower that* E. The following Theorem is a particular case of Corollary 4.2.7.

Theorem 5.3.3 Let P' be a program obtained by simultaneously replacing the conjunctions $\{\tilde{C}_1, \ldots, \tilde{C}_n\}$ with $\{\tilde{D}_1, \ldots, \tilde{D}_n\}$ in the bodies of the clauses of P. If for each $\tilde{C}_i$, there exists a (possibly empty) set of variables $\tilde{x}_i$ such that the following three conditions hold:

(a) [locality of the variables in $\tilde{x}_i$]. $\tilde{x}_i$ is a subset of the variables local to $\tilde{C}_i$ and $\tilde{D}_i$, that is, $\tilde{x}_i \subseteq Var(\tilde{C}_i) \cup Var(\tilde{D}_i)$ and the variables in $\tilde{x}_i$ don't occur in $\{\tilde{D}_1, \ldots, \tilde{D}_{i-1}, \tilde{D}_{i+1}, \ldots, \tilde{D}_n\}$ nor anywhere else in the clause where $\tilde{C}_i$ is found.

(b) [equivalence of the replacing and replaced parts]. $\exists \tilde{x}_i\, \tilde{D}_i \sim_P \exists \tilde{x}_i\, \tilde{C}_i$

(c) [the D_i's are not-slower than the C_i's]. $\exists \tilde{x}_i\, \tilde{D}_i$ is *not-slower* than $\exists \tilde{x}_i\, \tilde{C}_i$.

then $Fit(P) = Fit(P')$.

A property we will need in the sequel is the following.

Proposition 5.3.4 Suppose that $A \leftarrow \tilde{C}, \tilde{E}$ is a clause of P and that P' is obtained from P by replacing $\tilde{C}$ with $\tilde{D}$ in such a way that the conditions of Theorem 5.3.3 are satisfied (so that $Fit(P) = Fit(P')$). Then

- Each time that $A\theta$ is *true* (resp. *false*) in $\Phi_P^{\uparrow\alpha}$ then $A\theta$ is *true* (resp. *false*) in $\Phi_{P'}^{\uparrow\alpha}$

Proof. This is a consequence of the fact that the replacing conjunction is not-slower than the replaced one. The formal proof is omitted here, it can be inferred by analyzing the proof of Theorem 4.2.6. □

Before we provide the proof of the correctness of the four step schema, we need to establish some further preliminary results. The first one states that the converse of **A** holds in any case.

Proposition 5.3.5 Each time that $\exists \tilde{w}\, \tilde{B}\theta$ is *true* in some $\Phi_{P_1}^{\uparrow\beta}$, then there exists a non-limit ordinal $\alpha \leq \beta$ such that $\exists \tilde{w}\, \tilde{B}\theta$ is *true* in $\Phi_{P_1}^{\uparrow\alpha}$.

Proof. It follows at once from the definition of Fitting's operator. □

The following important transitive property holds:

Proposition 5.3.6 Let P and P' be normal programs, E and F be first order formulas;

- If $E \sim_P F$ and $Fit(P) = Fit(P')$, then $E \sim_{P'} F$. $\square$

Now we can provide the details of the proof.

Correctness of the four step schema

We now prove the correctness of the four step schema. For the sake of simplicity we restrict ourselves to the case in which *Step 1* introduces only one clause. The extension to the general case is straightforward.

Let $P_1, \ldots P_4$ be the sequence of programs obtained via the four step schema: P_1 is the initial program, i.e. the one that contains D_{def}. P_2, P_3 and P_4, are the programs obtained by applying steps *Step 2* through *Step 4*. In order to show that the Fitting's models of programs $P_1, \ldots P_4$ coincide, we proceed as follows:

By the correctness of the unfolding operation, Corollary 5.3.1 we have that $Fit(P_1) = Fit(P_2)$.

We perform some further unfolding on some atoms of P_2, obtaining a new program that we will call P_{2u}, again by Corollary 5.3.1 we have that $Fit(P_2) = Fit(P_{2u})$; then we produce a "parallel sequence" of programs P_{3u}, P_{4u} by applying the simultaneous replacement operation, miming, to some extent, the original transformation. By applying Theorem 5.3.3 we will show that $Fit(P_{2u}) = Fit(P_{3u}) = Fit(P_{4u})$.

Finally we show that programs P_{3u} and P_{4u} are obtainable respectively from P_3 and P_4 by appropriately applying the unfold operation, and hence, by Corollary 5.3.1, that $Fit(P_3) = Fit(P_{3u})$ and that $Fit(P_4) = Fit(P_{4u})$. This will end the proof. Fig.1 illustrates both the original transformation and its parallel sequence.

Initial program

Let us establish some notation: $P_1 \ldots P_4$ are the programs obtained by applying the four step schema to program P, and $c_0 : H \leftarrow \tilde{B}$. is the (only) clause added to program P in *Step 1*. We also denote by $\tilde{w}$ the set of the local variables of c_i, $\tilde{w} = Var(\tilde{B}) \backslash Var(H)$. For the moment, let us make the following restriction:

- till the end of 5.3, we assume that $\tilde{B}$ doesn't contain negative literals.

Later, in subsection 5.3, we will prove the general case.

A simple consequence of the fact that c_0 is the only clause defining the predicate symbol of H is the following.

Observation 5.3.7

- $H \sim_{P_1} \exists \tilde{w}\, \tilde{B}$; $\square$

P_2 and P_{2u}

P_2 is obtained by unfolding some of the atoms in $\tilde{B}$, so $P_2 = P \cup \{A_i \leftarrow \tilde{U}_i, \tilde{N}_i\}$, where the atoms in $\tilde{N}_i$ are those that have not been unfolded during *Step 1* (N stands for Not unfolded, while U for Unfolded), so $\tilde{N}_i$ is equal to a subset of an instance of $\tilde{B}$ and each A_i is an instance of H. We obtain P_{2u} from P_2 by further unfolding all the atoms in each $\tilde{N}_i$. We denote by $\{c_{i,j} : (A_i \leftarrow \tilde{U}_i)\gamma_{i,j}, \tilde{D}_{i,j}\}$ the set of clauses of P_{2u} obtained from clause c_i by unfolding the atoms in $\tilde{N}_i$. By the correctness of the unfolding operation, Corollary 5.3.1, we have that

$$Fit(P_1) = Fit(P_2) = Fit(P_{2u}) \tag{5.2}$$

$$\begin{tikzcd}
\begin{array}{c} P_1 = P \cup D_{\text{def}} \\ \text{where } D_{\text{def}} = \{c_0 : H \leftarrow \tilde{B}\} \end{array} \arrow[d] & \\
\begin{array}{c} P_2 = P \cup D_{\text{unf}} \\ \text{where } D_{\text{unf}} = \{c_i : A_i \leftarrow \tilde{U}_i, \tilde{N}_i\} \end{array} \arrow[r] \arrow[d] & \begin{array}{c} P_{2u} = P \cup D_{unf*} \\ \text{where } D_{unf*} = \{c'_{i,j} : (A_i \leftarrow \tilde{U}_i)\gamma_{i,j}, \tilde{D}_{i,j}\} \end{array} \arrow[d] \\
\begin{array}{c} P_3 = P \cup D_{\text{fold}} \\ \text{where } D_{\text{fold}} = \{c'_i : A_i \leftarrow \tilde{U}'_i, \tilde{N}_i\} \end{array} \arrow[r] \arrow[d] & \begin{array}{c} P_{3u} = P \cup D_{fold*} \\ \text{where } D_{fold*} = \{c'_{i,j} : (A_i \leftarrow \tilde{U}'_i)\gamma_{i,j}, \tilde{D}_{i,j}\} \end{array} \arrow[d] \\
\begin{array}{c} P_4 = P_{\text{fold}} \cup D_{\text{fold}} \\ \text{where } D_{\text{fold}} = \{c'_i : A_i \leftarrow \tilde{U}'_i, \tilde{N}_i\} \end{array} \arrow[r] & \begin{array}{c} P_{4u} = P_{\text{fold}} \cup D_{fold**} \\ \text{where } D_{fold**} = \{c'_{i,j} : (A_i \leftarrow \tilde{U}'_i)\gamma_{i,j}, \tilde{D}'_{i,j}\} \end{array}
\end{tikzcd}$$

Fig. 1. Diagram of the transformation (left) together with the “parallel sequence” (right).

Moreover, the following properties hold:

Observation 5.3.8

- $H \sim_{P_{2u}} \exists \tilde{w}\, \tilde{B}$;
- H is not-slower than $\exists \tilde{w}\, \tilde{B}$ in P_{2u}.

Proof. From Observation 5.3.7 we have that $H \sim_{P_1} \exists \tilde{w}\, \tilde{B}$. The first statement follows then from (5.2) and Proposition 5.3.6. For the second, fix θ and let β be the least ordinal such that $\exists \tilde{w}\, \tilde{B}\theta$ is *true* (or *false*) in $\Phi_{P_{2u}}^{\uparrow\beta}$. The clauses defining the atoms in $\tilde{B}$ are the same in P_1, P_2 and P_{2u}, so $\exists \tilde{w}\, \tilde{B}$ is *true* (resp. *false*) in $\Phi_{P_1}^{\uparrow\beta}$ as well. From condition **A** and Proposition 5.3.5 we have that β is a non-limit ordinal. Hence, by the definition of Φ, $H\theta$ is *true* (resp. *false*) in $\Phi_{P_1}^{\uparrow\beta+1}$, and, by Lemma 5.3.2 $H\theta$ is *true* (resp. *false*) in $\Phi_{P_{2u}}^{\uparrow\beta}$. □

P_3 and P_{3u}

P_{3u} is obtained from P_{2u} as follows.

Suppose that in *Step 2* we performed a recursive folding on the clause c_i : $A_i \leftarrow \tilde{B}\theta, \tilde{R}_i, \tilde{N}_i$ of P_2, obtaining c_i' : $A_i \leftarrow H\theta, \tilde{R}_i, \tilde{N}_i$ in P_3. In the diagram we denote by $\tilde{U}_i'$ the conjunction of literals resulting from the application of the recursive folding on the conjunction $\tilde{U}_i$ (so $\tilde{U}_i = \tilde{B}\theta, \tilde{R}_i$ and $\tilde{U}_i' = H\theta, \tilde{R}_i$).

On P_{2u} we then perform the following. In each of the clauses $c_{i,j}$ we transform $\tilde{U}_i\gamma_{i,j}$ into $\tilde{U}_i'\gamma_{i,j}$ by replacing conjunctions of literals of the form $\tilde{B}\theta\gamma_{i,j}$ with $H\theta\gamma_{i,j}$ wherever needed; we call the resulting clauses $c_{i,j}'$. It is easy to see that if we unfold all the atoms in $\tilde{N}_i$ in the body of clause c_i' in P_3, then the resulting clauses are exactly the $c_{i,j}'$ in P_{3u}; this is best shown by the diagram. Hence P_{3u} is obtainable from P_3 by appropriately applying the unfolding operation. From Corollary 5.3.1 it follows that

$$Fit(P_3) = Fit(P_{3u}) \tag{5.3}$$

Now we show that $Fit(P_{2u}) = Fit(P_{3u})$. First we need the following.

Proposition 5.3.9 Let Q be a program, A, B be atoms and $\tilde{y}$ be a set of variables, such that $A \sim_Q \exists \tilde{y}\, B$. Suppose also that η is a renaming over $\tilde{y}$ and that for each variable z that occurs in A or B, but not in $\tilde{y}$, $Var(z\eta) \cap Var(\tilde{y}\eta) = \emptyset$. Then

- $A\eta \sim_Q \exists(\tilde{y}\eta)\ B\eta$

Proof. Straightforward. □

Since $\gamma_{i,j}$ results from unfolding the atoms in $\tilde{N}_i$, we have that $Dom(\gamma_{i,j}) \cap Var(c_i) \subseteq Var(\tilde{N}_i)$. Hence, by the conditions on θ in *Step 2*, $Dom(\gamma_{i,j}) \cap \tilde{w}\theta = \emptyset$ and $\tilde{w}\theta\gamma_{i,j} = \tilde{w}\theta$; so $\theta\gamma_{i,j}$ is a renaming over $\tilde{w}$, and the variables in $\tilde{w}\theta\gamma_{i,j}$ do not occur anywhere else in $c_{i,j}$. From Observation 5.3.8 and Proposition 5.3.9 we have that

- $H\theta\gamma_{i,j} \sim_{P_{2u}} \exists(\tilde{w}\theta\gamma_{i,j})\ \tilde{B}\theta\gamma_{i,j}$;
- $H\theta\gamma_{i,j}$ is not-slower than $\exists(\tilde{w}\theta\gamma_{i,j})\ \tilde{B}\theta\gamma_{i,j}$ in P_{2u}.

Since we obtained P_{3u} from P_{2u} by simultaneously replacing conjunctions (of the form) $\tilde{B}\theta\gamma_{i,j}$ with $H\theta\gamma_{i,j}$, by Theorem 5.3.3

$$Fit(P_{2u}) = Fit(P_{3u}). \tag{5.4}$$

Moreover, the following properties hold:

Observation 5.3.10

- $H \sim_{P_{3u}} \exists \tilde{w}\, \tilde{B}$;

- H is not-slower than $\exists\tilde{w}\,\tilde{B}$ in P_{3u}.

Proof. The first statement follows from Observation 5.3.8, (5.4) and Proposition 5.3.6. For the second first note that going from P_{2u} to P_{3u} we have affected only clauses that define the predicate *new*, moreover no other predicates definition depends on these clauses, in particular the atoms in $\tilde{B}$ are independent from them, hence, since H is not-slower than $\exists\tilde{w}\,\tilde{B}$ in P_{2u}, the statement follows from Proposition 5.3.4. □

P_4 and P_{4u}

P_4 is obtained from P_3 by transforming some of the clauses of P of the form $A \leftarrow \tilde{B}\theta, \tilde{E}$ into $A \leftarrow H\theta, \tilde{E}$.

Now we want to obtain P_{4u} from P_{3u} in such a way that P_{4u} is obtainable also from P_4 by unfolding the atoms in the conjunctions $\tilde{N}_i$.

Let $d: \ A \leftarrow \tilde{B}\theta, \tilde{E}$ be one of the clauses of P_3 that are transformed in *Step 4*. First note that d belongs both to P_3 and P_{3u}, in fact d was already present it the original program P, and never modified. We can then apply the same operations to the clauses of P_{3u}. Observe that for the conditions on θ given in *Step 4*, and by Observation 5.3.10 we have that

Observation 5.3.11

- $H\theta \sim_{P_{3u}} \exists(\tilde{w}\theta)\ \tilde{B}\theta$
- $H\theta$ is not-slower than $\exists(\tilde{w}\theta)\ \tilde{B}\theta$ in P_{3u} □

Second, notice that in case that d was used as unfolding clause for going from P_2 to P_{2u}, then some instances of $\tilde{B}\theta$ were propagated into P_{3u}. Using the notation of the diagram, this is the case when some $\tilde{N}_i$ (in P_2) is of the form $A', \tilde{F}_i$ where A and A' are unifiable atoms, then one of the $\tilde{D}_{i,j}$ (in P_{2u}) is of the form $\tilde{D}_{i,j} = (\tilde{B}, \tilde{F}_i)\theta'$. However, if we unfold N_i in P_4, what we get is $\tilde{D}'_{i,j} = H\theta', \tilde{F}_i$, that has $H\theta'$ instead of $\tilde{B}\theta'$. By the same argument used for $\theta\gamma_{i,j}$ in 5.3, we have that

Observation 5.3.12

- $H\theta' \sim_{P_{3u}} \exists(\tilde{w}\theta')\ \tilde{B}\theta'$
- $H\theta'$ is not-slower than $\exists(\tilde{w}\theta')\ \tilde{B}\theta'$ in P_{3u} □

So in order to obtain P_{4u} from P_{3u} we have then to do two things: First, replace $\tilde{B}\theta$, with the corresponding $H\theta$ in all the clauses d that are transformed in *Step 4*. Second, replace $\tilde{B}\theta'$ with $H\theta'$ in the $\tilde{D}_{i,j}$ so that P_{4u} contains $\tilde{D}'_{i,j}$ instead of $\tilde{D}_{i,j}$. This tantamounts to the application of a simultaneous replacement.
From Observations 5.3.11 and 5.3.12, and Theorem 5.3.3 we have that

$$Fit(P_{3u}) = Fit(P_{4_u}) \tag{5.5}$$

Moreover P_{4u} is obtainable from P_4 by unfolding all the atoms in the conjunctions $\tilde{N}_i$ in the clauses where they occur. Hence

$$Fit(P_4) = Fit(P_{4u}). \tag{5.6}$$

So far, because of (1), (2), (3), (4) and (5), we have the following

Proposition 5.3.13 If condition **A** holds and $\tilde{B}$ does not contain negative literals, then

- $Fit(P_1) = Fit(P_2) = Fit(P_3) = Fit(P_4)$ $\square$

The general case

We can finally prove Theorem 5.2.3. Let us state it again.

Theorem 5.2.3. *Let $P_1, \ldots, P_4$ be a sequence of programs obtained applying the transformation* schema *to program P, Let also $D_{\mathrm{def}} = \{H_i \leftarrow \tilde{B}_i\}$ be the set of clauses introduced in* Step 1, *and, for each i, $\tilde{w}_i$ be the set of local variables of c_i: $\tilde{w}_i = Var(\tilde{B}_i) \backslash Var(H_i)$. If each c_i in D_{def} satisfies the following condition:*

A *each time that $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is* false *in some $\Phi_{P_1}^{\uparrow\beta}$, then there exists a non-limit ordinal $\alpha \leq \beta$ such that $\exists \tilde{w}_i\, \tilde{B}_i\theta$ is* false *in $\Phi_{P_1}^{\uparrow\alpha}$*

Then $Fit(P_1) = Fit(P_2) = Fit(P_3) = Fit(P_4)$.

Proof. We consider here the simplified case in which *Step 1* introduces only one clause which in turn contains only one negative literal in the body, i.e. $D_{\mathrm{def}} = \{c_0 :\ H \leftarrow \neg l(\tilde{y}), \tilde{B}'\}$. The generalization to the case of multiple clauses and multiple negative literals is straightforward and omitted here. Notice that if c_0 contained no negative literals, then the result would following directly from Proposition 5.3.13.

We now perform a double transformation on P_1: first, we enlarge it with the following new definition: $d : notl(\tilde{y}) \leftarrow \neg l(\tilde{y})$; then, we replace each instance $\neg l(\tilde{t})$ of $l(\tilde{y})$ that occurs in the body of a clause with the corresponding instance $notl(\tilde{t})$ of $notl(\tilde{y})$. This replacement operation clearly preserves Fitting's model of the programs, in fact it can be undone by unfolding. Let us call P_1' the program so obtained. We have that

$$Fit(P_1) = Fit(P_1')|_{B_{P_1}} \tag{5.7}$$

Where $Fit(P_1')|_{B_{P_1}}$ denotes the restriction of $Fit(P_1')$ to the atoms in the Herbrand base of P_1.

Now P_1' contains, instead of clause c_0, the following: $c_0' = H \leftarrow notl(\tilde{y}), \tilde{B}'$. which is a *definite* clause.

Now notice that, since the unfold operation is defined only for positive literals, then $\neg l(\tilde{y})$ is never unfolded in the transformation $P_1 \ldots P_4$. It follows that, by performing the same operations used for going from P_1 to P_4, we can obtain another "parallel sequence" $P_1' \ldots P_4'$ that starts with program P_1'. By the same arguments used to prove (5.7), we have that, for $i \in [1 \ldots 4]$,

$$Fit(P_i) = Fit(P_i')|_{B_{P_1}} \tag{5.8}$$

Moreover, by Proposition 5.3.13,

$$Fit(P_1') = Fit(P_2') = Fit(P_3') = Fit(P_4') \tag{5.9}$$

From (5.8) and (5.9) the thesis follows. $\square$

Chapter 6

Unfold/Fold Transformations of CLP Modules

In this chapter We propose a transformation system for CLP programs and modules. The framework is inspired by the one of Tamaki and Sato for pure logic programs [96]. 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 non-modular system w.r.t. the least model semantics.

6.1 Introduction

Modular Constraint Logic Programs

Constraint Logic Programming (CLP for short) is a powerful declarative programming paradigm in which constraints are primitive elements and the computation is specified by a logical inference rule. CLP has already been successfully employed in many diverse fields such as financial analysis [63], circuit synthesis [49] and combinatorial search problems [97]. Its success is partially due to the fact that the declarative nature of CLP allows us to solve complex problems by simple and concise programs. CLP's flexibility can be further enhanced by the adoption of constructs for structuring programs. This is an important step forward as the incremental and modular design is by now a well established software-engineering methodology used to design, verify and maintain large applications. Indeed, splitting a program into several smaller *modules* reduces the complexity of the design and of the validation phases. Moreover, it also helps to develop adaptable software, as changes in program's specification can affect only some modules rather that the whole program. For these reasons, modularity has been receiving received a considerable attention and, as the recent survey [24] shows, in the last few years several different proposals were introduced for integrating module constructs into logic languages. Here we adhere to the original approach of R. O'Keefe [76], and we consider a constraint logic program to be a combination of several separate modules, where different modules are combined together by a simple composition operator $\oplus$.

Motivation

All the (unfold/fold) transformation systems proposed so far for (constraint) logic programs, with the only exception of [69], 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*.

Now, a transformation system for modules requires ad-hoc applicability conditions: when we transform P into P' we don't just want P and P' to have the same (answer constraint) semantics: we want them to be observationally equivalent *whatever the context in which they are employed*. When this condition is satisfied we say that P and P' are observationally *congruent*.

In this chapter, 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 [96] 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 that the ones used previously.

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

This chapter is organized as follows. The next Section contains some preliminaries on CLP programs. In Section 6.3 we introduce the notion of module and we formalize the resultants semantics for CLP by using trees. Section 6.4 provides the definition of the transformation system. In Section 6.5 we add the applicability conditions needed to obtain a modular system and we state the main correctness result. In Section 6.6 we show that the Tamaki-Sato's system can be embedded into ours. As a consequence, the conditions given in Section 6.5 can also be added to those defined in [96] in order to obtain a modular unfold/fold system for pure logic programs. Section 6.7 concludes by comparing our results to those contained in two related works. The proof of the main technical result is deferred to the Appendix.

6.2 Preliminaries: CLP programs

The *Constraint Logic Programming* paradigm CLP(X) (CLP for short) has been proposed by Jaffar and Lassez [52, 51] 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 [52]. Indeed, as discussed in [69], 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 [65, 3] provide the necessary background material for logic programming theory. For constraint logic programs we refer to the original papers [52, 51] by Jaffar and Lassez and to the recent survey [53] by Jaffar and Maher.

The CLP framework was originally defined using a many-sorted first order language. In this chapter, to keep the notation simple, we consider a one sorted language (the extension of our results to the 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 $\Box$ 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

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

where c is a constraint, H (the head) and $B_1, \ldots, B_n$ (the body) are atomic formulas which use predicate symbols from Π_u only. A *goal* (or query), denoted by $c \Box B_1, \ldots, B_n$, 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) $\mathcal{D}$ consists of a set (the domain) D and an assignment of functions and relations on D to the function symbols in Σ and to the predicate symbols in Π_c respecting arities.

A $\mathcal{D}$-interpretation is an assignment that maps each predicate symbols in Π_u to a relation on the domain of the structure. A $\mathcal{D}$-interpretation I is called a $\mathcal{D}$-*model* of a CLP program P if all the rules of P evaluate to true under the assignment of relations and function provided by I and by $\mathcal{D}$. We recall that there exists ([51]) the least $\mathcal{D}$-model of a program P which is the natural CLP counterpart of the least Herbrand model for logic programs.

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

Here and in the sequel, given the atoms A, H, we write $A = H$ as a shorthand for:

- $a_1 = t_1 \wedge \ldots \wedge a_n = t_n$, if, for some predicate symbol p and natural n, $A \equiv p(a_1, \ldots, a_n)$ and $H \equiv p(t_1, \ldots, t_n)$

- *false*, otherwise.

This notation readily extends to conjunctions of atoms. We also find convenient to use the notation $\exists_{-\tilde{x}}\, \phi$ from [53] 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 $\mathcal{D}$-solvability for unifiability. More precisely, a derivation step for a goal $G:\ c_0 \Box B_1, \ldots, B_n$ in the program P results in the goal

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

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) $H \leftarrow c \Box \tilde{B}$ such that $(c_0 \wedge (B_i = H) \wedge c)$ is $\mathcal{D}$-satisfiable, that is, $\mathcal{D} \models \exists\, c_0 \wedge (B_i = H) \wedge c$. A derivation of length i for a goal G_0 in the program P is a sequence of goals $G_0, G_1, \ldots, G_i$ such that G_j is obtained from G_{j-1} in one derivation step in P, for $j \in [1, i]$. In the following a derivation ξ : $G_0, G_1, \ldots, G_i$ in P will be denoted by $G_0 \overset{P}{\leadsto} G_i$ and its length by $|\xi|$ Notice that, with this notation, a derivation of length zero is denoted by $G \overset{P}{\leadsto} G$. A *successful* derivation (*refutation*) is a finite derivation whose last element is a goal of the form $(c \; \Box)$. In this case, $\exists_{-Var(G)}\, c$ is called the *answer constraint* and is considered the result of the computation.

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

6.3 Modular CLP Programs

Following the original paper of R. O'Keefe [76], the approach to modular programming we consider here is based on a *meta-linguistic* programs composition mechanism. This provides a formal background to the usual software engineering techniques for the incremental development of programs.

Viewing modularity in terms of *meta-linguistic* operations on programs has several advantages. In fact it leads to the definition of a simple and powerful methodology for structuring programs which does not require to extend the CLP theory (this is not the case if one tries to extend CLP programs by *linguistic* mechanisms richer than those offered by clausal logic). Moreover, *meta-linguistic* operations are quite powerful, indeed the typical mechanisms of the object-oriented paradigm, such as encapsulation and information hiding, can be realized by means of simple composition operators ([16]).

Here, in order to keep the presentation simple, we follow [22] 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 6.3.1 (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. $\Box$

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 $\mathcal{I}$ 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 $\mathcal{E}$ which contains facts (unit clauses) describing the basic relations. In this case the extensional predicates which are defined in $\mathcal{E}$ are exported to $\mathcal{I}$, 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 [22] and use a simple program union operator. We denote by $Pred(E)$ set of predicate symbols which appear in the expression E.

Definition 6.3.2 (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. $\Box$

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

Definition 6.3.3 (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 $Q \overset{P_i}{\leadsto} c_i \,\Box$ then there exists a derivation $Q \overset{P_j}{\leadsto} c_j \,\Box$ such that $\mathcal{D} \models \exists_{-Var(Q)}\, c_i \leftrightarrow \exists_{-Var(Q)}\, c_j$. $\Box$

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

Example 6.3.4 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 wrt the $\oplus$ operator. Therefore the following.

Definition 6.3.5 (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 notions of equivalence and of congruence are used to define the correctness of our transformation system: 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.

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 [42]) 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 us a straightforward lifting to the CLP case of the compositional semantics defined in [22] for logic programs. The aim of [22] 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 $\mathcal{D}$-model and the computed answer semantics, are not compositional wrt $\oplus$; consider for instance the modules M_1 and M_2 in Example 6.3.4: they have the least $\mathcal{D}$-model, where $M_1 \oplus M$ and $M_2 \oplus M$ don't (the same reasoning applies for the answer constraint semantics of [43]). Following an idea first introduced in [44], 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 6.3.6 Let $cl_1 : \; A_1 \leftarrow c_1 \,\Box\, \tilde{B}_1$ and $cl_2 : \; A_2 \leftarrow c_2 \,\Box\, \tilde{B}_2$ be two clauses. We write $cl_1 \simeq cl_2$ iff for any $i, j \in [1, 2]$ and for any $\mathcal{D}$-solution ϑ of c_i there exists an $\mathcal{D}$-solution γ of c_j such that $A_i\vartheta = A_j\gamma$ and $\tilde{B}_i\vartheta$ and $\tilde{B}_j\gamma$ are equal as multisets.

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. $\square$

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 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 wrt the congruence relation $\approx_c$.

Lemma 6.3.7 [42] Let $M = \langle P, \pi\rangle$ and $M' = \langle P', \pi\rangle$ be two modules with the same set of open atoms. If $P \simeq P'$ then $M \approx_c M'$. $\square$

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

The following is the definition of the resultant semantics as it was originally given in [22] for pure logic programs and applied to CLP in [42].

Definition 6.3.9 (Resultants Semantics for CLP) Let $M = \langle P, Op(M)\rangle$ be a module. Then we define

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

$\square$

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 $c \,\square\, \tilde{A} \leftarrow d \,\square$. The compositionality of previous semantics w.r.t. $\oplus$ is proved in [42]. From such a result it follows the correctness of $\mathcal{O}$ w.r.t. $\approx_c$, stated by the following Corollary.

Corollary 6.3.10 (Correctness, [42]) 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. $\approx$ ([43]).

Example 6.3.11 Consider again the modules M_1 and M_2 of Example 6.3.4. Then

$$\begin{aligned} \mathcal{O}(M_1) = &\ \{p(X) \leftarrow X = a \,\square\,,\ q(X) \leftarrow X = a \,\square\,,\ q(X) \leftarrow true \,\square\, p(X)\} \\ \mathcal{O}(M_2) = &\ \{p(X) \leftarrow X = a \,\square\,,\ q(X) \leftarrow X = a \,\square\,\} \end{aligned}$$

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

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 labeled tree and the related terminology. Given a finite labeled 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 6.3.12 (Partial proof tree) Let A be an atom A *partial proof tree* for A is any finite labeled tree T satisfying the following conditions

1. The root node of T is labeled 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 $1 \leq j \leq n$.
3. All the clauses used in the labels of T are pairwise variable disjoint and have no variables in common with the atom in the lhs (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 subtrees 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 6.3.13 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, $1 \leq j \leq n$, 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. $\square$

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 6.3.14 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. $\square$

Definition 6.3.15 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. $\square$

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 6.3.16 Let π be a set of predicate symbols. We call π-atom any atom A such that $Pred(A) \in \pi$. An 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. $\square$

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

Proposition 6.3.17 (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 an 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. $\square$

6.4 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 [96] for pure logic programs (which is presented in chapter 1. 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 [96].

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 $\simeq$. As we formally prove later, this implies that we can always replace any clause cl in a program P by a clause cl', provided that $cl' \simeq cl$. This operation is often useful to *clean up* the constraints, and, in general, to present a clause in a more readable form. We start from the same requirements on the original (i.e. initial) program introduced in [96]. 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 6.4.1 (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} don't occur in P_{old} nor in the bodies of the clauses in P_{new}. □

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

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 6.4.2 (Computing an average) Consider the following CLP($\Re$)[1] program `AVERAGE` computing the average of the values in a list. Values may be given in different currencies, for this reason each element of the list contains a term of the form ⟨`Currency`, `Amount`⟩. The applicable exchange rates may be found by calling 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),
            weighted_sum(Rest, Rates, Sum´).
```

[1]CLP($\Re$) [55] is the CLP language obtained by considering the constraint domain $\Re$ of arithmetic over the real numbers.

```
len(List, Len) ←
    Len is the length of the elements in 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*. 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 are variable disjoint.

Definition 6.4.3 (Unfolding, for CLP) Let $cl : A \leftarrow c \,\Box\, H, \tilde{K}$ be a clause in the program P, and $\{H_1 \leftarrow c_1 \,\Box\, \tilde{B}_1, \ldots, H_n \leftarrow c_n \,\Box\, \tilde{B}_n\}$ be the set of the clauses in P such that $c \wedge c_i \wedge (H = H_i)$ is $\mathcal{D}$-satisfiable. For $i \in [1, n]$, let cl_i' be the clause

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

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 $\{H_1 \leftarrow c_1 \,\Box\, \tilde{B}_1, \ldots, H_n \leftarrow c_n \,\Box\, \tilde{B}_n\}$ are the *unfolding* clauses.

Example 6.4.2 (part 2) The transformation strategy which we use to optimize `AVERAGE` is often referred to as *tupling* (see [77]) or as *procedural join* (see [62]). 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 body of the program; to do that we need the following operation.

Definition 6.4.4 (Clause Removal) Let $cl : H \leftarrow c \,\Box\, \tilde{B}$ be a clause in the program P. If

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

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

Note 6.4.5 In [77] we find the definition of a *clause deletion* operation for pure logic programs which in CLP terms can be expressed as follows: if $cl : H \leftarrow c \,\Box\, \tilde{B}$ is a clause in P such that query $c \,\Box\, \tilde{B}$ has a finitely failed tree in P[2] then we can remove cl from P. Obviously, if $\mathcal{D} \models \neg\exists\ c$ then the goal $c \,\Box\, A$ 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 [77]. 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 $c \,\Box\, \tilde{B}$ has a finitely failed tree in P then, by a suitable sequence of unfoldings we can always transform the clause $A \leftarrow c \,\Box\, \tilde{B}$, 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'll move to the context of *modular* CLP programs the

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

first operation will remain unchanged while the latter would 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 are variable disjoint.

Definition 6.4.6 (Splitting) Let $cl : A \leftarrow c \,\Box\, H, \tilde{K}$ be a clause in the program P, and $\{H_1 \leftarrow c_1 \,\Box\, \tilde{B}_1, \ldots, H_n \leftarrow c_n \,\Box\, \tilde{B}_n\}$ be the set of the clauses in P such that $c \wedge c_i \wedge (H = H_i)$ is $\mathcal{D}$-satisfiable. For $i \in [1, n]$, let cl'_i be the clause

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

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).
```

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 6.4.7 We should mention that an operation called splitting has also been defined in a technical report of Tamaki and Sato [95]. However, the operation described here is substantially different from theirs. In CLP terms the splitting operation defined in [95] 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 $\mathcal{D}$-model (which corresponds to semantics used in [95]) but is does not produce $\approx$ equivalent programs. Indeed, if we consider the program $P = \{\mathtt{p(X)}.\}$ then by splitting its only clause w.r.t. the constraint $\mathtt{X} = \mathtt{a}$ we obtain the program $P' = \{\mathtt{p(X)} \leftarrow \mathtt{X} = \mathtt{a}\,\Box.,\ \mathtt{p(X)} \leftarrow \mathtt{X} \neq \mathtt{a}\,\Box.\}$. Clearly $P' \not\approx P$, since the query p(X) returns the answer constraint $\mathtt{X} = \mathtt{a}$ in P' only. □

Example 6.4.2 (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 `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 this 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 $\mathtt{Len}' > \mathtt{0}$. Clearly, adding $\mathtt{Len}' > \mathtt{0}$ to the body of `c8` cannot be done via a simple cleaning-up of the constraints, as it transforms `c8` in a non $\simeq$-equivalent clause. However, notice that the variable $\mathtt{Len}'$ in the atom $\mathtt{len([J|Rest]},\mathtt{Len}')$ (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 $\mathtt{Len}'$ will eventually be bounded to a numeric value greater than zero. We can then safely add the redundant constraint $\mathtt{Len}' > \mathtt{0}$ 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 $\mathtt{len([J|Rest]},\mathtt{Len}')$ succeeds in the current program with answer constraint c then c is equivalent to $c \wedge Len' > 0$).

Definition 6.4.8 (Constraint Replacement) Let $cl : H \leftarrow c_1 \,\square\, \tilde{B}$ be a clause of a program P and let c_2 be a constraint. If, for each successful derivation $true \,\square\, \tilde{B} \stackrel{P}{\leadsto} d \,\square$,

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

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

Constraint replacement has some similarities with the refinement operation as defined by Marriott and Stuckey in [73]. Refinement allows 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 $c_1 \,\Box\, \tilde{B}$, $\mathcal{D} \models d \rightarrow c$ holds, i.e. c is redundant in d. Clearly this case is covered by our definition. However, the similarities between this chapter and [73] end here. In [73], 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 6.4.2 (part 4) By performing a constraint replacement of

```
Len > 0  ∧ Len = Len´+1  ∧ Average*Len = Amount*Value+Sum´
```

by

```
Len > 0 ∧ Len = Len´+1 ∧ Average*Len = Amount*Value+Sum´ ∧ 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. $\Box$

We are now ready for the folding operation. Intuitively, this operation 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 [96], 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 and the folded 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 on its left hand side.

Definition 6.4.9 (Folding) Let $P_0, \ldots, P_i$, $i \geq 0$, be a transformation sequence. Let also
$cl : A \leftarrow c_A \,\Box\, \tilde{K}, \tilde{J}$ be a clause in P_i,
$d : D \leftarrow c_D \,\Box\, \tilde{H}$ be a clause in P_{new}.
If $c_A \,\Box\, \tilde{K}$ is an instance of *true* $\Box\, \tilde{H}$ 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

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

provided that the following three conditions hold:

(CLP1) (i) *"If we unfold D in cl′ using d as unfolding clause, then we obtain cl back" (modulo ≃),*
or, equivalently,
(ii) $\mathcal{D} \models \exists_{-Var(A,\tilde{J},\tilde{H})}\; c_A \wedge e \wedge c_D \;\leftrightarrow\; \exists_{-Var(A,\tilde{J},\tilde{H})}\; c_A \wedge (\tilde{H} = \tilde{K})$

(CLP2) *"d is the only clause of P_{new} that can be used to unfold D in cl′"*,
that is,
there is no clause $b :\; B \leftarrow c_B \,\Box\, \tilde{L}$ in P_{new} such that $b \neq d$ and $c_A \wedge e \wedge (D = B) \wedge c_B$ is $\mathcal{D}$-satisfiable.

(CLP3) *"No self-folding is allowed"*, that is
(a) either the predicate in A is an old predicate;
(b) or cl is the result of at least one unfolding in the sequence $P_0, \ldots, P_i$. $\Box$

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.

Conditions **CLP1** and **CLP2** 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 **CLP2** ensures that in P_{new} there exists no clause other than d that can be used as *unfolding clause.*

We now show that **CLP1**(i) and **CLP1**(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 $c_A \,\Box\, \tilde{K}$ is an instance of *true* $\Box\, \tilde{H}$, it follows that each solution of c_A can be extended to a solution of $c_A \wedge (\tilde{H} = \tilde{K})$. Hence

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

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

$$cl \;\simeq\; A \leftarrow c_A \wedge (\tilde{H} = \tilde{K}) \,\Box\, \tilde{H}, \tilde{J} \tag{6.1}$$

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

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

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

$$cl'' \;\simeq\; A \leftarrow c_A \wedge e \wedge c_D \,\square\, \tilde{H}, \tilde{J}. \tag{6.2}$$

From (6.1) and (6.2) it follows immediately that

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

This proves that condition **CLP1**(i) is equivalent to **CLP1**(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 **CLP3** 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 6.4.2 (part 5) We can now fold

$$\texttt{exchange\_rates(Rates)}, \texttt{sum}([\texttt{J}|\texttt{Rest}], \texttt{Rates}, \texttt{Sum}'), \texttt{len}([\texttt{J}|\texttt{Rest}], \texttt{Len}')$$

in `c9`, using `c2` as folding clause. In this case, the bridge constraint e has to be

$$\texttt{XS} = [\texttt{J}|\texttt{Rest}] \;\wedge\; \texttt{RATES} = \texttt{Rates} \;\wedge\; \texttt{LEN} = \texttt{Len}' \;\wedge\; \texttt{AV} = \texttt{Sum}'/\texttt{Len}'$$

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 **CLP1** 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

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

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. $\approx$, that is any transformed program together with a generic query Q will produce the same answer constraints of the original one. This is the content of the following result, which follows from the more general one contained in Section 6.5.

Corollary 6.4.10 (Correctness) If $P_0, \ldots, P_n$ is a transformation sequence then

(a) $P_0 \approx P_n$.
(b) The least $\mathcal{D}$-models of P_0 and P_n coincide.

Proof. Statement (a) is proven in Section 6.5 as a Corollary of Theorem 6.5.4. The fact that (a) implies (b) is proven in [42]. □

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 wrt the $\approx_c$ congruence is stated in Lemma 6.3.7. 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 6.4.11 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 6.4.9, 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 $P_{new}(\subset P_0)$.
- 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 one delicate step is taken care of by the following Observation.

Observation 6.4.12 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 (CLP1) holds iff $c_A \;\Box\; \tilde{K}$ is an instance of $c_D \;\Box\; \tilde{H}$ and (CLP1) 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 6.4.9 shows that, if $c_A \;\Box\; \tilde{K}$ is an instance of $true \;\Box\; \tilde{H}$ and (CLP1) holds, then we have the following equivalences:

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

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. $\Box$

This Observation shows that there is no loss of generality in modifying the applicability conditions of the folding operation Definition 6.4.9 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 $(CLP1) \wedge (CLP2) \wedge (CLP3)$ 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. $\Box$

6.5 A transformation system for CLP modules

Corollary 6.4.10 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 Corollary 6.4.10 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 $\{\texttt{c1}: \ \texttt{p} \leftarrow \texttt{q}.\}$ and where $Op(M_0) = \{\texttt{q}\}$. 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 without the 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: $\quad$ `p(X)` $\leftarrow$ *true* $\Box$ `q(X).`
$\quad\quad$ `q(a).`

where $Op(M_0) = \{\texttt{q}\}$. The only answer constraint to the query $\texttt{q(X)}$ in M_0 is $\texttt{X} = \texttt{a}$. Therefore, if we refer to the applicability conditions of Definition 6.4.8, we could add the constraint $\texttt{X} = \texttt{a}$ 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 $\texttt{p(b)}$ succeeds in $M_0 \oplus N$ and fails in $M_1 \oplus N$.

Definition 6.5.1 (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 non-modular programs notice that the applicability conditions of Definition 6.4.8 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 $true \,\square\, \tilde{B} \overset{P}{\leadsto} d \,\square\,$ we have that

$$H \leftarrow c_1 \wedge d \,\square \;\simeq\; H \leftarrow c_2 \wedge d \,\square$$

Now it is clear that the difference lies in the fact that here we cannot just refer to the successful derivations $true \,\square\, \tilde{B} \overset{P}{\leadsto} d \,\square\,$, 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 6.4.8 and 6.5.1 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 $M_{new} = \{\texttt{p} \leftarrow \texttt{q}\}$. 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 6.4.2, 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 6.5.2 (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 Corollary 6.4.10. 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 this chapter and shows that any modular transformation sequence preserves the resultants semantics.

Theorem 6.5.3 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 previous Theorem and the correctness result for the resultants semantics we can now derive easily the correctness of a modular transformation sequence.

Theorem 6.5.4 (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 6.5.3 and Corollary 6.3.10. □

In other words, for any module N such that $M_0 \oplus N$ is defined, $M_n \oplus N$ is also defined[3] 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 Corollary 6.4.10 of previous Section.

[3]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.

Corollary 6.4.10 If $P_0, \ldots, P_n$ is a transformation sequence, then,

$$P_0 \approx P_n.$$

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 6.5.4. □

Example 6.4.2 (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 6.5.4 guarantees that the final program will behave exactly as the initial one, even in this modular setting. □

6.6 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 [96] for LP, which is described in chapter 1. As a consequence, conditions **O1**
OCHO and **O4** can be used also in the LP case to transform a module into a congruent one.

Since clause removal, splitting and constraint replacement are new operations which were not in [96], 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[4].

Concerning the unfolding operation, it is easy to see that Definition 6.4.3 is the CLP counterpart of Definition 3.2.3. In fact, an LP clause is itself a CLP rule (with an empty constraint) and well known results ([64]) 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 3.2.3 iff we can unfold P according to Definition 6.4.3. 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.

[4]However, we should mention that in [96] also a more general replacement operation is taken into consideration, but this operation is beyond the scope of this chapter.

On the other hand, when considering the folding operation, the similarities between Definitions 3.2.5 and 6.4.9 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 6.6.1 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[5]. 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

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

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 3.2.5 are satisfied, by **F1** 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

$\mu(d) \ \equiv \ d^* : a(\tilde{x}) \leftarrow \tilde{x} = \tilde{s} \wedge \tilde{y} = \tilde{t} \ \Box \ b(\tilde{y})$,

$\mu(cl) \ \equiv \ cl^* : c(\tilde{z}) \leftarrow \tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \ \Box \ b(\tilde{w}), d(\tilde{k})$.

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

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

the result of the folding operation in CLP is then

[5] *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.

$cl'^* : c(\tilde{z}) \leftarrow \tilde{z} = \tilde{u} \wedge \tilde{w} = \tilde{t}\tau \wedge \tilde{k} = \tilde{v} \wedge \tilde{x} = \tilde{s}\tau \,\square\, a(\tilde{x}), d(\tilde{k})$.
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 **F1, F2** in P_n then d^*, cl^* and e satisfy **CLP1** in $\mu(P_n)$. Here the structure $\mathcal{D}$ is the Herbrand structure, whose domain is the Herbrand universe and where "=" is interpreted as the identity.

Now the condition **CLP1** is $\mathcal{D} \models \exists_{-\tilde{z},\tilde{y}}\, c_{left} \leftrightarrow \exists_{-\tilde{z},\tilde{y}}\, c_{right}$
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 **CLP1** to

$$\mathcal{D} \models \exists_{-\tilde{z},\tilde{y}}\;\; \tilde{z} = \tilde{u} \wedge \tilde{s} = \tilde{s}\tau \wedge \tilde{y} = \tilde{t} \;\;\leftrightarrow\;\; \exists_{-\tilde{z},\tilde{y}}\;\; \tilde{z} = \tilde{u} \wedge \tilde{y} = \tilde{t}\tau \tag{6.3}$$

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 $\mathcal{D} \models c\vartheta$.

We now show that for each solution η of one side of (6.3) there exists a solution η' of the other side of (6.3) 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' $Dom(\eta') = Dom(\eta) \cup Dom(\tau)$: for each $l \in Dom(\tau)$, we let

$$l\eta' \text{ be equal to } l\tau\eta. \tag{6.4}$$

η' is now also a solution of the left hand side of (6.3). In fact
$\tilde{s}\eta' = \tilde{s}\tau\eta$ (by (6.4))
$= \tilde{s}\tau\eta'$ (because η' is an extension of η).
Moreover
$\tilde{y}\eta' = \tilde{t}\tau\eta'$ (because η' is an extension of η, and η is a solution of $y = \tilde{t}\tau$)
$= t\eta'$ (by (6.4)).
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 **F2** we have that $\tilde{l}\tau$ is a tuple of distinct variables. Moreover, the variables in $\tilde{l}\tau$ don't occur anywhere else in the above formulas. So, for each $l_i \in \tilde{l}$, we can let

$$l_i\tau\eta' \text{ be equal to } l_i\eta. \tag{6.5}$$

Since η is already a solution of $\tilde{s} = \tilde{s}\tau$ and η' is an extension of η, by (6.5) 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 (6.3), which concludes the proof. □

Theorem 6.6.1 allows us to apply the results of the previous Section also to the Tamaki-Sato schema, thus obtaining a 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 $\approx^{LP}$ is defined as follows:

- $P_1 \approx^{LP} P_2$ 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 $Q\vartheta_i \equiv Q\vartheta_j$[6].

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

- $M_1 \approx_c^{LP} M_2$ 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, $M_1 \oplus N \approx^{LP} M_2 \oplus N$ holds.

Corollary 6.6.2 Let $M_0 : \langle P_0, \pi \rangle$ be a logic programming module, $P_0, \dots, P_n$ 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 6.6.1 and 6.5.4. □

6.7 Conclusions

Among the works on program's transformations, the most closely related to this chapter are Maher's [69] and the one of Bensaou and Guessarian [14].

Maher considers several kind of transformations for deductive databases modules with constraints (allowing negation in the bodies of the clauses) and refers to the perfect model semantics. However, the folding operation proposed in [69] 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 [69], 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 [14], yet there are some substantial differences between [14] and our proposal.

Firstly, since in an unfold/fold transformation sequence we allow more operations, we obtain a more powerful system. For instance, the transformation performed in

[6] 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.

Example 6.4.2 is not feasible with the tools of [14]. On the other hand, since in [14] the authors define also a goal replacement operation, there exist also some transformation which can be done with the tools of [14] and not with ours. However, such a replacement operation cannot be fitted in a 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 [14] 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 chapter.

Secondly, the semantics they refer to is an extension to the CLP case of the C-semantics ([29, 40]). Such a semantics characterizes the logical consequences of the program on $\mathcal{D}$-models, but does not allow to model answer constraints. For example, the C-semantics identifies the programs { `p(X,Y)` $\leftarrow$ `X=a,Y=b` $\square$`., p(X,Y).`} and { {`p(X,Y).` } which have different answer constraint for the goal `p(X,Y)`, and consequently are not identified by the answer constraint semantics in [43]. Since the C-semantics can be obtained as the upward closure of the answer constraint semantics, the result on the correctness of the unfold/fold system of [14] is a particular case of our Corollary 6.4.10. Moreover, we believe that the answer constraints semantics provides a better reference semantics for transformation systems, since answer constraints are the most natural properties that one would like to preserve while transforming programs.

A third relevant difference is due to the fact that since modularity is not take into account in [14], 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 [14] is that of the applicability conditions we propose are invariant under $\simeq$-equivalence (Proposition 6.4.11), while the ones in [14] are not: this means that in some cases the folding conditions of [14] may not be satisfiable unless we appropriately modify the constraints of the clauses (maintaining $\simeq$-equivalence).

To conclude, the contributions of this chapter 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 [96]. 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 $\mathcal{D}$-model of the original program.

A definition of a modular transformation sequence is given by adding some further applicability conditions. These conditions are shown to be sufficient to guarantee the correctness of the system w.r.t. the 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 [96]. This relation allows us to prove that, under conditions **O1** and **O4**, the system by Tamaki and Sato transforms a 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 [73] and for low-level local optimization (in which the constraint solving is partially compiled into imperative statements) [56, 54]. 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 [62] and further discussed in [77]. Also these strategies could easily be extended to CLP and integrated with our transformation rules.

6.8 Appendix

In this Appendix we first give the proof of Theorem 6.5.3 which shows that any modular transformation sequence preserves the resultants semantics. The proof, quite long an tedious, is split in two parts (partial an total correctness) and is inspired by the one given in [57].

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
$\pi = Op(M_0)$. □

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 6.5.3. Before proving such an inclusion we need to establish some further notation.

Definition 6.8.1 We say that two trees T and T' are *similar* if they are partial trees of 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 6.8.2 Let T be an 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. $\square$

Lemma 6.8.3 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 lhs of the label equation of the root node. If A' and A have the same relation symbol, and A' is variable-disjoint from T, then T' is a partial proof tree of A'.

Proof. Obvious. $\square$

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 6.8.4 Let P be a program and $A \leftarrow d \,\square\, \tilde{D}$ be an resultant. Equivalent are

- There exists a derivation $true \,\square\, A \overset{P}{\leadsto} d' \,\square\, \tilde{D}'$ such that $A \leftarrow d \,\square\, \tilde{D} \simeq A \leftarrow d' \,\square\, \tilde{D}'$;
- 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 6.8.5 ([42]) Let P be a program, if, for distinct $i, j \in [1, k]$, there exists a derivation

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

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 exist a derivation

$$true \,\square\, A_1, \ldots, A_k \overset{P}{\leadsto} c_1 \wedge \ldots \wedge c_k \,\square\, \tilde{F}_1, \ldots, \tilde{F}_k.$$

$\square$

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

Proposition 6.8.6 (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 $P_1, \ldots, P_n$ rather that to $M_1, \ldots, M_n$.

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 an π-tree T_A of 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 6.3.17, 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 an 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, that is, 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:

- $cl:\ A_0 \leftarrow c_A \,\square\, A_1, \ldots, A_n$
- $cl':\ A_0 \leftarrow c_A \wedge (A_1 = B) \wedge c_B \,\square\, A_1, \ldots, A_n$

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

$$(A = A_0) \wedge c_A \wedge (A_1 = B) \wedge c_B \wedge (A_1 = C) \wedge c_C \wedge \ \ldots \tag{6.6}$$

Now $C \leftarrow c_C \,\square\, \tilde{E}$ 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 (6.6) 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 clause cl' with cl in the label of the root of T_A, we obtain a tree T^1_A which is similar to T_A.

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

Finally, let T^2_A be the tree obtained from T^1_A by substituting each subtree T_{A_i} with $T^2_{A_i}$, by Lemma 6.8.2 we have that T^2_A is similar to T^1_A, and therefore to T_A. Since T^2_A is an 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' : $A \leftarrow c' \,\Box\, A_1, \ldots, A_n$, and
- cl : $A \leftarrow c \,\Box\, A_1, \ldots, A_n$. Where cl is the clause to which the replacement was applied. Let also $T_{A_1}, \ldots, T_{A_{n'}}$ be the subtrees of T_A (which we suppose attached to $A_1, \ldots, A_{n'}$), $c_{A_1}, \ldots, c_{A_{n'}}$ be their internal constraints and $\tilde{F}_{A_1}, \ldots, \tilde{F}_{A_{n'}}$ be their residuals. With this notation, the resultant of T_A is

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

By Lemma 6.8.4, the existence of $T_{A_1}, \ldots, T_{A_{n'}}$ implies that for $i \in [1, n']$ there exists a derivation $true \,\Box\, A_i \stackrel{P_{i+1}}{\leadsto} c_{A_i} \,\Box\, \tilde{F}_{A_i}$ (modulo $\simeq$). Since by inductive hypothesis each subtree of T_A has a similar subtree in P_i, Remark 6.8.4 also implies that, for $i \in [1, n']$ there exists a derivation which is equal (modulo $\simeq$) to

$$true \,\Box\, A_i \stackrel{P_i}{\leadsto} c_{A_i} \,\Box\, \tilde{F}_{A_i}.$$

By combining these derivations together (Remark 6.8.5) we have that there exists a derivation

$$true \,\Box\, A_1, \ldots, A_n \stackrel{P_i}{\leadsto} \tilde{c}_{A_1} \wedge \ \ldots \ \wedge c_{A_{n'}} \,\Box\, \tilde{F}_{A_1}, \ldots, \tilde{F}_{A_{n'}}, A_{n'+1}, \ldots, A_n. \tag{6.7}$$

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

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

From Remark 6.8.4 it follows that there exists an π-tree S_A of A in P_i whose resultant is

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

From (6.7) and the applicability conditions for the replacement operations 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 : $A_0 \leftarrow c_A \,\Box\, B_1^-, \ldots B_m^-, A_1, \ldots, A_n$ be the folded clause (in P_i)
- d : $B_0 \leftarrow c_B \,\Box\, B_1, \ldots, B_m$ be the folding clause (in P_{new}),

so we have that

- cl' : $A_0 \leftarrow c_A \wedge e \,\Box\, B_0, A_1, \ldots, A_n$ 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 an 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 6.3.17 it follows that there exists an 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 **CLP2**, the label clause of the root of S_{B_0} is an appropriate renaming of d. Let

- $d^* : B_0^* \leftarrow c_B^* \,\Box\, B_1^*, \ldots, B_m^*$ 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 6.8.2 we have that

$$R \simeq R^2 \tag{6.8}$$

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

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

By **CLP1**, this reduces to

$$(A = A_0) \wedge c_A \wedge (B_0^* = B_0) \wedge (\wedge_{j=1}^{m} B_j^* = B_j) \wedge (\wedge_{j=1}^{m'} B_j^* = C_j) \wedge c_{rest} \tag{6.9}$$

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 $(\wedge_{j=1}^{m} B_j^* = B_j)$ or the variables x and y do not occur anywhere else in (6.9), nor in R^2. So (6.9) becomes

$$(A = A_0) \wedge c_A \wedge (\wedge_{j=1}^{m} B_j^* = B_j) \wedge (\wedge_{j=1}^{m'} B_j^* = C_j) \wedge c_{rest} \tag{6.10}$$

On the other hand, by replacing B_j^* with B_j^- in the lhs 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 6.8.3, 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 $A \leftarrow c_{tot}^3 \,\Box\, \tilde{F}, B_{m'+1}^-, \ldots, B_m^-, A_{n'+1}, \ldots, A_n$, where c_{tot}^3 is

$$c_A \wedge (\wedge_{j=1}^m B_j^- = B_j) \wedge (\wedge_{j=1}^{m'} B_j^- = C_j) \wedge c_{rest}$$

By this, (6.10) and (6.8), 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, by replacing each $S_{B_j^-}$ with it in T_A^3, obtaining T_A^4 by Lemma 6.8.2 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. □

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 6.8.7 If *cl* is a clause of P_i that does not satisfy condition **CLP3** 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 6.8.8 (weight)

- The weight of an 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 6.8.9 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 $T_1, \ldots, T_k$ are trees in P_0;
- if $T_1, \ldots, T_k$ are trees of $A_1, \ldots, A_k$ and $R_1, \ldots, R_k$ are their resultants, then
  (a) $w(A, R) \geq w(A_1, R_1) + \ldots + w(A_k, R_k)$;
  (b) $w(A, R) > w(A_1, R_1) + \ldots + w(A_k, R_k)$ if *cl* satisfies **CLP3**. □

The above definition is a generalization of the definition of *descent clause* of [57].

Definition 6.8.10 We call P_i *weight complete* iff for each atom A and resultant R, if there is an 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 6.8.11 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 an π-tree of A in P_0 with resultant R. Since P_i is weight complete, there exist descent tree T_A of A in $P_i \cup P_0$ with resultant R. Let also
- $cl : A_0 \leftarrow c_A \;\Box\; A_1, \ldots A_n$ (in P_i) be the label clause of its root,
- $A_1, \ldots, A_{n'}$ be those atoms of cl that have an immediate subtree attached to
- $T_{A_1}, \ldots, T_{A_{n'}}$ be the immediate subtrees of T_A (in P_0) and $R_{A_1}, \ldots, R_{A_{n'}}$ be their resultants.
Then, since T_A is a descent tree,
$w(A, R) \geq w(A_1, R_{A_1}) + \ldots + w(A_{n'}, R_{A_{n'}})$.
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 doesn't satisfy **CLP3**, by Remark 6.8.7, 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 6.8.2 the tree T''_A, obtained from T_A by replacing each subtree T_{A_j} with T''_{A_j}, is an π-tree of A in P_i with resultant R. This proves the Proposition. $\Box$

We we are now ready to prove our total correctness Theorem.

Theorem 6.5.3 (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 an 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 6.8.9. Let cl be the label clause of the root of T, notice that cl satisfies **CLP3** iff its head is an *old* atom, just like the elements of

its body. From the Definition of weight 6.8.8 and the minimality of T, it follows that condition (b) in Definition 6.8.9 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 6.8.6 and 6.8.11 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 an 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 6.8.2 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 an 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 $d : \; B_0 \leftarrow c_B \,\Box\, B_1, \ldots, B_m$ (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 an π-tree of A in $P_i \cup P_0$; let R'_A be its resultant, by Lemma 6.8.2 and the usual assumption on the variables in the clauses of the subtrees, we have that

$$R \simeq R'_A \tag{6.11}$$

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 6.8.2 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

$$R'_A \simeq A \leftarrow (A = A_0) \wedge c_A \wedge (A_1 = B_0) \wedge c_B \wedge c_{rest} \,\Box\, \tilde{F}, B_{m'+1}, \ldots, B_m, A_{n'+1}, \ldots, A_n \tag{6.12}$$

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

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

By (6.11) and (6.12) 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 6.8.9 are satisfied.
Now
$w(A, R_A) \geq w(A_1, R_{A_1}) + \ldots + w(A_{n'}, R_{A_{n'}})$ (since T_A is a descent tree),
$\geq w(B_1, R_{B_1}) + \ldots + w(B_{m'}, R_{B_{m'}}) + w(A_2, R_{A_2}) + \ldots + w(A_{n'}, R_{A_{n'}})$ (since T_{A_1} is a descent tree)
Moreover, if d satisfies **CLP3** then, by condition (b) in Definition 6.8.9.
$w(A_1, R_{A_1}) > w(B_1, R_{B_1}) + \ldots + w(B_{m'}, R_{B_{m'}})$
On the other hand if d does not satisfy **CLP3**, then by Remark 6.8.7 the predicate of B_0 and A_1 must be a new predicate; again, by Remark 6.8.7 we have that cl must satisfy **CLP3**. It follows that
$w(A, R_A) > w(A_1, R_{A_1}) + \ldots + w(A_{n'}, R_{A_{n'}})$
So, in any case, we have that
$w(A, R_A) > w(T_{B_1}) + \ldots + w(T_{B_{m'}}) + w(T_{A_2}) + \ldots + w(T_{A_{n'}})$
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 $n' \geq 1$.

Since $\mathcal{O}(P_0) = \mathcal{O}(P_i)$, we have that for $i \in [1, n']$ there exist an 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 $T_{A_1}, \ldots, T_{A_{n'}}$ with $S_{A_1}, \ldots, S_{A_{n'}}$. From Lemma 6.8.2 and the usual standardization apart of the clauses in the subtrees, it follows that S_A is an π-tree of A in P_i and that S_A is similar to T_A.

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

$$A \leftarrow (A = A_0) \wedge c_A \wedge (A_1 = B_0) \wedge c_B \wedge c_{rest} \,\square\, Residual \qquad (6.13)$$

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

Where d^* : $B_0^* \leftarrow c_B^* \,\Box\, B_1^*, \ldots, B_m^*$ 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 (6.13) it follows that the resultant of T'_A is ($\simeq$ equivalent to)

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

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 (6.13) 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, that is, that it satisfies conditions (a) and (b) of Definition 6.8.9, 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 **CLP3** 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 an 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 $T_{A_1}, \ldots, T_{A_{n'}}$ with $S_{A_1}, \ldots, S_{A_{n'}}$. From Lemma 6.8.2 and the usual standardization apart of the subtrees it follows that S_A is an π-tree of A in P_i and that S_A is similar to T_A.

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

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

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 $S_{A_1}, \ldots, S_{A_{n'}}$ and from Remarks 6.8.4 and 6.8.5 it follows that there exist a derivation

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

Now, let the result of the constraint replacement operation be the clause

- cl' : $A_0 \leftarrow c'_A \,\Box\, A_1, \ldots, A_n$.

From the applicability conditions of the constraint replacement operation it follows that the resultant

$$A_0 \leftarrow (A = A_0) \wedge c_A \wedge c_{A_1} \wedge \ldots \wedge c_{A_{n'}} \,\Box\, \tilde{F}_{A_1}, \ldots, \tilde{F}_{A_{n'}}, A_{n'+1}, \ldots, A_n, \not\preceq \quad (6.14)$$
$$A_0 \leftarrow (A = A_0) \wedge c'_A \wedge c_{A_1} \wedge \ldots \wedge c_{A_{n'}} \,\Box\, \tilde{F}_{A_1}, \ldots, \tilde{F}_{A_{n'}}, A_{n'+1}, \ldots, A_n,$$

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

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

And from (6.14) 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, that is, that it satisfies conditions (a) and (b) of Definition 6.8.9, 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 **CLP3** iff cl does.

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

$$R \simeq A \leftarrow (A = A_0) \wedge c_A \wedge (\wedge_{j=1}^{n'} A_j = C_j) \wedge c_{rest} \,\Box\, \tilde{F}, A_{n'+1}, \ldots, A_n. \tag{6.15}$$

Now let the folding clause (in P_{new}) be
$d: \; B_0 \leftarrow B_1, \ldots, B_m$
There is no loss in generality in assuming that there exists an index k such that $A_k, \ldots, A_{k+m}$ are the unfolded atoms, so for $j \in [1, m]$, A_{k+j} and B_j are unifiable atoms. The result of the folding operation is then
$cl': \; A_0 \leftarrow c_A \wedge e \,\Box\, A_1, \ldots A_k, B_0, A_{k+m+1}, \ldots A_{n'}$
Now notice that of the atoms of cl that are going to be folded, $A_{k+1}, \ldots, A_{n'}$ are the ones that have an immediate subtree attached to in T_A, These atoms correspond to $B_1, \ldots, B_{n'-k}$ in d, (we should also consider explicitly the cases all have or have not a subtree attached to, that is, the cases in which $n' < k$ or $n' \geq m + k$, however these are easy corollaries of the general case, so we now assume that $k \leq n' < m + k$).
Now let T_{B_0} be the π-tree of B_0 in P_0 built as follows:
- $d': \; B'_0 \leftarrow c'_B \,\Box\, B'_1, \ldots, B'_m$ (an appropriate renaming of d) is the label clause of its root node,
- $B_0 = B'_0$ is then the label equations of its root node,
- $T_{B'_1}, \ldots, T_{B'_{n'-k}}$ are its immediate subtrees, which are obtained, as explained in Lemma 6.8.3, from the trees $T_{A_{k+1}}, \ldots, T_{A_{n'}}$ by replacing A_{k+j} with B'_j in the lhs of the label equations of their root nodes.
- $B'_{n'-k+1}, \ldots, B'_m$ 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}).
- $T_{A_1}, \ldots, T_{A_{k-1}}, T_{B_0}$ are its immediate subtrees (in P_0).
Let R'' be its resultant, we have that

$$R'' = A \leftarrow c_{tot} \,\Box\, \tilde{F}, B'_{n'-k+1}, \ldots, B'_m, A_{k+m+1}, \ldots, A_n \tag{6.16}$$

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

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

By **CLP1** this becomes:

$$(A = A_0) \wedge c_A \wedge (B_0 = B'_0) \wedge (\wedge_{j=1}^{m} B_j = B'_j) \wedge (\wedge_{j=1}^{k} A_j = C_j) \wedge (\wedge_{j=k+1}^{n'} B'_{j-k} = C_j) \wedge c_{rest} \tag{6.17}$$

As we did in Proposition 6.8.6, 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 $(\wedge_{j=1}^{m} B_j = B'_j)$, or the variables x and y do not occur anywhere else in (6.17), nor in R''.

Thus c_{tot} can be rewritten as follows:

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

By making explicit the constraint $(\wedge_{j=1}^{m} B_j = B'_j)$ and comparing the result with (6.15) we see that T''_A is an 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, that is, that it satisfies the conditions (a), (b) of the Definition 6.8.9.

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 **CLP2**, 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 6.8.8 we then have that

$$w(B_0, R_{B_0}) \leq w(T_{B_1}) + \ldots + w(T_{B_{n'-k}}) \tag{6.18}$$

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 **CLP3**, by Definition 6.8.8 we have that

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

$= w(A_1, R_{A_1}) + \ldots + w(A_k, R_{A_k}) + w(A_{k+1}, R_{A_{k+1}}) + \ldots + w(A_{n'}, R_{A_{n'}})$

$= w(A_1, R_{A_1}) + \ldots + w(A_k, R_{A_k}) + w(T_{A_{k+1}}) + \ldots + w(T_{A_{n'}})$ (by the minimality of the T_{A_j})

$= w(A_1, R_{A_1}) + \ldots + w(A_k, R_{A_k}) + w(T_{B_1}) + \ldots + w(T_{B_{n'-k}})$ (by the definition of T_{B_j})

$\geq w(A_1, R_{A_1}) + \ldots + w(A_k, R_{A_k}) + w(B_0, R_{B_0})$ (by (6.18)).

Thus T''_A satisfies conditions (a) and (b) of Definition 6.8.9. □

Chapter 7

The Replacement Operation for CLP Modules

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

7.1 Introduction

Central to the development of large and efficient applications is now the study of optimization techniques for programs and modules. Concerning specifically the CLP paradigm, the literature on this subject can be divided into two main branches. On one hand we find methods which focus exclusively on the manipulation of the constraint for compile-time [73] and for low-level local optimization (in which the constraint solving may be partially compiled into imperative statements) [56]. Compile time optimizations based on static analysis have also been investigated [72]. On the other hand there are techniques such as the unfold/fold transformation systems, which were developed initially for Logic Programs [96] and then applied to CLP in [69, 14] and in chapter 6 of this thesis. These latter methods focus primarily on the declarative side of the program.

Replacement is a program transformation technique flexible enough to encompass both the above kind of optimization: it can be profitably used to manipulate both the constraint and the "declarative" side of a CLP program. In fact the replacement operation, which was introduced in the field of Logic Programming by Tamaki and Sato [96] and later applied to CLP in [69, 14], syntactically consists in replacing a conjunction of atoms in the body of a program clause by another conjunction. It is therefore a very general operation and it is able to mimic many other transformations, such as thinning, fattening [18] and folding (see [77] for a survey on transformation techniques for logic languages).

Clearly, a primary requirement a transformation operation should satisfy is *correctness*: the original and the transformed program should be equivalent wrt to some (operational or declarative) reference semantics. In the logic programming area, a lot of research [96, 67, 47, 88, 20, 69, 14, 32, 80] has been devoted to the definition of *applicability conditions* sufficient to guarantee the correctness of replacement wrt several different semantics. Unfortunately, apart from [69], none of these transformation systems can be correctly applied to *modules*. In fact, since they all refer to semantics which are not compositional wrt $\oplus$, they provide correctness results which
OCHO are adequate only if programs are seen as stand alone units. As we already explained in chapter 6, when we transform a *module* M into M' we don't just want M and M' to have the same behavior: we want them semantically equivalent *whatever is the context in which we use them*. In other words we need some further applicability conditions which guarantee that, given any other module Q, $M \oplus Q$ and $M' \oplus Q$ will be equivalent to each other. When this condition is satisfied we say that M and M' are *compositionally* equivalent or *congruent*[1].

Furthermore, even when restricting to the non modular setting, the applicability conditions so far provided for the replacement transformations suffer from drawbacks which, in our opinion, prevented a wider diffusion of the operation. On one hand, some of them [47, 88, 67, 69] do not allow replacement to introduce recursion, which, as we will shortly see, is an important feature for optimizing Constraint Logic Programs. On the other hand, other approaches [96, 20, 80] do exploit the full potentiality of replacement, but at the price of applicability conditions which are discouragingly complicated.

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

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

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

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

(a) *When the replacing conjunction is at least as efficient as the replaced one.* Referring to the operational semantics this means that each time we can compute an "answer" constraint c for the replaced conjunction (in the given program) in n steps, we can also compute the answer c for the replacing one in m steps with $m \leq n$. This is undoubtedly a desirable situation which fits well in the natural context in which the transformation is performed in order to increase program's execution speed. Moreover, this condition is flexible enough to allow us to introduce recursion (which can be seen as an example of *non-fatal* loop) in the definition of the predicates.
(b) *When the replacing conjunction is independent from the clause that is going to be transformed.*
This clearly guarantees that no loops are introduced.

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

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

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

This chapter is organized as follows. In Section 7.2 we state the applicability conditions needed to obtain compositionally equivalent programs, wrt the answer constraints notion of observable, and we present the main correctness result. In Section 7.3 we illustrate the optimization technique based on replacement through a simple example. Section 7.4 shows how the applicability conditions can be modified (weakened) when we refer to other semantic properties of modules. Section 7.5 concludes by comparing our results to those contained in some related papers. Some proofs are deferred to the Appendix.

Preliminaries

The notations and the necessary preliminary notions are given in the previous chapter, sections 6.2 and 6.3. The only difference is that in this chapter we'll use a slightly more restrictive form of $\cong$-equivalence: given two clauses having the same head, $cl_1 : A \leftarrow c_1 \,\Box\, \tilde{B}_1$ and $cl_2 : A \leftarrow c_2 \,\Box\, \tilde{B}_2$. We say that cl_1 is *similar* to cl_2, $cl_1 \simeq cl_2$, iff for $i, j \in [1, 2]$, for any $\mathcal{D}$-solution ϑ of c_i there exists a $\mathcal{D}$-solution γ of c_j such that $\tilde{B}_i\vartheta$ and $\tilde{B}_j\gamma$ are equal as multisets. Notice that, as opposed to definition 6.3.6 here we also require that two clauses, in order to be similar, must have exactly the same heads (this will simplify the proofs).

7.2 Operational correctness of Replacement

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

In this section we first define an *operational* notion of correctness based on the *answer constraints*. Then we provide some applicability conditions for replacement in form of a natural formalization of the requirements (i) and (ii) discussed in the introduction. Then we show that, whenever these conditions are satisfied, the replacement operation is operationally correct. Later, in Section 7.4, we will also show how these conditions can be modified (weakened) when considering correctness based on different operational and logical notions.

Operational congruence

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

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

Definition 7.2.1 Let P, P' be two programs, $\xi :\ c \,\Box\, \tilde{C} \overset{P}{\leadsto} b \,\Box\, \tilde{B}$ and $\xi' :\ c \,\Box\, \tilde{C} \overset{P'}{\leadsto} b' \,\Box\, \tilde{B}'$ be two derivations starting in the same goal. Let also $\tilde{x} = Var(c \,\Box\, \tilde{C})$. We say that

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

iff $q(\tilde{x}) \leftarrow b \,\Box\, \tilde{B} \ \simeq\ q(\tilde{x}) \leftarrow b' \,\Box\, \tilde{B}'$, where q is any (dummy) predicate symbol[2]. □

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

Definition 7.2.2 (Operational Congruence) Let M_1 and M_2 be CLP modules that have the same set of open predicates. We say that

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

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

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

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

We now give a result which provides a condition sufficient to guarantee the operational congruence of two modules. Here, and in the sequel, given a set of predicate symbols π we call a π-*derivation* any derivation $c \,\Box\, \tilde{C} \leadsto b \,\Box\, \tilde{B}$ such that $Pred(\tilde{B}) \subseteq \pi$.

Theorem 7.2.3 [42] Let $M_1 = \langle P_1, \pi\rangle$ and $M_2 = \langle P_2, \pi\rangle$ be two modules. If

- for each π-derivation in M_1 there exists a similar π-derivation in M_2

then, for every module M such that $M_1 \oplus M$ and $M_2 \oplus M$ are defined, we have that for any refutation in $M_1 \oplus M$ there exists a similar refutation in $M_2 \oplus M$. $\Box$

Partial correctness

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

Definition 7.2.4 (Query's operational equivalence) Let $M = \langle P, \pi\rangle$ be a module, $c_1 \,\Box\, \tilde{C}_1$ and $c_2 \,\Box\, \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

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

iff for each π-derivation $c_i \,\Box\, \tilde{C}_i \overset{P}{\leadsto} b_i \,\Box\, \tilde{B}_i$, renamed apart wrt $\tilde{x}$, there exists a derivation $c_j \,\Box\, \tilde{C}_j \overset{P}{\leadsto} b_j \,\Box\, \tilde{B}_j$, renamed apart wrt $\tilde{x}$ such that $q(\tilde{x}) \leftarrow b_i \,\Box\, \tilde{B}_i \;\simeq\; q(\tilde{x}) \leftarrow b_j \,\Box\, \tilde{B}_j$, where $i, j \in [1, 2]$, $i \neq j$ and q is any (dummy) predicate symbol[3]. $\Box$

The idea behind the above definition, and which distinguishes it from all the previous approaches, is that in a modular context we cannot just refer to refutations, but we also have to take into account those partial derivations that end in a tuple of open atoms, whose definition could eventually be modified. Notice that the larger is the set of open predicates we consider, the stronger becomes the definition of equivalence. Indeed, having more open predicates implies that the derivations we consider are more likely to be influenced by the adjoining of external definitions.

As we informally mentioned in the introduction, when we replace $c \,\Box\, \tilde{C}$ by $d \,\Box\, \tilde{D}$ in the clause $cl : \; A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$, our first requirement will be the equivalence of $c \,\Box\, \tilde{C}$ and $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M. We now show that if this requirement is satisfied then the operation is at least *partially* correct. This is the content of the following.

Theorem 7.2.5 (Partial Correctness) Let $cl : \; A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$ be a clause in the module $M : \langle P, \pi\rangle$ and $M' : \langle P', \pi\rangle$ be the result of replacing $c \,\Box\, \tilde{C}$ by $d \,\Box\, \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}\}$. If

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

- $d \,\Box\, \tilde{D}$ is *$\mathcal{O}$-equivalent to* $c \,\Box\, \tilde{C}$ under $Var(A, \tilde{E})$ in M,

then for each π-derivation ξ' in M' there exist a similar π-derivation ξ in M.

Proof. Here, as well as in the proof of some other theorems that will follow, some equations will be labeled with the special sign †. We do this because we are also going to refer to such equations also in the sequel, however, as far as this proof is concerned, these labels are of no relevance. First, we need to state a couple of preliminary results. The proof of the first one is immediate, and thus it is omitted.

Claim 7.1 Let P be a program, and $c \,\Box\, \tilde{C}$ be a query. Then, for any n, there exists a derivation $c \,\Box\, \tilde{C} \overset{P}{\leadsto} d \,\Box\, \tilde{D}$ of length n iff there exists a derivation $true \,\Box\, \tilde{C} \overset{P}{\leadsto} d' \,\Box\, \tilde{D}$ of length n such that

(i) $d \equiv c \wedge d'$
(ii) the variables that $d' \,\Box\, \tilde{D}$ and c have in common are a subset of the variables of $\tilde{C}$.

$\Box$

Claim 7.2 [42] Let P be a program, and $c_1 \wedge c_2 \,\Box\, \tilde{C}_1, \tilde{C}_2$ be a query. Then, there exists a derivation $c_1 \wedge c_2 \,\Box\, \tilde{C}_1, \tilde{C}_2 \overset{P}{\leadsto} d \,\Box\, \tilde{D}$ of length n iff there exist two derivations $\xi_1 :\ c_1 \,\Box\, \tilde{C}_1 \overset{P}{\leadsto} d_1 \,\Box\, \tilde{D}_1$ and $\xi_2 :\ c_2 \,\Box\, \tilde{C}_2 \overset{P}{\leadsto} d_2 \,\Box\, \tilde{D}_2$ such that

(i) $\tilde{D} \equiv \tilde{D}_1, \tilde{D}_2$, and $d \equiv d_1 \wedge d_2$ is satisfiable,
(ii) the variables that ξ_1 and ξ_2 have in common are exactly those that $c_1 \,\Box\, \tilde{C}_1$ and $c_2 \,\Box\, \tilde{C}_2$ have in common,
(iii) $|\xi_1| + |\xi_2| = n$. $\Box$

We can now continue with the proof of the Theorem, so let ξ' be a π-derivation in M'. We have to show that there exists a derivation ξ in M which is similar to ξ'. For this we proceed by induction on the length of the derivation. The base case, $|\xi'| = 0$, is trivial, as the derivations of length zero are (by definition) the ones of the form $b \,\Box\, \tilde{B} \overset{M'}{\leadsto} b \,\Box\, \tilde{B}$. Therefore we proceed with the inductive step. By Claims 7.1 and 7.2, ξ' can be chosen of the form

$$\xi' :\ true \,\Box\, H \overset{M'}{\leadsto} b \,\Box\, \tilde{B}.$$

Where $\tilde{B}$ contains only π-atoms, and where (since this derivation has length greater than 0) we can assume that $Var(H) \cap Var(\tilde{B}) = \emptyset$. By the definition of derivation, there has to exist a (renaming of a) clause of M',

$$J \leftarrow c_L \,\Box\, \tilde{L} \tag{7.1}$$

and a π-derivation

$$\zeta' :\ (H = J) \wedge c_L \,\Box\, \tilde{L} \overset{M'}{\leadsto} b \,\Box\, \tilde{B}.$$

Where $|\xi'| = |\zeta'| + 1$. By the inductive hypothesis, there exists a derivation ζ in M such that $\zeta \simeq \zeta'$ †. Now, if the clause of (7.1) was also a clause of M (that is, if it was not a result of the transformation), then there would exist a derivation ξ in M such that $\xi \simeq \xi'^{\dagger}$, concluding the proof. So we have to consider the case in which $J \leftarrow c_L \,\square\, \tilde{L} \in M' \backslash M$; in this situation, $J \leftarrow c_L \,\square\, \tilde{L}$ is exactly (a variant of) the clause $cl' : A \leftarrow d \,\square\, \tilde{D}, \tilde{E}$. By appropriately renaming all the variables in the clauses and the derivations considered so far, we can assume that ζ' is exactly the derivation

$$\zeta' : (H = A) \wedge d \,\square\, \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b \,\square\, \tilde{B}.$$

By Claim 7.2, there exist two derivations ζ'_1 and ζ'_2 such that

$$\begin{array}{c}
\zeta'_1 : d \,\square\, \tilde{D} \overset{M'}{\leadsto} b_1 \,\square\, \tilde{B}_1, \\
\zeta'_2 : (H = A) \,\square\, \tilde{E} \overset{M'}{\leadsto} b_2 \,\square\, \tilde{B}_2, \\
b \equiv b_1 \wedge b_2 \text{ and } \tilde{B} \equiv \tilde{B}_1, \tilde{B}_2, \\
|\zeta'_1| + |\zeta'_2| = |\zeta'| = |\xi'| - 1, \\
Var(b_1 \,\square\, \tilde{B}_1) \cap Var(b_2 \,\square\, \tilde{B}_2) \subseteq Var(d \,\square\, \tilde{D}) \cap Var((H = A) \,\square\, \tilde{E}).
\end{array} \tag{7.2}$$

Here and in the sequel, we make the following assumption:

Assumption 7.2.6 Each time we consider a new clause or a new derivation, the variables that the new expression has in common with the ones previously mentioned are only the ones that are strictly necessary.

By the inductive hypothesis, there exist two derivations ζ_1 and ζ_2 in M, such that

$$\zeta_1 : d \,\square\, \tilde{D} \overset{M'}{\leadsto} b_1^* \,\square\, \tilde{B}_1^*,$$

$$\zeta_2 : (H = A) \,\square\, \tilde{E} \overset{M'}{\leadsto} b_2^* \,\square\, \tilde{B}_2^*,$$

$$\zeta_1 \simeq \zeta_1'^{\dagger} \quad \text{and} \quad \zeta_2 \simeq \zeta_2'^{\dagger}, \tag{7.3}$$

$$Var(b_1^* \,\square\, \tilde{B}_1^*) \cap Var(b_2^* \,\square\, \tilde{B}_2^*) \subseteq Var(d \,\square\, \tilde{D}) \cap Var((H = A) \,\square\, \tilde{E}). \tag{7.4}$$

Since $d \,\square\, \tilde{D}$ is equivalent to $c \,\square\, \tilde{C}$ under $Var(A, \tilde{E})$ in M, it follows that there exists a derivation

$$\zeta_3 : c \,\square\, \tilde{C} \overset{M}{\leadsto} b_3 \,\square\, \tilde{B}_3$$

such that for any dummy predicate symbol q, if we let $\tilde{x} = Var(A, \tilde{E})$,

$$q(\tilde{x}) \leftarrow b_1 \,\square\, \tilde{B}_1 \simeq q(\tilde{x}) \leftarrow b_3 \,\square\, \tilde{B}_3. \tag{7.5}$$

Here there is no loss in generality in assuming that the variables of $b_3 \,\square\, \tilde{B}_3$ which do not occur in $d \,\square\, \tilde{D}$, also do not occur in the derivations considered so far. So, by Claim 7.2, we can put together ζ_3 and ζ_2, and obtain the derivation

$$\zeta_4 : (H = A) \wedge d \,\square\, \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b_3 \wedge b_2^* \,\square\, \tilde{B}_3, \tilde{B}_2^*.$$

Since in M we find the clause $cl : A \leftarrow c \,\square\, \tilde{C}, \tilde{E}$, by the definition of derivation there exists a derivation ξ which uses only clauses of M and which is similar to

$$true \,\square\, H \overset{M}{\leadsto} b_3 \wedge b_2^* \,\square\, \tilde{B}_3, \tilde{B}_2^*.$$

Since the variables that $b_3 \,\Box\, \tilde{B}_3$ has in common with the rest of this expression are certainly contained in $Var(A, \tilde{E})$, from (7.2), (7.3) and (7.5) it follows that $\xi \simeq \xi'$ †. Hence the thesis. $\Box$

Combined with Theorem 7.2.3, this Theorem shows that, when its hypothesis are satisfied, for every module N such that $M \oplus N$ and $M' \oplus N$ are defined and for each refutation in $M' \oplus N$ there exists a similar refutation in $M \oplus N$. In other words, that the transformation has not added to the program any extra semantic information.

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

An immediate consequence of Theorem 7.2.5 is the following simple Corollary which characterizes the situations in which we have total correctness.

Corollary 7.2.7 Let $cl: \; A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi \rangle$, and $M' : \langle P', \pi \rangle$ be the result of replacing $c \,\Box\, \tilde{C}$ with $d \,\Box\, \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}\}$. If $c \,\Box\, \tilde{C}$ is $\mathcal{O}$-equivalent to $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M then

- $M \approx_{\mathcal{O}} M'$ iff $c \,\Box\, \tilde{C}$ is equivalent to $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M'.

Proof.

($\Rightarrow$). It is easy to see that if $c \,\Box\, \tilde{C}$ is $\mathcal{O}$-equivalent to $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M and $M \approx_{\mathcal{O}} M'$ then $c \,\Box\, \tilde{C}$ is also $\mathcal{O}$-equivalent to $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M'.

($\Leftarrow$). By Theorem 7.2.5 we have that each π-derivation in M' has a similar π-derivation in M'. Now M can be re-obtained from M' by replacing back $d \,\Box\, \tilde{D}$ by $c \,\Box\, \tilde{C}$. Since by hypothesis $c \,\Box\, \tilde{C}$ is also $\mathcal{O}$-equivalent to $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M', from Theorem 7.2.5 we also have that each π-derivation in M has a similar π-derivation in M', therefore, by Theorem 7.2.3 $M \approx_{\mathcal{O}} M'$. $\Box$

Roughly speaking, the previous Corollary states that the operation is operationally correct if the replacing and the replaced conjunctions are operationally equivalent both in the initial and the resulting program. Of course this result requires some knowledge of the the semantics of the resulting program and therefore cannot be used as an applicability condition for the replacement operation: for that purpose we want conditions which are based solely on the semantic properties of the *initial* program. To this is devoted the rest of this section.

Total correctness

When we replace $c \,\Box\, \tilde{C}$ by $d \,\Box\, \tilde{D}$ in the clause $cl: \; A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$, the equivalence of $c \,\Box\, \tilde{C}$ and $d \,\Box\, \tilde{D}$ under $Var(A, \tilde{E})$ in M is not sufficient to guarantee *total* correctness, as there may be computations which can be done in the original module M, but not in the transformed on M'. In fact, when $\tilde{D}$ depends on the modified clause the replacement can introduce a loop thus affecting the total correctness. This is shown by the following classical counter-example.

Example 7.2.8 Let $\langle P, \emptyset\rangle$ be the module consisting of the following clauses.

```
cl:   q ←r.
      r.
```

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

```
cl´:  q ←q.
      r.
```

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

Now we propose two methods for guaranteeing that no "fatal" loops are introduced. These methods formalize the requirement (ii) we mentioned in the introduction. The first one is the most complex but in our opinion is also the most useful for program's optimization. It is based on the following Definition.

Definition 7.2.9 (Not Slower) Let $M = \langle P, \pi\rangle$ be a module, $c_1 \,\Box\, \tilde{C}_1$ and $c_2 \,\Box\, \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

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

iff for each π-derivation $\xi_1 :\ c_1 \,\Box\, \tilde{C}_1 \overset{P}{\leadsto} b_1 \,\Box\, \tilde{B}_1$, renamed apart wrt $\tilde{x}$, there exists a derivation $\xi_2 :\ c_2 \,\Box\, \tilde{C}_2 \overset{P}{\leadsto} b_2 \,\Box\, \tilde{B}_2$, renamed apart wrt $\tilde{x}$ such that $|\xi_2| \leq |\xi_1|$ and that $q(\tilde{x}) \leftarrow b_1 \,\Box\, \tilde{B}_1 \ \simeq\ q(\tilde{x}) \leftarrow b_2 \,\Box\, \tilde{B}_2$, where q is any (dummy) predicate symbol[4].
□

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

Theorem 7.2.10 (Correctness I) Let $cl :\ A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$ be a clause in the module $M : \langle P, \pi\rangle$ and $M' : \langle P', \pi\rangle$ be the result of replacing $c \,\Box\, \tilde{C}$ by $d \,\Box\, \tilde{D}$ in cl. So $P' = P\backslash\{cl\} \cup \{cl' : A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}\}$. If

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

then $M \approx_{\mathcal{O}} M'$.

Proof. For practical reasons, we now divide the proof in two parts: the first one is the counterpart of the first part of the proof of Theorem 7.2.5, and will also be referred to in the proof of Theorem 4.7.
Part 1. By Theorem 7.2.5 it follows that each π-derivation ξ' in M' there is a derivation ξ in M such that $\xi' \simeq \xi^{\dagger}$, therefore, by Theorem 7.2.3, in order to prove the thesis we have to show that also the converse holds, that is, that for each π-derivation ξ in M there is a derivation ξ' in M' such that $\xi \simeq \xi'^{\dagger}$. With no further

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

effort we'll show that in this situation we can always find a ξ' such that $|\xi| \geq |\xi'|$. This will be used to prove Corollary 7.2.12.

We proceed by induction on the length of the derivation. Let ξ be a π-derivation in M.

Base case $|\xi| = 0$. This case is trivial, as the derivations of length zero are the ones of the form $b \,\Box\, \tilde{B} \overset{M}{\leadsto} b \,\Box\, \tilde{B}$.

Inductive step. By Claims 7.1 and 7.2, ξ can be chosen of the form

$$\xi : \ true \,\Box\, H \overset{M}{\leadsto} b \,\Box\, \tilde{B}$$

where $\tilde{B}$ contains only π-atoms, and where (since this derivation has length greater than 0) we can assume that $Var(H) \cap Var(\tilde{B}) = \emptyset$. By the definition of derivation, there has to exist a (renaming of a) clause of M,

$$J \leftarrow c_L \,\Box\, \tilde{L} \tag{7.6}$$

and a π-derivation

$$\zeta : \ (H = J) \wedge c_L \,\Box\, \tilde{L} \overset{M}{\leadsto} b \,\Box\, \tilde{B}$$

where $|\xi| = |\zeta| + 1$. By the inductive hypothesis, there exists a derivation ζ' in M' such that $\zeta \simeq \zeta'^{\,\dagger}$ and that $|\zeta| \geq |\zeta'|$. Now, if the clause of (7.6) was also a clause of M' (that is, if it was not affected by the transformation), then there would exist a derivation ξ' in M' such that $\xi \simeq \xi'^{\,\dagger}$, and that $|\xi| \geq |\xi'|$ concluding the proof. So we have to consider the case in which $J \leftarrow c_L \,\Box\, \tilde{L} \in M \backslash M'$; in this situation, $J \leftarrow c_L \,\Box\, \tilde{L}$ is exactly (a variant of) the clause $cl : A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$. By appropriately renaming all the variables in the clauses and the derivations considered so far, we can assume that ζ is exactly the derivation

$$\zeta : \ (H = A) \wedge c \,\Box\, \tilde{C}, \tilde{E} \overset{M}{\leadsto} b \,\Box\, \tilde{B}.$$

By Claims 7.2, there exist two derivations ζ_1 and ζ_2 such that

$$\begin{gathered}
\zeta_1 : \ c \,\Box\, \tilde{C} \overset{M}{\leadsto} b_1 \,\Box\, \tilde{B}_1, \\
\zeta_2 : \ (H = A) \,\Box\, \tilde{E} \overset{M}{\leadsto} b_2 \,\Box\, \tilde{B}_2, \\
b \equiv b_1 \wedge b_2 \text{ and } \tilde{B} \equiv \tilde{B}_1, \tilde{B}_2, \\
|\zeta_1| + |\zeta_2| = |\zeta| = |\xi| - 1, \\
Var(b_1 \,\Box\, \tilde{B}_1) \cap Var(b_2 \,\Box\, \tilde{B}_2) \subseteq Var(c \,\Box\, \tilde{C}) \cap Var((H = A) \,\Box\, \tilde{E}).
\end{gathered} \tag{7.7}$$

Here, like in the proof of 7.2.5 we follow Assumption 7.2.6, so the variables that each new expression has in common with the ones previously mentioned are only the ones that are strictly necessary.

Part 2. So, by the fact that $d \,\Box\, \tilde{D}$ is equivalent to and not-slower than $c \,\Box\, \tilde{C}$ under $Var(A, \tilde{E})$ in M, it follows that there exists a derivation

$$\zeta_3 : \ d \,\Box\, \tilde{D} \overset{M}{\leadsto} b_3 \,\Box\, \tilde{B}_3$$

such that $|\zeta_3| \leq |\zeta_1|$, and that for any dummy predicate symbol q, if we let $\tilde{x} = Var(A, \tilde{E})$,

$$q(\tilde{x}) \leftarrow b_1 \,\Box\, \tilde{B}_1 \simeq q(\tilde{x}) \leftarrow b_3 \,\Box\, \tilde{B}_3 \tag{7.8}$$

Here there is no loss in generality in assuming that the variables of $b_3 \,\Box\, \tilde{B}_3$ which do not occur in $d \,\Box\, \tilde{D}$, also do not occur in the derivations considered so far. So, by Claims 7.2, we can put together ζ_3 and ζ_2, and obtain the derivation

$$\zeta_4 : \ (H = A) \wedge d \,\Box\, \tilde{D}, \tilde{E} \stackrel{M}{\leadsto} b_3 \wedge b_2 \,\Box\, \tilde{B}_3, \tilde{B}_2.$$

Here we obviously have that:

Observation 7.2.11 The variables that $b_3 \,\Box\, \tilde{B}_3$ has in common with the rest of this expression are certainly contained in $Var(A, \tilde{E})$.

Moreover,the following holds: $|\zeta_4| = |\zeta_3| + |\zeta_2| \leq |\zeta_1| + |\zeta_2| = |\zeta| = |\xi| - 1$. Therefore, by the inductive hypothesis, there exists a derivation $\zeta' : \ (H = A) \wedge d \,\Box\, \tilde{D}, \tilde{E} \stackrel{M'}{\leadsto} b'_3 \wedge b'_2 \,\Box\, \tilde{B}'_3, \tilde{B}'_2$ such that

$$\zeta_4 \simeq \zeta'^{\dagger} \text{ and } |\zeta_4| \geq |\zeta'| \tag{7.9}$$

Since in M' we find the clause $cl' : \ A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}$, by the definition of derivation there exists a derivation $\xi' : true \,\Box\, H \stackrel{M'}{\leadsto} b'_3 \wedge b'_2 \,\Box\, \tilde{B}_3,' \tilde{B}'_2$. From (7.7), Observation 7.2.11, (7.8), and (7.9) it follows that $\xi \simeq \xi'^{\dagger}$ and that $|\xi| \geq |\xi'|$. Hence the thesis. □

Note that that $d \,\Box\, \tilde{D}$ is (operationally) not-slower than $c \,\Box\, \tilde{C}$ in M if computing an answer for $d \,\Box\, \tilde{D}$ in M, under any $\oplus$-context, never requires more iterations that computing the corresponding answer for $c \,\Box\, \tilde{C}$. Clearly, this means that the definition of $d \,\Box\, \tilde{D}$ is at least as efficient as the one of $c \,\Box\, \tilde{C}$. Therefore, the requirement of the above theorem, namely that the replacing conjunction has to be not-slower than the replaced one, fits well in a context where transformation operations are intended to increase the performances of programs. Indeed, it is easy to show that, when the hypothesis of the above theorem are satisfied, then the resulting module is (computationally) at least as efficient as the initial one. This is the content of next Corollary.

Corollary 7.2.12 Let M and M' be modules. Suppose that M' was obtained from M by applying a replacement operation in which the conditions of theorem 7.2.10 were satisfied. Then for each π-derivation ξ in M there exists a similar π-derivation ξ' in M' such that ξ' is *not longer than* ξ.

Proof. It is included in the proof of Theorem 7.2.10. □

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

Definition 7.2.13 (Dependency) Let P be a program, p and q be relations. We say that p *refers to* q in P iff there is a clause in P with p in the head and q in the body. We say that p *depends on* q in P iff (p, q) is in the reflexive and transitive closure of the relation *refers to*. $\Box$

We can now state our second result on total correctness.

Theorem 7.2.14 (Correctness II) Let $cl : A \leftarrow c \Box \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi \rangle$, and $M' : \langle P', \pi \rangle$ be the result of replacing $c \Box \tilde{C}$ by $d \Box \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \Box \tilde{D}, \tilde{E}\}$. If

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

then $M \approx_{\mathcal{O}} M'$. $\Box$

Proof. The first part of the proof is identical to **Part 1** of the proof of Theorem 4.3, so we just refer to it, and proceed with the second part.

Part 2b. So, by the fact that $d \Box \tilde{D}$ is equivalent to $c \Box \tilde{C}$ under $Var(A, \tilde{E})$ in M, It follows that there exists a derivation

$$\zeta_3 : \ d \Box \tilde{D} \overset{M}{\leadsto} b_3 \Box \tilde{B}_3$$

such that for any dummy predicate symbol q, if we let $\tilde{x} = Var(A, \tilde{E})$,

$$q(\tilde{x}) \leftarrow b_1 \Box \tilde{B}_1 \simeq q(\tilde{x}) \leftarrow b_3 \Box \tilde{B}_3. \tag{7.10}$$

Since the atoms in $d \Box \tilde{D}$ are independent from cl, the clauses used in ζ_3 are also clauses of M', so in M' there exists a derivation ζ_3', which is identical to ζ_3, $\zeta_3' : d \Box \tilde{D} \overset{M'}{\leadsto} b_3 \Box \tilde{B}_3$. Moreover, since $|\zeta_2| < |\xi|$, by the inductive hypothesis there exists a derivation ζ_2' such that

$$\begin{aligned} \zeta_2' : \ (H = A) \Box \tilde{E} \overset{M'}{\leadsto} b_2' \Box \tilde{B}_2', \\ \zeta_2 \simeq \zeta_2'{}^{\dagger}. \end{aligned} \tag{7.11}$$

By Claim 7.2 we can put together ζ_2' and ζ_3' and obtain the derivation

$$\zeta_4' : \ (H = A) \wedge d \Box \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b_3 \wedge b_2' \Box \tilde{B}_3, \tilde{B}_2'.$$

Since in M' we find the clause $cl' : \ A \leftarrow d \Box \tilde{D}, \tilde{E}$, by the definition of derivation there exists a derivation ξ' which uses only clauses of M' and which is similar to

$$true \Box H \overset{M'}{\leadsto} b_3 \wedge b_2' \Box \tilde{B}_2', \tilde{B}_3.$$

Since the variables that $b_3 \Box \tilde{B}_3$ has in common with the rest of this expression are certainly contained in $Var(A, \tilde{E})$, from (7.7), (7.10) and (7.11) it follows that $\xi \simeq \xi'^{\dagger}$. Hence the thesis. $\Box$

7.3 An Example

In this section we show what kind of optimizations can be achieved via replacement through a worked example. In particular, we'll show that, under the given applicability conditions, replacement allow us to introduce recursion in the definition of predicates. For this we employ a transformation strategy which is typically used in unfold/fold systems such as the one in [96]. Indeed, the applicability conditions we will give are general enough to let replacement mimic most of the transformations feasible with the tools of [96]. One advantage of replacement over folding is that the applicability conditions for the former refer solely to the (semantic) properties of the program we are working on, while for folding these depend also on the history of the transformation (that is, on the transformation steps previously performed). In any case, to the replacement operation there is much more than just mimicking the folding one, since the replacing and the replaced conjunction can be totally independent from each other.

The following example is a simplified version of the one used in chapter 6.

Example 7.3.1 (Computing an average) Consider the following CLP($\Re$)[5] program `AVERAGE` computing the average of the values in a list. Values may be given in different currencies, for this reason each element of the list contains a term of the form $\langle$`Currency, Amount`$\rangle$. The applicable exchange rates may be found by calling the predicate `exchange_rates`, which will return a list containing terms of the form $\langle$`Currency, Exchange_Rate`$\rangle$, where `Exchange_Rate` is the exchange rate relative to `Currency`. As we already mentioned in chapter 6, despite its simplicity, this is a typical program that can be used in a modular context. Indeed, if we consider that the exchange rates between currencies are typically fluctuating ratios, it comes natural to assume `exchange_rates` as an *open* (or imported) predicate, which may refer to some external information server to access always the most up-to-date information.

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

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

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

      weighted_sum([], 0).
      weighted_sum([ ⟨Currency, Amount⟩ | Ts], Rates, Sum) ←
```

[5] CLP($\Re$) [55] is the CLP language obtained by considering the constraint domain $\Re$ of arithmetic over the real numbers. The signature for $\Re$ contains the constant symbols 0 and 1, the binary function symbols $+$ and $*$, and the binary predicate symbols $+, <, \leq$ for constraints which are interpreted on the real numbers as usual.

```
    Sum = Amount*Value + Sum´ □
    member(⟨Currency, Value⟩, Rates),
    weighted_sum(Ts, Rates, Sum´).

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

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

Notice that the definition of `average` needs to scan the list `Xs` twice. This is a source of inefficiency that can be fixed via unfolding and replacement operations. The transformation strategy which we are going use use is often referred to as *tupling* [77] or as *procedural join* (see [62]). First, we introduce a new predicate `w_sum_and_len` defined by the following clause

```
c2:  w_sum_and_len(XS, RATES, SUM, LEN) ← □
         exchange_rates(RATES),
         weighted_sum(XS, RATES, SUM),
         len(XS, LEN).
```

`w_sum_and_len` reports the weighted sum of the values in `XS`, together with the length of `Xs` itself and the list of the exchange rates. Notice that `w_sum_and_len`, as it is now, needs to traverse the list `Xs` twice as well. We start to transform `AVERAGE` by unfolding both `weighted_sum(XS, RATES, SUM)` and `len(XS, LEN)` in the body of `c2`. This operations yield the module $\mathtt{AV}_1$ which contains the following two clauses:

```
c3:  w_sum_and_len([], Rates, 0, 0) ← □ exchange_rates(Rates).
c4:  w_sum_and_len([⟨Currency,Amount⟩|Rest], Rates, Sum, Len) ←
         Len = Len´+1  ∧ Sum = Amount*Value+Sum´ □
         exchange_rates(Rates),
         member(⟨Currency, Value⟩, Rates),
         weighted_sum(Rest, Rates, Sum´),
         len(Rest, Len´).
```

From the correctness of the unfolding operation it follows that $\mathtt{AVERAGE} \approx \mathtt{AV}_1$.

Now, we can *replace* `exchange_rates(Rates), weighted_sum(Rest, Rates, Sum´), len(Rest, Len´)` by `w_sum_and_len(Rest, Rates, Sum´, Len´)` in the body of `c4`. In the resulting module $\mathtt{AV}_2$, after cleaning up the constraints[6][7], the predicate `w_sum_and_len` is defined by the following clauses:

```
c3:  w_sum_and_len([], Rates, 0, 0) ← □ exchange_rates(Rates).
c5:  w_sum_and_len([⟨Currency,Amount⟩|Rest], Rates, Sum, Len) ←
```

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

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

```
Len = Len´+1  ∧ Sum = Amount*Value+Sum´  □
w_sum_and_len(Rest, Rates, Sum´, Len´),
member(⟨Currency, Value⟩, Rates).
```

Notice that, because of this last operation, the definition of `w_sum_and_len` is now recursive and it needs to traverse the list only once. Indeed, this operation constitutes the crucial optimization step. We now show that the applicability conditions of Theorem 7.2.10 were satisfied, and therefore that $\mathtt{AV}_2 \approx_{\mathcal{O}} \mathtt{AV}_1$. For this we use the following proposition.

Proposition 7.3.2 Let $cl: \ H \leftarrow b \,\Box\, \tilde{B}$ be the unique clause which defines $Pred(H)$ in the module $M : \langle P, \pi\rangle$ and assume $Pred(H) \notin \pi$. Then $true \,\Box\, H$ is operationally equivalent to $b \ \Box\, \tilde{B}$ under $Var(H)$ in M.
Moreover, if $M' : \langle P', \pi\rangle$ is the module obtained by unfolding some atoms $A_1, \ldots, A_n$ in the body of cl such that $Pred(A_i) \notin \pi$ for all $i \in [1, m]$, then $true \,\Box\, H$ is operationally not-slower than $b \ \Box\, \tilde{B}$ under $Var(H)$ in M'.

Proof. The first part is obvious. For the second one we prove the case in which only one atom A is unfolded in the body of cl. The generalization to n atoms is immediate. We first need the following.

Claim 7.3 Let cl, P, P' and A be defined as above and let $e \ \Box\, \tilde{E}$ be a generic query. Then, for any derivation $\xi : \ e \ \Box\, \tilde{E} \overset{P}{\leadsto} d \,\Box\, \tilde{D}$ such that $\tilde{D}$ does not contain any renamed version of the atom A, there exists a derivation $\xi' : \ e \ \Box\, \tilde{E} \overset{P'}{\leadsto} d' \,\Box\, \tilde{D}'$ such that ξ and ξ' are similar and $|\xi'| \leq |\xi|$. Moreover, if (a renamed version of) clause cl is used in ξ, then $|\xi'| < |\xi|$.

Proof. To simplify the notation in the following we will denote by A and cl also any renamed version of the atom A and of the clause cl, respectively. We also assume that $\tilde{B}$ (the body of cl) has the form $A, \tilde{G}$. The proof is by induction on the number of times h that cl is used in the derivation ξ.

For the base case $h = 0$ the thesis holds immediately, since P' differs from P only in the fact that the clause cl has been replaced for its unfolded versions.

For the inductive case $h > 0$ first observe that any occurrence of A in the derivation ξ will eventually be rewritten by using a clause in P, since $\tilde{D}$ does not contain the atom A. Moreover, we can assume without loss of generality that the selection rule used in ξ is such that as soon as A appears in the derivation A is immediately selected. In fact, to prove the claim clearly we can consider derivations up to $\simeq$, i.e. we can identify similar derivations. Since conjunction of constraints is associative and commutative, it is immediate to see that changing the selection rule of ξ into the one assumed before does not affect $\simeq$ equivalence. For the same reason we can also assume that the bodies of clauses are suitably reordered.

According to these assumptions ξ has the form

$$a \ \Box\, \tilde{A} \overset{P}{\leadsto} c \,\Box\, \tilde{C}, H' \overset{P}{\leadsto} c \wedge (H = H)' \wedge b \,\Box\, \tilde{C}, A, \tilde{G} \overset{P}{\leadsto} d' \,\Box\, \tilde{C}, \tilde{K}, \tilde{G} \overset{P}{\leadsto} d \,\Box\, \tilde{D}$$

where $d' \equiv (c \ \wedge \ (H = H') \ \wedge \ b \ \wedge \ (A = A') \ \wedge \ k)$, a renamed version of the clause $A' \leftarrow k \ \Box \ \tilde{K}$ defines $Pred(A)$ in P and the clause cl is not used in the derivation $d' \ \Box \ \tilde{C}, \tilde{K}, \tilde{G} \overset{P}{\leadsto} d \ \Box \ \tilde{D}$.

By inductive hypothesis there exists a derivation ξ'_1 in P' which is similar to $\xi_1 : \ a \ \Box \ \tilde{A} \overset{P}{\leadsto} c \ \Box \ \tilde{C}, H$ and such that $|\xi'_1| \leq |\xi_1|$. By definition of unfolding in P' we find the (renamed version of the) clause $H \leftarrow b \wedge (A = A') \ \Box \ \tilde{K}, G$. Therefore, by Definition 7.2.1, there exists a derivation ξ'_2 in P' which is similar to $\xi'_2 : \ a \ \Box \ \tilde{A} \overset{P}{\leadsto} d' \ \Box \ \tilde{C}, \tilde{K}, \tilde{G}$ and such that $|\xi'_2| < |\xi_2|$. Since the clause cl is not used in $d' \ \Box \ \tilde{C}, \tilde{K}, \tilde{G} \overset{P}{\leadsto} d \ \Box \ \tilde{D}$, we can conclude that there exists a derivation ξ' in P' which is similar to ξ and such that $|\xi'| < |\xi|$, thus completing the proof of the Claim. □

To prove the Proposition consider now a generic π-derivation $b \ \Box \ \tilde{B} \overset{P}{\leadsto} c \ \Box \ \tilde{C}$. Since in P we find the clause $cl : \ H \leftarrow b \ \Box \ \tilde{B}$, clearly there exists also a π-derivation $\xi : \ true \ \Box \ \tilde{H} \overset{P}{\leadsto} c' \ \Box \ \tilde{C}'$ such that

$$q(\tilde{x}) \leftarrow c \ \Box \ \tilde{C} \ \simeq \ q(\tilde{x}) \leftarrow c' \ \Box \ \tilde{C}' \tag{7.12}$$

where $\tilde{x} = Var(H)$ and q is any (dummy) predicate symbol.

Note that in the derivation ξ the clause cl is used at least once, since it is the only clause defining $Pred(H)$ in P. Moreover the hypothesis $Pred(A) \notin \pi$ and the definition of π-derivation imply that $\tilde{C}'$ does not contain any renamed version of the atom A. Therefore we can apply previous Claim thus obtaining that there exists a derivation ξ' in P' which is similar to ξ and such that $|\xi'| < |\xi|$. This, together with (7.12), Definition 7.2.1 and Definition 7.2.9 completes the proof. □

Because of the above Proposition, denoting by c_4 the constraint which appear in the clause `c4`, we have that c_4 □ `w_sum_and_len(Rest,Rates,Sum´,Len´)` is $\mathcal{O}$-equivalent to and $\mathcal{O}$-not-slower than c_4 □ `exchange_rates(Rates), weighted_sum(Rest, Rates, Sum´), len(Rest, Len´)` y under { `Currency, Amount,Rest,Rates, Sum, Len` } in $\texttt{AV}_1$. Therefore the conditions of Theorem 7.2.10 are satisfied and `AVERAGE` $\approx_{\mathcal{O}}$ $\texttt{AV}_2$ holds. More generally, Proposition 7.3.2 shows also that the applicability conditions given in Theorem 7.2.10 allow the replacement to mimic, to a large extent, the unfold/fold transformation as defined in [96].

Finally, in order to let also the definition of `average` enjoy of these improvements, we simply replace `exchange_rates(Rates), weighted_sum(Xs, Rates, Sum), len(Xs, Len)` by `w_sum_and_len(Xs, Rates, Sum, Len)` in the body of `c1`. After the cleaning-up the resulting clause is

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

So, we have obtained the module $\texttt{AV}_3$, consisting of the clauses `c6`, `c3` and `c5`, where we find a definition of `average` which needs to scan the list only once. The correctness of this last transformation step, i.e. the compositional equivalence of $\texttt{AV}_3$ with $\texttt{AV}_2$ (and consequently also with the original module `AVERAGE`), can be easily proven using Theorem 7.2.14 as follows. As before, because of Proposition 7.3.2 we have that `exchange_rates(Rates), weighted_sum(Rest, Rates, Sum´), len(Rest, Len´)` is $\mathcal{O}$-equivalent to `w_sum_and_len(Rest, Rates, Sum´, Len´)` under { `Rest, Rates, Sum´, Len´` } in $\mathtt{AV}_1$. This equivalence holds also in $\mathtt{AV}_2$, since the correctness of the first replacement implies $\mathtt{AV}_1 \approx_{\mathcal{O}} \mathtt{AV}_2$. From this it follows that $\mathtt{c}_1 \;\Box$ `exchange_rates(Rates), weighted_sum(Xs, Rates, Sum), len(Xs, Len)` is $\mathcal{O}$-equivalent to $\mathtt{c}_1 \;\Box$ `w_sum_and_len(List, Rates, Sum, Len)` under {`List, Av`}. Moreover, `w_sum_and_len` does not depend on clause `c1` in $\mathtt{AV}_2$. Therefore, from Theorem 7.2.14 it follows that $\mathtt{AV}_3 \approx_{\mathcal{O}} \mathtt{AV}_2$, and therefore, from the correctness of the previous transformation steps, that `AVERAGE` $\approx_{\mathcal{O}} \mathtt{AV}_3$, i.e. that the whole transformation is correct. $\Box$

7.4 Correctness wrt other congruences

In some cases one can be interested in preserving other kind of properties of modules rather than their answer constraints. Indeed in the literature, together with the answer constraint semantics [43], we find two other semantics for CLP without negation. One is the so-called $\mathcal{C}$-semantics which was defined for pure logic programs [29, 39] and then adapted to CLP (specifically for program's transformation) in [14] by using an operational definition. The $\mathcal{C}$-semantics characterizes the most general answer constraints of a CLP program. The second, and more notable one, is the least model semantics (on the relevant algebraic structure $\mathcal{D}$) [51]. This semantics is the CLP counterpart of the least Herbrand model and it is commonly considered the standard declarative semantics for CLP.

In this Section we consider the congruences induced by these two semantics. We show that we can easily adapt to both the contexts the applicability conditions used in Theorems 7.2.10 and 7.2.14. Moreover, since these congruences are weaker than the operational one, the resulting applicability conditions are weaker than the previous ones, thus allowing more optimizations on the modules.

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

Definition 7.4.1 Let P, P' be two programs, $\xi : \; c \,\Box\, \tilde{C} \overset{P}{\leadsto} b \,\Box\, \tilde{B}$ and $\xi' : \; c \,\Box\, \tilde{C} \overset{P'}{\leadsto} b' \,\Box\, \tilde{B}'$ be two derivations starting in the same goal, let also $\tilde{x} = Var(c \,\Box\, \tilde{C})$. We say that

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

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

Notice that $\mathcal{D} \models \; \exists_{-\tilde{x}}\, b \,\Box\, \tilde{B} \;\rightarrow\; \exists_{-\tilde{x}}\, b' \,\Box\, \tilde{B}'$ holds iff, for each solution θ of b, there exists a solution θ' of b' such that θ and θ' agree on the variables $\tilde{x}$ and each element in the conjunction $\tilde{B}'\theta'$ is also an element of the conjunction $\tilde{B}\theta$. It is also worth noticing that $\preceq$ does not represent "one side" of $\simeq$, since we can have that $\xi \preceq \xi'$, $\xi' \preceq \xi$ and still $\xi \not\simeq \xi'$.

This is due to the fact that in the definition of $\simeq$ the goals have to be considered as multisets, while here considering them as sets is sufficient. For instance, this is the case when we consider the derivations $\xi : \ p(x) \rightsquigarrow x = y \,\Box\, q(y), q(y).$ and $\xi' : \ p(x) \rightsquigarrow x = y \,\Box\, q(y).$

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

Definition 7.4.2 ($\mathcal{C}$- and $\mathcal{M}$-congruence) Let M_1 and M_2 be CLP modules that have the same set of open predicates. We say that

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

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

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

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

The operational congruence is stronger than the $\mathcal{C}$-congruence, which in turn is stronger than the $\mathcal{M}$-congruence. This will be formally proved in the sequel. To clarify the difference among the three kind of relations let us consider the following simple modules where we assume the set of open atoms to be empty.

M_1 :	M_2 :	M_3 :
`p(X).`	`p(X).`	`p(X)` $\leftarrow$ `X = Y+1` $\Box$ `p(Y).`
	`p(0).`	`p(0).`

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

Note 7.4.3 For the reader familiar with the original definition of the $\mathcal{C}$-semantics [29] some explanations are in order here. The $\mathcal{C}$-semantics of a pure logic program P is defined indifferently as

(a) the set of atomic logical consequences of P, or
(b) the set of *most general* answers computed by P.

It is also proven ([68]) that, if the underlying language is infinite, then two pure logic programs have the same $\mathcal{C}$ semantics iff they have the same least Herbrand model.

Now, the CLP counterpart of the $\mathcal{C}$-semantics is defined in [14] just as the counterpart of (b) above. The fact is that, for CLP programs the statements (a) and (b) are not equivalent to each other. This is shown for example by the programs

```
p(X) ← X = a  ∨ X = b.
```

and

```
p(X) ← X = a.
p(X) ← X = b.
```

Moreover, since in the CLP context we need the domain $\mathcal{D}$ for evaluating the constraint, it makes little sense talking about the *logical consequences of* P (which are the formulae ϕ such that $P \models \phi$). On the other hand, it is meaningful talk about the logical consequences of P "*under* $\mathcal{D}$", by this we mean the set of formulae ϕ such that $\mathcal{D} \models P \rightarrow \phi$. Now, since the domain of $\mathcal{D}$ determines the universe of our interpretations and models, we have that two CLP programs have the same "set of atomic[8] logical consequences under $\mathcal{D}$" iff they have the same least $\mathcal{D}$-model, but this does not imply that they have the same most general answers. Indeed, if we consider the programs in M_1 and M_3 above, we have that, if $\mathcal{D}$ is the usual additive structure on the set of natural numbers, M_1 and M_3 (seen as programs) have the same least $\mathcal{D}$ models, therefore the same set of logical consequences "*under* $\mathcal{D}$", but they do not have the same set of most general answers. Notice that this is the case even though our structure contains the infinite set of constants corresponding to the natural numbers. $\square$

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

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

In this Subsection we provide the applicability conditions for the replacement operation in the case we refer to the $\mathcal{C}$-congruence. More precisely, we are going to reformulate appropriately Theorems 7.2.10 and 7.2.14. This provides a generalization of the result on the correctness of the replacement operation given in [14].

We start with a Theorem which gives a condition sufficient to guarantee that two modules are $\mathcal{C}$-congruent, thus providing a $\mathcal{C}$-counterpart of Theorem 7.2.3. Its proof can easily be obtained from the one of Theorem 7.2.3 and thus it is omitted.

Theorem 7.4.4 Let $M_1 = \langle P_1, \pi \rangle$ and $M_2 = \langle P_2, \pi \rangle$ be two modules. If, for each π-derivation ξ_i in M_i there exists a π-derivation ξ_j in M_j such that $\xi_i \preceq \xi_j$ ($i, j \in [1, 2]$, $i \neq j$), then $M_1 \approx_{\mathcal{C}} M_2$. $\square$

This result also shows that the $\mathcal{C}$-congruence is strictly weaker that the operational one. Now, in order to provide the $\mathcal{C}$-version of the applicability conditions for the replacement operation, we restate the Definitions 7.2.4 and 7.2.9 to adapt them to the new context.

Definition 7.4.5 Let $M = \langle P, \pi \rangle$ be a module, $c_1 \,\square\, \tilde{C}_1$ and $c_2 \,\square\, \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

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

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

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

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

iff for each π-derivation ξ_1 : $c_1 \,\square\, \tilde{C}_1 \overset{P}{\leadsto} b_1 \,\square\, \tilde{B}_1$ there exists a π-derivation ξ_2 : $c_2 \,\square\, \tilde{C}_2 \overset{P}{\leadsto} b_2 \,\square\, \tilde{B}_2$ such that $|\xi_2| \leq |\xi_1|$ and $\mathcal{D} \models \exists_{-\tilde{x}}\, b_1 \,\square\, \tilde{B}_1 \rightarrow \exists_{-\tilde{x}}\, b_2 \,\square\, \tilde{B}_2$.

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

It is easy to see that the concepts of $\mathcal{C}$-equivalence and of $\mathcal{C}$-not-slower are weaker than their operational counterparts given in Definitions 7.2.4 and 7.2.9. Intuitively, the difference in terms of derivations lies in the fact that for the former we want a one-to-one correspondence between all the partial derivations ending with open atoms, while the latter requires this one-to-one correspondence to hold only for the "most general" ones. Now when we refer to the $\mathcal{C}$-congruence we can weaken the hypothesis of Theorems 7.2.10 and 7.2.14 by replacing the concepts of *equivalent* and *not-slower* by their $\mathcal{C}$-counterparts. Namely, we have the following.

Theorem 7.4.6 ($\mathcal{C}$-correctness) Let cl : $A \leftarrow c \,\square\, \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi \rangle$, and $M' : \langle P', \pi \rangle$ be the result of replacing $c \,\square\, \tilde{C}$ by $d \,\square\, \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \,\square\, \tilde{D}, \tilde{E}\}$. If

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

then $M \approx_{\mathcal{C}} M'$ $\square$

Proof. We now show that: (a) for each π-derivation ξ' in M' there is a derivation ξ in M such that $\xi' \preceq \xi$ and that (b) (the vice-versa) for each π-derivation ξ in M there is a derivation ξ' in M' such that $\xi \preceq \xi'$. From Theorem 7.4.4 this will imply the thesis.

Actually, the proof is almost identical to a combination of the proofs of Theorems 7.2.5, 4.3 and 4.7. So it is much more convenient if we just show how these have to be modified in order to adapt them to the context of the $\mathcal{C}$-congruence.

Part (a). In order to show that for each derivation ξ' in M' there is a derivation ξ in M such that $\xi' \preceq \xi$ it is sufficient to apply the following syntactic changes to the proof of Theorem 7.2.5:

- In each equation labeled by the † sign, we replace the $\simeq$ operator with $\succeq$ (where, obviously, we define $\xi \succeq \xi'$ iff $\xi' \preceq \xi$).
- The equation (7.5) has to be replaced by $\mathcal{D} \models \exists_{-\tilde{x}}\, b_1 \,\square\, \tilde{B}_1 \rightarrow \exists_{-\tilde{x}}\, b_3 \,\square\, \tilde{B}_3$.

Part (b). In order to show that for each derivation ξ in M there is a derivation ξ' in M' such that $\xi \preceq \xi'$ it is sufficient to combine together the proofs of Theorems 4.3 and 4.7 and apply the following syntactic changes:

- In each equation labeled by the † sign, replace the $\simeq$ operator with $\preceq$.

- The equations (7.8) and (7.10) have to be replaced by: $\mathcal{D} \models \exists_{-\tilde{x}}\, b_1 \,\Box\, \tilde{B}_1 \;\rightarrow\; \exists_{-\tilde{x}}\, b_3 \,\Box\, \tilde{B}_3$ $\Box$

This result can also be seen as a generalization of Proposition 4.6 in [14]. In fact, it is easy to check that when the hypothesis of that proposition are satisfied then the replacing and the replaced conjunction are always $\mathcal{C}$-equivalent to each other and that the replacing conjunction is always not-slower than the replaced one (under an appropriate set of variables).

The applicability conditions in the previous Theorem are weaker than the ones in Theorems 7.2.10 and 7.2.14. This reflects the fact that some replacement operations which are correct wrt $\mathcal{C}$ congruence may not be so wrt the operational one. A typical example of a replacement operation which always satisfies the hypothesis of Theorem 7.4.6, but which is possibly not operationally correct, and therefore does not satisfy the hypothesis of Theorems 7.2.10 and 7.2.14, is the elimination of duplicate atoms in the body of a clause. Indeed, consider a program M consisting the following clause

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

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

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

In this subsection we give the $\mathcal{M}$-counterpart of the results stated in the previous one. We formulate (and prove correct) the applicability conditions for the replacement operation in case we want to preserve the $\mathcal{M}$-congruence.

As we mentioned before, the $\mathcal{M}$-congruence is strictly weaker then the $\mathcal{C}$-congruence. Indeed, we have already seen that two modules which are $\mathcal{M}$-congruent do not need to be $\mathcal{C}$-congruent (consider previous programs M_1 and M_3). For the other implication we have the following result, whose proof is given in the Appendix.

Proposition 7.4.7 If two modules are $\mathcal{C}$-congruent then they are also $\mathcal{M}$-congruent. $\Box$

When considering the $\mathcal{M}$-congruence we can further weaken the applicability conditions for the replacement operation by defining the notions of $\mathcal{M}$-equivalent and of $\mathcal{M}$-not-slower as follows.

Definition 7.4.8 Let $M = \langle P, \pi \rangle$ be a module, $c_1 \,\Box\, \tilde{C}_1$ and $c_2 \,\Box\, \tilde{C}_2$ be two queries and $\tilde{x}$ be a tuple of variables. Then we say that

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

iff for each π-derivation $c_i \,\Box\, \tilde{C}_i \overset{P}{\leadsto} b_i \,\Box\, \tilde{B}_i$ and each solution ϑ_i of b_i, there exists a derivation $c_j \,\Box\, \tilde{C}_j \overset{P}{\leadsto} b_j \,\Box\, \tilde{B}_j$ and a solution ϑ_j of b_j such that $\mathcal{D} \models \tilde{B}_i\vartheta_i \;\rightarrow\; \tilde{B}_j\vartheta_j$ and $\tilde{x}\vartheta_1 = \tilde{x}\vartheta_2$ $(i, j \in [1, 2], i \neq j)$.
Moreover, we say that

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

iff for each π-derivation $\xi_1 : \; c_1 \,\Box\, \tilde{C}_1 \overset{P}{\leadsto} b_1 \,\Box\, \tilde{B}_1$ and for each solution ϑ_1 of b_1, there exists a derivation $\xi_2 : \; c_2 \,\Box\, \tilde{C}_2 \overset{P}{\leadsto} b_2 \,\Box\, \tilde{B}_2$ and a solution ϑ_2 of b_2 such that $|\xi_2| \leq |\xi_1|$, $\mathcal{D} \models \tilde{B}_1\vartheta_1 \;\rightarrow\; \tilde{B}_2\vartheta_2$ and $\tilde{x}\vartheta_1 = \tilde{x}\vartheta_2$.
Again, all the considered derivations here considered are supposed to be renamed apart wrt $\tilde{x}$. □

From this definition it follows immediately that the $\mathcal{M}$-equivalence is the weakest of the three equivalences we have introduced, as it checks only the "ground" derivations. Theorem 7.4.6 can now be restated for the case of $\mathcal{M}$-congruence as follows.

Theorem 7.4.9 ($\mathcal{M}$-correctness) Let $cl : \; A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi\rangle$, and $M' : \langle P', \pi\rangle$ be the result of replacing $c \,\Box\, \tilde{C}$ by $d \,\Box\, \tilde{D}$ in cl. So $P' = P\backslash\{cl\} \cup \{cl' : A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}\}$. If

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

then $M \approx_{\mathcal{M}} M'$.

Proof. See Appendix □

7.4.3 The non-modular case

We discuss now how the previous results can be applied to the non-modular case, that is when programs are considered as stand-alone units. In this case, since we do not have to consider $\oplus$-contexts, the notion of correctness for the replacement operation is defined wrt the following equivalences.

Definition 7.4.10 Let P_1 and P_2 be CLP programs. We say that P_1 and P_2 are

- *operationally equivalent* iff for each refutation in P_1 there exists a similar refutation in P_2 and vice-versa,
- $\mathcal{C}$-equivalent iff for each refutation in P_1 there exists a more general refutation in P_2 and vice-versa,
- $\mathcal{M}$-equivalent iff P_1 and P_2 have the same least $\mathcal{D}$-model.

Here, the use of the term *equivalence*, rather than *congruence* reflects the fact that we are not considering modules, but (stand-alone) programs.

According to the above definition, we say that the replacement operation on CLP programs is operationally ($\mathcal{C}$-, $\mathcal{M}$-) *correct* iff it maps programs into operationally ($\mathcal{C}$-, $\mathcal{M}$-) equivalent ones.

From previous definition it follows immediately that the non-modular case can be naturally regarded as a particular instance of the modular one. In fact, if we assume that the set of open predicates is empty, then the concepts of equivalence and congruence coincide. Moreover, according to Definition 6.3.2 if $\pi = \emptyset$, then composition is allowed only between predicate disjoint modules, and, semantically, this is like allowing no composition at all. Therefore the correctness results in the non-modular case can be obtained by just setting $\pi = \emptyset$ in Theorems 7.2.10, 7.2.14 and 7.4.6.

From the definitions it is also clear that the smaller is the set of open predicates, the weaker become the applicability conditions needed to ensure correctness of replacement, for all the three congruences considered. In particular, the applicability conditions for the non-modular case are quite weaker than the ones for the modular setting.

7.5 Related papers and conclusions

In this section we try to highlight the similarities and the differences between the approach we follow and the ones proposed in the literature.

Let us start by considering Maher's paper [69], which, to the best of our knowledge, is the only paper in the literature that deals with the replacement operation in the context of modular (constraint) logic programs. Firstly it should be mentioned that [69] takes into considerations also the unfold and the fold operations, which are beyond the scope of this chapter. Apart from that, the main difference between this chapter and [69] is that Maher takes into consideration *normal* programs (i.e. programs which contain negated atoms in the bodies of their clauses). Since the tools needed to handle normal programs are quite different and heavier than those sufficient to deal with definite programs, it follows that the techniques adopted to prove the correctness of the replacement operation are quite different as well, and comparison between the two articles are difficult. For instance, the applicability conditions of [69] guarantee the preservation of the Perfect Model Semantics [6, 81], which is incomparable to the semantics used here. It is of no surprise then that if we restrict our attention to definite programs, then our results extend those of [69]. In particular each time that the requirements of [69] are satisfied also the hypothesis of Theorem 7.4.9 are satisfied as well. This implies that [69] requires the replacing conjunction to be always independent from the modified clause (therefore forbidding the introduction of recursion via the replacement operation). Finally, another difference is due to the fact that we adopt a more flexible definition of modular program, which allows, for instance, mutual recursion among modules.

Apart from [69], in the literature we find only another paper which investigates the replacement operation for CLP: The one by Bensaou and Guessarian [14]. In [14] the authors provide applicability conditions for the replacement operation (and also for the operations of unfold and fold, which, we repeat ourselves, have been studied in Chapter 6 and are beyond the scope of this chapter) which guarantee the correctness of the operation wrt the $\mathcal{C}$-semantics. Of course, the main difference between the approach to the replacement operation given in this chapter and the one of [14] is that in [14] modularity is not an issue. In any case, the $\mathcal{C}$-correctness result in Theorem 7.4.6 provides us with a generalization of Proposition 4.6 in [14]: each time that the applicability conditions given in [14] are satisfied we can also apply the replacement. The converse is not true (even in the non-modular case). For instance the replacements performed in Example 7.3.1 are not feasible using the tools of [14].

In the Logic Programming Area

As we mentioned in the introduction, the replacement operation was introduced in the area of pure logic programs by Tamaki and Sato in [96]. Later, developments were provided by the works of Sato himself [88], Gardner and Shepherdson [47], Bossi, Cocco and Etalle [20], Proietti and Pettorossi [79, 80] and Cook and Gallagher [32]. The main improvement of this chapter over all the papers just mentioned is that we take into consideration modular programs. So, in the rest of this section we restrict our attention to *non-modular* programs, and we try, in this more restrictive case, to highlight the other main differences (and relations) between our approach and the other ones.

In [96] the replacement operation is part of an unfold/fold transformation system and the applicability conditions are devised in order to fit with the other two operations. Apart from this, the main differences between this chapter and [96] are due to the fact that the applicability conditions of [96] guarantee the correctness of the operation wrt the least Herbrand model semantics, while we also consider stronger semantics (the $\mathcal{C}$ and the operational semantics). Still there are some similarities between [96] and this chapter which are worth noticing. Namely, the applicability conditions given in [96] can also be seen as being based on two requirements:

(a) The replacing conjunction must be equivalent to the replaced one in $P\backslash\{cl\}$, where P and cl are respectively the modified program and clause. Unfortunately, as pointed out in [47], the fact of referring to $P\backslash\{cl\}$ rather than to P alone, leads to an error in the applicability conditions.
(b) for each proof for the replaced query there has to be a corresponding proof for the replacing one such that the *rank* of the latter is not greater than the rank of the former. Intuitively, the rank of a proof can be associated to the size of a proof tree. Of course this condition relates to (it actually inspires) the concept of not-slower query which is extensively used here.

Later, Sato in [88] considered replacement of tautologically equivalent formulas in the context of first-order programs. Being the context so different than the one considered here, [88] is practically unrelated to this chapter.

A more related paper is the one of Gardner and Shepherdson [47]. [47] deals also with the operations of unfold and fold in the context of normal program, however, the section on replacement is quite separate from the rest of the paper, as it deals with definite programs and refers to the $\mathcal{C}$-semantics. In fact the main result of [47] states that if the replacing conjunction is equivalent to the replaced one then for every computation feasible in the original program P there exists a more general computation feasible in the transformed program P' and vice-versa. The introduction of a loop is avoided by adopting a quite restrictive definition of equivalence: it is required that the most general answers to the replaced and the replacing queries are not affected by the presence or the absence of the modified clause cl in the program. In practice both queries have to be *semantically independent* from the modified clause. Therefore, for those programs (we hope the great majority) for which *semantic independence* coincides with *physical independence*[9] Theorem 7.4.6 provides a generalization of Theorem 5.1 in [47] in the following two ways: (a) it is not required that the replaced conjunction is independent from the (predicate in the head of the) replaced clause, and (b) it provides a condition (the one that uses the concept of being not-slower) that allows also the replacing conjunction to be dependent on the (predicate in the head of the) replaced clause, therefore allowing the introduction of recursion.

Going on with our small survey, we can now consider [20], which can be regarded as the ancestor of this chapter. In [20], Bossi et al. give some conditions sufficient to guarantee the correctness of the replacement operation wrt the operational semantics (of logic programs). Of course the main difference between this chapter and [20] is that in the latter only non-modular logic programs are considered. Apart from that there are other differences, namely

- [20] uses a quite more complicated yet more general method to prevent the introduction of a loop: the replacing conjunction may be dependent on the head of the replaced clause and still be slower than the replaced conjunction, as long as the difference in "speed" (the delay) is bounded by the *dependency degree* of the replacing conjunction on the head of the modified clause. In this sense the approach we follow here is slightly more restrictive. However, we believe that the gain in generality is not worth the loss in clarity. This applies in particular to this chapter, in which things are further complicated by the presence of modularity. Recall that, as we mentioned in the introduction, one of our main goals is to propose applicability conditions which are not "discouragingly complicated".
- A second difference is due to the fact that [20] referred to a bottom-up construction of the semantics. The top down method we adopted here is not only more intuitive, but it also more flexible. In particular the second part of Proposition 7.3.2 is not obtainable with the tools of [20].

The results of [20] have also been applied to normal programs in Chapter 4 of this thesis). These papers provide applicability conditions which guarantee the correctness of the operation wrt Fitting's and Kunen's semantics.

Other related papers are the ones of Proietti and Pettorossi [80], and Cook and

[9] Here we say that a a query is physically independent from a clause $A \leftarrow \tilde{B}$, if no predicate in the query depends on $Pred(A)$ in the sense of the Dependency Definition 7.2.13.

Gallagher [32].

In [80] it is proposed a method based on program's manipulation. The underlying idea is the following: suppose that we want to obtain the program P' from P by applying a replacement operation. To guarantee total correctness, we may manipulate (an augmented version of) P via the syntactic operations of unfolding and folding until we obtain a program Q which validates *syntactically* the operation. This guarantees that P' will have the same operational semantics of P. This method is clearly totally different (hence incomparable) from the one we propose.

Finally, Cook and Gallagher [32] present an approach to the replacement operation which is based on termination analysis. In addition to the usual condition that the replacing conjunction has to be equivalent to the replaced one, they avoid the introduction of a loop by simply requiring (a subprogram of) the resulting program to be *terminating* [5].

In the Functional Programming Area

Without pretending to be exhaustive, we want to mention a recent paper on the replacement operation for functional programs which, independently, follows substantially the same approach we do. In [86], Sands guarantees total correctness by requiring firstly the replacing expression to be equivalent to the replaced one and secondly by avoiding the introduction of a loop by

- requiring the replacing expression to be independent from the modified clause (corresponding to the method used in Theorem 7.2.14),
- or requiring the replacing expression to be an *improvement* over the replaced one. This clearly corresponds to the condition we give in Theorem 7.2.10. The underlying intuition given in [86] is that in this case, the evaluation of the replacing expression converges "faster" than one of the replaced one, consequently, all evaluations will converge faster in the transformed program than in the original one and, parallelly, no dangerous loop may be introduced.

Concluding remarks

We have investigated optimizations of CLP modules based on the replacement transformation. As discussed above, our results extend previous ones in the field of transformations for logic programs in that we have defined applicability conditions for replacement which guarantee that the original and the transformed module are semantically equivalent under any $\oplus$-context. These conditions have been instantiated to consider three different semantic notions. Moreover, also when restricting to the non-modular setting, we provide generalizations of previous results for replacement of CLP programs.

We believe that our setting is suitable as a theoretical basis to define tools for the optimization of CLP modules. In particular, the applicability conditions which allow one to obtain operationally congruent modules are the more natural for practical applications, since answer constraints are the standard results of CLP computations.

7.6 Appendix

In this Appendix we give the proofs of Proposition 7.4.7 and Theorem 7.4.9. The proof of the Theorem follows the guidelines of the one of Theorem 7.4.6. First we introduce an operational characterization of the $\mathcal{M}$-congruence. To this end we need the following.

Definition 7.6.1 Let π be a set of predicate symbols, $\xi:\ c_A \,\Box\, \tilde{A} \rightsquigarrow b \,\Box\, \tilde{B}$ be a π-derivation, and θ be a valuation. We say that

$$\langle \xi, \theta \rangle \text{ is a } \pi\textit{-derivation-solution pair},$$

If $Dom(\vartheta) = Var(\xi)$ and ϑ is a solution of b. □

When π is not specified in the previous definition we mean that ξ can be any derivation (and not just a π-derivation). Moreover, if ξ is a derivation *in M* then we say that $\langle \xi, \theta \rangle$ is a pair *in M*. We now need to extend Definition 7.4.1 to derivation-solution pairs. The underlying idea is that $\langle \xi_1, \theta_1 \rangle \preceq \langle \xi_2, \theta_2 \rangle$ iff ξ_1 and ξ_2 are derivations starting in the same goal and $\xi_1\theta_1 \preceq \xi_2\theta_2$. Therefore the following.

Definition 7.6.2 Let P, P' be two programs, $\xi_1:\ c_A \,\Box\, \tilde{A} \stackrel{P}{\rightsquigarrow} b_1 \,\Box\, \tilde{B}_1$ and $\xi_2:\ c_A \,\Box\, \tilde{A} \stackrel{P'}{\rightsquigarrow} b_2 \,\Box\, \tilde{B}_2$ be two derivations starting in the same goal. Let also θ_1 and θ_2 be solution of ξ_1 and ξ_2, respectively. We say that

$$\langle \xi_2, \theta_2 \rangle \text{ is more general than } \langle \xi_1, \theta_1 \rangle,\ \langle \xi_1, \theta_1 \rangle \preceq \langle \xi_2, \theta_2 \rangle,$$

if $\mathcal{D} \models \tilde{B}_1\theta_1 \ \rightarrow\ \tilde{B}_2\theta_2$. □

We can now characterize the concept of $\mathcal{M}$-congruence.

Theorem 7.6.3 Let $M_1 = \langle P_1, \pi \rangle$ and $M_2 = \langle P_2, \pi \rangle$ be two modules. Equivalent are

- for each π-derivation-solution pair $\langle \xi_i, \theta_i \rangle$ in M_i there exists a π-derivation-solution pair $\langle \xi_j, \theta_j \rangle$ in M_j $(i \neq j)$ such that $\langle \xi_i, \theta_i \rangle \preceq \langle \xi_j, \theta_j \rangle$,
- $M \approx_{\mathcal{M}} M'$. □

Proof. An analogous result, for the case of pure logic programs, is proved in [22]. The extension to the CLP case is straightforward. □

This Theorem represents the $\mathcal{M}$- counterpart of Theorems 7.2.3 and 7.4.4. Notice that, as opposed to the previous cases, here we have a bidirectional implication. An immediate consequence of this result is Proposition 7.4.7; let us state it again.

Proposition 7.4.7 If two modules are $\mathcal{C}$-congruent then they are $\mathcal{M}$-congruent.

Proof. Straightforward from Theorem 7.6.3 and Definitions 7.4.2, 7.4.1 and 7.6.2. □

Before proving Theorem 7.4.9 we need to strengthen Claim 7.2 as follows. Here and in the following, given a derivation $\xi:\ c_A \,\Box\, \tilde{A} \rightsquigarrow b \,\Box\, \tilde{B}$, we say that the valuation θ is a *solution* of ξ if $Dom(\theta) = Var(\xi)$ and θ is a solution of b.

Claim 7.4 Let P be a program, and $c_1 \wedge c_2 \,\Box\, \tilde{C}_1, \tilde{C}_2$ be a query. Then, there exists a derivation $c_1 \wedge c_2 \,\Box\, \tilde{C}_1, \tilde{C}_2 \overset{P}{\leadsto} d \,\Box\, \tilde{D}$ of length n iff there exist two derivations $\xi_1 :\ c_1 \,\Box\, \tilde{C}_1 \overset{P}{\leadsto} d_1 \,\Box\, \tilde{D}_1$ and $\xi_2 :\ c_2 \,\Box\, \tilde{C}_2 \overset{P}{\leadsto} d_2 \,\Box\, \tilde{D}_2$ such that

(i) $\tilde{D} \equiv \tilde{D}_1, \tilde{D}_2$, and $d \equiv d_1 \wedge d_2$ is satisfiable,

(ii) the variables that ξ_1 and ξ_2 have in common are exactly those that $c_1 \,\Box\, \tilde{C}_1$ and $c_2 \,\Box\, \tilde{C}_2$ have in common,

(iii) $|\xi_1| + |\xi_2| = n$.

(iv) if θ is a solution of ξ then $\theta|_{Var(\xi_i)}$ is a solution of ξ_i,

(v) if θ_1 is a solution of ξ_1 and θ_2 is a solution of ξ_2, such that θ_1 and θ_2 agree con the set of variables $Var(c_1 \,\Box\, \tilde{C}_1) \cap Var(c_2 \,\Box\, \tilde{C}_2)$ then $\theta_1\theta_2$ is a solution of ξ. Moreover $\theta_1\theta_2|_{Var(\xi_i)} = \theta_i$.

Proof. The first part coincides with Claim 7.2. The second part is a straightforward consequence of the first one. $\Box$

We can eventually prove the Theorem 7.4.9.

Theorem 7.4.9 ($\mathcal{M}$-correctness) Let $cl\ :\ A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$ be a clause of the module $M : \langle P, \pi \rangle$, and $M' : \langle P', \pi \rangle$ be the result of replacing $c \,\Box\, \tilde{C}$ by $d \,\Box\, \tilde{D}$ in cl. So $P' = P \backslash \{cl\} \cup \{cl' : A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}\}$. If

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

then $M \approx_{\mathcal{M}} M'$.

Proof. As in Theorem 7.4.6 we divide the proof in two parts. In part (a) we prove *partial correctness:* we show that for each pair π-derivation-solution $\langle \xi', \theta' \rangle$ in M' there is a pair π-derivation-solution $\langle \xi, \theta \rangle$ in M such that $\langle \xi', \theta' \rangle \preceq \langle \xi, \theta \rangle$. In part (b) we show the vice-versa: that for each π-derivation-solution $\langle \xi, \theta \rangle$ in M there is a π-derivation-solution $\langle \xi', \theta' \rangle$ in M' such that $\langle \xi, \theta \rangle \preceq \langle \xi', \theta' \rangle$. By Theorem 7.6.3 this implies the thesis. In the following, for the sake of simplicity, derivation-solution pairs will be referred to simply as *pairs*, and, as in the proof of Theorem 7.2.5, we follow Assumption 7.2.6.

Part (a). We proceed by induction on the length of the derivation. Let $\langle \xi', \theta' \rangle$ be a π-derivation-solution in M'.

Base case $|\xi'| = 0$. This case is trivial, as the derivations of length zero are the ones of the form $b \,\Box\, \tilde{B} \overset{M'}{\leadsto} b \,\Box\, \tilde{B}$.

Inductive step. By Claims 7.1 and 7.4 the derivation ξ' can be chosen of the form.

$$\xi' :\ true \,\Box\, H \overset{M'}{\leadsto} b \,\Box\, \tilde{B}$$

where $\tilde{B}$ contains only π-atoms and $Var(H) \cap Var(\tilde{B}) = \emptyset$ (since ξ' has length greater than 0). By the definition of derivation it follows that there exists a (renaming of a) clause of M',

$$J \leftarrow c_L \,\Box\, \tilde{L} \tag{7.13}$$

and a π-derivation

$$\zeta' : \ (H = J) \wedge c_L \,\Box\, \tilde{L} \overset{M'}{\leadsto} b \,\Box\, \tilde{B}$$

such that $|\xi'| = |\zeta'| + 1$, $Var(\zeta') = Var(\xi')$, and θ' is a solution of ζ'. By inductive hypothesis there exists a pair $\langle \zeta, \theta \rangle$ in M such that $\langle \zeta', \theta' \rangle \preceq \langle \zeta, \theta \rangle$. Now, if the clause of (7.13) was also a clause of M (that is, if it was not a result of the transformation), then there would exist a pair $\langle \xi, \theta \rangle$ in M such that $\langle \xi', \theta' \rangle \preceq \langle \xi, \theta \rangle$, thus concluding the proof of part (a). So we have to consider the case in which $J \leftarrow c_L \,\Box\, \tilde{L} \in M' \backslash M$. In this situation $J \leftarrow c_L \,\Box\, \tilde{L}$ is exactly (a variant of) the clause $cl' : A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}$. By appropriately renaming all the variables in the clauses and the derivations considered so far, we can assume that ζ' is the derivation

$$\zeta' : \ (H = A) \wedge d \,\Box\, \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b \,\Box\, \tilde{B}$$

By Claim 7.4 there exist two derivations ζ'_1 and ζ'_2 such that

$$\begin{gathered}
\zeta'_1 : \ d \,\Box\, \tilde{D} \overset{M'}{\leadsto} b_1 \,\Box\, \tilde{B}_1, \\
\zeta'_2 : \ (H = A) \,\Box\, \tilde{E} \overset{M'}{\leadsto} b_2 \,\Box\, \tilde{B}_2, \\
b \equiv b_1 \wedge b_2 \text{ and } \tilde{B} \equiv \tilde{B}_1, \tilde{B}_2, \\
|\zeta'_1| + |\zeta'_2| = |\zeta'| = |\xi'| - 1, \\
Var(\zeta'_1) \cap Var(\zeta'_2) \subseteq Var(d \,\Box\, \tilde{D}) \cap Var((H = A) \,\Box\, \tilde{E}),
\end{gathered}$$

and such that $\theta'|_{Var(\zeta'_1)}$ is a solution of ζ'_1 and $\theta'|_{Var(\zeta'_2)}$ is a solution of ζ'_2. By the inductive hypothesis there exist two pairs $\langle \zeta_1, \eta_1 \rangle$ and $\langle \zeta_2, \eta_2 \rangle$ in M, such that

$$\begin{gathered}
\zeta_1 : \ d \,\Box\, \tilde{D} \overset{M'}{\leadsto} b_1^* \,\Box\, \tilde{B}_1^*, \\
\zeta_2 : \ (H = A) \,\Box\, \tilde{E} \overset{M'}{\leadsto} b_2^* \,\Box\, \tilde{B}_2^*, \\
\langle \zeta_1, \eta_1 \rangle \succeq \langle \zeta'_1, \theta'|_{Var(\zeta'_1)} \rangle \quad \text{and} \quad \langle \zeta_2, \eta_2 \rangle \succeq \langle \zeta'_2, \theta'|_{Var(\zeta'_2)} \rangle, \\
Var(\zeta_1) \cap Var(\zeta_2) \subseteq Var(d \,\Box\, \tilde{D}) \cap Var((H = A) \,\Box\, \tilde{E}).
\end{gathered}$$

Since $d \,\Box\, \tilde{D}$ is ($\mathcal{M}$-)equivalent to $c \,\Box\, \tilde{C}$ under $Var(A, \tilde{E})$ in M it follows that there exists a derivation-solution pair $\langle \zeta_3, \eta_3 \rangle$, where

$$\zeta_3 : \ c \,\Box\, \tilde{C} \overset{M}{\leadsto} b_3 \,\Box\, \tilde{B}_3,$$

such that, if we let $\tilde{x} = Var(A, \tilde{E})$,

$$\eta_1|_{\tilde{x}} = \eta_3|_{\tilde{x}} \quad \text{and} \quad \mathcal{D} \models \tilde{B}_1\eta_1 \ \rightarrow \ \tilde{B}_3\eta_3. \tag{7.14}$$

By Assumption 7.2.6, the variables of $b_3 \,\Box\, \tilde{B}_3$ which do not occur in $d \,\Box\, \tilde{D}$, do not occur either in the derivations considered so far. Therefore the variables that ζ_2 and ζ_3 have in common are certainly contained in $\tilde{x}$. This together with the fact that $b_1^* \wedge b_2^*$ is satisfiable and the left hand side of (7.14) implies that also $b_3 \wedge b_2^*$ is satisfiable. Then, by Claim 7.4, we can put together ζ_3 and ζ_2 thus obtaining the derivation

$$\zeta_4 : \ (H = A) \wedge d \,\Box\, \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b_3 \wedge b_2^* \,\Box\, \tilde{B}_3, \tilde{B}_2^*$$

such that $\theta_4 = \eta_2\eta_3$ is a solution of ζ_4 and

$$\theta_4|_{Var(\zeta_3)} = \eta_3 \quad \text{and} \quad \theta_4|_{Var(\zeta_2)} = \eta_2. \tag{7.15}$$

Since in M we find the clause $cl:\ A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$, by the definition of derivation it follows that there exists a derivation ξ which uses only clauses of M such that ξ is similar to

$$true \,\Box\, H \overset{M}{\leadsto} b_3 \wedge b_2^* \,\Box\, \tilde{B}_3, \tilde{B}_2^*$$

and θ_4 is a solution of ξ. Since the variables that $b_3 \,\Box\, \tilde{B}_3$ has in common with the rest of this expression are certainly contained in $Var(A, \tilde{E})$, from (7.14) and (7.15) it follows that $\xi' \preceq \xi$, thus concluding the proof of part 1.

Part (b). We now show that for each π-derivation-solution $\langle \xi, \theta \rangle$ in M there is a π-derivation-solution $\langle \xi', \theta' \rangle$ in M' such that $\langle \xi, \theta \rangle \preceq \langle \xi', \theta' \rangle$. The first part of this is perfectly symmetrical to the one of Part (a): We proceed by induction on the length of the derivation ξ in M.

Base case $|\xi| = 0$. This case is trivial, as the derivations of length zero are the ones of the form $b \,\Box\, \tilde{B} \overset{M}{\leadsto} b \,\Box\, \tilde{B}$.

Inductive step. By Claims 7.1 and 7.4, ξ can be chosen of the form

$$\xi:\ true \,\Box\, H \overset{M}{\leadsto} b \,\Box\, \tilde{B}$$

where $\tilde{B}$ contains only π-atoms and $Var(H) \cap Var(\tilde{B}) = \emptyset$. By the definition of derivation there exist a (renaming of a) clause of M,

$$J \leftarrow c_L \,\Box\, \tilde{L} \tag{7.16}$$

and a π-derivation

$$\zeta:\ (H = J) \wedge c_L \,\Box\, \tilde{L} \overset{M}{\leadsto} b \,\Box\, \tilde{B}$$

such that $|\xi| = |\zeta| + 1$, $Var(\zeta) = Var(\xi)$ and θ is a solution of ζ. By the inductive hypothesis, there exists a pair $\langle \zeta', \theta' \rangle$ in M' such that $\langle \zeta, \theta \rangle \preceq \langle \zeta', \theta' \rangle$. Now, if the clause of (7.16) was also a clause of M' (that is, if it was not a result of the transformation), then there would exist a derivation-solution pair $\langle \xi', \theta' \rangle$ in M' such that $\langle \xi, \theta \rangle \preceq \langle \xi', \theta' \rangle$, thus concluding the proof of part (b).

So we have to consider the case in which $J \leftarrow c_L \,\Box\, \tilde{L} \in M \backslash M'$. In this situation, $J \leftarrow c_L \,\Box\, \tilde{L}$ is exactly (a variant of) the clause $cl : A \leftarrow c \,\Box\, \tilde{C}, \tilde{E}$. By appropriately renaming all the variables in the clauses and the derivations considered so far, we can assume that ζ is exactly the derivation

$$\zeta:\ (H = A) \wedge c \,\Box\, \tilde{C}, \tilde{E} \overset{M}{\leadsto} b \,\Box\, \tilde{B}.$$

By Claim 7.4, there exist two derivations ζ_1 and ζ_2 such that

$$\begin{gathered}
\zeta_1:\ c \,\Box\, \tilde{C} \overset{M}{\leadsto} b_1 \,\Box\, \tilde{B}_1, \\
\zeta_2:\ (H = A) \,\Box\, \tilde{E} \overset{M}{\leadsto} b_2 \,\Box\, \tilde{B}_2, \\
b \equiv b_1 \wedge b_2 \text{ and } \tilde{B} \equiv \tilde{B}_1, \tilde{B}_2, \\
|\zeta_1| + |\zeta_2| = |\zeta| = |\xi| - 1 \\
Var(\zeta_1) \cap Var(\zeta_2) \subseteq Var(c \,\Box\, \tilde{C}) \cap Var((H = A) \,\Box\, \tilde{E}),
\end{gathered}$$

and such that $\theta|_{Var(\zeta_1)}$ is a solution of ζ_1 and $\theta|_{Var(\zeta_2)}$ is a solution of ζ_2.
From the fact that $d \,\square\, \tilde{D}$ is ($\mathcal{M}$-) equivalent to $c \,\square\, \tilde{C}$ under $Var(A, \tilde{E})$ in M it follows that there exists a pair $\langle \zeta_3, \eta_3 \rangle$, where

$$\zeta_3 : \; d \,\square\, \tilde{D} \overset{M}{\leadsto} b_3 \,\square\, \tilde{B}_3,$$

such that ,

$$\eta_3|_{\tilde{x}} = \theta|_{\tilde{x}} \quad \text{and} \quad \mathcal{D} \models \tilde{B}_1\theta \;\rightarrow\; B_3\eta_3 \tag{7.17}$$

for $\tilde{x} = Var(A, \tilde{E})$. We now have to distinguish two cases.

Case 1. First we consider the case in which $d \,\square\, \tilde{D}$ is ($\mathcal{M}$-) not slower than $c \,\square\, \tilde{C}$ under $Var(A, \tilde{E})$ in M. In this case, we can assume that $|\zeta_3| \leq |\zeta_1|$.

There is no loss in generality in assuming that the variables of $b_3 \,\square\, \tilde{B}_3$ which do not occur in $d \,\square\, \tilde{D}$ do not occur in the derivations considered so far. Therefore, the variables that ζ_2 and ζ_3 have in common are certainly contained in $\tilde{x}$. From this, the fact that $b_1 \wedge b_2$ is satisfiable and the left hand side of (7.17) it follows that also $b_3 \wedge b_2$ is satisfiable. By Claim 7.4, we can then put together ζ_3 and ζ_2, and obtain the derivation

$$\zeta_4 : \; (H = A) \wedge d \,\square\, \tilde{D}, \tilde{E} \overset{M}{\leadsto} b_3 \wedge b_2 \,\square\, \tilde{B}_3, \tilde{B}_2 \tag{7.18}$$

where we have that $\theta_4 = \eta_2\eta_3$ is a solution of ζ_4 and that

$$\theta_4|_{Var(\zeta_3)} = \eta_3 \quad \text{and} \quad \theta_4|_{Var(\zeta_2)} = \eta_2. \tag{7.19}$$

Here we have also that

Observation 7.6.4 the variables that $b_3 \,\square\, \tilde{B}_3$ has in common with the rest of (7.18) are certainly contained in $Var(A, \tilde{E})$.

Moreover, the following inequality holds: $|\zeta_4| = |\zeta_3| + |\zeta_2| \leq |\zeta_1| + |\zeta_2| = |\zeta| = |\xi| - 1$. Therefore, by the inductive hypothesis, there exists a pair $\langle \zeta', \theta' \rangle$ such that $\zeta' : \; (H = A) \wedge d \,\square\, \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b'_3 \wedge b'_2 \,\square\, \tilde{B}'_3, \tilde{B}'_2$ and

$$\langle \zeta_4, \theta_4 \rangle \preceq \langle \zeta', \theta' \rangle \tag{7.20}$$

Since in M' we find the clause $cl' : \; A \leftarrow d \,\square\, \tilde{D}, \tilde{E}$, by the definition of derivation there exists a derivation $\xi' : true \,\square\, H \overset{M'}{\leadsto} b'_3 \wedge b'_2 \,\square\, \tilde{B}_3,' \tilde{B}'_2$ such that θ' is a solution of ξ'. Now Observation 7.6.4, (7.17), (7.19) and (7.20) imply that $\xi \preceq \xi'$, thus concluding the proof of Case 1.

Case 2. We consider now the case in which $d \,\square\, \tilde{D}$ is not ($\mathcal{M}$-) not-slower than $c \,\square\, \tilde{C}$ under $Var(A, \tilde{E})$ in M. From the hypothesis it follows then that $d \,\square\, \tilde{D}$ is independent from cl. So, the clauses used in ζ_3 are also clauses of M' and we have that in M' there exists a derivation ζ'_3 which is identical to ζ_3, that is $\zeta'_3 : \; d \,\square\, \tilde{D} \overset{M'}{\leadsto} b_3 \,\square\, \tilde{B}_3$. Moreover, since $|\zeta_2| < |\xi|$, by the inductive hypothesis there exists a pair $\langle \zeta'_2, \eta'_2 \rangle$ such that

$$\begin{gathered} \zeta'_2 : \; (H = A) \,\square\, \tilde{E} \overset{M'}{\leadsto} b'_2 \,\square\, \tilde{B}'_2 \text{ and} \\ \langle \zeta_2, \theta|_{Var(\zeta_2)} \rangle \preceq \langle \zeta'_2, \eta'_2 \rangle. \end{gathered} \tag{7.21}$$

By Assumption 7.2.6, the variables that ζ_2' and ζ_3' have in common are contained in $\tilde{x}$. Therefore, from the fact that $b_1 \wedge b_2$ is satisfiable and the left hand side of (7.17) it follows that also $b_3 \wedge b_2$ is satisfiable. The relation (7.21) implies that $b_3 \wedge b_2$ is satisfiable. From Claim 7.4 it follows that we can put together ζ_2' and ζ_3' thus obtaining the derivation

$$\zeta_4' : \ (H = A) \wedge d \,\Box\, \tilde{D}, \tilde{E} \overset{M'}{\leadsto} b_3 \wedge b_2' \,\Box\, \tilde{B}_3, \tilde{B}_2'$$

such that $\theta_4' = \eta_2'\eta_3$ is a solution of ζ_4' and the following holds:

$$\theta_4'|_{Var(\zeta_3')} = \eta_3 \quad \text{and} \quad \theta_4'|_{Var(\zeta_2')} = \eta_2'. \tag{7.22}$$

Since in M' we find the clause $cl' : \ A \leftarrow d \,\Box\, \tilde{D}, \tilde{E}$, by the definition of derivation there exists a derivation $\xi' : true \,\Box\, H \overset{M'}{\leadsto} b_3 \wedge b_2' \,\Box\, \tilde{B}_2', \tilde{B}_3$ such that θ_4' is a solution of ξ'. Since the variables that $b_3 \,\Box\, \tilde{B}_3$ has in common with the rest of this expression are certainly contained in $Var(A, \tilde{E})$, from (7.17), (7.22) and (7.11) it follows that $\langle \xi, \theta \rangle \preceq \langle \xi', \theta_4' \rangle$, thus completing the proof. $\Box$

Chapter 8

On Unification-Free Prolog Programs

We provide new simple conditions which allow us to conclude that in case of several well-known Prolog programs the unification algorithm can be replaced by iterated matching. As already noticed by other researchers, such a replacement offers a possibility of improving the efficiency of program's execution. The results we prove improve on those in our previous paper ([7]) both because they allow to prove unification-freeness for a larger class of programs and because the conditions are, in many cases, checkable in a much more efficient way.

8.1 Introduction

Unification is the core of the resolution method employed by PROLOG, and its efficiency has great influence on the overall performance of the interpreter. The best sequential unification algorithm employs linear time (see for example Martelli-Montanari [74]), and, most likely, this result cannot be improved by the adoption of a parallel algorithm: Dwork et al. [36] have shown that, unless PTIME $\subseteq$ NC (which is quite improbable) unification does not admit an algorithm that run polilogarithmic time using a polynomially bounded number of processors.

On the other hand, fast parallel algorithms are available for *term matching*: a special case of unification where one of the terms is always an instance of the other one [36, 37]. This motivates the research for sufficient conditions for the replacement of unification with term matching (see, for instance [34, 70, 13] and, more recently, [7, 71]).

In Deransart and Maluszynski [34], Maluszynski and Komorowski [70] and Attali and Franchi-Zannettacci [13], the problem was tackled by using *modes*. Intuitively, a *mode* is a function that labels as *input* or *output* the positions of each relation in order to indicate how the arguments of a relation should be used. A limit of this approach is that the input positions of the queries are expected to be filled in by ground (i.e. variable-free) terms. Apt and Etalle [7] improved upon the previous results by additionally using *types*, which allow to deal with non-ground inputs.

Here, we generalize the results of [7]. The main tools of our approach can be summarized as follows:

First, in addition to *input* and *output* positions, we introduce here U-positions. Here "U" can be read as *unknown*, as the U-positions of a query can be filled in by any term. It turns out that for many of the programs mentioned in [7] we could simply turn some positions into U positions, both enlarging significantly the class of allowed queries and, when this process was applied to the nonground input positions, simplifying dramatically the method for proving that the program is unification-free.

Second, we now allow also *pure terms* to fill in *output* positions of the queries, again this enlarges the class of allowed queries.

Finally, by following Apt [4], we adopt here a more flexible definition of *well-typed program*.

As in our previous paper, the conditions we provide can be statically checked without analyzing the search trees for the queries.

This chapter is organized as follows. In the next section we introduce the concepts of solvability by sequential matching and of unification-free prolog program. Section 3 contains the basic definitions of modes and types, which are the main tools we need in the sequel. Both concept are used in order to specify how the arguments of an atom should be used, and, ultimately, to restrict the set of allowed queries. In section 4 we begin to tackle the problem of how to prove that a program is unification-free: we introduce the definition of a Nicely Typed program and we show that, in some cases, this concept alone is sufficient for our purposes. This section can be also seen as an intermediate step: in the subsequent one we report the definition of Well-typed program. Programs which are both Well and Nicely Typed are the ones that will enable us to prove, in Section 5, our most general theorem (8.5.18). In Section 6 we give a more restrictive version of our Main Theorem. The relevance of this result lies in the fact that its applicability conditions can be tested in a much more efficient way. Section 7 contains some practical examples, and in Section 8 we conclude by comparing this chapter with our previous paper [7] and with another recent related paper [71].

8.2 Preliminaries

In what follows we study logic programs executed by means of the *LD-resolution*, which consists of the SLD-resolution combined with the leftmost selection rule. An SLD-derivation in which the leftmost selection rule is used is called an *LD-derivation*. We allow in programs various first-order built-in's, like $=$, $\neq$, $>$, etc, and assume that they are resolved in the way conforming to their interpretation.

We work here with *queries*, that is sequences of atoms, instead of *goals*, that is constructs of the form $\leftarrow Q$, where Q is a query. Apart from this we use the standard notation of Lloyd [65] and Apt [3]. In particular, given a syntactic construct E (so for example, a term, an atom or a set of equations) we denote by $Var(E)$ the set of the variables appearing in E. Given a substitution $\theta = \{x_1/t_1, ..., x_n/t_n\}$ we denote by $Dom(\theta)$ the set of variables $\{x_1, \ldots, x_n\}$, by $Range(\theta)$ the set of terms $\{t_1, \ldots, t_n\}$, and by $Ran(\theta)$ the set of variables appearing in $\{t_1, \ldots, t_n\}$. Finally, we define $Var(\theta) = Dom(\theta) \cup Ran(\theta)$.

Recall that a substitution θ is called *grounding* if $Ran(\theta)$ is empty, and is called a *renaming* if it is a permutation of the variables in $Dom(\theta)$. Given a substitution θ and a set of variables V, we denote by $\theta|V$ the substitution obtained from θ by restricting its domain to V.

Unifiers

Given two sequences of terms $\tilde{s} = s_1, ..., s_n$ and $\tilde{t} = t_1, ..., t_n$ of the same length we abbreviate the set of equations $\{s_1 = t_1, ..., s_n = t_n\}$ to $\{\tilde{s} = \tilde{t}\}$ and the sequence $s_1\theta, ..., s_n\theta$ to $\tilde{s}\theta$. Two atoms can unify only if they have the same relation symbol, and with two atoms $p(\tilde{s})$ and $p(\tilde{t})$ to be unified we associate the set of equations $\{\tilde{s} = \tilde{t}\}$. In the applications we often refer to this set as $p(\tilde{s}) = p(\tilde{t})$. A substitution θ such that $\tilde{s}\theta = \tilde{t}\theta$ is called a *unifier* of the set of equations $\{\tilde{s} = \tilde{t}\}$. Thus the set of equations $\{\tilde{s} = \tilde{t}\}$ has the same unifiers as the atoms $p(\tilde{s})$ and $p(\tilde{t})$.

A unifier θ of a set of equations E is called a *most general unifier* (in short *mgu*) of E if it is more general than all unifiers of E. An mgu θ of a set of equations E is called *relevant* if $Var(\theta) \subseteq Var(E)$.

The following Lemma was proved in Lassez, Marriot and Maher [64].

Lemma 8.2.1 Let θ_1 and θ_2 be mgu's of a set of equations. Then for some renaming η we have $\theta_2 = \theta_1\eta$. □

Finally, the following well-known Lemma allows us to search for mgu's in an iterative fashion.

Lemma 8.2.2 Let E_1, E_2 be two sets of equations. Suppose that θ_1 is a relevant mgu of E_1 and θ_2 is a relevant mgu of $E_2\theta_1$. Then $\theta_1\theta_2$ is a relevant mgu of $E_1 \cup E_2$. Moreover, if $E_1 \cup E_2$ is unifiable then θ_1 exists and for any such θ_1 an appropriate θ_2 exists, as well. □

Solvability by (sequential) Matching

Following the notation of Apt and Etalle, [7], we begin by recalling the following concepts.

Definition 8.2.3 Consider a set of equations $E = \{\tilde{s} = \tilde{t}\}$.

- A substitution θ such that either $Dom(\theta) \subseteq Var(\tilde{s})$ and $\tilde{s}\theta = \tilde{t}$ or $Dom(\theta) \subseteq Var(\tilde{t})$ and $\tilde{s} = \tilde{t}\theta$, is called a *match* for E.
- E is called *left-right disjoint* if $Var(\tilde{s}) \cap Var(\tilde{t}) = \emptyset$. □

Clearly, if E is left-right disjoint, then a match for E is also a relevant mgu of E. The sets of equations we consider in this chapter will always satisfy this disjointness proviso due to the standardization apart.

Definition 8.2.4 Let E be a left-right disjoint set of equations. We say that E is *solvable by matching* if E is unifiable implies that a match for E exists. □

Consider a selected atom $p(t_1, \ldots, t_n)$ and the head $p(s_1, \ldots, s_n)$ of an input clause used to resolve it. The unification mechanism tries then to find a mgu of the set of equations $t_1 = s_1, \ldots, t_n = s_n$. Sometimes such a set is not solvable by matching as a whole, but it can be solved by a *sequential* matching, that is, by considering the equations one at a time.

To formalize this idea we introduce the following notion.

Definition 8.2.5 Let $E = E_1, \ldots, E_n$ be a left-right disjoint sequence of (sets of) equations.

- We say that E is *solvable by sequential matching* if E is unifiable implies that for some substitutions $\theta_1, \ldots, \theta_n$, and for $i \in [1, n]$
  - $E_i\theta_1 \ldots \theta_{i-1}$ is left-right disjoint,
  - θ_i is a match for $E_i\theta_1 \ldots \theta_{i-1}$.
- We say that E is *solvable by sequential matching* wrt π if π is a permutation of $1, \ldots, n$, and
  - $E_{\pi(1)}, \ldots, E_{\pi(n)}$ is solvable by sequential matching. □

Note that when $\theta_1, \ldots, \theta_n$ satisfy the above two conditions, then by Lemma 8.2.2 $\theta_1\theta_2 \ldots \theta_n$ is a relevant mgu of E.

This Definition corresponds to the one considered by Maluszynski and Komorowski [70], and is slightly less general than the one of *iterated* matching given in [7], which makes no explicit reference to the order in which the equations are to be solved. Intuitively, E is solvable by *iterated* matching iff there exists a π such that E is solvable by sequential matching wrt π.

Unification Free Programs

Recall that the aim of this chapter is to clarify for what Prolog programs unification can be replaced by sequential matching. The following Definition is then the key one. Here we denote by $rel(A)$ the relation symbol of the atom A.

Definition 8.2.6

- Let ξ be an LD-derivation. Let A be an atom selected in ξ and H the head of the input clause selected to resolve A in ξ. Suppose that A and H have the same relation symbol. Then we say that the system $A = H$ *is considered in* ξ.
- Suppose that each system of equations $A = H$ considered in the LD-derivations of $P \cup \{Q\}$ is solvable by sequential matching wrt a permutation $\pi_{rel(A)}$, where $\pi_{rel(A)}$ is uniquely determined by the relation symbol of A. Then we say that $P \cup \{Q\}$ is *unification free*. □

A slightly more flexible definition of unification-free program was given in Apt-Etalle [7], where the equation $A = H$ may be solvable by *iterated* matching, i.e. the sequence π needs not to be determinable from the relations symbol of A.

8.3 Types and Modes

The main tools that we are going to use in this chapter are types and modes. The following very general definition of type is sufficient for our purposes.

Definition 8.3.1

- A *type* is a set of atoms with the same relation symbol;
- A *type* is a type for a relation symbol p. □

Notice that, as opposed to [7], here we are also considering types which are not closed under substitution.

For the purpose of this chapter, types for relations are always built by suitably combining set of terms.

Definition 8.3.2

- A *term_type* is a set of terms. □

Here, we sometimes overload the term *type* to denote either a type or a term_type; the actual meaning will be clear from the context.

Certain term_types will be of special interest:
U — the set of all terms,
Var — the set of variables,
$List$ — the set of lists,
$BinTree$ — the set of binary trees,
$Ground$ — the set of ground terms.

Of course, the use of the term_type $List$ assumes the existence of the empty list `[]` and the list constructor `[.|.]` in the language, and the use of the type Nat assumes the existence of the numeral `0` and the successor function `s(.)`, etc.

The following notation will be used throughout the chapter. Let p be an n-ary relation symbol, and let $T_1, \ldots, T_n$ be term_types. we denote by

$$p: T_1 \times \ldots \times T_n$$

the type for p given by the following set of atoms.

$$\{p(t_1, \ldots, t_n) \mid \text{ for } i \in [1, n], t_i \in T_i\}$$

Given a program P, a *typing* for P is a function that associate to each relation symbol p in P a type of the form $p: T_1 \times \ldots \times T_n$, consequently we also say that T_i is the term_type associated to the i-th *position* of p.

We need one final Definition.

Definition 8.3.3 Let $p: T_1 \times \ldots \times T_n$ be the type for p.

- We say that an atom $p(t_1, \ldots, t_n)$ is *correctly typed* in his i-th position if $t_i \in T_i$;
- We say that an atom $p(t_1, \ldots, t_n)$ *correctly typed* if it is correctly type in all its positions. □

In the sequel we assume that each program has a (n often unspecified) typing associated to. The typing specifies how the argument of a relation should be used: as a general rule, we expect that the atoms selected in a LD-derivation are correctly typed (to make sure of this we'll introduce appropriate tools). Consider for instance the well-known program `append`:

```
app([X | Xs], Ys, [X | Zs]) ← app(Xs, Ys, Zs).
app([], Ys, Ys).
```

`append` can be used for concatenating two lists, and this can be reflected by the adoption of the following "natural" typing:

`app` : $List \times List \times Var$

This typing expresses the fact that each time an atom of the form `:- append(s, t, u)` is selected in by the (leftmost) selection rule, we expect `s` and `t` to be lists, and `u` to be a variable. Multiple typings can be obtained by simply renaming the relations.

Before introducing modes, we need a last definition.

Definition 8.3.4

- We call an atom (resp. a term) a *pure atom* (resp. *pure term*) if it is of the form $p(\tilde{x})$ with $\tilde{x}$ a sequence of different variables.
- Two atoms (resp. terms) are called *disjoint* if they have no variables in common. □

To study solvability by matching, we keep in special consideration the following term_types.

- Var - the set of all variables;
- Pt - the set of variables and pure terms;
- U - the set of all terms.

Notice that $Var \subseteq Pt \subseteq U$. According to the typing used, we'll make some distinctions among the positions of an atom. Consider the case of a selected atom A and the head H of an input clause used to resolve A. In presence of types, we expect A to be correctly typed. It is then natural to consider the positions of A which are typed *Var* or *Pt*, which are filled in by variables or pure terms as *output* positions, as they contain no information. On the other hand for those positions which are typed U, since we really have no clue over the kind of parameter-passing that will take place in them, we use the special name of U-positions. The remaining positions will then by convention be considered as *input*. These considerations are at the base of the following Definition.

Definition 8.3.5 Let $p : T_1 \times \ldots \times T_n$ be the type of the relation symbol p. We call the i-th position of an atom $p(t_1, \ldots, t_n)$

- A U-position if $T_i = U$
- An *output* position if $T_i = Var$ or $T_i = Pt$;
- An *input* position otherwise. □

This classification is actually a *moding*. Modes for logic programs were first considered by Mellish [75] and then more extensively studied in Reddy [83] and in Dembinski and Maluszynski [35]. Here we are departing from the previous works by using also the mode U, which can be seen as a way to avoid to commit ourselves to a specific mode when such a commitment is not necessary.

8.4 Avoiding Unification using the modes "U" and "output"

In order to introduce the tools we need in a gradual manner, we begin by excluding the presence of input positions.

Surprisingly, in many cases, this restriction does not represent a problem: in order to pass the information from the selected atom to the head of the input clause we can still use the U-positions. Consider for instance again the program `append`, as we mentioned before, when it is used for concatenating two lists, the "natural" typing is

`append`: $List \times List \times Var$.

Now, if we want to avoid the presence of input positions, we can simply use the following typing.

`append`: $U \times U \times Var$

Notice that the first two positions are U-positions, while the third one is and output one. The only practical difference between this and the "natural" typing is that in the query `app(s, t, u)` we now allow `s` and `t` to be any term, rather than just list. This is obviously no restriction. In general, using the U-positions for the parameter-passing task has the advantage of flexibility: since every term belongs to U we are making here no *a priori* assumption on the structure of the data. Moreover, as we'll show in the rest of this Section, proving unification-freeness is in this context particularly simple.

Throughout this Section we assume that the atoms have only U- and output positions: by Definition 8.3.5 this is equivalent to considering typings built only with the following term_types: U, Var and Pt.

Sequential Matching via Pure Terms

We start with a simple test allowing us to determine whether a given set of equations is solvable by matching.

Lemma 8.4.1 (Matching 1) Consider two disjoint atoms A and H with the same relation symbol. Suppose that

- one of them is ground or pure.

Then $A = H$ is solvable by matching.

Proof. Clear. □

Now let us go back to the example of the (correctly typed) selected atom A and the head H of a clause used to resolve it. In order to apply the Matching 1 Lemma 8.4.1 to the part of $A = H$ corresponding to the U-positions, since we have no information about the shape of the terms filling in the U-positions of A, we have to impose some restrictions on H. Here we call a family of terms *linear* if every variable occurs at most once in it.

Definition 8.4.2 (U-safe$^-$) An atom H is called *U-safe$^-$* if the family of terms filling in its U-positions is linear and consists of only variables and pure terms. □

The minus sign in *U-safe$^-$* is motivated by the fact that in Section 8.5 we'll introduce a more general definition of U-safeness, which will also take into account the presence of input positions. We need now one further notion.

Definition 8.4.3 An atom A is called *output independent* if each term occurring in an output position is disjoint from the rest of A. □

Now we prove a result allowing us to conclude that $A = H$ is solvable by sequential matching.

Lemma 8.4.4 (Sequential Matching 1) Consider two disjoint atoms A and H with the same relation symbol p. Suppose that p has no input positions. If

- A is correctly typed and output independent,
- H is U-safe$^-$,

then there exists a permutation π such that $A = H$ is solvable by sequential matching wrt π.

In particular, $A = H$ is solvable by sequential matching wrt any permutation π of $1, \ldots, n$ such that, according to the order given by $\pi(1), \ldots, \pi(n)$, we have that the U-positions of p come first and the output positions come last.

Proof. Suppose that $A = H$ is unifiable, we can then assume that A is $p(s_1, \ldots, s_n)$ and that H is equal to $p(t_1, \ldots, t_n)$, where $s_1, \ldots, s_n, t_1, \ldots, t_n$ have been reordered in such a way that U-positions come first (on the left) and the output positions are the rightmost ones.

We now need to prove that $s_1 = t_1, \ldots, s_n = t_n$ is solvable by sequential matching, that is we need to find $\theta_1, \ldots, \theta_n$ such that each θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$. For each i, we distinguish upon the kind of position where the equation $s_i = t_i$ is found.

If $s_i = t_i$ is found in a U-position then, since H is U-safe$^-$, we have that t_i is a variable or a pure term and $Var(t_i) \cap Var(\theta_1 \ldots \theta_{i-1}) = \emptyset$, so $t_i\theta_1 \ldots \theta_{i-1}$ is still a variable or a pure term and by the Matching 1 Lemma 8.4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching.

Finally, if $s_i = t_i$ is found in an output position then, from the assumptions we made on A, it follows that s_i is a variable or a pure term and that $Var(s_i) \cap Var(\theta_1, \ldots, \theta_{i-1}) =$

$\emptyset$. So $s_i\theta_1, \ldots, \theta_{i-1}$ is still a variable or a pure term, and by the Matching 1 Lemma 8.4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching. □

When A and H satisfy the conditions of this Lemma, we can then solve $A = H$ by sequentially matching one position at a time. Still, we can improve on this result by showing that there exist some subsets of $A = H$ which correspond to more than one position and which can be solved by a single matching. This issue will be discussed in the Appendix.

We need one further notion.

Definition 8.4.5 We call an LD-derivation *i/o driven* if all atoms selected in it are correctly typed and output independent. □

i/o driven derivations were introduced in [7], but the definition we give here is more general than the previous one. This is due to the fact that now we consider also U-positions, and that we allow Pt as a term_type for the output positions (in [7] the only term_type allowed for the output positions is Var).

The Sequential Matching Lemma 8.4.4 allows us to combine the notions of U-safe atom and of i/o driven derivation for concluding that $P \cup \{Q\}$ is unification free.

Theorem 8.4.6 Suppose that each predicate symbol occurring in P has no input positions. If

- the head of every clause of P is U-safe$^-$,
- all LD-derivations of $P \cup \{Q\}$ are i/o driven.

Then $P \cup \{Q\}$ is unification free. □

Taking care of the output positions: Nicely Typed programs

In order to apply Theorem 8.4.6 we need to find conditions which imply that all considered LD-derivations are i/o driven. Since here we exclude the existence of input positions, all we have to do is to ensure that the selected atom A is correctly typed in its output position and output independent. For this we'll introduce the new concept of Nicely Typed program.

We start with the following notion which was introduced in Chadha and Plaisted [27]. Here we use the notation of Apt and Pellegrini [9]: when writing an atom as $p(\tilde{r}, \tilde{o})$, we now assume that $\tilde{o}$ is the sequence of terms filling in the output positions of p, while that $\tilde{r}$ is the sequence of terms filling its remaining positions.

Definition 8.4.7 (Nicely Moded)

- A query $p_1(\tilde{r}_1, \tilde{o}_1), \ldots, p_n(\tilde{r}_n, \tilde{o}_n)$ is called *nicely moded* if $\tilde{o}_1, \ldots \tilde{o}_n$ is a linear family of terms and for $j \in [1, n]$

$$Var(\tilde{r}_j) \cap (\bigcup_{k=j}^{n} Var(\tilde{o}_k)) = \emptyset. \tag{8.1}$$

- A clause
$$p_0(\tilde{r}_0, \tilde{o}_0) \leftarrow p_1(\tilde{r}_1, \tilde{o}_1), \ldots, p_n(\tilde{r}_n, \tilde{o}_n)$$
is called *nicely moded* if $p_1(\tilde{r}_1, \tilde{o}_1), \ldots, p_n(\tilde{r}_n, \tilde{o}_n)$ is nicely moded and
$$Var(\tilde{r}_0) \cap (\bigcup_{k=1}^{n} Var(\tilde{o}_k)) = \emptyset. \tag{8.2}$$
In particular, every unit clause is nicely moded.
- A program is called *nicely moded* if every clause of it is. □

Thus, assuming that in every atom the output positions are the rightmost ones, a query is nicely moded if

- every variable occurring in an output position of an atom does not occur earlier in the query.

And a clause is nicely moded if

- every variable occurring in an output position of a body atom occurs neither earlier in the body nor in a non-output position of the head.

So, intuitively, the concept of being nicely moded prevents a "speculative binding" of the variables which occur in output positions — these variables are required to be "fresh".

From the definition it follows that, if the query is nicely moded, then the selected atom is output independent. In order to fulfill the requirements of i/o drivenness we also ask the output positions to be correctly typed. For this reason we introduce a further Definition. Here and in the sequel, given an atom A, we denote by $VarOut(A)$ the set of variables occurring in the output positions of A. Similar notation is used for sequences of atoms.

Definition 8.4.8 (Nicely Typed)

- A nicely moded query $\tilde{B}$ is called *nicely typed* if it is correctly typed in its output positions.
- a nicely moded clause $H \leftarrow \tilde{B}$ is called *nicely typed* if $\tilde{B}$ is nicely typed, and each term t filling in a position of H of type Pt satisfies the following

$$\text{If } t \text{ is a variable and } t \cap VarOut(\tilde{B}) \neq \emptyset \text{ then } t \text{ fills in a position of } \tilde{B} \text{ of type } Pt. \tag{8.3}$$

- A program is called *nicely typed* if every clause of it is. □

Nicely typed programs can be seen as a generalization of simply moded programs of [7]. The additional condition (8.3) that we impose on the clauses is needed to ensure the persistence of the notion of being nicely typed, which is proven in the following key Lemma.

Lemma 8.4.9 An LD-resolvent of a nicely typed query and a disjoint with it nicely typed clause is nicely typed. □

Proof. Consider a nicely typed query $A, \tilde{A}$ and a disjoint with it nicely typed clause $H \leftarrow \tilde{B}$, such that A and H unify. Take as E_0 the subset of $A = H$ corresponding to the non-output positions, and as $E_1, \ldots, E_n$ the subsets of $A = H$ each corresponding to an output position.

The proof is divided in steps.

Claim 8.1 There exist $\theta_0, \ldots, \theta_n$ such that, for $i \in [0, n]$,

(a) θ_i is a relevant mgu of $E_i\theta_0 \ldots \theta_{i-1}$,
(b) $\tilde{B}\theta_0, \ldots, \theta_i$ is correctly typed in its output positions.

Proof. We proceed by induction.

Base case: $i = 0$.
Let θ_0 be any relevant mgu of E_0. Since $H \leftarrow \tilde{B}$ is nicely moded, the variables in $VarOut(\tilde{B})$ do not occur in the non-output positions of H, therefore the output positions of $\tilde{B}$ are not affected by θ_0. Since by hypothesis $\tilde{B}$ is correctly typed in its output positions, $\tilde{B}\theta_0$ is correctly typed in its output positions as well.

Induction step: $i > 0$.
Let $E_i \equiv s = t$, where s and t are the terms filling the i-th output position respectively of A and H. First notice that since A is nicely moded, the variables of s do not occur anywhere else in A. Moreover, from the disjointness hypothesis (and the relevance of each θ_i) it follows then that $Var(s) \cap Var(\theta_0 \ldots \theta_{i-1}) = \emptyset$. Therefore we have that

$$s\theta_0 \ldots \theta_{i-1} = s$$

Keep in mind that by the inductive hypothesis $\tilde{B}\theta_0 \ldots \theta_{i-1}$ is correctly typed in its output positions, and that $s = s\theta_0 \ldots \theta_{i-1}$. Since A is nicely typed, s may only be a variable or a pure term. Let us consider those two cases separately, and let us suppose that s is

a variable. Then we can take θ_i to be exactly $[s/t\theta_0 \ldots \theta_{i-1}]$. Therefore $Dom(\theta_i) = s$, and $\tilde{B}\theta_0 \ldots \theta_{i-1}$ is not affected by θ_i, and the result follows from the inductive hypothesis.

a pure term. Since A is nicely typed, the type of the the i-th output position of A (and H) must be Pt. Let θ_i be any relevant mgu of $s\theta_1 \ldots \theta_{i-1} = t\theta_1 \ldots \theta_{i-1}$ We have to distinguish three cases:
First we consider the case in which $t\theta_0 \ldots \theta_{i-1}$ is a variable and it occurs in $VarOut(\tilde{B}\theta_0 \ldots \theta_{i-1})$. Obviously, in this case t itself is a variable as well. Now notice that if r is any term filling in an output position of $\tilde{B}$ then we have that

$$\text{if} \quad Var(r\theta_0 \ldots \theta_{i-1}) \cap t\theta_0 \ldots \theta_{i-1} \neq \emptyset \quad \text{then} \quad Var(r) \cap t \neq \emptyset \tag{8.4}$$

In other words, if r is disjoint from t then also $r\theta_0 \ldots \theta_{i-1}$ is disjoint from $t\theta_0 \ldots \theta_{i-1}$. This is due to the fact that, since $H \leftarrow \tilde{B}$ is nicely moded, the variables of r may not occur in the input positions of H but only in the output ones, and, since A is output independent, the substitutions $\theta_0 \ldots \theta_{i-1}$ cannot bind them to other variables of $H \leftarrow \tilde{B}$.

Since $t\theta_0 \ldots \theta_{i-1}$ occurs in $VarOut(\tilde{B}\theta_0 \ldots \theta_{i-1})$, from (8.4) it follows that t occurs in $VarOut(\tilde{B})$. Furthermore, from (8.4) and the fact that $H \leftarrow \tilde{B}$ is nicely typed it follows that $t\theta_0 \ldots \theta_{i-1}$ fills in an output position of $\tilde{B}\theta_0 \ldots \theta_{i-1}$, and (being $H \leftarrow \tilde{B}$ nicely moded) it does not occur anywhere also in $\tilde{B}\theta_0 \ldots \theta_{i-1}$. Now, $s\theta_0 \ldots \theta_{i-1}$ is a pure term and $t\theta_0 \ldots \theta_{i-1}$ is a variable, therefore we have that $t\theta_0 \ldots \theta_{i-1}\theta_i$ is a pure term, and, since $t\theta_0 \ldots \theta_{i-1}$ fills in an output position of $\tilde{B}\theta_0 \ldots \theta_{i-1}$ of type Pt, from the inductive hypothesis it follows that $\tilde{B}\theta_0 \ldots \theta_{i-1}\theta_i$ is correctly typed in its output positions.

Secondly, if $t\theta_0 \ldots \theta_{i-1}$ is a variable and it does not occur in $VarOut(\tilde{B})\theta_0 \ldots \theta_{i-1}$, then the output positions of $\tilde{B}\theta_0 \ldots \theta_{i-1}$ are not affected by θ_i, and the result follows by the inductive hypothesis.

Finally, if $t\theta_0 \ldots \theta_{i-1}$ is not a variable, then, since $s\theta_0 \ldots \theta_{i-1}(= s)$ is a pure term, and since $(s = t)\theta_0 \ldots \theta_{i-1}$ is unifiable, we have that $t\theta_0 \ldots \theta_{i-1}$ is an instance of $s\theta_0 \ldots \theta_{i-1}$. We can then take θ_i such that $Dom(\theta_i) = s\theta_0 \ldots \theta_{i-1}$. It follows that $t\theta_0 \ldots \theta_{i-1}$ is not affected by θ_i Consequently, $\tilde{B}\theta_0 \ldots \theta_{i-1}$ is not affected by θ_i as well and the result follows from the inductive hypothesis.

This ends the proof of Claim 8.1. □

Now let $\theta = \theta_0 \ldots \theta_i$. By Lemma 8.2.2 θ is a relevant mgu of $A = H$. So far we have established that

$$\tilde{B}\theta \text{ is correctly typed in its output positions.} \tag{8.5}$$

In order to prove that also $(\tilde{B}, \tilde{A})\theta$ is nicely typed we have to go through a few more steps.

Claim 8.2 $\tilde{A}\theta$ is correctly typed in its output position.

Proof. $\tilde{A}$ is nicely moded, therefore $VarOut(\tilde{A}) \cap Var(A) = \emptyset$. Since θ is relevant, from the disjointness hypothesis it follows then that $Var(\theta) \cap VarOut(\tilde{A}) = \emptyset$. Since $\tilde{A}$ is correctly typed in its output position, also $\tilde{A}\theta$ is. □

Finally we have that

Claim 8.3 $(\tilde{B}, \tilde{A})\theta$ is nicely moded.

Proof. This is due to the fact that the resolvent of a nicely moded query and a (disjoint with it) nicely moded clause is nicely moded (Apt and Pellegrini in [9, Lemma 5.3]). □

From (8.5) and the last two Claims it follows that $(\tilde{B}, \tilde{A})\theta$ is nicely typed. Now $\theta = \theta_1 \ldots \theta_n$ is just one specific mgu of $A = H$. By Lemma 8.2.1 every other mgu of $A = H$ is of the form $\theta\eta$ for a renaming η. But a renaming of a nicely typed query is nicely typed, so we conclude that every LD-resolvent of $A, \tilde{A}$ and $H \leftarrow \tilde{B}$ is nicely typed. □

The following is an immediate consequence of Lemma 8.4.9 which will be soon needed.

Corollary 8.4.10 Let P and Q be nicely typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ are correctly typed in their output positions and are output independent. □

Avoiding Unification with Nicely Typed Programs

Recall that in order to prove that $P \cup \{Q\}$ is unification-free using Theorem 8.4.6 we are looking for conditions which imply that all the LD-derivations starting in Q are i/o driven and that, since we are excluding the presence of input positions, this reduces to requiring that the selected atom are correctly typed in their output positions and output independent. By Corollary 8.4.10 the concept of being nicely typed is the one we need.

Lemma 8.4.11 Suppose that each predicate symbol p occurring in P has no input positions. If

- P and Q are nicely typed.

Then all LD-derivations of $P \cup \{Q\}$ are i/o driven.

Proof. This follows directly form Corollary 8.4.10. □

We can now state the main result of this Section.

Theorem 8.4.12 Suppose that each predicate symbol p occurring in P has no input positions. If

- P and Q are nicely typed,
- the head of every clause of P is U-safe$^-$

Then $P \cup \{Q\}$ is unification free.

Proof. From Lemma 8.4.11 and Theorem 8.4.6 □

This result, though rather simple, can be applied to a large number of programs.

Example 8.4.13
(i) Consider again the program `append`, together with the following typing:

`app` : $U \times U \times Pt$

First note that `append` is nicely typed and that the head of both clauses are U-safe$^-$. Now let `t, s` be terms, and `u` be a variable (or a pure term), disjoint from `t, s`; `append(t,s,u)` is then a nicely typed query, and, from Theorem 8.4.12, it follows that `append` $\cup$ { `app(s, t, u)`} is unification free.

(ii) `append` can be used not only for concatenating two lists, but also for splitting a list in two. This is reflected by the adoption of the following typing:

`app` : $Pt \times Pt \times U$

Again, `append` is nicely typed, and the head of both clauses are U-safe$^-$. Theorem

8.4.12 yields that, for disjoint terms `u, v ,t`, where `u` and `v` are variables or pure terms, `append` $\cup$ { `app(u, v, t)`} is unification free.

(iii) Let us now consider the following permutation program:

```
perm(Xs, Ys) ← Ys is a permutation of the list Xs.

perm(Xs, [X | Ys]) ←
    app1(X1s, [X | X2s], Xs),
    app2(X1s, X2s, Zs),
    perm(Zs, Ys).
perm([], []).
```

augmented by the `app1` and `app2` programs.

Where both `app1` and `app2` are renamings of the `append` program; we use here two distinct renamings in order to adopt two different types, namely

`app1` : $Pt \times Pt \times U$
`app2` : $U \times U \times Pt$

By the previous example we have that both `app1` and `app2` are nicely typed. Let us consider the following typing:

`perm` : $U \times Pt$

It is easy to check that `perm` is nicely typed, and that both clause's heads are U-safe^{-}. Hence, when `u` a variable or a pure term disjoint from `t`, `permutation` $\cup$ { `perm(t, u)`} is unification free. $\square$

More examples of programs and typings that satisfy the hypothesis of Theorem 8.4.12 are provided by the list in Section 8.7.

8.5 Avoiding Unification using also the mode "input"

In the previous Section we have been using only the modes U and output. Therefore the parameter passing from the selected atom to the head of the input clause was always done via the U-positions. As we remarked before, this has the advantage of flexibility, as there is no assumption on the data structure used. However, in some cases, if we can be more precise about the kind of data structure is being used, we'll be able to broaden the range of of programs and queries that we can prove to be unification-free. Consider for instance the well-known `member` program.

```
member(Element, List) ←
    Element is an element of the list List.

member(X, [X | Xs]).
member(X, [Y | Xs]) ← member(X, Xs).
```

It is easy to check (see Example 8.6.7 for a formalization of this statement) when the typing is `member` : $Pt \times U$, `member` satisfies the conditions of Theorem 8.4.12, therefore if `s` is in *Pt* and `t` is disjoint from `s`, then `member` $\cup$ { `member(s, t)` } is unification-free. On the other hand, it is also easy to (manually) check that if we know that `t` is ground, then we can drop the assumption that `s` is in *Pt*: `member` $\cup$ { `member(s, t)` } is still unification-free. In order to capture this situation, we need an extension of Theorem 8.4.12 that is applicable when the typing adopted is `member` : $U \times Ground$. In this situation, according to the convention of Definition 8.3.5, the second position is moded as *input*.

In this Section we provide the tools necessary to handle the presence of input positions. First notice that by Definition 8.3.5, the input positions of an atom are exactly the ones that are not typed *Var, Pt* or *U*. Consequently, considering also input positions tantamounts to considering also term_types which are not in { *Var, Pt, U* }.

The new types we interested in are *monotonic*, that is, they are closed under substitution. This property will simplify a lot the discussion.

Definition 8.5.1 We call a term_type T *monotonic* iff, for each substitution θ

- $t \in T$ implies $t\theta \in T$ □

From now on we make the following Assumption.

Assumption 8.5.2

- with the exception of term_types *Var, Pt*, all the term_types we refer to are monotonic. □

Notice that types *Ground, U* are by definition monotonic. Recall that we assume also that the type associated to a relation symbol p is always of the form $p: T_1 \times \ldots \times T_n$. The basic implication of Assumption 8.5.2 is then that the T_is corresponding to the input positions are always monotonic term_types.

Sequential Matching via Generic Expressions

Generic expressions were introduced by Apt-Etalle in [7], and can be used to obtain a new interesting condition for solvability by matching. For example, assume the standard list notation and consider a term $t = [x, y|z]$ with x, y and z variables. Note that (despite the fact that t is not a pure term), whenever a list l unifies with t, then l is an instance of t, i.e $l = t$ is solvable by matching.

Thus solvability by matching can be sometimes deduced from the shape of the considered terms. In this subsection we will follow closely Apt and Etalle [7], and we begin with the following Definition.

Definition 8.5.3 Let T be a term_type. A term t is a *generic expression* for T if for every $s \in T$ disjoint with t, if s unifies with t then s is an instance of t. □

In other words, t is a generic expression for the term_type T iff all left-right disjoint equations $s = t$, where $s \in T$, are solvable by matching.

Example 8.5.4

- 0, $s(x)$, $s(s(x))$, ... are generic expressions for the term_type *Nat*,

- $[\,]$, $[x]$, $[x|y]$, $[x,y|z]$, ... are generic expressions for the term_type *List*. □

Note that a generic expression for T needs not to be a member of T.

Next, we provide some important examples of generic expressions which will be used in the sequel. Here and in the following we call a (term_) type T *ground* if all its elements are ground, and *non-ground* if some of its elements is non-ground; consequently the *non-ground* positions of an atom H are those positions of H whose associated term_type is not a *ground* type.

Lemma 8.5.5 Let T be a term_type. Then

- variables are generic expressions for T,
- the only generic expressions for the term_type U are variables,
- if T does not contain variables, then every pure term is a generic expression for T,
- if T is ground, then every term is a generic expression for T.

Proof. Clear. □

When the term_types are defined by structural induction (as for example in Bronsard, Lakshman and Reddy [23] or in Yardeni, T. Frühwirth and E. Shapiro [98]), then it is easy to characterize the generic expressions for each type by structural induction.

We can now provide another simple test for establishing solvability by matching.

Lemma 8.5.6 (Matching 2, [7]) Consider two disjoint atoms A and H with the same relation symbol. Suppose that

- A is correctly typed,
- the positions of H are filled in by mutually disjoint terms and each of them is a generic expression for its positions type.

Then $A = H$ is solvable by matching. Moreover, if A and H are unifiable, then a substitution θ with $Dom(\theta) \subseteq Var(H)$ exists such that $A = H\theta$.

Proof. Clear. □

Consider again the case of a selected atom A and the head H of a clause used to resolve A. In presence of arbitrary term_types, in order to apply the Matching 2 Lemma 8.5.6 to the subset of $A = H$ corresponding to the input positions, we have to impose some restrictions on H.

Definition 8.5.7 An atom H is called *input safe* if each term t filling in a non-ground input position of H satisfies the following two conditions:

(i) t is a generic expression for this positions type,
(ii) t is disjoint from all the other terms occurring in the non-ground input positions of H. □

We also need to upgrade the Definition of U-safe$^-$ atom in order to take into account the presence of input positions.

Definition 8.5.8 (U-safe) An atom H is called *U-safe* if for each term t filling in one of its U-positions one of the following two conditions holds:

(i) t is a variable or a pure term and it is disjoint from the terms occurring in the input and the other U-positions of H;
(ii) each variable occurring in t appears also in an *input* position of H of *ground* type. □

Note that when there are no input positions this Definition coincides with the one of U-safe$^-$ atom.

The above two conditions reflect two different way in which we can apply the Matching 1 Lemma 8.4.1 to the U-positions of $A = H$: the first conditions ensures that the term in the position we are considering is a variable or a pure term, and that it is not affected by the matching of the input and the other U-positions. On the other hand the second makes sure that after having matched the input positions of $A = H$, the term will be ground, so that the Matching 1 Lemma will still be applicable.

The above Definitions allow us to generalize Lemma 8.4.4 to the case in which we have also input positions.

Lemma 8.5.9 (Sequential Matching 2) Consider two disjoint atoms A and H with the same relation symbol. If

- A is correctly typed and output independent,
- H is input safe and U-safe,

Then there exists a permutation π such that $A = H$ is solvable by sequential matching wrt π.

In particular, $A = H$ is solvable by sequential matching wrt any permutation of $1,\dots,n$ such that, according to the order given by $\pi(1),\dots,\pi(n)$, we have that the non-ground input positions of p come first, the ground input positions come next, the U-positions come after them and the output positions come last.

Proof. Suppose that $A = H$ is unifiable, we can then assume that A and H are equal respectively to $p(s_1,\dots,s_n)$ and $p(t_1,\dots,t_n)$, where $s_1,\dots,s_n,\ t_1,\dots,t_n$ have been reordered in such a way that non-ground input positions come first (on the left), the ground (input) positions come next, the U-positions come third and the output positions are the rightmost ones.

We now need to prove that $s_1 = t_1,\dots,s_n = t_n$ is solvable by sequential matching, that is we need to find $\theta_1,\dots,\theta_n$ such that each θ_i is a match of $(s_i = t_i)\theta_1\dots\theta_{i-1}$.

Let T_i be the term_type associated to the i-th position of p. Each equation $s_i = t_i$ corresponds to one position of $A = H$, we now distinguish four cases upon the kind of position the equation $s_i = t_i$ corresponds to.

First we consider the case when $s_i = t_i$ corresponds to a *non-ground* input position. Since H is input safe, t_i is a generic expression for T_i and $Var(t_i) \cap Var(\theta_1\dots\theta_{i-1}) = \emptyset$, so $t_i\theta_1\dots\theta_{i-1}$ is still a generic expression for T_i and, since $\theta_1\dots\theta_{i-1}$ are relevant, $t_i\theta_1\dots\theta_{i-1}$ is disjoint from $s_i\theta_1\dots\theta_{i-1}$. Moreover, A is correctly typed, thus s_i belongs to T_i, and, since by Assumption 8.5.2, T_i is monotonic, $s_i\theta_1\dots\theta_{i-1}$ belongs to

T_i as well. From the Matching 2 Lemma 8.5.9 it follows then that $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching.

Second, we consider the case when $s_i = t_i$ corresponds to a *ground* input position. Since A is correctly typed, s_i is a ground term. From the Matching 1 Lemma 8.4.1 it follows then that $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching. Moreover, if $t_j, \ldots, t_k$ are the terms found in the ground input position of H, we also have that $(t_j, \ldots, t_k)\theta_1 \ldots \theta_k$ are ground terms.

Third, if $s_i = t_i$ is found in a U-position then, depending on which of the two conditions of U-safeness is satisfied we have that: (i) t_i is a variable or a pure term and $Var(t_i) \cap Var(\theta_1 \ldots \theta_{i-1}) = \emptyset$, so $t_i\theta_1 \ldots \theta_{i-1}$ is still a variable or a pure term and by the Matching 1 Lemma 8.4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching; (ii) $Var(t_i) \subseteq Var(t_j, \ldots, t_k)$ and, by the order hypothesis, the equations $1, \ldots, k$ have already been processed, from what noticed before it follows that $t_i\theta_1 \ldots \theta_{i-1}$ is a ground term, and again, by the Matching 1 Lemma 8.4.1, $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching.

Finally, if $s_i = t_i$ is found in an output position then s_i is a variable or a pure term and, since A is output independent, $Var(s_i) \cap Var(\theta_1, \ldots, \theta_{i-1}) = \emptyset$. So $s_i\theta_1, \ldots, \theta_{i-1}$ is still a variable or a pure term, and by the Matching 1 Lemma 8.4.1 $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ is solvable by matching. □

This allows us to generalize Theorem 8.4.6. Recall that an LD-derivation is called *i/o driven* if all atoms selected in it are correctly typed and output independent.

Theorem 8.5.10 Suppose that

- the head of every clause of P is input safe and U-safe,
- all LD-derivations of $P \cup \{Q\}$ are i/o driven.

Then $P \cup \{Q\}$ is unification free. □

Taking care of the input positions: Well-Typed Programs

In order to apply Theorem 8.5.10, we need again to find some conditions sufficient to ensure that the LD-derivations will be i/o-driven. As in the previous Section, the output positions will be taken care of by the fact that the programs we consider are nicely typed. Consequently, our concern is now to guarantee that the selected atoms will be correctly typed in their input positions. In presence of arbitrary term_types, the task is not trivial.

Substantially, the approach that we follow here is originally due to Bossi and Cocco [17], where it was used for proving partial correctness. We use the concept of Well-Typed program, which was introduced by Bronsard, Lakshman and Reddy [23], and we adopt the notation of Apt [4].

We begin with the following Definition, where we assume that the input positions of atom are grouped on the left.

Definition 8.5.11 Let $rel(A) : T_1 \times \ldots \times T_n$ be the type associated to the relation symbol of the atom A. Assume that the input positions of A are its leftmost m positions, then

- the *pre-type* for $rel(A)$ is the type

$$pre_{rel(A)} : T_1 \times \ldots \times T_m \times U \times \ldots \times U$$

and it is obtained by projecting $rel(A) : T_1 \times \ldots \times T_n$ onto its input positions. □

The pre-type of $rel(A)$ is then uniquely determined by the type of $rel(A)$; therefore from the assumption that each relation symbol has always a type associated to it it follows that each relation symbol has automatically also a pre-type associated to. The advantage of referring to the pre-type instead of the type is that by Assumption 8.5.2 the pre-type is always monotonic.

To give the definition of Well-Typed program we need two more notions.

Definition 8.5.12 Let $A_1, \ldots, A_{n+1}$ be atoms and $\mathcal{T}_1, \ldots, \mathcal{T}_{n+1}$ be monotonic types

- By a *type judgement* we mean a statement of the form

$$\models A_1 \in \mathcal{T}_1 \wedge \ldots \wedge A_n \in \mathcal{T}_n \Rightarrow \; A_{n+1} \in \mathcal{T}_{n+1}$$

which denotes that, for all substitutions θ, $Dom(\theta) = Var(A_1, \ldots, A_n)$:

$$\text{if} \;\; A_1\theta \in \mathcal{T}_1 \wedge \ldots \wedge A_n\theta \in \mathcal{T}_n \text{ then } \; A_{n+1}\theta \in \mathcal{T}_{n+1}$$

□

Recall that in order to apply Theorem 8.5.10, we have to prove that each selected atom belongs to its pre-type; to do this we use type judgements and associate to each relation symbol also a *post-type*.

Definition 8.5.13 A *post-type* for a relation symbol p, is a monotonic type for p. □

From now on we assume that each relations symbol has, together with the type, also a post-type associated to it.

As opposed to the type, we want the post-type to contain information about the state of the arguments of a query *after* the query itself has been successfully resolved. For example, consider again the program `append`. A typical typing for it is `app`: *List* $\times$ *List* $\times$ *Pt*[1]. This formalizes the idea that when and atom of the form `app(s, t, u)` is *selected*, we expect `s` and `t` to be variables and `u` to be a variable, or, at most, a pure term. On the other hand, we require the post-type to hold some knowledge over the situation of `s`, `t` and `u` after that the query `app(s, t, u)` has been successfully resolved. In this situation a natural post-type would be $post_{\texttt{app}}$: *List* $\times$ *List* $\times$ *List*, indicating that, after `app(s, t, u)` has succeeded, we also expect `u` to be a list. Notice also that when the type adopted is the above one, the the pre-type is $pre_{\texttt{app}}$: *List* $\times$ *List* $\times$ U.

In the following we write $pre(A)$ (resp. $post(A)$) as shorthand for $A \in pre_{rel(A)}$ (resp. $A \in pre_{rel(A)}$), where $pre_{rel(A)}$ and $post_{rel(A)}$ are the pre- and post-type of the relation symbol of A.

[1]This is a slight extension of the "natural" typing `app`: *List* $\times$ *List* $\times$ Var that we mentioned in Sections 8.3 and 8.4

Definition 8.5.14

- A query $A_1, \ldots, A_n$ is called *well-typed* if, for $j \in [1, n]$,

$$\models post(A_1) \wedge \ldots \wedge post(A_{j-1}) \;\Rightarrow\; pre(A_j).$$

- A clause $H \leftarrow B_1, \ldots, B_n$ is called *well-typed* if, for $j \in [1, n+1]$,

$$\models pre(H) \wedge post(B_1) \wedge \ldots \wedge post(B_{j-1}) \;\Rightarrow\; pre(B_j),$$

  where $pre(B_{n+1}) := post(H)$.
- A program is called *well-typed* if every clause of it is. □

Thus, a query is well-typed if

- the pre-type of an atom can be deduced from the post-types of previous atoms.

And a clause is well-typed if

- ($j \in [1, n]$) the pre-type a body atom can be deduced from the pre-type of the head and the post-types of the previous body atoms,
- ($j = n + 1$) the post-types of the head can be deduced from the pre-type of the head and the post-types of the body atoms.

In particular a query A is well-typed iff $\models pre(A)$, while a unit clause $A \leftarrow$ is well-typed iff $\models pre(A) \Rightarrow post(A)$.

The following result states the persistence of the notion of being well-typed (see Bossi-Cocco [17] or an account of it Apt-Marchiori [10]).

Lemma 8.5.15 (Persistence) An LD-resolvent of a well-typed query and a well-typed clause that is variable disjoint with it, is well-typed. □

This brings us to the following conclusion.

Corollary 8.5.16 Let P and Q be well-typed, and let ξ be an LD-derivation of $P \cup \{Q\}$. Then every atom selected in ξ is correctly typed in its input positions.

Proof. A variant of a well-typed clause is well-typed and for a well-typed query $A_1, \ldots, A_n$ we have $\models pre(A_1)$. □

Avoiding Unification with Well+Nicely Typed Programs

Recall that in order to prove that $P \cup \{Q\}$ is unification-free using Theorem 8.4.6 we are looking again for conditions which imply that all the LD-derivations starting in Q are i/o driven: we want that the selected atom is correctly typed and output independent.

The combination of the concepts of being well-typed and being nicely typed allows us to deal with all the cases in which the types used satisfy Assumption 8.5.2: well-typedness takes care of the input position, while nicely typedness takes care of the output ones.

Lemma 8.5.17 Suppose that

- P and Q are nicely typed and well-typed.

Then all LD-derivations of $P \cup \{Q\}$ are i/o driven.

Proof. It follows from Corollaries 8.5.16 and 8.4.10. □

This brings us to the main result of this chapter.

Theorem 8.5.18 (Main) Suppose that

- P and Q are nicely typed and well-typed,
- the head of every clause of P is input safe and U-safe

Then $P \cup \{Q\}$ is unification free.

Proof. From Lemma 8.5.17 and Theorem 8.5.10. □

In particular, from the Sequential Matching 2 Lemma 8.5.9 it follows that each of the equations $A = H$ considered in the LD-derivations can be solved by sequentially matching (one by one) each of the atoms positions, provided that we observe the following order: first the nonground input positions, then the ground input positions, after that the U-positions and finally the output ones. In the Appendix we'll show how we can improve on this result by grouping some positions under the same match.

It is not difficult to check that this Theorem 8.5.18 generalizes our previous result, Theorem 8.4.12. Indeed if the program P and the query Q satisfy the conditions of Theorem 8.4.12, then, since the atoms have no input positions, we have that the heads of the clauses of P are trivially input-safe and, by assigning to each predicate symbol p the trivial post-type $p : U \times \ldots \times U$, we have that P and Q are well-typed. Therefore P and Q satisfy the hypothesis of Theorem 8.5.18 as well.

Example 8.5.19 Consider now the program `permutation sort` which is often used as a benchmark program.

```
ps(Xs, Ys) ← permutation(Xs, Ys), ordered(Ys).

permutation(Xs, [Y | Ys]) ←
   select(Y, Xs, Zs),
   permutation(Zs, Ys).
permutation([], []).

select(X, [X | Xs], Xs).
select(X, [Z | Xs], [Z | Zs]) ← select(X, Xs, Zs).

ordered([]).
ordered([X]).
ordered([X, Y | Xs]) ← X ≤ Y, ordered([Y| Xs]).
```

Let us associate to it the following typing,

		type	post-type
`ps`	:	$List \times Pt$	$List \times List$
`permutation`	:	$List \times Pt$	$List \times List$
`select`	:	$Pt \times List \times Pt$	$U \times List \times List$
`ordered`	:	$List$	$List$

Now, `permutation sort` is well-typed and nicely typed. Moreover, the heads of all clauses are input safe and U-safe[2]. By the Main Theorem 8.5.18 we get that for a list `s` and a disjoint with it variable or pure term `t`, `permutation sort` $\cup$ { `ps(s, t)`} is unification free.

Observe that the terms `[X]` and `[X, Y | Xs]`, filling in the input positions of, respectively, the first and the third clause defining the relation `ordered`, are generic expressions for *List*, but are not pure terms. In a sense we could say that `[X]` and `[X, Y | Xs]` are nontrivial generic expressions. □

8.6 A simpler special case: Ground input positions

Sometimes, a lot of the machinery needed by Theorem 8.5.18 is actually superfluous. In particular, this happens when the input positions are all of ground type. In this case, instead of requiring the program to be well-typed, we can use the more restrictive concept of well-moded program. This has two relevant advantages:

First, that we do not need to associate a post-type to each relation symbol.

Second, while checking that a program is well-typed is an algorithmically intractable problem, testing well-modedness can be done in polynomial (quadratic) time. A discussion on the algorithmic tractability of the concepts used in this chapter is reported in Section 8.6.1.

In this Section we'll assume that the only term_type used for the input positions in *Ground*. Informally, this means that the information we pass to the program consists always of ground terms. By Definition 8.3.5 this is equivalent to assuming that we use types which are built using only the following term_types: *Ground, Pt, Var, U.*

Well-Moded programs

The concept of Well-Moded program is essentially due to Dembinski and Maluszynski [35]; here we make use of the elegant formulation of Rosemblueth [85] and of the same notation of [7]. In particular, when writing an atom as $p(\tilde{u}, \tilde{v})$, we now assume that $\tilde{u}$ is a sequence of terms filling in the input positions of p and that $\tilde{v}$ is a sequence of terms filling in the output and the U-positions of p (notice that this shorthand is different from the one used for Definition 8.4.7).

Definition 8.6.1

[2]The latter statement is trivial, as there are no U-positions: the fact that U appears in a post-type is of no relevance here.

- A query $p_1(\tilde{s}_1, \tilde{t}_1), \ldots, p_n(\tilde{s}_n, \tilde{t}_n)$ is called *well-moded* if for $i \in [1, n]$

$$Var(\tilde{s}_i) \subseteq \bigcup_{j=1}^{i-1} Var(\tilde{t}_j).$$

- A clause

$$p_0(\tilde{t}_0, \tilde{s}_{n+1}) \leftarrow p_1(\tilde{s}_1, \tilde{t}_1), \ldots, p_n(\tilde{s}_n, \tilde{t}_n)$$

  is called *well-moded* if for $i \in [1, n+1]$

$$Var(\tilde{s}_i) \subseteq \bigcup_{j=0}^{i-1} Var(\tilde{t}_j).$$

- A program is called *well-moded* if every clause of it is. □

Thus, a query is well-moded if

- every variable occurring in an input position of an atom ($i \in [1, n]$) occurs in a non-input position of an earlier ($j \in [1, i-1]$) atom.

And a clause is well-moded if

- ($i \in [1, n]$) every variable occurring in an input position of a body atom occurs either in an input position of the head ($j = 0$), or in a non-input position of an earlier ($j \in [1, i-1]$) body atom,
- ($i = n+1$) every variable occurring in an non-input position of the head occurs in an input position of the head ($j = 0$), or in an output position of a body atom ($j \in [1, n]$).

It is important to notice that the concept of a well-moded program (resp. query) is a particular case of that of a well-typed program. Indeed, if the only term_type used for the input positions is *Ground*, and the post-type associated to each relation symbol p is $p : Ground \times \ldots \times Ground$, then the notions of a well-typed program (resp. query) and a well-moded program (resp. query) coincide.

The following Lemma states the persistence of the notion of being well-moded. A proof of it can be found in Apt and Marchiori [7].

Lemma 8.6.2 An LD-resolvent of a well-moded query and a disjoint with it well-moded clause is well-moded. □

The next result is originally due to Dembinski and Maluszynski and follows directly from the definition of well-moded program.

Corollary 8.6.3 Let P and Q be well-moded, and let ξ be an LD-derivation of $P \cup \{Q\}$. All atoms selected in ξ contain ground terms in their input positions. □

Avoiding Unification with Well-Moded Nicely Typed Programs

As we anticipated at the beginning of this Section, here we assume that the only term_type used for the input position is *Ground*, this is equivalent to making the following

Assumption 8.6.4 In this subsection we each predicate symbol has a type associated to it of the form $p : T_1 \times \ldots \times T_n$, where for $i \in [1, n]$, $T_i \in \{Ground, Var, Pt, U\}$. □

Once again we are going to use Theorem 8.4.6 for proving that $P \cup \{Q\}$ is unification-free. Therefore we are looking again for conditions which imply that all the LD-derivations starting in Q are i/o driven: the selected atoms in a LD-derivation need to be correctly typed and output independent. As in the previous two Sections, the concept of being nicely typed will take care of the output positions.

Since we are assuming that the input positions are always of ground type, from Corollary 8.6.3 it follows that well-modedness is what we need for taking care of the input positions.

Lemma 8.6.5 If Assumption 8.6.4 is satisfied and

- P and Q are nicely typed and well-moded.

Then all LD-derivations of $P \cup \{Q\}$ are i/o driven.

Proof. Let A be a selected atom in an LD-derivation of $P \cup \{Q\}$. By Corollary 8.6.3 the input positions of A are correctly typed, and by Corollary 8.4.10, A is correctly typed in its output positions is output independent. □

This, together with Theorem 8.4.6, brings us to the following conclusion.

Theorem 8.6.6 If Assumption 8.6.4 is satisfied and

- P and Q are nicely typed and well-moded,
- the head of every clause of P is U-safe

Then $P \cup \{Q\}$ is unification free.

Proof. It follows directly from Lemma 8.6.5 and Theorem 8.4.6. □

It is easy to check that this is a special case of Theorem 8.5.18: if P and Q satisfy its hypothesis, then P and Q are well-moded and, as we mentioned before, well-moded programs (and queries) are a special case of well-typed programs in which the only term_type used for the input positions is *Ground*. Therefore P and Q satisfy also the condition of being well-typed, moreover, we also have that the heads of P are (trivially) input safe. Consequently P and Q satisfy the hypothesis of Theorem 8.5.18 as well.

Example 8.6.7
(i) First, let us go back to what we stated at the beginning of Section 8.5, and let us consider again the program `member`. With the typing `member`: $U \times Ground$, `member` is well-moded and (trivially, as there are no output positions) nicely typed; moreover,

all clause's heads are U-safe. By Theorem 8.6.6 if `t` is a ground term, then, for any `s`, `member` $\cup$ { `member(s, t)`} is unification free.

Let us compare this with what we could have obtained by using the result (namely, Theorem 8.4.12) given in the Section 8.4. Without using input positions we can prove that, when the following type is used:

`member` : $Pt \times U$

then `member` is nicely typed and all clause's heads are U-safe. By Theorem 8.4.12 this implies that if `s` is a variable or a pure term disjoint from `t`, then `member` $\cup$ { `member(s, t)`} is unification free. In this case, the advantage of Theorem 8.6.6 over Theorem 8.4.12 is that we can allow `s` to be any term. The price we have to pay for this is that Theorem 8.6.6 requires `t` to be ground. Symmetrically, Theorem 8.4.12 imposes no conditions on `t` (which can be then a nonground list, or any other term) but requires `s` to be a variable or a pure term.

Notice also that, when the above types are used, Theorem 8.6.6 is not applicable, as the program is not well-moded. This shows that Theorem 8.6.6 is not more general that Theorem 8.4.12.

(ii) Consider now the `MapColor` program:

```
color_map(Map, Colors) ←
    Map is correctly typed using Colors.

color_map([Region | Regions], Colors) ←
    color_region(Region, Colors),
    color_map(Regions, Colors),
color_map([], _ ).

color_region(Region, Colors) ←
    Region and its neighbors are correctly colored using Colors.

color_region(region(Name, Color, Neighbors) , Colors) ←
    select(Color, Colors, ColorsLeft),
    subset(Neighbors, ColorsLeft).

select(X, Xs, Zs) ←
    Zs is the result of deleting one occurrence of X from the list Zs.

select(X, [X | Xs], Xs).
select(X, [Z | Xs], [Z | Zs]) ← select(X, Xs, Zs).

subset(Xs, Ys) ←
    each element of the list Xs is also an element of the list Ys.

subset([X | Xs], Ys) ←member(X, Ys), subset(Xs, Ys).
subset([] , _ ).
```

augmented by the `member` program.

Let us associate to it the following typing:

$$\begin{array}{rcl}
\texttt{color_map} & : & U \times Ground \\
\texttt{color_region} & : & U \times Ground \\
\texttt{select} & : & U \times Ground \times Pt \\
\texttt{subset} & : & U \times Ground \\
\texttt{member} & : & U \times Ground
\end{array}$$

It is straightforward to check that with the above typing, `MapColor` is well-moded and nicely typed. Since the head of all clauses are U-safe, by Theorem 8.6.6 we have that, if `t` is a ground term, then, for any `s`, `color_map` $\cup$ { `color_map(s, t)`} is unification free. □

It is worth noticing that the U-positions have been used in (at least) two opposite ways: in Section 8.4 we they were actually used as "input" positions, in the sense that they were used to transfer information from the selected atom to the head of the clause used to resolve it, while in Section 8.6 they were more used as "output". This becomes noticeable in the moment that we compare Example 8.4.13 with Example 8.6.7. However, it should be mentioned that this distinction is not always so clear: consider for instance the program `select` (which is a subprogram of the above `MapColor`): A query `select(s, t, u)` can be used in two main ways: to delete the element `s` from the list `t` and report the result in `u`, or as a generalized `member` program, to report in `s` an element of `t`, and in `u` the remains of the list. In the first case the first position is used as "input", in the second as "output", but for both cases we can simply use the typing `select` : $U \times Ground \times Pt$. In this case the mode U takes care of the ambivalence of the first position. Notice also that when we adopt this typing the hypothesis of Theorem 8.6.6 are satisfied, therefore if `t` is ground, `u` is in Pt and `s` is disjoint from `s` then `select`$\cup$`select(s, t, u)` is unification-free.

8.6.1 Comparing Theorems 8.4.12, 8.5.18 and 8.6.6: efficiency issues

Theorem 8.5.18 is a generalization of Theorems 8.4.12 and 8.6.6, but the latter two are much more suitable for being used in an automatic way.

In fact, it is worth noticing that the applicability conditions of Theorems 8.4.12 and 8.6.6 can be statically and efficiently tested: in order to check that a program is nicely typed, well-moded and the head of its clauses are input safe, one can easily find some naive algorithms whose complexity is quadratic in the size of the clauses and linear in the number of clauses in a program. Indeed, all three concepts require procedures like the following one.

```
for each clause cl in P do
    for each variable v occurring in cl do
        begin
           check that all the other occurrences of v in cl satisfy the
           required conditions (this require re-scanning cl)
        end
```

On the other hand, to test the hypothesis of Theorem 8.5.18 one needs to check if some *type judgements* hold, and this is a much more complex problem, in fact, for artificially built types, it can even be undecidable. Aiken and Lakshman in [2] have investigated the problem of checking type judgements for monotonic types: they prove that it is EXPTIME-hard and they state that no upper bound is known, moreover, they show that also in the case that we use only *discriminative types*[3] then the problem has a a lower complexity bound of PSPACE, and a upper bound of NEXPTIME. In other words, even in this more restrictive case, the problem remains highly untractable.

Thus, checking the conditions of Theorems 8.4.12 and 8.6.6 is much simpler than checking the ones of Theorem 8.5.18, moreover, by checking the list in Section 8.7, one can easily realise that the practical cases in which Theorem 8.5.18 is really useful are a minority: in most cases Theorems 8.4.12 and 8.6.6 are sufficient for our purposes.

8.7 What have we done and what have we not done

What have we done: the List

To apply the established results to a program and a query, one needs to find appropriate typings for the considered relations such that the conditions of one of the Theorems 8.4.12, 8.5.18 or 8.6.6, are satisfied. In the table below several programs taken from the book of Sterling and Shapiro [94] are listed. For each program it is indicated for which typings these theorems are applicable.

In programs which use difference-lists we replace "\" by ",", thus splitting a position filled in by a difference-list into two positions. Because of this change in some relations additional arguments are introduced, and so certain clauses have to be modified in an obvious way. For example, in the parsing program on page 258 each clause of the form `p(X)` $\leftarrow$ `r(X)` has to be replaced by `p(X,Y)` $\leftarrow$ `r(X,Y)`. Such changes are purely syntactic and they allow us to draw conclusions about the original program.

We also report between parenthesis typings which are "subsumed" by other typings in the list, that is, typings for which there exists another typing which is more

[3] a *discriminative type* is a type built using to some specific rules which include a fixpoint set construction; according to Aiken and Lakshman "The important restriction of discriminative set expressions are that no intersection operation is allowed and all union are formed from expressions with distinct outermost constructor". In any case, discriminative types are descriptive enough to be able to handle all the examples presented here.

general. We report them here because they provide further examples of typings wrt which these programs are (unification-free and) well-typed (or well-moded).

program	page	Thm.	Typing	
member	45	8.4.12	$Pt \times U$	
		8.6.6	$U \times Ground$	
		(8.5.18)	$(Pt \times List)$	
prefix	45	8.4.12	$Pt \times U$	
		8.6.6	$Ground \times Ground$	
		(8.6.6)	$(Pt \times Ground)$	
		(8.5.18)	$(Pt \times List)$	
suffix	45	8.4.12	$Pt \times U$	
		8.6.6	$Ground \times Ground$	
		(8.6.6)	$(Pt \times Ground)$	
		(8.5.18)	$(Pt \times List)$	
naive reverse	48	8.4.12	$U \times Pt$	
		8.6.6	$Ground \times U$	
		(8.5.18)	$(List \times Pt)$	
reverse-accum.	48	8.4.12	$U \times Pt,$	$U \times U \times Pt$
		8.6.6	$Ground \times U,$	$Ground \times Ground \times U$
		(8.5.18)	$(List \times Pt,$	$List \times List \times Pt)$
delete	53	8.5.18	$Ground \times U \times Pt$	
		8.5.18	$Ground \times U \times Ground$	
		(8.6.6)	$(Ground \times Ground \times Pt)$	
select	53	8.4.12	$Pt \times U \times Pt$	
		8.4.12	$U \times Pt \times U$	
		8.6.6	$U \times Ground \times Pt$	
		8.6.6	$Ground \times Ground \times Ground$	
		(8.6.6)	$(Ground \times Ground \times Pt)$	
		(8.5.18)	$(Pt \times List \times Pt)$	
insertion sort	55	8.4.12	$s : U \times Pt,$	$i : U \times U \times Pt$
		(8.6.6)	$(s : Ground \times Pt,$	$i : Ground \times Ground \times Pt)$
		(8.5.18)	$(s : List \times Pt,$	$i : U \times List \times Pt)$
quicksort	56	8.4.12	$q : U \times Pt,$	$p : U \times U \times Var \times Var$
		(8.6.6)	$(q : Ground \times Pt,$	$p : Ground \times Ground \times Var \times Var)$
		(8.5.18)	$(q : List \times Pt,$	$p : U \times List \times Pt)$

tree-member	58	8.4.12	$Pt \times U$	
		8.6.6	$U \times Ground$	
		8.6.6	$Ground \times Ground$	
		(8.5.18)	$(Pt \times BinTree)$	
isotree	58	8.4.12	$U \times Pt$	
		8.4.12	$Pt \times U$	
		8.6.6	$Ground \times Ground$	
		(8.6.6)	$(Ground \times Pt)$	
		(8.6.6)	$(Pt \times Ground)$	
		(8.5.18)	$(BinTree \times Pt)$	
		(8.5.18)	$(Pt \times BinTree)$	
substitute	60	8.5.18	$U \times U \times Ground \times Pt$	
		8.5.18	$U \times U \times Pt \times Ground$	
		8.5.18	$U \times U \times Ground \times Ground$	
		(8.6.6)	$(Ground \times Ground \times Ground \times Pt)$	
		(8.6.6)	$(Ground \times Ground \times Pt \times Ground)$	
pre-order	60	8.4.12	$U \times Pt$	
		8.6.6	$Ground \times U$	
		(8.5.18)	$(BinTree \times Pt)$	
in-order	60	8.4.12	$U \times Pt$	
		8.6.6	$Ground \times U$	
		(8.5.18)	$(BinTree \times Pt)$	
post-order	60	8.4.12	$U \times Pt$	
		8.6.6	$Ground \times U$	
		(8.5.18)	$(BinTree \times Pt)$	
polynomial	62	8.6.6	$Ground \times U$	
derivative	63	8.6.6	$Ground \times U \times Pt$	
		8.6.6	$Ground \times U \times Ground$	
hanoi	64	8.4.12	$U \times U \times U \times U \times Pt$	
		8.6.6	$U \times Ground \times Ground \times Ground \times U$	
reverse_dl	244	8.4.12	$r : U \times Pt,$	$r\_dl : U \times Pt \times U$
		8.6.6	$r : Ground \times U,$	$r\_dl : Ground \times U \times Ground$
		(8.5.18)	$(r : List \times Pt,$	$r\_dl : List \times Pt \times List)$

dutch	246	8.4.12	$dutch : U \times Pt$,	$di : U \times Pt \times Pt \times Pt$
		8.6.6	$dutch : Ground \times U$,	$di : Ground \times Pt \times Pt \times Pt$
dutch_dl	246	8.4.12	$dutch : U \times Pt$,	$di : U \times Pt \times Pt \times Pt \times U$
parsing	258	8.6.6	all $Ground \times U$	

What have we not done

Still, there are some natural programs that when executed do not require unification, while they cannot be proven unification-free using our method. We are aware of the following two examples: `quicksort_dl` and `flatten_dl` [94, pag. 244, 241].

First, let us consider `quicksort_dl`.

```
qs(Xs, Ys) ← qs_dl(Xs, Ys, []).
qs_dl([X | Xs], Ys, Zs ) ←
   partition(X, Xs, Littles, Bigs),
   qs_dl(Littles, Ys, [X|Y1s]),
   qs_dl(Bigs, Y1s, Zs).
qs_dl([], Xs, Xs).

partition(X, [Y | Xs], [Y | Ls], Bs) ← X > Y, partition(X, Xs, Ls, Bs).
partition(X, [Y | Xs], Ls, [Y | Bs]) ← X ≤ Y, partition(X, Xs, Ls, Bs).
partition(X, [], [], []).
```

By looking at the trace of the program, it is easy to see that, if `t` is a list and `s` is a variable disjoint with `t`, then `quicksort_dl`$\cup\{$ `qs(t, s)` $\}$ is unification free. Indeed, if we use the following types:

$$\begin{array}{rcl} \texttt{qs} & : & List \times Var \\ \texttt{qs\_dl} & : & List \times Var \times U \\ \texttt{partition} & : & U \times List \times Var \times Var \end{array}$$

then we have that the heads of all the clauses are input safe and U-safe, moreover, we can check "by hand" that, if $\{$ `qs(t, s)` $\}$ is correctly typed and output independent, all LD-derivations of `quicksort_dl` $\cup$ $\{$ `qs(t, s)` $\}$ are i/o driven, therefore, by Theorem 8.5.10, `quicksort_dl` $\cup$ $\{$ `qs(t, s)` $\}$ is unification-free. The problem here is that the program is not nicely typed: `Y1s` appears first in the U-position of `qs_dl(Littles, Ys, [X|Y1s])` and then in the output position of `qs_dl(Bigs, Y1s, Zs)`, therefore, with the tools in our possession, we cannot *prove* that the derivations are i/o driven, in particular we can't show that each time that an atom of the form `qs_dl(t, s, r)` is selected, `s` will be a variable[4].

Now, let us consider the program `flatten_dl`.

[4] It may be interesting to notice that, if we want to prove "by hand" that this program is unification-free, then the key step is indeed represented by showing that each time that an atom of the form `qs_dl(t, s, r)` is selected, `s` will be a variable.

```
flatten(Xs, Ys) ← flatten_dl(Xs, Ys, []).
flatten_dl([X | Xs], Ys, Zs ) ←
   flatten_dl(X, Ys, Ys1),
   flatten_dl(Xs, Ys1, Zs).
flatten_dl(X, [X | Xs], Xs) ←
   constant(X), X ≠ [ ].
flatten_dl([], Xs, Xs).
```

Incidentally, the reasons why we cannot `flatten_dl` to be unification-free are the same ones found for the program `quicksort_dl`. If we associate to it the following types:

$$\begin{array}{rcl} \texttt{flatten} & : & \mathit{Ground} \times \mathit{Var} \\ \texttt{flatten\_dl} & : & \mathit{Ground} \times \mathit{Var} \times U \end{array}$$

We have that the heads of all the clauses are input safe and U-safe, and, in the case that `t` is a list and `s` is a variable disjoint with `t`, all LD-derivations of `flatten_dl` $\cup$ { `flatten(t, s)` } are i/o driven, therefore, by Theorem 8.5.10, `flatten_dl` $\cup$ { `flatten(t, s)` } is unification-free. Again, the problem here is that the program is not nicely typed: `Y1s` appears first in the U-position of `flatten_dl(X, Ys, Ys1)` and then in the output position of `flatten_dl(Xs, Ys1, Zs)`; consequently, with our tools we cannot guarantee the i/o drivenness of the derivations.

In the literature we do find tools that would enable us to prove these two programs to be unification-free, namely *asserted programs*. *Assertions* can be viewed as extension of types, and provide a more expressive formalism for proving run-time properties like groundness of terms and independence of variables (see Apt-Marchiori [10]). Two are the reasons why we decided not to use assertions in this chapter: in the first place, the machinery involved is far more complicated and computationally expensive than with types, and when we use types in full generality we already face the algorithmically intractable problem of checking type judgements. Secondly, the only two programs that we know of that can be proven to be unification-free using assertions and not with types are precisely `flatten_dl` and `quicksort_dl`. Summarizing, we strongly believe that the gain in generality is far not worth the loss in clarity and efficiency.

Of course, the results of this chapter allow us to can prove `quicksort_dl` and `flatten_dl` are unification-free wrt the following types:

$$\begin{array}{rcl} \texttt{qs} & : & \mathit{Ground} \times \mathit{Ground} \\ \texttt{qs\_dl} & : & \mathit{Ground} \times \mathit{Ground} \times U \\ \texttt{partition} & : & \mathit{Ground} \times \mathit{Ground} \times \mathit{Var} \times \mathit{Var} \end{array}$$

$$\begin{array}{rcl} \texttt{flatten} & : & \mathit{Ground} \times \mathit{Ground} \\ \texttt{flatten\_dl} & : & \mathit{Ground} \times \mathit{Ground} \times U \end{array}$$

However this are not the natural typings for these programs: for instance they require that in the queries `qs(t, s)` and `flatten(t, s)` both `t` and `s` are ground terms. In practice we have to know the result of the computation in advance.

What cannot be done: when is unification needed

Considering the surprisingly large number of programs that could be proven to be unification-free, in [7] we raised the question of whether unification was actually intrinsically needed in Prolog programs: "A canonic example (of a program requiring unification) is the Prolog program `curry` which computes a type assignment to a lambda term, if such an assignment exists (see e.g. Reddy [84]). We are not aware of other natural examples, though it should be added that for complicated queries which anticipate in their output positions the form of computed answers, almost any program will necessitate the use of unification."

In one year we have been running into a couple of interesting examples. The first one is the program `append_dl` [94, Pag. 241].

```
append_dl(As, Bs, Cs) ←
    the difference-list Cs is the result concatenating the difference-lists As and Bs.
append_dl(Xs \ Ys, Ys \ Zs, Xs \ Zs).
```

`append_dl` can concatenate the difference lists `As` and `Bs` in constant time, a relevant improvement over the ordinary `append`, which takes linear time. However, it is easy to see that in most cases `append_dl` does requires the use unification.

A second example is provided by the Prolog formalization of a problem from Coelho and Cotta [31, pag. 193]: arrange three 1's, three 2's, ..., three 9's in sequence so that for all $i \in [1, 9]$ there are exactly i numbers between successive occurrences of i.

```
sublist(Xs, Ys) ← Xs is a sublist of the list Ys.
sublist(Xs, Ys) ← app(_, Zs, Ys), app(Xs, _, Zs).

sequence(Xs) ← Xs is a list of 27 elements.
sequence([_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_,_]).

question(Ss) ← Ss is a list of 27 elements forming the desired sequence.
question(Ss) ←
    sequence(Ss),
    sublist([1,_,1,_,1], Ss),
    sublist([2,_,_,2,_,_,2], Ss),
    sublist([3,_,_,_,3,_,_,_,3], Ss),
    sublist([4,_,_,_,_,4,_,_,_,_,4], Ss),
    sublist([5,_,_,_,_,_,5,_,_,_,_,_,5], Ss),
    sublist([6,_,_,_,_,_,_,6,_,_,_,_,_,_,6], Ss),
    sublist([7,_,_,_,_,_,_,_,7,_,_,_,_,_,_,_,7], Ss),
    sublist([8,_,_,_,_,_,_,_,_,8,_,_,_,_,_,_,_,_,8], Ss),
    sublist([9,_,_,_,_,_,_,_,_,_,9,_,_,_,_,_,_,_,_,_,9], Ss).

augmented by the append program.
```

In this case Prolog provides a straightforward and elegant way of formalizing the problem, however by looking at the trace of the execution it is easy to check that, in order to run properly, the program fully uses unification.

8.8 Conclusions

Relations with [7]

This chapter can be seen as an extension of Apt and Etalle [7]. Technically, the main differences between this and [7] can be summarized as follows:

- In [7] only input and output positions are considered while here we introduce and use U-positions as well.
- In [7] the only terms that are allowed to fill in the output positions of the queries are variables. Here, by using the type Pt, we often allow the presence of pure terms, and this broadens the class of programs and queries that we can prove to be unification-free.
- Like in here, in [7], the programs considered needed always to be well-typed[5], however, the definition of well-typed programs used in [7] is more restrictive than the present ones.

The practical consequence of these facts are manifold.

- The results can be applied to a larger class of programs.
  Examples of programs that could not be handled with the tools of [7] and that can be handled now are `permutation` and `color_map`.
- The results can be applied to a larger class of queries.
  In almost all cases, programs which could be handled in [7] can be now handled better, i.e. the class of allowed queries is now broader. To give a simple example, let us consider the program `member`. Using the tools of [7], we can prove to be unification-free wrt the following typings:

  (1) `member`: $Ground \times Ground$,
  (2) `member`: $Var \times Ground$,
  (3) `member`: $Var \times List$

  On the other hand, using the tools given in this chapter we can prove `member` to be unification-free wrt the following typings:

  (a) `member`: $U \times Ground$
  (b) `member`: $Pt \times U$

  It is easy to see that the typing (a) is more general than both (1) and (2), while (b) is more general than both (2) (again) and (3): the class of queries for which we can prove unification freedom is now quite larger, and we can do this using a reduced number of different typings (two instead of three), thus reducing the machinery involved in the proof.
- The hypothesis of the theorems are often checkable in a much more efficient way.
  In order to provide an example, let us consider again the `member` program, together with the typings given above. First recall that the typing (b) is more

[5] recall that in the discussion after Theorem 8.5.18 we showed that, by appropriately choosing the *type* and the *post-type* for a relation symbol, all the programs that satisfy the conditions of Theorem 8.4.12 or the ones of Theorem 8.6.6 are well-typed.

general that both typings (2) and (3). Now, an important advantage of (b) over (3) is the following: in order to use (3) we have to use Theorem 30 of [7][6] which requires to check some non-trivial type judgement, and this is, as discussed before, an algorithmically intractable problem. On the other hand, in order to prove unification freedom using typing (b), can use Theorem 8.4.12, our simplest result, whose hypothesis can be simply and efficiently tested.
This situation is not incidental: by looking at the list of programs reported in [7, Section 8][7] and comparing it with the one in Section 8.7 of this chapter, we see that in most of the cases in which we had some nonground input positions, we could simply turn these positions into U-positions, and prove unification freedom using Theorem 8.4.12 instead of Theorem 30 of [7], both enlarging the class of allowed queries and simplifying dramatically the process of proving that the program is unification-free.

Other related work

Another recent related work is the one of M. Marchiori [71]: Marchiori concentrates on Well-Moded programs and studies *maximal localizations* of the property of being Unification-Free. In order to compare his paper with our chapter we have to introduce a bit of notation. Let us be brief and informal.

We say that a property $\mathcal{P}$ is *local* if for any two programs P and Q that satisfy it, we have that the program $P \cup Q$ satisfies $\mathcal{P}$ as well. In other words, $\mathcal{P}$ is local if it can be checked clause by clause. For instance the property *"P is Well-Moded and Nicely typed wrt the typing $\mathcal{T}$"* is local, while the property *"there exists a typing $\mathcal{T}$ such that P is Well-Moded and Nicely typed wrt it"* is not local, as we need to traverse the program more than once to check it (eventually we have to try different $\mathcal{T}$s). We also say that a property $\mathcal{Q}$ is more general than $\mathcal{P}$ if each program that satisfies $\mathcal{P}$ satisfies $\mathcal{Q}$ as well.

Now, the question addressed in [71] is the following:

- assume that to each relation symbol is already associated a typing of the form

$$p: \ T_1 \times \ldots \times T_n, \text{ where, for each } i, \ T_i \in \{Ground, U\}. \tag{8.6}$$

  we want to find (if it exists) a *local* property $\mathcal{P}$ such that
  - each program that satisfies $\mathcal{P}$ is Well-Moded (wrt the give typing (8.6));
  - each program that satisfies $\mathcal{P}$ is Unification-Free;
  - $\mathcal{P}$ is maximal, that is, there is no other local property $\mathcal{Q}$ which is more general than $\mathcal{P}$ and that satisfies the above two conditions.

[6]Roughly speaking, [7, Theorem 30] is a restricted version of Theorem 8.5.18, and it is the most general result of [7].

[7]the reader who actually does so has to be warned that the notation is a bit different: for instance the type `select` $(-{:}U, +{:}List, -{:}List)$ of [7] corresponds to our type `select` : *Var*, *List*, *Var*. together with the post-type `select` : *U*, *List*, *List*.

In [71] it is proven that such properties exist, in particular two of them are defined in detail[8]. Of course there exist other maximal properties that satisfy the above conditions.

Summarizing, the goal of [71] is quite different from our own: [71] focuses more on the theoretical aspects of local properties in the context of well-moded program, while here we want to provide (possibly simple) tools for proving unification freedom for a (possibly) large class of programs and queries. Indeed the class of programs and queries for which we can prove unification freedom is substantially larger than in [71]; this is mainly due to two reason: firstly, because restricting to the class of Well-moded program already narrows sensibly the set of allowed queries (recall that of the programs of the List, the ones that are Well-Moded are the ones which are proven to be Unification-Free via Theorem 8.6.6); secondly, because *local* properties are, at least in this context, intrinsically rather weak.

8.9 Appendix: reducing the number of matches

Let $A = p(\tilde{s})$ and $H = p(\tilde{t})$ be two atoms. We know that if the hypothesis of the Sequential Matching 2 Lemma 8.5.9 are satisfied, then the equations in $\tilde{s} = \tilde{t}$ are solvable, *one at a time*, by matching.

Here we want to show that some subsets of $\tilde{s} = \tilde{t}$ containing more than one equation can be solved by a single matching. This reduces the total number of matchings needed to solve $\tilde{s} = \tilde{t}$, and results in an efficiency gain: since there are parallel algorithms for term matching that run in polilogarithmic time [36, 37], matching more positions at once increases the execution speed.

Lemma 8.9.1 Consider two disjoint atoms $A = p(\tilde{s})$ and $H = p(\tilde{t})$ with the same relation symbol. Assume that A correctly typed and output independent, and that H is input safe and U-safe. Let us now divide the set of equations $\tilde{s} = \tilde{t}$ into the following subsets: let

- $\tilde{s}_1 = \tilde{t}_1$ be the subset of $\tilde{s} = \tilde{t}$ corresponding to the *nonground* input positions.
- $\tilde{s}_2 = \tilde{t}_2$ be the subset of $\tilde{s} = \tilde{t}$ corresponding to the *ground* input positions.
- $\tilde{s}_3 = \tilde{t}_3$ be the subset of $\tilde{s} = \tilde{t}$ corresponding to the U-positions with respect to which H satisfies condition (ii) of U-safeness (Definition 8.5.8).
- $\tilde{s}_4 = \tilde{t}_4$ be the subset of $\tilde{s} = \tilde{t}$ corresponding to those of the remaining U-positions of H which are filled in by a variable.
- $s_5 = t_5, \ldots, s_k = t_k$ be the subsets of $\tilde{s} = \tilde{t}$ such that for $i \in [5, k]$, each $s_i = t_i$ corresponds to one of the remaining U-positions.
- $s_{k+1} = t_{k+1}, \ldots, s_l = t_l$ be the subsets of $\tilde{s} = \tilde{t}$ such that for $i \in [k+1, l]$, each $s_i = t_i$ corresponds to a position of type Pt.
- $\tilde{s}_{l+1} = \tilde{t}_{l+1}$ be the subset of $\tilde{s} = \tilde{t}$ corresponding to the positions typed Var.

[8]These two properties are named "(the property of being) *Flatly-Well-Moded*" and "*coFlatly-Well-Moded*"

Then

$$\tilde{s}_1 = \tilde{t}_1,\ \tilde{s}_2 = \tilde{t}_2,\ \tilde{s}_3 = \tilde{t}_3,\ \tilde{s}_4 = \tilde{t}_4,\ s_5 = t_5, \ldots, s_k = t_k,\ s_{k+1} = t_{k+1}, \ldots, s_l = t_l,\ \tilde{s}_{l+1} = \tilde{t}_{l+1}$$

is solvable by sequential matching.

Here notice that $\tilde{s}_1 = \tilde{t}_1$, $\tilde{s}_2 = \tilde{t}_2$, $\tilde{s}_3 = \tilde{t}_3$, $\tilde{s}_4 = \tilde{t}_4$ and $\tilde{s}_{l+1} = \tilde{t}_{l+1}$ are *sets* of equations, and these are precisely the subsets of $\tilde{s} = \tilde{t}$ whose content can be processed by a single matching.

Proof. We proceed as in the proof of Lemma 8.5.9: we'll find some substitutions $\theta_1, \ldots, \theta_l$ such that, for $i \in [1, l+1]$, θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$ (here, for the sake of precision, for $i \in \{1, 2, 3, 4, l+1\}$, we should have used bold letters, and written $(\tilde{s}_i = \tilde{t}_i)$). We have to consider seven distinct cases.

In $\tilde{s}_1 = \tilde{t}_1$, since H is input safe, each term in $\tilde{t}_1$ is a generic expressions for the type of the positions it corresponds to; moreover, the terms in $\tilde{t}_1$ are pairwise disjoint. Since A is correctly typed, from the Matching 2 Lemma 8.5.6 it follows that $\tilde{s}_1 = \tilde{t}_1$ is solvable by matching. Let θ_1 be a match of $\tilde{s}_1 = \tilde{t}_1$.

In $(\tilde{s}_2 = \tilde{t}_2)\theta_1$, since A is correctly typed, the terms in $\tilde{s}_2$ are all ground. By the Matching 1 Lemma 8.4.1 $(\tilde{s}_2 = \tilde{t}_2)\theta_1$ is then solvable by matching. Let θ_2 be a match of it, and notice that $\tilde{t}_2\theta_1\theta_2$ is a set of ground terms.

In $(\tilde{s}_3 = \tilde{t}_3)\theta_1\theta_2$, becsuse of the way $\tilde{s}_3 = \tilde{t}_3$ was defined, we have that $Var(\tilde{t}_3) \subseteq Var(\tilde{t}_2)$, therefore $\tilde{t}_3\theta_1\theta_2$ is a set of ground terms. Again, by the Matching 1 Lemma 8.4.1 $(\tilde{s}_3 = \tilde{t}_3)\theta_1\theta_2$ is then solvable by matching. Let θ_3 be a match of it.

In $(\tilde{s}_4 = \tilde{t}_4)\theta_1\theta_2\theta_3$, by the way $\tilde{s}_4 = \tilde{t}_4$ was defined, $\tilde{t}_4$ consists of distinct variables, moreover $Var(\tilde{t}_4) \cap Var(\tilde{t}_1, \ldots, \tilde{t}_3) = \emptyset$. By the relevance of $\theta_1, \theta_2, \theta_3$ (a match is always a *relevant* mgu) we then have that $\tilde{t}_4\theta_1\theta_2\theta_3$ is a set of distinct variables. Again, by the Matching 1 Lemma 8.4.1 $(\tilde{s}_4 = \tilde{t}_4)\theta_1\theta_2\theta_3$ is then solvable by matching. Let θ_4 be a match of it.

The equations $(s_5 = t_5, \ldots, s_k = t_k, s_{k+1} = t_{k+1}, \ldots, s_l = t_l)\theta_1 \ldots \theta_4$ are then solvable (one at a time) by sequential matching. This follows at once from the proof of the Sequential Matching 2 Lemma 8.5.9. In particular we have that: for $i \in [5, k]$, since H is U-safe, $t_i\theta_1 \ldots \theta_{i-1}$ is a variable or a pure term, while for $i \in [k+1, l]$, since A is correctly typed and output independent, $s_i\theta_1 \ldots \theta_{i-1}$ is a variable or a pure term; here we (inductively) assume that for $i \in [5, l]$, θ_i is a match of $(s_i = t_i)\theta_1 \ldots \theta_{i-1}$.

Finally, in $(\tilde{s}_{l+1} = \tilde{t}_{l+1})\theta_1 \ldots \theta_l$, since A is correctly typed and output independent, from the relevance of $\theta_1, \ldots, \theta_l$ it follows that the terms in $\tilde{s}_{l+1}\theta_1 \ldots \theta_l$ are all distinct variables. Therefore, by the Matching 1 Lemma 8.4.1, $(\tilde{s}_{l+1} = \tilde{t}_{l+1})\theta_1 \ldots \theta_l$ is solvable by matching. This proves the Lemma. □

In practice, Lemma 8.9.1 states that we can solve by a single matching each of the following groups of positions:

- the *nonground* input positions.
- the *ground* input positions.
- the U-positions with respect to which H satisfies condition (ii) of U-safeness (Definition 8.5.8).

- those of the remaining U-positions of H which are filled in by a variable.
- the positions typed Var.

While the remaining positions should be processed one by one. These are

- the remaining U-positions.
- the position of type Pt.

The following Example shows that these last positions actually need to be processed one at a time.

Example 8.9.2

(i) Consider $A = p(x, f(x, x))$ and $H = p(g(y), f(z, w))$, together with the typing $p : U \times U$. We have that A is correctly typed and that H is U-safe. Since here there are no input nor output positions, it follows that the hypothesis of the Sequential matching 2 Lemma 8.5.9 are satisfied, therefore $A = H$ is solvable by sequential matching. However $A = H$ is not solvable by matching, as there is no θ such that $A\theta = H$ or $A = H\theta$. This shows that the U positions of H which are filled in by pure terms and for which H satisfies condition (i) of U-safeness (Definition 8.5.8) need to be processed one at a time.

(ii) A perfectly symmetric reasoning applies for the positions typed Pt: consider $A = p(y, f(z, w))$ and $H = p(x, x)$, together with the typing $p : Pt \times Pt$. A is correctly typed and output independent, and since there are no input and U-positions, this is sufficient to satisfy the hypothesis of the Sequential 2 Lemma 8.5.9. Therefore $A = H$ is solvable by sequential matching, but not by a simple matching. As before, this is confirmed by the fact that there is no θ such that $A\theta = H$ or $A = H\theta$. □

Lemma 8.9.1 is an improved version of the Sequential Matching 2 Lemma 8.5.9, which in turn was the crucial step of Theorem 8.5.18. Therefore, its basic implication is that, when A and H are respectively the selected atom and the head of the input clause used to resolve it, then some positions of $A = H$ can be grouped in the same match (while others may not).

For this reason, in some situations, we might find convenient to adopt a typing which is more restrictive than another one, but which allows us to prove that we can solve the equations in the LD-derivations with a smaller number of matchings.

Consider for instance once again the program `append`, suppose that we want to use it for splitting a ground list in two. We might then want to adopt the following typing:

$$\mathcal{T}_1 = \ \texttt{app} : Pt \times U \times Ground$$

Here the (only) input position in the third one. From Theorem 8.6.6 it follows that, if `t` is a ground list, `r` is in Pt, then, for any term `s` disjoint from `s`, `append` $\cup$ { `app(r, s, t)`} is unification free.

However, if the kind of queries we are interested in are the ones in which the first two positions of `append` are filled in by variables (and this is a common situation), then we might find convenient to use the following typing:

$$\mathcal{T}_2 = \texttt{app} : Var \times Var \times Ground$$

Of course $\mathcal{T}_2$ is more restrictive than $\mathcal{T}_1$: every query that is correctly typed wrt $\mathcal{T}_2$ is also correctly typed wrt $\mathcal{T}_1$ (and not vice-versa). However, when we adopt $\mathcal{T}_1$, the best that we can prove is that all the equations considered in the LD-derivations of `append` $\cup$ { `app(r, s, t)`} are solvable by *triple* matching: first we match the rightmost position, then we match the middle one, and finally we match the leftmost one. On the other hand, if we adopt $\mathcal{T}_2$, from Lemma 8.9.1 it follows that all the equations considered in the LD-derivations of `append` $\cup$ { `app(r, s, t)`} are solvable by *double* (rather than triple) matching: first we match the rightmost position, then with a single match we can take care of the first two ones. Of course this holds provided that the queries satisfy the conditions of Theorem 8.5.18 wrt the adopted typing, and that is when they are correctly typed and output independent.

Finally, as a further example consider again the program `select`, which is reported in Example 8.6.7. As we mentioned in the discussion after Example 8.6.7, a query `select(s, t, u)` can be used in two main ways: to delete the element `s` from the list `t` and report the result in `u`, or as a generalized `member` program, to report in `s` an element of `t`, and in `u` the remains of the list. For both cases we can use the typing

$$\mathcal{T}_1 = \texttt{select} : U \times Ground \times Pt$$

When we use this typing, (assuming that the query satisfies the hypothesys of Theorem 8.5.18), from Lemma 8.9.1 it follows that all the equations condidered in the LD derivations of `select` $\cup$ { `select(s, t, u)` } are solvable by triple matching.

However, when `select` is used in the first of the ways outlined above, then the first two arguments of the query are possibly ground terms. This allows us to use the typing

$$\mathcal{T}_2 = \texttt{select} : Ground \times Ground \times Pt$$

in this case, by Lemma 8.9.1, the equations considered in LD-derivations of `select` $\cup$ { `select(s, t, u)` } are solvable by *double* matching: first we match simultaneously the first two positions, then we match the third one.

A similar reasoning applies when we want to use select only as a generalized member program: we can reduce the number of matching needed in the LD-derivations by restricting the range of allowed queries, in particular by adopting the following typing:

$$\mathcal{T}_3 = \texttt{select} : Var \times Ground \times Var$$

In this case, from Lemma 8.9.1 it follows that the equations considered in the LD-derivations are again solvable by double matching, but this time we (obviously) match first the second position (the input one) and then, simultaneously, the first and third one (again, here we naturally assume that the queries satisfy the conditions of Theorem 8.5.18).

Bibliography

[1] L. Aiello, G. Attardi, and G. Prini. Towards a more declarative programming style. In E. J. Neuhold, editor, *Proc. of the IFIP Conference on Formal Description of Programming Concepts*, pages 121–137. North-Holland, 1978.

[2] A. Aiken and T.K. Lakshman. Type checking directionally typed logic programs. Technical report, Department of Computer Science, University of Illinois at Urbana Champaign, november 1993.

[3] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics, pages 495–574. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[4] K. R. Apt. Program verification and prolog. In E. Børgher, editor, *Specification and validation methods for programming languages and systems*. Oxford University Press, 1994. to appear.

[5] K. R. Apt and M. Bezem. Acyclic programs. *New Generation Computing*, 9(3&4):335–363, 1991.

[6] K. R. Apt, H. A. Blair, and A. Walker. Towards a theory of declarative knowledge. In editor J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 89–148. Morgan Kaufmann, 1988.

[7] K. R. Apt and S. Etalle. On the unification free Prolog programs. In A. Borzyszkowski and S. Sokolowski, editors, *Proceedings of the Conference on Mathematical Foundations of Computer Science (MFCS 93)*, Lecture Notes in Computer Science, pages 1–19, Berlin, 1993. Springer-Verlag.

[8] K. R. Apt and D. Pedreschi. Reasoning about termination of pure Prolog programs. *Information and Computation*, 106(1):109–157, 1993.

[9] K. R. Apt and A. Pellegrini. Why the occur-check is not a problem. In M. Bruynooghe and M. Wirsing, editors, *Proceedings of the Fourth International Symposium on Programming Language Implementation and Logic Programming (PLILP 92)*, Lecture Notes in Computer Science 631, pages 69–86, Berlin, 1992. Springer-Verlag.

[10] K.R. Apt and E. Marchiori. Reasoning about Prolog programs: from modes

through types to assertions. Technical Report CS-R9358, CWI, Amsterdam, The Netherlands, 1993. Available via anonymous ftp at ftp.cwi.nl, or via xmosaic at http://www.cwi.nl/cwi/publications/index.html.

[11] C. Aravidan and P. M. Dung. Partial deduction of logic programs w.r.t. well-founded semantics. In H. Kirchner G. Levi, editor, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, pages 384–402. Springer-Verlag, 1992.

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

[13] I. Attali and P. Franchi-Zannettacci. Unification-free execution of TYPOL programs by semantic attribute evaluation. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the Fifth International Conference on Logic Programming*, pages 160–177. The MIT Press, 1988.

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

[15] M. Bezem. Strong termination of logic programs. *Journal of Logic Programming*, 15(1&2):79–97, 1993.

[16] A. Bossi, M. Bugliesi, M. Gabbrielli, G. Levi, and M. C. Meo. Differential logic programming. In *Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 359–370. ACM Press, 1993.

[17] A. Bossi and N. Cocco. Verifying correctness of logic programs. In J. Diaz and F. Orejas, editors, *TAPSOFT '89, Barcelona, Spain, March 1989, (Lecture Notes in Computer Science, vol. 352)*, pages 96–110. Springer-Verlag, 1989.

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

[19] A. Bossi, N. Cocco, and S. Dulli. A method for specializing logic programs. *ACM Transactions on Programming Languages and Systems*, 12(2):253–302, April 1990.

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

[21] A. Bossi, N. Cocco, and S. Etalle. Transforming Normal Programs by Replacement. In A. Pettorossi, editor, *Meta Programming in Logic - Proceedings META'92*, volume 649 of *Lecture Notes in Computer Science*, pages 265–279. Springer-Verlag, Berlin, 1992.

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

[23] F. Bronsard, T.K. Lakshman, and U.S. Reddy. A framework of directionality for proving termination of logic programs. In K.R. Apt, editor, *Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages

321–335. MIT Press, 1992.

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

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

[26] L. Cavedon. Acyclic programs and the completeness of SLDNF-resolution. *Theoretical Computer Science*, 86:81–92, 1991.

[27] R. Chadha and D.A. Plaisted. Correctness of unification without occur check in Prolog. Technical report, Department of Computer Science, University of North Carolina, Chapel Hill, N.C., 1991.

[28] K. L. Clark. Negation as failure rule. In H. Gallaire and G. Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

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

[30] K.L. Clark and S. Sickel. Predicate logic: a calculus for deriving programs. In *Proceedings of IJCAI'77*, pages 419–120, 1977.

[31] H. Coelho and J.C. Cotta. *Prolog by Example*. Springer-Verlag, Berlin, 1988.

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

[33] D. A. de Waal and J. P Gallagher. Specialization of a unification algorithm. In T. Clement and K.-K. Lau, editors, *Proc. First Workshop on Logic Program Synthesis and Transformation*, pages 205–221. Springer-Verlag, 1991.

[34] P. Deransart and J. Maluszynski. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2:119–156, 1985.

[35] P. Deransart and J. Maluszyński. Relating Logic Programs and Attribute Grammars. *Journal of Logic Programming*, 2:119–156, 1985.

[36] C. Dwork, P.C. Kanellakis, and J.C. Mitchell. On the sequential nature of unification. *Journal of Logic Programming*, 1(1):35–50, 1984.

[37] C. Dwork, P. Kannellakis, and L. Stockmeyer. Parallel algorithms for term matching. In J. H. Siekmann, editor, *Proc. Eighth International Conference on Automated Deduction*, Lecture Notes in Computer Science 230, pages 416–430. Springer-Verlag, 1986.

[38] S. Etalle. Transformazione dei programmi logici con negazione, Tesi di Laurea, Dip. Matematica Pura e Applicata, Università di Padova, Padova, Italy, July 1991.

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

[40] M. Falaschi, G. Levi, M. Martelli, and C. Palamidessi. A Model-Theoretic Reconstruction of the Operational Semantics of Logic Programs. *Information and Computation*, 102(1):86–113, 1993.

[41] M. Fitting. A Kripke-Kleene semantics for Logic Programs. *Journal of Logic Programming*, 2(4):295–312, 1985.

[42] M. Gabbrielli, G.M. Dore, and G. Levi. Observable Semantics for Constraint Logic Programs. *Journal of Logic and Computation*, 5(2):133–171, 1995.

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

[44] H. Gaifman and E. Shapiro. Fully abstract compositional semantics for logic programs. In *Proc. Sixteenth Annual ACM Symp. on Principles of Programming Languages*, pages 134–142. ACM, 1989.

[45] J. Gallagher and M. Bruynooghe. Some low-level source transformations for logic programs. In M. Bruynooghe, editor, *Proceedings of the Second Workshop on Meta-Programming in Logic, April 1990, Leuven, Belgium*, pages 229–246. Department of Computer Science, KU Leuven, Belgium, 1990.

[46] J. Gallagher, M. Codish, and E. Shapiro. Specialisation of Prolog and FCP programs using abstract interpretation. *New Generation Computing*, 6(2,3):159–186, 1988.

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

[48] A. Van Gelder, K. Ross, and J. S. Schlipf. Unfounded sets and the Well-Founded Semantics for General Logic Programs. In *Proc. Seventh ACM symposium on Principles of Database System*, pages 211–230, 1988.

[49] Nevin Heintze, Spiro Michaylov, and Peter J. Stuckey. CLP($\mathcal{R}$) and some electrical engineering problems. In Jean-Louis Lassez, editor, *ICLP'87: Proceedings 4th International Conference on Logic Programming*, pages 675–703, Melbourne, Victoria, Australia, May 1987. MIT Press. Also in Journal of Automated Reasoning vol. 9, pages 231–260, October 1992.

[50] C.J. Hogger. Derivation of logic programs. *Journal of the ACM*, 28(2):372–392, April 1981.

[51] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In *Proc. Fourteenth Annual ACM Symp. on Principles of Programming Languages*, pages 111–119. ACM, 1987.

[52] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Technical report, Department of Computer Science, Monash University, June 1986.

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

[54] Joxan Jaffar, Spiro Michaylov, Peter J. Stuckey, and Roland H. C. Yap. An abstract machine for CLP($\mathcal{R}$). In *Proceedings ACM SIGPLAN Symposium on Programming Language Design and Implementation (PLDI), San Francisco*, pages 128–139, June 1992.

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

[56] Niels Jørgensen, Kim Marriott, and Spiro Michaylov. Some global compile-time optimizations for CLP($\mathcal{R}$). In Vijay Saraswat and Kazunori Ueda, edit-

ors, *ILPS'91: Proceedings of the International Logic Programming Symposium*, pages 420–434, San Diego, October 1991. MIT Press.

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

[58] T. Kawamura and T. Kanamori. Preservation of Stronger Equivalence in Unfold/Fold Logic Programming Transformation. *Theoretical Computer Science*, 75(1&2):139–156, 1990.

[59] S.C. Kleene. *Introduction to Metamathematics.* D. van Nostrand, Princeton, New Jersey, 1952.

[60] H. Komorowski. Partial evaluation as a means for inferencing data structures in an applicative language: A theory and implementation in the case of Prolog. In Ninth ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, pages 255–267. ACM, 1982.

[61] K. Kunen. Negation in Logic Programming. *Journal of Logic Programming*, 4:289–308, 1987.

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

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

[64] J.-L. Lassez, M. J. Maher, and K. Marriott. Unification Revisited. In J. Minker, editor, *Foundations of Deductive Databases and Logic Programming*, pages 587–625. Morgan Kaufmann, Los Altos, Ca., 1988.

[65] J. W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, Berlin, 1987. Second edition.

[66] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

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

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

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

[70] J. Maluszynski and H. J. Komorowski. Unification-free execution of logic programs. In *Proceedings of the 1985 IEEE Symposium on Logic Programming*, pages 78–86, Boston, 1985. IEEE Computer Society Press.

[71] M. Marchiori. Localizations of unification freedom through matching directions. In M. Bruynooghe, editor, *Proc. Eleventh International Logic Programming Symposium.* MIT Press, 1994.

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

1990. MIT Press.

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

[74] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4:258–282, 1982.

[75] C. S. Mellish. The Automatic Generation of Mode Declarations for Prolog Programs. DAI Research Paper 163, Department of Artificial Intelligence, Univ. of Edinburgh, August 1981.

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

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

[78] M. Proietti and A. Pettorossi. Unfolding, definition, folding, in this order for avoiding unnecessary variables in logic programs. In Maluszynski and M. Wirsing, editors, *PLILP 91, Passau, Germany (Lecture Notes in Computer Science, Vol.528)*, pages 347–358. Springer-Verlag, 1991.

[79] M. Proietti and A. Pettorossi. Synthesis of programs from unfold/fold proofs. In Y. Deville, editor, *Proc. Thirs Workshop on Logic Program Synthesis and Transformation*. Springer-Verlag, 1993.

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

[81] T. C. Przymusinki. Perfect model semantics. In *Fifth international Conference and Symposium on Logic programming, Seattle, U.S.A.*, pages 1081–1096, 1988.

[82] T. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Proceedings of the Eighth Symposium on Principles of Database Systems*, pages 11–21. ACM SIGACT-SIGMOD, 1989.

[83] U. S. Reddy. Transformation of logic programs into functional programs. In *International Symposium on Logic Programming*, pages 187–198, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[84] U.S. Reddy. On the relationship between logic and functional languages. In D. DeGroot and G. Lindstrom, editors, *Functional and Logic Programming*, pages 3–36. Prentice-Hall, 1986.

[85] D.A. Rosenblueth. Using program transformation to obtain methods for eliminating backtracking in fixed-mode logic programs. Technical Report 7, Universidad Nacional Autonoma de Mexico, Instituto de Investigaciones en Matematicas Aplicadas y en Sistemas, 1991.

[86] D. Sands. Total correctness by local improvement in program transformation. In *Proceedings of the 22nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. ACM Press, January 1995. (To Appear).

[87] T. Sato. An equivalence preserving first order unfold/fold transformation system.

In *Second Int. Conference on Algebraic and Logic Programming, Nancy, France, October 1990, (Lecture Notes in Computer Science, Vol. 463)*, pages 175–188. Springer-Verlag, 1990.

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

[89] H. Seki. Unfold/fold transformation of stratified programs. In G. Levi and M. Martelli, editors, *6th International Conference on Logic Programming*, pages 554–568. The MIT Press, 1989.

[90] H. Seki. A comparative study of the Well-Founded and Stable model semantics: Transformation's viewpoint. In D. Pedreschi W. Marek, A. Nerode and V.S. Subrahmanian, editors, *Workshop on Logic Programming and Non-Monotonic Logic, Austin, Texas, October 1990*, pages 115–123. Association for Logic Programming and Mathematical Sciences Institute, Cornell University, 1990.

[91] H. Seki. Unfold/fold transformation of stratified programs. *Theoretical Computer Science*, 86(1):107–139, 1991.

[92] H. Seki. Unfold/fold transformation of general logic programs for the Well-Founded semantics. *Journal of Logic Programming*, 16(1&2):5–23, 1993.

[93] J. C. Shepherdson. Language and equality theory in logic programming. Technical Report PM-88-08, University Walk, Bristol, England, 1988.

[94] L. Sterling and E. Y. Shapiro. *The Art of Prolog*. The MIT Press, Cambridge, Mass., 1986.

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

[96] H. Tamaki and T. Sato. Unfold/Fold Transformations of Logic Programs. In Sten-Åke Tärnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 127–139, 1984.

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

[98] E. Yardeni, T. Frühwirth, and E. Shapiro. Polymorphically typed logic programs. In F. Pfenning, editor, *Types in Logic Programming*, pages 63–90. MIT Press, Cambridge, Massachussets, 1992.

Samenvatting

Het proefschrift is als volgt opgebouwd. Hoofdstuk 1 bevat een korte introductie op het gebied van logisch programmeren en programma transformaties. In Hoofdstuk 2 wordt de semantiek van normale logische programma's behandeld. Dit hoofdstuk dient als introductie voor de daaropvolgende drie hoofdstukken. Daarnaast bevat het hoofdstuk een nieuw resultaat waarin programma equivalentie met betrekking tot de Kunen semantiek wordt gekarakteriseert. In Hoofdstuk 3 beginnen we met de studie van eigenschappen van Unfold/Fold transformatie systemen. In dit hoofdstuk bewijzen we dat de Unfold/Fold methode van Tamaki en Sato, toegepast op een *terminerend* programma, resulteert in een programma dat zelf ook terminerend is. In Hoofdstuk 4 introduceren we de vervangingsoperatie, en onderzoeken enkele nieuwe toepassingscondities, in de context van normale logische programma's. De resultaten uit dit hoofdstuk worden in het daaropvolgende hoofdstuk gebruikt om nieuwe toepassingscondities voor de Fold operatie te vinden, die de correctheid van deze operatie met betrekking tot de Fitting semantiek garanderen. In Hoofdstuk 5 definiëren we een transformatiesysteem voor zogenaamde 'Modular Constraint Logic Programs'; logische programma's met een modulaire opbouw, waarin programmaregels randvoorwaarden kunnen bevatten. Daarnaast geven we een aantal toepassingscondities die er voor zorgen dat het systeem *compositioneel* is; we bewijzen dat onder deze condities de getransformeerde module dezelfde antwoordformules heeft als het orgineel, ook wanneer deze modules met andere modules samengevoegd worden. In Hoofdstuk 6 gaan we dieper in op de problemen die spelen bij het transformeren van 'Modular Constraint Logic Programs', met name bij de vervangingsoperatie. In dit hoofdstuk definiëren we nieuwe toepassingscondities, onder welke tijdens de transformatie bepaalde observeerbare eigenschappen behouden blijven, ook onder *compositie* van modules. Er dient opgemerkt te worden dat, binnen onze aanpak, de toepassingscondities niet gebonden zijn aan specifieke observeerbare eigenschappen. Het is vaak mogelijk deze condities zodanig aan te passen, dat ze voldoen voor de observeerbare eigenschappen waar we het meest in zijn geïnteresseerd. In Hoofdstuk 7 laten we programma transformaties voor wat ze zijn, en houden we ons bezig met programma analyse. Het is algemeen bekend dat unificatie het hart is van de resolutie methode die in PROLOG gebruikt wordt, en dat de efficiëntie waarmee dit gebeurt een grote invloed heeft op de prestaties van de interpreter. In dit hoofdstuk presenteren we eenvoudige condities onder welke het mogelijk is unificatie te vervangen door 'iterated matching', een procedure die een stuk efficiënter is te implementeren dan unificatie. We gebruiken deze condities vervolgens om aan te tonen dat 'iterated matching' volstaat bij een aantal veelgebruikte PROLOG programma's. Met deze kennis is het mogelijk de executie van deze programma's te versnellen.

Titles in the ILLC Dissertation Series:

Transsentential Meditations; Ups and downs in dynamic semantics
Paul Dekker
ILLC Dissertation series 1993-1

93.1.ned.tex93.1.eng.tex90-74795-20-x00: SOLD OUT *Resource Bounded Reductions*
Harry Buhrman
ILLC Dissertation series 1993-2

93.2.ned.tex93.2.eng.tex90-74795-16-120,00 *Efficient Metamathematics*
Rineke Verbrugge
ILLC Dissertation series 1993-3

93.3.ned.tex93.3.eng.tex90-800769-8-820,00 *Extending Modal Logic*
Maarten de Rijke
ILLC Dissertation series 1993-4

93.4.ned.tex93.4.eng.tex90-800769-9-622,50 *Studied Flexibility*
Herman Hendriks
ILLC Dissertation series 1993-5

93.5.ned.tex93.5.eng.tex90-74795-01-325,00 *Aspects of Algorithms and Complexity*
John Tromp
ILLC Dissertation series 1993-6

93.6.ned.tex93.6.eng.tex90-74795-17-x20,00 *The Noble Art of Linear Decorating*
Harold Schellinx
ILLC Dissertation series 1994-1

94.1.ned.tex94.1.eng.tex90-74795-02-122,50 *Generating Uniform User-Interfaces for Interactive Programming Environments*
Jan Willem Cornelis Koorn
ILLC Dissertation series 1994-2

94.2.ned.tex94.2.eng.tex90-74795-03-x20,00 *Process Theory and Equation Solving*
Nicoline Johanna Drost
ILLC Dissertation series 1994-3

94.3.ned.tex94.3.eng.tex90-74795-04-820,00 *Calculi for Constructive Communication, a Study of the Dynamics of Partial States*
Jan Jaspars
ILLC Dissertation series 1994-4

94.4.ned.tex94.4.eng.tex90-74795-08-020,00 *Executable Language Definitions, Case Studies and Origin Tracking Techniques*
Arie van Deursen
ILLC Dissertation series 1994-5

94.5.ned.tex90-74795-09-922,50 *Chapters on Bounded Arithmetic & on Provability Logic*
Domenico Zambella
ILLC Dissertation series 1994-6

94.6.ned.tex94.6.eng.tex90-74795-10-215,00 *Adventures in Diagonalizable Algebras*
V. Yu. Shavrukov
ILLC Dissertation series 1994-7

94.7.ned.tex94.7.eng.tex90-74795-18-817,50 *Learnable Classes of Categorial Grammars*
Makoto Kanazawa
ILLC Dissertation series 1994-8

94.8.ned.tex94.8.eng.tex90-74795-19-622,50 *Clocks, Trees and Stars in Process Theory*
Wan Fokkink
ILLC Dissertation series 1994-9

94.9.eng.tex90-74795-11-017,50 *Logics for Agents with Bounded Rationality*
Zhisheng Huang
ILLC Dissertation series 1994-10

94.10.eng.tex90-74795-13-720,00 *On Modular Algebraic Prototol Specification*
Jacob Brunekreef
ILLC Dissertation series 1995-1

90-74795-12-9(to appear) *Investigating Bounded Contraction*
Andreja Prijatelj
ILLC Dissertation series 1995-2

90-74795-14-515,00 *Algebraic Relativization and Arrow Logic*
Maarten Marx
ILLC Dissertation series 1995-3

90-74795-15-317,50 *Study on the Formal Semantics of Pictures*
Dejuan Wang
ILLC Dissertation series 1995-4

(to appear) *Generation of Program Analysis Tools*
Frank Tip
ILLC Dissertation series 1995-5

90-74795-22-620,00 *Verification Techniques for Elementary Data Types and Retransmission Protocols*
Jos van Wamel
ILLC Dissertation series 1995-6

Transformation and Analysis of (Constraint) Logic Programs
Sandro Etalle
ILLC Dissertation series 1995-7

90-74795-27-7 *Frames and Labels. A Modal Analysis of Categorial Inference*
Natasha Kurtonina
ILLC Dissertation series 1995-8

90-74795-28-5 *Tools for PSF*
G.J. Veltink
ILLC Dissertation series 1995-9

90-74795-29-3 *(to be announced)*
Giovanna Ceparello
ILLC Dissertation series 1995-10

90-74795-30-7 *Instantial Logic. An Investigation into Reasoning with Instances*
W.P.M. Meyer Viol
ILLC Dissertation series 1995-11

90-74795-31-5