Gondor: Gödel on Andorra

*Andrea Domenici**
Scuola Superiore di Studi Universitari
e di Perfezionamento S. Anna
Via Carducci, 40 - 56100 Pisa

andrea@sssup2.sssup.it

Abstract

This paper describes Gondor, a tool that translates Gödel programs into Andorra-I Prolog programs. Gödel is a logic programming language with types, modules, a flexible computation rule, and an original pruning operator. Andorra-I Prolog is a language supported by the Andorra parallel execution system. Gondor is then a parallel implementation of Gödel.

1 Introduction

The logic programming language Gödel [12] has been developed to support the application of advanced meta-programming techniques to software engineering. Its main design aim is "to have functionality and expressiveness similar to Prolog, but to have greatly improved declarative semantics compared with Prolog" [15]. This paper describes Gondor [6, 7], a tool that translates Gödel programs into Andorra-I Prolog [8, 21], a language supported by the Andorra-I System [20, 26] for the parallel execution of logic programming languages on multiprocessor architectures. The Gondor preprocessor and the Andorra-I System together constitute the first parallel implementation of the Gödel language.

Gödel's main facilities include types, modules, support for meta-programming, and a set of control features.

Gödel's *parametric polymorphic* [4] type system and its module system provide the well known advantages of types and modules for large scale software systems, particularly from the point of view of verification and software reusability. The meta-programming facilities are an extended collection of tools (mainly predefined types and procedures) to support program transformations. Its core is the *ground representation* [2, 11], a technique whereby an *object program* (e.g. a program to be interpreted or compiled, or a database query to be evaluated) is represented as a ground (variable-free) term in a *meta-program* (e.g. an interpreter or compiler, or a database system). This contrasts with the approach encouraged by traditional logic languages, such as Prolog, where the distinction between object program and meta-program is often made unclear by the presence of both object-level and meta-level variables in the

*This work was partially supported by the Ministero dell'Università e della Ricerca Scientifica e Tecnologica, Italy.

same (meta-)program. Gödel's control features are *control declarations*, *constraint solving*, and *pruning*.

Control (or *delay*) declarations are a means to specify that the execution of certain procedure calls must suspend until its arguments are instantiated in a way specified by the programmer. These declarations can be used to program more efficient execution strategies when the standard left-to-right computation rule is less fit for a given problem. Gödel's computation rule is said to be *flexible*, as it can be modified by delay declarations. Constraint solving [10] handles systems of linear and non-linear constraints on integers, and linear constraints on rationals. The Gödel pruning operator [14], called *commit*, generalizes the commit used in the concurrent logic programming languages [23] but has a better procedural semantics and lends itself to program transformations, while *conditionals* replace a typical unsound usage of cut.

The Andorra-I System is an implementation of a parallel execution model called *Basic Andorra Model*. This model combines OR-parallelism with a form of AND-parallelism (*determinate dependent AND-Parallelism*) where control of execution relies on the property of *determinacy*: a *determinate* subgoal is a call that can be matched by only one clause, and determinate calls are executed in parallel before nondeterminate calls. When no determinate calls exist in a goal, a nondeterminate call is selected and executed in OR-parallel.

The Gondor preprocessor is a first solution for the task of executing Gödel on a parallel architecture. The approach of translating between two high-level languages has made it possible to exploit the mechanisms of an existing parallel logic programming system without dealing explicitly with the system's implementation.

The language accepted by the preprocessor is the one described in [13]. This earlier version of the language differs from the current one ([12]) mainly in the lack of constraint solving. Accordingly, constraint solving will not be discussed further in this paper.

Section 2 discusses the features of Gödel and Andorra most relevant to a parallel implementation. Section 3 describes the techniques used to translate Gödel's constructs. Section 4 refers to some related research work, and Section 5 is a brief discussion.

2 Some features of Gödel and Andorra

For the purpose of a parallel implementation, the relevant features of Gödel are its pruning mechanism and its computation rule. This section gives a short informal exposition of these two issues in Gödel and in the Andorra system, following a simplified description of the language.

2.1 Gödel in brief

In Gödel, identifiers beginning with a lowercase letter denote variables, while uppercase letters begin identifiers of constants, functors, type names and constructors, and keywords.

A Gödel program is a set of modules. A module consists in general in an *export* part and a *local* part, identified by the 'EXPORT' and 'LOCAL' keywords, respectively; the 'MODULE' keyword is used in modules with no export part. The export part declares symbols available to other modules, while the local part contains declarations of private symbols and definitions of predicates. *Import declarations*, of the form 'IMPORT *modules*' are used in a module to obtain access to the symbols exported by other modules.

Language declarations introduce the types, constants, functions, propositions, and predicates used in a module. *Base* types are denoted by identifiers occurring in declarations beginning with 'BASE', and can be used, together with *constructors*, to create complex type names. For

example, the following Gödel fragment declares the base type 'People' and all the types whose names are the "terms" built from the functor 'List', the constant 'People', and arbitrary variables called *parameters*:

```
BASE         People.
CONSTRUCTOR  List/1.
```

Some of the types introduced by the above declarations are 'People', 'List(People)' (list of people), and 'List(alpha)' (list of anything). The last example is a *parametric* type: the *parameter* 'alpha' is a variable that can be instantiated as a type name. The type of a function is declared by listing the types of its arguments and of its range, and the type of a predicate is the tuple of the types of its arguments. Gödel variables are not declared, as their type is inferred from the positions of their occurrences.

A *delay declaration* for a predicate P has the form 'DELAY *atom* UNTIL *condition*', where *atom* is an atom having P's functor and arity, and *condition* is an expression composed of conjunctions and disjunctions of the simple conditions NONVAR(v), GROUND(v), and TRUE, with v any variable in *atom*. The meaning of delay declarations will be discussed in Section 2.2.

The *definition* of a predicate is the set of *statements* whose head is an atom with the predicate's name and arity. Definitions will also be called *procedures*, as is customary in Prolog. The body of a Gödel statement is a formula in a polymorphic typed first-order language extended with pruning operators and conditionals. The symbol '<-' stands for left implication and replaces Prolog's ':-', negation is represented by '~', conjunction by '&', disjunction by '\/', right implication and equivalence by '->' and '<->', respectively. Expressions of the form 'SOME [*variables*]' and 'ALL [*variables*]' are existential and, respectively, universal quantifiers. A variable with a single occurrence can be represented by an underscore '_'. Such *underscore variables* are implicitly affected by an existential quantifier placed immediately before the atom where they occur.

Conditionals have the following syntax:

conditional ⟶ (*conditional*)
| *formula*
| IF *condition* THEN *formula*
| IF *condition* THEN *formula* ELSE *conditional*
| IF SOME [*variables*] *condition* THEN *formula*
| IF SOME [*variables*] *condition* THEN *formula* ELSE *conditional*

where *condition* is a formula that does not begin with an existential quantifier and *formula* is a formula that does not contain conditionals. Intuitively, the 'THEN' part is evaluated if *condition* succeeds, while the 'ELSE' part, if present, is evaluated if *condition* fails. If an existential quantifier precedes *condition*, its scope is composed of *condition* and of the 'THEN' part.

The general form of a *commit* is '{ *scope* }_ *label*', where the *scope* is a formula that may contain commits or conditionals, and the *label* is a positive integer. When, in the execution of a statement S, the scope of a commit with label n has been solved, the other statements in the same definition of S *containing commits with label* n are pruned away. All further solutions for the scope are also pruned.

The behavior of the Gödel commit can be understood in terms of a generalized SLDNF-tree where goals are allowed to have commits appearing in them. In such a tree T, let G_0 be a

non-leaf node, and G_1 a child of G_0. We say that G_1 is an *l-child* of G_0 if either i) G_0 contains a commit labelled l and the selected literal in G_0 is in the scope of this commit; or ii) G_1 is derived from G_0 using an input statement which contains a commit labelled l (after standardization apart of the commit labels). Now, let S be a subtree of an SLDNF-tree. We say that a tree S' is obtained from S by a *pruning step* in S at G_0 if i) S has a node G_0 with distinct l-children G_1 and G_2, and there is an l-free node G_2' in S which is either equal to or below G_2; and ii) S' is obtained from S by removing the subtree of S rooted at G_1. The concept of pruning step is the basis of the procedural meaning of the Gödel commit. In an actual implementation, when the scope of a commit has been solved (i.e., a goal with the properties of G_2' above has been derived), all the subtrees rooted at goals (as G_1 above) corresponding to other statements containing commits with the same label are discarded in one step [12, 14].

This pruning operator has the property that the programs where it is used can be transformed (e.g. by partial evaluation) without altering their operational semantics.

If a commit label occurs in only one statement of a definition, the effect of the commit reduces to finding only one solution to its scope. In this case, the label and its preceding underscore can be dropped. If some statements in a definition have bodies of the form '{ V }_ n W', where n is the same in all these statements, they can be written as 'V | W'. The expression 'V |' is called a *bar commit* and its scope V cannot contain another bar commit.

2.2 Pruning and computation rule in Gödel

Pruning in Gödel relies on conditionals and on the pruning operator. Conditionals, introduced in NU-Prolog [24], select one alternative according to the outcome of a condition. The pruning operator allows a selective pruning of statements within a procedure, i.e., in general only a subset of all the statements of a procedure is pruned, rather than all but one (as in committed choice languages), or all remaining statements (as with Prolog's cut). An important feature of Gödel pruning is what may be called *safe pruning*: pruning is disabled inside a negated call and in the execution of the condition in a conditional.

The syntax of Gödel's pruning operator allows very complex constructs, as more than one commit may appear in a statement body and each commit may be preceded or followed by other formulas, or embed other commits. Commits with a given label l may occur in all statements of a definition or just in some statements, and a definition may both have statements with commits and without commits. This variety of allowable forms contrasts with the syntax for commits enforced by most languages, where only one commit per statement is allowed and either a commit operator is assumed to occur (explicitly or implicitly) in each clause (e.g. Parlog) or the behavior of a procedure where commits do not occur in all statements (or clauses) is implementation-dependent (e.g. Andorra-I Prolog).

Gödel's computation rule calls for a suspension mechanism involved in the execution of negated calls, of calls to the inequality predicate ('~='), of conditionals, and of calls to predicates for which delay declarations exist. More precisely, at each resolution step each literal or conditional is either *delayed* or *runnable*, i.e., not delayed. The selected literal or conditional is then the leftmost runnable one. If no literal is runnable, the computation cannot proceed, and we say that it *flounders*.

To ensure soundness, a negated call is delayed until its arguments are ground. This treatment of negation is called *safe negation*. A call to inequality is delayed if its arguments unify but only by binding at least one non-underscore variable. This rule is a relaxation of the general suspension rule for safe negation, which may result in unnecessary delays for calls to inequality. A conditional is delayed until its condition contains no free variables.

If a predicate has delay declarations of the form 'DELAY A_1 UNTIL *condition-1*', ..., 'DELAY A_i UNTIL *condition-i*', ..., 'DELAY A_m UNTIL *condition-m*', a call A to the predicate is delayed if i) A has a common instance with some atom A_i in the delay declarations but is not an instance of this atom; or ii) A is an instance of an atom A_i in the delay declarations but does not satisfy the corresponding condition *condition-i*. Recall that the conditions in the delay declarations refer to the state of instantiation of the arguments. The effect of delay declarations is then similar to the control that can be achieved in Prolog by using `var/1` and `nonvar/1`, but Gödel's delay declarations are cleaner, since this kind of non-logical control information does not occur in the predicate definitions, and much more flexible, since delay declarations affect the order of computation, whereas `var/1` an `nonvar/1` can just succeed or fail.

We may observe that the Gödel computation rule is insensitive of the presence of pruning operators, i.e., the fact that a literal occurs in the scope of a commit does not influence its eligibility for selection at any computation step. For example, if a commit is not the leftmost formula in the body, the literals within its scope could be selected after literals outside the scope. This contrasts with languages such as Parlog, where the scope of a commit is the formula to its left, and the literals occurring in the scope are guaranteed to be selected before those outside.

2.3 Pruning and computation rule in Andorra

In the Andorra system, both cut and commit (in the style of committed choice languages) are supported. As in Prolog, a cut prunes all (further) solutions to the calls on its left and all the following clauses in the procedure. A commit prunes solutions of the calls to its left and the preceding and following clauses in the procedure. No provision is made for pruning selected clauses, or for safe pruning.

The Andorra system has an implicit suspension rule based on the concept of *determinacy*: nondeterminate calls are delayed until either they become determinate (when some of their arguments are instantiated by other calls) or there is no determinate call in the current goal. In this case, one nondeterminate call is selected for reduction. This is an aspect of the *Basic Andorra model*: determinate calls are executed, possibly in And-parallel, before nondeterminate ones. A nondeterminate call is executed, possibly in Or-parallel, when no call is determinate. To support this execution model, the Andorra system is able to detect determinism and to suspend calls. Detection of determinism involves a static analysis [22] of the Andorra-I Prolog programs to generate determinacy code, i.e. WAM-like [25] instruction that test arguments, select clauses and handle suspension at run time.

Suspension of nondeterminate calls is managed implicitly by the system. However, in the Andorra-I Prolog language the testing and suspension mechanism is made accessible to the programmer through built-in predicates, such as `data/1` and `ap_ground/1`, that suspend until their argument is instantiated or, respectively, ground.

In Andorra the selection of calls is affected by the pruning operators, since calls to the right of a cut or commit are executed after the calls to its left.

Another aspect of the Andorra control is the possibility of overriding the basic "eager" And-parallel computation rule for determinate calls by forcing a sequential, left-to-right execution of certain sequences of calls. This possibility is needed by the system to guarantee a correct execution of built-ins with side-effects, and sequential execution can be programmed explicitly, in Andorra-I Prolog, by using a sequential conjunction connective.

3 From Gödel to Andorra-I Prolog

This section describes the techniques used to implement on an already existing parallel logic programming system the features of Gödel introduced in the previous section. The implementation is based on transforming Gödel programs into programs in a language supported by the system.

The execution model of Andorra-I Prolog, together with its explicit control constructs, make it adequate to implement a version of Gödel that exploits the Basic Andorra principle to achieve determinate dependent And-parallelism and Or-parallelism.

This version differs from the original definition of Gödel in the computation rule and in the lack of safe pruning. We may also recall that the definition referred to here is not the most recent.

3.1 Computation rule and pruning operator

As has been mentioned in previous sections, the Gödel pruning operator has a much more complex syntax than the Andorra commit. Furthermore, the Gödel computation rule does not take pruning operators into account, whereas in Andorra-I Prolog, as usual in logic programming languages, literals to the left of a commit will be selected before those to the right. When a Gödel procedure is translated into a target language supporting a commit operator in the style of the concurrent logic languages, the result is a new procedure, usually containing new predicates, whose execution should produce the same solutions that are produced by the execution (on a Gödel system) of the original procedure. The procedure obtained by the translation follows the target language's syntax, which in turn determines in part the order of literal selection. Therefore choosing a particular schema of translation for Gödel procedures implicitly defines a new computation rule for the language.

It is desirable that the new computation rule be as close as possible to the original one, but we have chosen to introduce selection criteria that take commits into account. In some cases, this causes the transformed program to find fewer solutions than the original program. This situation may arise when the different selection order leads to different pruning steps being executed, as will be shown in Section 5. In particular, under the standard Gödel rule some pruning operators may be ignored, whereas the computation rule introduced by the transformations requires literals within commits to be evaluated before literals outside commits, and this usually ensures that the commits are indeed executed if their scope succeeds. This computation rule is arguably more intuitive since a programmer usually wants the scope of a commit to be executed before the rest of the statement, and also wants the pruning to take place when the scope succeeds.

The next paragraphs describe the transformations applied by Gondor to statements and definitions containing commits, then the resulting computation rule is described, in terms of a set of constraints to be applied to the original computation rule. The transformations are a set of rules that rewrite single statements or predicate definitions, and are applied repeatedly as many times as possible. When no rule is applicable, the program is in a form where each definition is in *standard form*, i.e., either it is commit-free or each of its statements has only one commit, the commit occurs leftmost in the body, and all commits have the same label. A definition in standard form can be rewritten immediately into a syntax of the form generally used in committed choice languages, which is also part of the Andorra-I Prolog syntax.

In the following, commits whose label occurs in more than one statement of a procedure are called *guards*, otherwise *one-solutions*. The word *commit* will be used hereafter when a

rule refers to a construct that may be either a guard or a one-solution. For better readability, instead of the standard Gödel syntax the following notation will be used:

$$\begin{array}{rcl} \mathit{statement} & \longrightarrow & \mathit{atom} \leftarrow \mathit{c\text{–}formula} \\ & | & \mathit{atom} \leftarrow \\ \mathit{c\text{–}formula} & \longrightarrow & \mathit{formula} \\ & | & \{\mathit{c\text{–}formula}\}_l \\ & | & \mathit{c\text{–}formula} \wedge \mathit{c\text{–}formula} \end{array}$$

where *formula* is a conjunction of literals, l is the commit label and $\wedge$ is the conjunction connective. We may assume the above syntax without losing generality, as any Gödel program stripped of the commit braces (the *underlying program*) can be transformed to a normal form where each body is a conjunction of literals.

3.1.1 Transformations for statements

Repeated application of the following rules transforms a Gödel program P into a program P' where no guards are embedded in other guards, all guards precede commit-free formulas in statement bodies, and each label occurs at most once within each body. A program having the above properties is said to be in *linear guarded* form.

In the following, letters F, C, and G, possibly subscripted, denote *formulas*, *c–formulas*, and conjunctions of guards, respectively.

Rule 1 For all statements containing a commit C of the form $\{F \wedge \{C_1\}_k \wedge C_2\}_l$, where $\{C_1\}_k$ is a guard and F and C_2 may be empty, rewrite C in the form $\{C_1\}_k \wedge \{F \wedge C_2\}_l$. If both F and C_2 are empty, replace $\{F \wedge C_2\}_l$ with $\{true\}_l$.

Rule 2 For all statements whose body B has the form $F_1 \wedge G_1 \wedge F_2 \wedge \ldots \wedge G_k \wedge F_l$, where one or both of F_1 and F_l may be empty, rewrite B in the form $G_1 \wedge \ldots \wedge G_k \wedge F_1 \wedge \ldots \wedge F_l$.

Rule 3 For all statements containing a conjunction of guards G of the form $\{C_1\}_l \wedge G_1 \wedge \{C_2\}_l$, where G_1 does not contain commits with label l, rewrite G in the form $\{C_1 \wedge C_2\}_l \wedge G_1$.

Rule 1 extracts a guard from an enclosing commit, thus "flattening out" nested guards. Rule 2 rewrites the body so that all guards precede formulas and one-solutions in textual order, and the textual order between guards is preserved. Rule 3 packs guards with the same label and disjoint scopes into one guard.

3.1.2 Transformations for definitions

Repeated application of the following rules transforms a program in linear guarded form P into a program P' where each definition is in standard form.

Each commit may be *marked* or *unmarked*. A marked commit is not considered for further transformations, all commits are initially unmarked, and a rule may mark a commit. The conditions for application of the rules in this section implicitly refer to unmarked commits.

Finally, the *least common generalization* (or *most specific generalization*) $\sqcup S$ of a set of atomic formulas is defined as in [19]. The same notation will denote the least common generalization of a set S of terms, defined analogously.

In the following, letters F, C, and D, possibly subscripted, denote *formulas*, *statements*, and sets of *statements*, respectively. Letter B will be used for whole statement bodies or for

the part of a statement body following a sequence of guards. Letters p and t will be used for predicates and terms, respectively. In particular, in each rule p (unadorned) will be the predicate, of arity a, whose definition $D = \{C_i\}_{i=1}^{n}$ is to be transformed into a new definition D'. When D is expressed as the union of two sets of statements, the sets are understood to be disjoint. Definitions for new predicates introduced by the transformations will be denoted by D'', possibly subscripted.

Rule 4 For all predicates whose definition contains both guarded and unguarded statements:

1. Let $D = D_u \cup D_g$, where D_g is the set of guarded statements in D.
2. Let each statement be of the form $p(t_{i1}, \ldots, t_{ia}) \leftarrow B_i$.
3. Let $p(t_1^*, \ldots, t_a^*) = \sqcup\{p(t_{i1}, \ldots, t_{ia}) \mid C_i \in G\}$, and $D' = D_u \cup \{C'\}$, where
$$C' = p(t_1^*, \ldots, t_a^*) \leftarrow p'(t_1^*, \ldots, t_a^*)$$
and p' is a new symbol.
4. Define p' by $D'' = \{p'(t_{i1}, \ldots, t_{ia}) \leftarrow B_i \mid C_i \in D_g\}$.

Rule 5 For each predicate such that all statements in its definition are guarded, and no label occurs leftmost in all statements:

1. Let each statement C_i be of the form
$$p(t_{i1}, \ldots, t_{ia}) \leftarrow \{F_{i1}\}_{l_{i1}} \wedge \ldots \wedge \{F_{im_i}\}_{l_{im_i}} \wedge B_i$$
where F_{ij} (l_{ij}) is the scope (label) of the j-th guard in the i-th statement, and $m_i \geq 1$ is the number of guards in the i-th statement.
2. Let $D' = \{p(t_{i1}, \ldots, t_{ia}) \leftarrow \{F_{i1}\}_k \wedge p_i'(t_{i1}, \ldots, t_{ia}, v_{i1}, \ldots, v_{ir_i})\}_{i=1}^{n}$, where k is a positive integer, $v_{i1}, \ldots, v_{ir_i}$ are the r_i distinct variables occurring in the body of C_i, and each p_i' a new predicate symbol.
3. Define the n new predicates p_i' by
$$D_i'' = \{\ p_i'(x_1, \ldots, x_a, v_1, \ldots, v_{r_i}) \leftarrow \{F_{i2}\}_{l_{i2}} \wedge \ldots \wedge \{F_{im_i}\}_{l_{im_i}} \wedge F_i\ \}$$
$$\cup\ \{\ p_i'(t_{j1}, \ldots, t_{ja}, x_1, \ldots, x_{r_i}) \leftarrow \{F_{j1}\}_{l_{j1}} \wedge \ldots \wedge \{F_{jm_j}\}_{l_{jm_j}} \wedge F_j\ \}$$
where the values of j are such that statements C_j in D do not contain label l_{i1}, and each x_k is a variable that does not occur elsewhere in the statement.

Rule 6 For each predicate such that all statements in its definition D are guarded, and the leftmost guards of each statement have the same label k:

1. Let $D = K \cup L$, where $K = \{C_i\}_{i=1}^{m}$ is the set of statements where only k occurs as a guard label.
2. Let each statement be of the form $p(t_{i1}, \ldots, t_{ia}) \leftarrow \{F_i\}_k \wedge B_i$.
3. Let $D' = K \cup L'$, with $L' = \{p(t_{i1}, \ldots, t_{ia}) \leftarrow \{F_i\}_k \wedge p_i'(v_{i1}, \ldots, v_{ir_i})\}_{i=m}^{n}$, where $v_{i1}, \ldots, v_{ir_i}$ are the r_i distinct variables occurring in B_i, and each p_i' is a new symbol.
4. Define the $n - m + 1$ predicates p_i', $i = m, \ldots, n$, by $D_i'' = \{p_i'(v_{i1}, \ldots, v_{ir_i}) \leftarrow B_i\}$.

Rule 7 For each predicate such that a statement C_i in its definition D contains a one-solution $\{Q\}_l$:

1. Let D' be the definition obtained from D by replacing $\{Q\}_l$ in C_i with $p'(v_1, \ldots, v_r)$, where $v_1, \ldots, v_r$ are the r distinct variables occurring in Q, and p' is a new symbol.
2. Define predicate p' by $D'' = \{p'(v_1, \ldots, v_r) \leftarrow \{Q\}_l\}$, where $\{Q\}_l$ is marked.

Rule 4 processes definitions containing both statements with guards and statements without guards. The set of statements with guards is replaced by one statement whose head is a generalization of the heads of the replaced statements; this statement calls a new predicate whose definition is composed of those statements, with their heads renamed.

Rule 5 transforms procedures where guards occur in all statements, but the leftmost commits in each statement do not have the same label. This rule produces a set of new definitions where the leftmost commits have the same label. In doing this transformation, we must take into account the fact that a statement may contain commits with different labels. After one of the commits has been executed, another one may still do some further pruning. In the following example:

$$p(a,x) \leftarrow \{s_1(x,y)\}_1 \wedge q(y) \quad (1)$$
$$p(b,x) \leftarrow \{s_2(x)\}_2 \wedge r(x) \quad (2)$$
$$p(c,x) \leftarrow \{s_3(y)\}_2 \wedge \{s_4(y,z)\}_1 \wedge s(y,z) \quad (3)$$

let us ussume $s_3(y)$ in statement 3 succeeds. Statement 2 should then be pruned, and execution should continue with the rest of the body of statement 3, and with statement 1 still eligible for execution. This is achieved if the definition is transformed into

$$p(a,x) \leftarrow \{s_1(x,y)\}_{10} \wedge p'_1(a,x,x,y)$$
$$p(b,x) \leftarrow \{s_2(x)\}_{10} \wedge p'_2(b,x,x)$$
$$p(c,x) \leftarrow \{s_3(y)\}_{10} \wedge p'_3(c,x,y,z)$$

and p'_3 is defined by

$$p'_3(x_1,x_2,y,z) \leftarrow \{s_4(y,z)\}_1 \wedge s(y,z) \quad (4)$$
$$p'_3(a,x,x_1,x_2) \leftarrow \{s_1(x,y)\}_1 \wedge q(y) \quad (5)$$

This definition contains one statement (4) whose body is the body of statement 3 minus the first guard, and one statement (5) whose body is the body of statement 1. The arguments of the head of statement 4 are such that the unifications carried out when statement 3 was called are not repeated.

Rule 6 deals with definitions where all statements have guards, the leftmost guards have the same label, and some statements have more than one guard. It produces definitions where each statement has only one guard by substituting the body of each statement after the first guard by a call to a new predicate.

Rule 7 transforms statements where one-solutions occur together with other formulas, producing statements where the one-solution is replaced by a call to a new predicate, whose definition contains only that one-solution.

3.1.3 Computation rule

We may now state the constraints on the Gödel computation rule introduced by the above transformations. First, the concept of *depth* of commits and literals is introduced.

Definition The *depth* of a commit in a goal (body) is the number of unpaired left brackets between the first symbol of the goal (body) and the left bracket (included) of the commit itself. The *depth* of a literal L is the maximum depth of the commits containing L. The depth of a literal not occurring inside any guard is defined to be 0.

The constraints on the Gödel computation rule can be stated as follows: i) literals inside guards will be selected before literals outside all guards; ii) literals within a guard will be selected before literals outside it and contained in a commit enclosing that guard; and iii) literals within different guards of the same depth embedded in the same commit, or within guards of depth 1, are selected from the leftmost guard first. More precisely:

1. Literals of depth $n > 0$ are selected before literals of depth 0.
2. If G is a commit of depth n containing a guard G_1 of depth $n+1$, the literals in G_1 are selected before the literals of depth n contained in G.
3. If two guards G_1 and G_2 have depth 1, or they have depth $n+1$ and are contained in a commit G of depth n, the literals in G_1 are selected before the literals in G_2 iff G_1 precedes G_2 in textual order.

Once these constraints are satisfied, other selection criteria can be applied, such as Gödel delay rules for negated calls, inequality, and predicates with delay declarations, and the Andorra principle for parallel execution of determinate calls.

Computation rule and implementation of commits are discussed at some length, and in a slightly more general setting, in [5], where outlines of proofs for soundness and completeness properties of the above rules are given.

3.2 Delay declarations

Gondor's implementation of delay declarations relies on Andorra's suspension mechanism. In Andorra, each call is tested for determinacy, and the call may be suspended depending on the outcome of the test. The determinacy test, in turn, is composed of elementary tests for instantiation on the call's arguments and of their subterms. These tests are implicitly carried out by the determinacy code introduced by the Andorra preprocessor, by they may also be programmed explicitly by means of built-in predicates in Andorra-I Prolog.

The purpose of delay declarations is also to suspend a call according to the way its arguments are instantiated. The Gondor preprocessor includes tests in the definition of each predicate that has delay declarations. More precisely, a predicate p for which delay declarations exist is renamed, say as p', and p's definition is replaced by one whose statements contain the tests required by the delay declarations. If the tests for a delay declaration are satisfied, a call to p' is issued, thus executing p's original definition.

Again, we depart from the standard Gödel syntax, and also from the standard Andorra-I Prolog syntax. Delay declaration are represented as:

$$\begin{aligned} \textit{delay–declaration} &\longrightarrow \textit{atom} \mapsto \textit{delay–condition} \\ \textit{delay–condition} &\longrightarrow \textit{simple–condition} \end{aligned}$$

$$
\begin{array}{rcl}
 & | & \mathit{simple\text{–}condition} \wedge \mathit{And\text{–}seq} \\
 & | & \mathit{simple\text{–}condition} \vee \mathit{Or\text{–}seq} \\
\mathit{simple\text{–}condition} & \longrightarrow & \mathit{nonvar}(\mathit{var}) \\
 & | & \mathit{ground}(\mathit{var}) \\
 & | & \mathit{true} \\
 & | & (\mathit{delay\text{–}condition}) \\
\mathit{And\text{–}seq} & \longrightarrow & \mathit{simple\text{–}condition} \\
 & | & \mathit{simple\text{–}condition} \wedge \mathit{And\text{–}seq} \\
\mathit{Or\text{–}seq} & \longrightarrow & \mathit{simple\text{–}condition} \\
 & | & \mathit{simple\text{–}condition} \vee \mathit{Or\text{–}seq}
\end{array}
$$

Statements in Andorra-I Prolog will be represented in a similar way as Gödel statements, except that *nonvar*(*var*) and *ground*(*var*) will stand in that context for the built-ins that suspend until their argument is instantiated or, respectively, ground. Symbols '&' and '|' denote, respectively, sequential conjunction and Andorra-I Prolog commit.

In the following, p is a predicate of arity a defined by $D = \{p(s_{i1}, \ldots, s_{ia}) \leftarrow B_i\}_{i=1}^{n}$, and the set of p's delay declarations is $\Xi = \{K_i\}_{i=1}^{m}$, with

$$K_i = p(t_{i1}, \ldots, t_{ia}) \mapsto Y_i(v_1, \ldots, v_{r_i})$$

where $Y_i(v_1, \ldots, v_{r_i})$ is a delay condition on the variables $v_1, \ldots, v_{r_i}$ occurring in terms $t_{i1}, \ldots, t_{ia}$.

The definition for p is then translated according to the following algorithm:

1. For each K_i, rewrite Y_i in disjunctive normal form:

   $$Y_{i1}(v_1, \ldots, v_{r_i}) \vee \ldots \vee Y_{il_i}(v_1, \ldots, v_{r_i})$$

   where each $Y_{i1}(v_1, \ldots, v_{r_i})$ is of the form $\epsilon_1 \wedge \ldots \wedge \epsilon_{r_i}$ and each ϵ_k is *nonvar*(v_k), or *ground*(v_k), or *true*.
2. With $\{t_{ik_1}, \ldots, t_{ik_h}\}$ the non-variable arguments of the atom in K_i, let

   $$\Delta_i = \mathit{nonvar}(x_{k_1}) \wedge \ldots \wedge \mathit{nonvar}(x_{k_h}) \qquad \Delta_i' = x_{k_1} \equiv t_{ik_1} \wedge \ldots \wedge x_{k_h} \equiv t_{ik_h}$$

   where each x_k is a unique variable and '$\equiv$' stands for equality.
3. For each K_i, let D_i' be the set of l_i statements

   $$\begin{array}{rcl}
   \{p(x_1, \ldots, x_a) & \leftarrow & \Gamma_{i1} \mid p'(x_1, \ldots, x_a)\,, \\
   & \vdots & \\
   p(x_1, \ldots, x_a) & \leftarrow & \Gamma_{il_i} \mid p'(x_1, \ldots, x_a)\}
   \end{array}$$

   with $\Gamma_{ij} = \Delta_i \;\&\; \Delta_i' \;\&\; Y_{ij}(v_1, \ldots, v_{r_i})$.
4. Let $D' = \cup\{D_i'\}_{i=1}^{m}$ be p's new definition.
5. Define p' by $D'' = \{p'(s_{i1}, \ldots, s_{ia}) \leftarrow B_i\}_{i=1}^{n}$.

In Step 2, a new variable is associated with each non-variable argument in the atom of each delay declaration. These variables are tested by expressions Δ_i and Δ_i'. The former suspends until the corresponding call arguments are instantiated, the latter checks that the call arguments unify with the declaration arguments. When the tests have been passed, expression $Y_{ij}(v_1, \ldots, v_{r_i})$ checks if one of the alternative components of the delay condition is verified, suspending if necessary. After this test has been satisfied, the commit is executed and control passes to p', which is identical to the original predicate but for its name.

3.3 Safe negation, inequality, conditionals

Safe negation is implemented by replacing each negated formula by a call to a new predicate whose arguments are the variables of the formula that are not existentially quantified. This predicate calls the Andorra builtin that tests for groundness (and suspends otherwise), then calls the standard unsafe negation of the formula, implemented with Prolog's `call/1` and cut.

This transformation takes place in the following steps:

1. Let F be an expression of the form $\sim G$, where G is a formula, and let $V = \{v_1, \ldots, v_r\}$ be the set of the free variables occurring in F.
2. Replace F with $p'(v_1, \ldots, v_r)$, where p' is a new predicate symbol.
3. Define p' by

$$D'' = \{p'(v_1, \ldots, v_r) \leftarrow ground(v_1) \wedge \ldots \wedge ground(v_r) \ \& \ not(G)\}$$

where $not(G)$ is the unsafe negation of G.

Inequality is simply (and incompletely) implemented as the safe negation of equality. This is more restrictive than required by Gödel, which allows a call to inequality to fail if its arguments can unify without binding any non-underscore variable.

Conditionals are replaced by a call to a predicate p' that first checks for groundness the variables that are not existentially quantified, then calls another predicate p'' that uses the Prolog cut to select the appropriate branch of the conditional. Conditionals where the condition is existentially quantified (*quantified conditionals*) will be expressed as

$$condq(condition, then\text{–}part, else\text{–}part)$$

where *else–part* may be empty. Unquantified conditionals are similarly expressed as

$$cond(condition, then\text{–}part, else\text{–}part)\ .$$

The transformations applied to conditionals are then as follows:

1. Let F be a conditional of the form $cond(C, T, E)$ or $condq(C, T, E)$, and let $V = \{v_1, \ldots, v_r\}$ the set of the free variables occurring in F.
2. Replace F with $p'(v_1, \ldots, v_r)$, where p' is a new predicate symbol.
3. Define p' by

$$D'' = \{p'(v_1, \ldots, v_r) \leftarrow ground(v_1) \wedge \ldots \wedge ground(v_r) \ \& \ p''(v_1, \ldots, v_r)\}$$

where p'' is a new predicate symbol.

4. If F is unquantified, define p'' by

$$D''' = \{p''(v_1, \ldots, v_r) \leftarrow C \wedge\, ! \wedge T\ ,$$
$$p''(v_1, \ldots, v_r) \leftarrow E\}$$

otherwise by

$$D''' = \{p''(v_1, \ldots, v_r) \leftarrow C \wedge T \wedge\, !\ ,$$
$$p''(v_1, \ldots, v_r) \leftarrow E\}$$

The definition of p'' for quantified conditionals allows multiple solutions for C and T to be found.

4 Related work

Program transformations and parallel implementation of logic languages are the subject of many research projects. In this section, we only attempt to refer to very few works that, in this author's view and to his knowledge, are particularly close to the topics of this paper.

Another parallel implementation of Gödel is being developed [17] at the University of Uppsala. This implementation is based on the Reform Prolog system [1], that uses *Reform compilation* [16] to achieve *recursion-parallelism*, a form of AND-parallelism where recursive invocations of a predicate are executed in parallel.

Gödel's delay declarations and conditionals are inherited from NU-Prolog [24]. A parallel implementation of NU-Prolog is NUA-Prolog [18].

Source-to-source transformations from Prolog to the Andorra Kernel Language [9] have been developed [3] at the Universidad Politécnica de Madrid.

5 Conclusions

The translation techniques described in this paper have implemented as a tool, written in SICStus Prolog, that has been used to produce Andorra-I Prolog programs from Gödel programs. The resulting programs have been executed on a shared memory multiprocessor. No performance measurements have been made, as this research is aimed at establishing basic implementation techniques for Gödel. Gondor is clearly an early prototype, and there is much scope for improvements in efficiency. Some improvements will be gained by refining the transformation rules, that in some cases lead to repeated computations, and by merging the generation of Andorra-I Prolog code with determinacy analysis.

The main differences between standard Gödel and the language implemented by Gondor (aside from the user interface and constraints, that have been introduced after Gondor was designed) are the computation rule and safe pruning. In most cases, Gondor-processed programs give the same results as programs executed on the Gödel interpreter, except when particular combinations of commits lead to different pruning steps in the original and the transformed program. This happens, for example, when the goal `<- P( x )` is evaluated with respect to the program

```
P( x ) <- { G1 & { G2 }_2 }_1 & Q( x ).
P( x ) <- { G3 }_1 & R( x ).
P( x ) <- { G4 }_2 & S( x ).

Q( C ).  R( D ).  S( E ).      G2.     G3.     G4.
```

With the standard computation rule, `G1` is selected first in the first statement for `P`, `G1` fails and `{G2}_2` is not executed, so that the third statement is not pruned. Solutions `x = D` and `x = E` are then available. With the computation rule enforced by the transformations, in the first statement `G2` is executed first and the third statement is pruned; then `G1` fails, and only solution `x = D` remains. Although the computation rule implemented by Gondor is different from the standard Gödel rule, it is similar to the one expected by programmers accustomed to committed choice languages, and commit structures as complex as the one shown in the above example are unlikely to occur in actual Gödel programs.

Safe pruning is not currently supported. It can be implemented in a straightforward way by writing two versions of each procedure using commit: one version is the original definition, the

other is that definition stripped of the commits. One version or another is called according to whether the call being executed is negated or not. This solution has not been adopted as it would produce too much bulk, unless some form of code sharing between the two versions is used. An other possible approach would require a small addition to the Andorra implementation: "conditional pruning" instructions could be introduced, whose effect would depend on the state of a flag set in the execution of negated calls.

This research shows that there is no fundamental obstacle to a parallel implementation of Gödel on a multiprocessor architecture, and demonstrates the adequacy of the Andorra system to support a language, such as Gödel, with a few unconventional feàtures. The techniques used in Gondor should be useful in future developments of parallel Gödel implementations, and could be used in other applications, such as metaprogramming techniques for committed choice languages.

Acknowledgements

The research described in this paper has been carried out at the Department of Computer Science of the University of Bristol, under supervision of John W. Lloyd, and in collaboration with David H. D. Warren and his group. I wish to thank them and their co-workers. Very special thanks to Antony Bowers, who introduced me to the language and the machinery of its interpreter, and to Vítor Santos Costa, Inês Dutra, and Rong Yang, who explained me ideas, features and quirks of the Andorra system. John Gallagher kindly let me use his program for term generalization. All (and others) have given me time, ideas, informations, and a very nice environment to work and live in.

References

[1] J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: The Language and its Implementation. In *International Conference on Logic Programming*, Budapest, 1993. Submitted.

[2] K.A. Bowen and R.A. Kowalski. Amalgamating language and metalanguage in logic programming. In K.L. Clark and S.-A. Tarnlund, editors, *Logic Programming*, pages 153–172. Academic Press, 1982.

[3] F. Bueno and M. Hermenegildo. Towards a Translation Algorithm from Prolog to the Andorra Kernel Language. In *Sesto Convegno sulla Programmazione Logica GULP-91*, pages 489–503, 1991.

[4] L. Cardelli and P. Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17:471–522, 1985.

[5] A. Domenici. Implementing labelled commits. Unpublished manuscript, Department of Computer Science, University of Bristol, 1991.

[6] A. Domenici. The Gondor preprocessor. Technical Report ARTS Lab 92-09, Scuola Superiore S. Anna, Pisa, 1992.

[7] A. Domenici. *Un'implementazione parallela del linguaggio Gödel.* PhD thesis, Dip. di Ingegneria dell'Informazione: Elettronica, Informatica, Telecomunicazioni, U. di Pisa, 1992.

[8] S. Haridi and P. Brand. Andorra Prolog: An integration of Prolog and committed choice languages. In *Proceedings of the International Conference on Fifth Generation Computer Systems,* Tokyo, pages 745–754, 1988.

[9] S. Haridi and S. Janson. Kernel Andorra Prolog and its computation model. In *7th International Conference on Logic Programming*. MIT Press, 1990.

[10] L. Van Hentenryck. *Constraint satisfaction in Logic Programming*. MIT Press, 1989.

[11] P.M. Hill and J.W. Lloyd. Analysis of meta-programs. In H.D. Abramson and M.H. Rogers, editors, *Meta-Programming in Logic Programming*, pages 23–52. MIT Press, 1989. Proceedings of the Meta88 Workshop, June 1988.

[12] P.M. Hill and J.W. Lloyd. The Gödel report. Technical Report TR-91-02, Department of Computer Science, University of Bristol, 1991.

[13] P.M. Hill and J.W. Lloyd. The Gödel report (preliminary version). Technical Report TR-91-02, Department of Computer Science, University of Bristol, 1991.

[14] P.M. Hill, J.W. Lloyd, and J.C. Shepherdson. Properties of a pruning operator. *Journal of Logic and Computation*, 1(1):99–143, 1990.

[15] J.W. Lloyd. Designing Logic Programming Languages. Technical Report TR-92-01, Department of Computer Science, University of Bristol, 1992. Invited paper, CompEuro, The Hague, May 1992.

[16] H. Millroth. Reforming Compilation of Logic Programs. In *International Symposium on Logic Programming*, San Diego, Calif., 1991. MIT Press.

[17] H. Millroth. personal communication, Dec. 1992.

[18] D. Palmer and L. Naish. NUA-Prolog: An Extension of the WAM for Parallel Andorra. In K. Furukawa, editor, *8th International Conference on Logic Programming*, Paris , pages 430–442, Paris, 1991. MIT Press.

[19] J.C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 5, pages 135–151, 1969.

[20] V. Santos Costa, D.H.D. Warren, and R. Yang. Andorra-I: A parallel Prolog System that transparently exploits both and- and or-Parallelism. In *ACM Symposium on Principles and Practices of Parallel Programming*, 1991.

[21] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I engine : A parallel implementation of the Basic Andorra model. In K. Furukawa, editor, *8th International Conference on Logic Programming*, Paris, Paris, 1991. MIT Press.

[22] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I preprocessor: Supporting full Prolog on the Basic Andorra model. In K. Furukawa, editor, *8th International Conference on Logic Programming*, Paris, pages 443–456, Paris, 1991. MIT Press.

[23] E. Shapiro. The family of concurrent logic programming languages. *ACM Computing Surveys*, 21(3):412–510, 1989.

[24] J. A. Thom and J. Zobel. Nu-prolog reference manual, version 1.3. Technical report, Machine Intelligence Project, Department of Computer Science, University of Melbourne, 1988.

[25] D. H. D. Warren. An abstract prolog instruction set. Technical Note 309, SRI International, 1983.

[26] R. Yang, T. Beaumont, I. Dutra, V. Santos Costa, and D.H.D. Warren. Performance of the Compiler-based Andorra-I. In *9th International Conference on Logic Programming*, Budapest, 1993. Submitted.

COMPLETENESS OF SOME TRANSFORMATION STRATEGIES FOR AVOIDING UNNECESSARY LOGICAL VARIABLES

Maurizio Proietti

IASI-CNR
Viale Manzoni 30
00185 Roma, Italy
proietti@iasi.rm.cnr.it

Alberto Pettorossi

Electronics Department
University of Rome II
00133 Roma, Italy
adp@iasi.rm.cnr.it

ABSTRACT

An unnecessary variable of a logic clause is a variable which either occurs more than once in the body or does not occur in the head. Unnecessary variables often cause inefficiency in program execution, because they force redundant computations or the construction of unnecessary intermediate structures.

We consider the problem of applying transformation strategies based on the application of unfold/fold rules, with the goal of eliminating the unnecessary variables from a given program. These strategies are based on various versions of the Elimination Procedure presented in a previous paper of ours [20].

We have proved some completeness results for those strategies. Our notion of completeness can be formulated as follows: given a set of transformation rules R, *we say that a strategy* S *is complete w.r.t.* R *iff for any given program* P, *we can transform* P *by using the strategy* S *into an equivalent program without unnecessary variables, if the elimination of the unnecessary variables of* P *is possible by an arbitrary use of the rules in* R.

1. Introduction

A variable X of a clause C in a logic program is said to be *unnecessary* if at least one of the following two conditions are true: 1) X occurs more than once in the body of C (in this case we say that X is a *shared* variable), 2) X does not occur in the head of C (in this case we say that X is an *existential* variable).

The use of unnecessary variables is often helpful for writing logic programs which can easily be understood and proved correct w.r.t. their intended meaning. Indeed, a popular style of programming, which we may call *compositional*, consists in decomposing a given goal in small and easy subgoals, then writing pieces of program which solve these small subgoals, and finally, composing the various pieces together. In logic programming this final composition is often performed by means of variables shared among the atoms which denote the computations for solving the various subgoals.

Unfortunately, this programming style often produces inefficient programs, because the composition of the various subgoals does not take into account the interactions which may occur among the evaluations of these subgoals. For instance, let us consider a clause of the form:

$$p(X) \leftarrow q(X,Y), r(Y).$$

where for solving the goal p(X) we are required to solve q(X,Y) and r(Y). Y is an unnecessary variable being shared and existential, and its binding is not explicitly needed because Y does not

This work has been partially supported by "Progetto Finalizzato Sistemi Informatici e Calcolo Parallelo" of CNR under grant n. 89.00026.69, and by Esprit Project, Computational Logic (Compulog II).

occur in the head of the clause. If the construction and the destruction of that binding is expensive, then our program is inefficient.

Similar problems occur when the compositional style of programming is applied for writing programs using other languages, different from logic. In imperative languages one may construct several procedures which are then combined together by using various kinds of sequential or parallel composition. In functional languages, the various small subgoals in which a given goal is decomposed are solved by means of individual functions which are then combined together by using function application or tupling.

There are various papers in the literature which present techniques for improving the efficiency of the evaluation of programs written according to the compositional style of programming. The following two approaches have been followed: 1) the improvement of the evaluator by using, for instance, garbage collection, memoization, and various forms of laziness and coroutining, and 2) the transformation of the given program into a semantically equivalent one which can be more efficiently evaluated by a non-improved evaluator.

We will consider the transformational approach, for which various techniques have been proposed, such as, for instance: *finite differencing* [18], *composition* or *deforestation strategy* [12, 26], *tupling strategy* [19], *promotion strategy* [2], *compiling control* [5], and *unnecessary variable elimination* [20]. (See also [13] for a survey of some transformation techniques, mainly in the imperative and functional cases.)

Many of the transformation techniques we have mentioned, are based on the *unfold/fold* rules introduced by Burstall and Darlington [6]. Unfolding and folding are elementary, semantics preserving transformations which can be combined with suitable properties of the program at hand to obtain more complex, efficiency improving transformations. The algorithms used for combining the transformation rules are called *strategies*.

The degree of automation of many transformation strategies is not fully satisfactory yet, because in the general case they are nondeterministic and require the exploration of a large search space. However, there are relevant subcases which are particularly suitable for mechanization (see, for instance, [7]).

Unfold/fold based transformations have been widely used in the field of logic programming and various specialized versions of these rules have been proved to preserve various semantics of logic programs (see, for instance, [3, 15, 23, 24]). Here we will consider the simple case of definite programs with the least Herbrand model semantics [16] and a slight variant of the transformation rules by Tamaki and Sato, which preserve this semantics. We will then be concerned with the transformation strategies and the problem of their completeness.

We first notice that the unfold/fold rules are not semantically complete, because one can easily find examples of two programs which have the same semantics, and neither of them can be transformed into the other one, using the unfolding and folding rules only [27]. However, since we are interested in the use of unfold/fold transformations for eliminating unnecessary variables, we consider a weaker notion of completeness of strategies which we now define.

DEFINITION 1. (*Complete strategies*) Given a set of transformation rules R, a strategy S is *complete w.r.t.* R iff for any given program P, if P can be transformed into an equivalent program without unnecessary variables by an arbitrary use of the rules in R, then P can be transformed into an equivalent program without unnecessary variables by using the strategy S. ■

Notice that, for the completeness of a strategy S, we do not require that by using S we can obtain *all* programs without unnecessary variables which are equivalent to the given one.

In the sequel, we will consider various sets of transformation rules: 1) the basic one, called UFD,

which contains the unfolding, folding, and definition rules, 2) the UFDR, which includes UFD and the goal replacement rule, and 3) the UFDRC, which includes UFDR and the clause deletion rule.

We will then consider three versions of the strategy for eliminating unnecessary variables presented in [20], and we will prove the completeness of those three strategies w.r.t. UFD, UFDR, and UFDRC, respectively.

2. Preliminary Definitions

Given a term t, we denote by vars(t) the set of variables occurring in t. Similar notation will be used for variables occurring in atoms, goals, and clauses. Given a clause C, we denote its head by hd(C) and its body by bd(C).

Given a syntactic structure S, we denote by preds(S) the set of predicate symbols occurring in S. In particular, given the program P, preds(P) denotes the set of predicate symbols occurring in P.

A predicate symbol p in preds(P) is said to be a *top predicate* iff it occurs in the head of a clause in P and not elsewhere in P.

2.1 The Transformation Rules

We now briefly describe the rules which will be used for transforming programs. We assume that when the transformation rules are applied, two distinct clauses do *not* have variables in common.

Unfolding Rule. Let C and D be two clauses and A be an atom in bd(C) unifiable with hd(D), with most general unifier θ. By *unfolding* C *w.r.t.* A *using* D we derive a clause U such that: 1) $hd(U) = hd(C)\theta$, and 2) $bd(U) = ((bd(C) - \{A\}) \cup bd(D))\theta$.

Folding Rule. Let C and D be two clauses and B a subset of bd(C) for which there exists a substitution θ such that $B = bd(D)\theta$. Consider the clause F such that: 1) hd(F) = hd(C) and 2) $bd(F) = (bd(C) - B) \cup hd(D)\theta$. Suppose that by unfolding clause F w.r.t. $hd(D)\theta$ using D we get a variant of C. Then we say that F is derived by *folding* C *w.r.t.* B *using* D.

Definition Rule. Let P be a program. By *definition* from P we derive a clause D of the form $newp(X_1,\ldots,X_m) \leftarrow A_1,\ldots,A_n$, such that: 1) newp does not occur in P, 2) $X_1,\ldots,X_m$ are distinct variables, and 3) every predicate occurring in the body of D occurs in P as well. We say that newp is the *predicate defined by* D.

Goal Replacement Rule. We assume that we are given a finite set $L = \{G_i \equiv H_i \mid i = 1,\ldots,n\}$, where for i=1,...,n, the *replacement law* $G_i \equiv H_i$ denotes an ordered pair of finite sets of atoms. Let C be a clause, B a subset of bd(C), $G \equiv H$ a replacement law in L, and θ a substitution for the variables of G such that: a) $vars(C) \cap vars(G \equiv H) = \emptyset$, and b) $B = G\theta$. By *replacement* of B in C using $G \equiv H$ we derive the clause D such that: 1) $hd(D) = hd(C)$ and 2) $bd(D) = (bd(C) - B) \cup H\theta$.

Notice that we do *not* assume that $\equiv$ is commutative, in the sense that we use $G \equiv H$ for replacing $G\theta$ by $H\theta$, but not viceversa.

Clause Deletion Rule. Given a program P and a set of atoms B, we say that B is *failing in* P iff it contains an atom which is *not* unifiable with the head of any clause in P. If for some clause C in P, bd(C) is failing in P, then C can be deleted from P. In this case we will also say that from C we derive the *true clause* T (recall that for any program P we have that $P \wedge T$ is equivalent to P).

2.2. The Transformation Process

For preserving the semantics of the program to be transformed, the above transformation rules should be applied according to some metarules. For describing those metarules, it is helpful to

represent the transformation process as a set of trees as we now specify.

Suppose that we are given a program P and a set of replacement laws L such that all predicates occurring in L occur also in P.

A *transformation forest* F for P and L is a set of directed trees whose nodes are labeled by clauses and whose arcs are labeled by transformation rules. By Roots(F) and Leaves(F) we denote the sets of all clauses different from T which label the roots and the leaves, respectively, of the trees in F.

The transformation forest F is constructed in a non-deterministic way as follows. We start from F equal to the empty set of trees and, given a transformation forest E, we may either halt or expand E according to one of the following metarules:

1) If C is a clause of P not labeling any root node of E, then we add to E a new tree with one node only labeled by C.
2) (*Definition Step*) If a clause D can be derived by applying the definition rule and the predicate defined by D does not occur in any root of the trees of E, then we add to E a root labeled by D.
3) (*Unfolding Step*) Let N be a node of a tree in E labeled by a clause C and let A be an atom in bd(C) which is unifiable with the head of at least one clause in P.
If $\{U_1,\ldots,U_k\}$ is the largest set of clauses which can be derived by unfolding C w.r.t. A using clauses in P, then we construct the sons $N_1,\ldots,N_k$ of N. For i=1,...,k, node N_i is labeled by U_i and the arc from N to N_i is labeled by 'unfolding'.
4) (*Folding Step*) Let N be a node of a tree T in E labeled by a clause C and let D be a clause labeling a root of E such that the predicate occurring in hd(D) is a top predicate of P. Suppose that on the path from the root of T to N there exists an arc labeled by 'unfolding'. If clause G can be derived by folding C using D, then we construct a son N_1 of N. Node N_1 is labeled by G and the arc from N to N_1 is labeled by 'folding'.
5) (*Replacement Step*) Let N be a node in E labeled by a clause C. If by replacement of a goal in the body of C using a replacement law in L we derive a clause D, then we construct a son N_1 of N. Node N_1 is labeled by D and the arc from N to N_1 is labeled by 'goal replacement'.
6) (*Deletion Step*) Let N be a node in E labeled by a clause C. If bd(C) is failing in $P \cup Roots(E)$, then we construct a son N_1 of of N. Node N_1 is labeled by T, that is, the true clause, and the arc from N to N_1 is labeled by 'clause deletion'.

During the construction of a tree of a transformation forest we say that Step S2 *is performed after* Step S1 iff the nodes generated by applying S2 are descendants of a node generated by applying S1.

From the transformation forest F for P and L constructed as described above, we derive a new program TransfP = (P – Roots(F)) ∪ Leaves(F). F will also be called a *transformation forest from* P *to* TransfP.

Obviously, the semantic equivalence between P and TransfP depends on the choice of the set of replacement laws L. We will not consider this problem here, and we refer the reader to [24] where it is shown the correctness of a similar set of transformation rules. The main difference between our rules and the ones in [24] is that when we apply the definition, unfolding, and clause deletion rules we take into consideration the initial program P, instead of the current program.

3. Elimination of Unnecessary Variables by Transformation: An Example

We now present a program for computing the relation common(T,U,V) which holds among the trees T, U, and V iff T contains U and V contains U, where the binary relation 'contains' is defined as follows.

Given two labeled binary trees T and U, we say that T *contains* U iff there exists a sequence of trees $T_1,\ldots,T_n$ such that T_1=T, T_n=U, and for i=1,...,n–1, T_{i+1} can be obtained from T_i by replacing a subtree S of T_i by a subtree of S itself.

1. common(T, U, V) ← contains(T, U), contains(V, U).
2. contains(tip(N), tip(N)).
3. contains(t(L, N, R), tip(N)).
4. contains(t(L1, N, R1), t(L2, N, R2)) ← contains(L1, L2), contains(R1, R2).
5. contains(t(L, N, R), U) ← contains(L, U).
6. contains(t(L, N, R), U) ← contains(R, U).

In clause 1, U is an unnecessary variable, which is a shared variable. We would like to transform the above program into an equivalent one without shared variables.

This task can be performed by the transformation steps described below. The corresponding transformation forest is depicted in Figure 1.

By unfolding clause 1 w.r.t. the first atom of its body we get the following clauses:

7. common(tip(N), tip(N), V) ← contains(V, tip(N)).
8. common(t(LT, N, RT), tip(N), V) ← contains(V, tip(N)).
9. common(t(LT, N, RT), t(LU, N, RU), V) ← contains(LT, LU), contains(RT, RU), contains(V, t(LU, N, RU)).
10. common(t(LT, N, RT), U, V) ← contains(LT, U), contains(VT, U).
11. common(t(LT, N, RT), U, V) ← contains(RT, U), contains(V, U).

From the body of clause 9 we introduce by the definition rule a new predicate as follows:

12. new(LT, N, RT, V, LU, RU) ← contains(LT, LU), contains(RT, RU), contains(V, t(LU, N, RU)).

By folding clause 9 using clause 12, we get:

13. common(t(LT, N, RT), t(LU, N, RU), V) ← new(LT, N, RT, V, LU, RU).

By folding clause 10 using clause 1, we get:

14. common(t(LT, N, RT), U, V) ← common(LT, U, V).

By folding clause 11 using clause 1 again, we get:

15. common(t(LT, N, RT), U, V) ← common(RT, U, V).

By unfolding clause 12 w.r.t. the third atom of its body, we get the following three clauses:

16. new(LT, N, RT, t(LV, N, RV), LU, RU) ← contains(LT, LU), contains(RT, RU), contains(LV, LU), contains(RV,RU).
17. new(LT, N, RT, t(LV,M,RV), LU, RU) ← contains(LT,LU), contains(RT,RU), contains(LV, t(LU,N, RU)).
18. new(LT, N, RT, t(LV,M,RV), LU, RU) ← contains(LT,LU), contains(RT,RU), contains(RV,t(LU,N, RU)).

By folding clause 16 using clause 1 we get:

19. new(LT, N, RT, t(LV, N, RV), LU, RU) ← common(LT, LU, LV), contains(RT, RU), contains(RV, RU).

By folding clause 19 using clause 1 we get:

20. new(LT, N, RT, t(LV, N, RV), LU, RU) ← common(LT, LU, LV), common(RT, RU, RV).

By folding clause 17 using clause 12 we get:

21. new(LT, N, RT, t(LV, M, RV), LU, RU) ← new(LT, N, RT, LV, LU, RU).

By folding clause 18 using clause 12 we get:

22. new(LT, N, RT, t(LV, M, RV), LU, RU) ← new(LT, N, RT, RV, LU, RU).

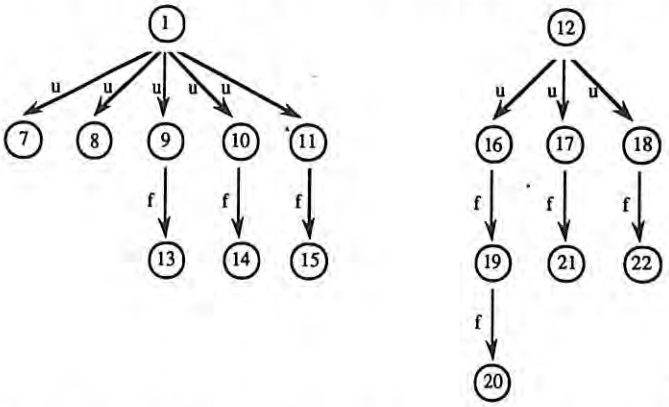

Figure 1. A transformation forest. Numbers refer to clauses and labels 'u' and 'f' stand for 'unfolding' and 'folding', respectively.

The final program can be obtained from the initial one (that is, clauses 1 through 6), by replacing clause 1 by the clauses labeling the leaves of the transformation forest of Figure 1. These leaf-clauses do not contain shared variables, and therefore, the final program does not contain any unnecessary variable.

4. A Complete Elimination Strategy Using the Unfolding, Folding, and Definition Rules

In this section we will consider the set of transformation rules UFD = {unfolding, folding, definition}. The application of these rules will be considered w.r.t. the given initial program in the sense that, for instance, we will perform the unfolding steps using the clauses in the initial program.

Throughout this section we will denote the initial program by Prog. We will then introduce a strategy and we will prove its completeness w.r.t. UFD. This strategy is a slight variant of the one presented in [20].

DEFINITION 2. (*Blocks of a clause*) Consider a set A of atoms. We define a binary relation $\downarrow$ over A as follows. Given two atoms A_1 and A_2 in A, we have that:

$A_1 \downarrow A_2$ iff $vars(A_1) \cap vars(A_2) \neq \emptyset$.

Let $\Downarrow$ be the reflexive and transitive closure of the relation $\downarrow$ over A. We denote by Part(A) the *partition* of A into *blocks* w.r.t. the equivalence relation $\Downarrow$. ■

Example 1. Let C be the following clause taken from the example of Section 3:

16. new(LT, N, RT, t(LV, N, RV), LU, RU) ← contains(LT, LU), contains(RT, RU), contains(LV, LU), contains(RV,RU).

Part(bd(C)) consists of the following two blocks: {contains(LT, LU), contains(LV, LU)} and

{contains(RT, RU), contains(RV, RU)}. ■

DEFINITION 3. (*Block-folding*) Let C and D be two clauses without common variables. Suppose that C can be folded using D, that is, there exist a subset B of bd(C) and a substitution θ such that $B = bd(D)\theta$. Consider the clause F derived by folding C w.r.t. B using D.
We say that F has been derived by *block-folding* from C iff the following conditions hold:

1) $Part(bd(D\theta)) \subseteq Part(bd(C))$,
2) X is an existential variable of D iff $X\theta$ is an existential variable of C, and
3) $vars(hd(D)) \subseteq vars(bd(D))$.

We also say that clause D is a *block-generalization* of C. ■

PROPOSITION 4. Suppose that, by using rules in UFD, the program Prog can be transformed into a program Q without unnecessary variables.
Then there exists a transformation forest F from Prog to a program without unnecessary variables such that for each tree T in F: 1) in T we perform precisely one unfolding step, 2) every folding step is a block-folding step, 3) all folding steps are performed after the unfolding step, and 4) the root of T is labeled by a clause whose body has one block only and this block contains at least one unnecessary variable. ■

Let us now introduce the following strategy.
THE ELIMINATION STRATEGY Σ.
Input. A program Prog.
Output. A program TransfProg equivalent to Prog such that no unnecessary variable occurs in TransfProg.

InClauses := {C | C is a clause in Prog with unnecessary variables};
Defs := {D | D is a clause of InClauses such that the predicate of hd(D) is a top predicate of Prog};
TransfProg := Prog – InClauses;
while there exists a clause $C \in$ InClauses *do*
  InClauses := InClauses – {C};
  Unfolding. Select an atom A in bd(C) which is unifiable with the head of at least one clause in Prog, and collect in a set C_U all clauses which can be derived by unfolding C w.r.t. A using clauses of Prog;
  Definition. For every clause E in C_U and for every block $B \in Part(bd(E))$ containing at least one unnecessary variable, consider a clause NewB such that:
    1) NewB is a block-generalization of E,
    2) B is an instance of bd(NewB)
  and add NewB to both InClauses and Defs, unless in Defs there exists a block-generalization of NewB;
  Folding. For every clause E in C_U add to TransfProg the clause derived from E by folding every block B of Part(bd(E)) with at least one unnecessary variable (in particular, if no block of Part(bd(E)) has unnecessary variables, then add E to TransfProg). ■

Notice that the strategy Σ is nondeterministic. Indeed, for the unfolding phase in Σ we have to select an atom in the body of the clause to be unfolded, and for the definition phase in Σ we have to choose one among several possible block-generalizations of the given clause.

We will not address here the problem of controlling the nondeterminism of Σ (and similarly for the extensions of Σ, called Σ_R and Σ_{RC}, which we will introduce below). We only want to mention

that one can specialize the techniques presented in [21].

THEOREM 5. The strategy Σ is partially correct. Moreover, Σ is complete w.r.t. the set of transformation rules UFD.
PROOF. 1) Partial correctness. By the correctness of the transformation rules [24], we have that TransfProg is equivalent to Prog. The absence of unnecessary variables in TransfProg follows from the fact that, during the folding phase in Σ, the clauses which are added to TransfProg do not contain any unnecessary variable.
2) Completeness. By Proposition 4 and by the fact that the use of 'old' definitions for performing folding steps does not affect the termination of Σ. ■

5. Enhancing The Elimination Strategy by Using Goal Replacement

In this section we consider also the goal replacement rule in the set of transformations which may be performed for eliminating the unnecessary variables of a given program Prog. We will then present an extension of the strategy Σ and we will prove its completeness w.r.t. UFDR = {unfolding, folding, definition, goal replacement}.

Throughout this section, we assume that L is the set of replacement laws which may be used for the application of the goal replacement rule.

DEFINITION 6. We say that $G \equiv H$ in L is a *block-replacement law* iff Part(G) is made out of one block only. ■

Example 2. The following replacement law, expressing the associativity of append, is a block-replacement law:
append(L, M, N), append(N, P, Q) ≡ append(L, R, Q), append(M, P, R) ■

PROPOSITION 7. Suppose that every law in L is a block-replacement law, no top predicate of Prog occurs in L, and preds(L) $\subseteq$ preds(Prog). Suppose also that, by using rules in UFDR, Prog can be transformed into a program without unnecessary variables.
Then there exists a transformation forest F from Prog to a program without unnecessary variables such that for each tree T in F: 1) in T we perform at most one unfolding step and if T contains a folding step then in T we perform precisely one unfolding step, 2) every folding step is a block-folding step, 3) all folding steps are performed after the unfolding step and the replacement steps, and 4) the root of T is labeled by a clause whose body has one block only and this block contains at least one unnecessary variable. ■

Let us now consider the following extended strategy.
THE ELIMINATION STRATEGY Σ_R.
Input. A program Prog and a set L of replacement laws such that every law in L is a block-replacement law, no top predicate of Prog occurs in L, and preds(L) $\subseteq$ preds(Prog).
Output. A program TransfProg equivalent to Prog such that no unnecessary variable occurs in TransfProg.

InClauses := {C | C is a clause in Prog with unnecessary variables};
Defs := {D | D is a clause of InClauses such that the predicate of hd(D) is a top predicate of Prog};
TransfProg := Prog – InClauses;
while there exists a clause C ∈ InClauses *do*
    InClauses := InClauses – {C};
    TransfC := {C};

Replacement & Unfolding. Replace C in TransfC by a clause D obtained from C by zero or more applications of the goal replacement rule;
if D does not contain any unnecessary variable
then Add D to TransfProg and delete it from TransfC
else *begin* Select an atom A in bd(D) which is unifiable with the head of at least one clause in Prog, and replace D in TransfC by the set of all clauses which can be derived by unfolding D w.r.t. A using clauses of Prog;
Apply zero or more times the goal replacement rule to the clauses of TransfC
end;
Definition. For every clause E in TransfC and for every block B ∈ Part(bd(E)) containing at least one unnecessary variable, consider a clause NewB such that:
1) NewB is a block-generalization of E,
2) B is an instance of bd(NewB)
and add NewB to InClauses and Defs, unless in Defs there exists a block-generalization of NewB;
Folding. For every clause E in TransfC add to TransfProg the clause derived from E by folding every block B of Part(bd(E)) with at least one unnecessary variable (in particular, if no block of Part(bd(E)) has unnecessary variables, then add E to TransfProg). ■

THEOREM 8. The strategy Σ_R is partially correct. If every law in L is a block-replacement law, no top predicate of Prog occurs in L, and preds(L) ⊆ preds(Prog), then the strategy Σ_R is complete w.r.t. the set of rules UFDR. ■

6. Completeness Results of a Strategy which includes Clause Deletions

In this section we present an extension of the strategy Σ_R which is complete w.r.t. the set of transformation rules UFDRC = {unfolding, folding, definition, goal replacement, clause deletion}.

Throughout this section, we assume that Prog is the initial program and L is the set of replacement laws which may be used for the application of the goal replacement rule. For L we make the same assumptions we have indicated in Section 5.

PROPOSITION 9. Suppose that, by using rules in UFDRC, Prog can be transformed into a program without unnecessary variables. Then there exist two programs Q and R such that:

- R does not contain unnecessary variables,
- R can be obtained from Q by repeated applications of the clause deletion rule and by deletion of every clause C such that no predicate in preds(Prog) depends on the predicate occurring in hd(C) (where the 'depends' relation is defined in the usual way), and
- there exists a transformation forest F from Prog to Q such that for each tree T labeling a node of F: 1) in T we perform at most one unfolding step and if T contains a folding step then in T we perform precisely one unfolding step, 2) every folding step is a block-folding step, 3) all folding steps are performed after the unfolding step and the replacement steps, and 4) the root of T is labeled by a clause whose body has precisely one block. ■

Let us now consider the following strategy.
THE ELIMINATION STRATEGY Σ_{RC}.
Input. A program Prog and a set L of replacement laws such that every law in L is a block-replacement law, no top predicate of Prog occurs in L, and preds(L) ⊆ preds(Prog).
Output. A program TransfProg equivalent to Prog such that no unnecessary variable occurs in TransfProg.

InClauses := {C | C is a clause in Prog with unnecessary variables};
Defs := {D | D is a clause of InClauses such that the predicate of hd(D) is a top predicate of Prog};
TransfProg := Prog – InClauses;
while there exists a clause C ∈ InClauses *do*
InClauses := InClauses – {C};
TransfC := {C};
Clause Deletion. Repeatedly apply the clause deletion rule to TransfProg ∪ InClauses and also delete from TransfProg ∪ InClauses every clause G such that no predicate in preds(Prog) depends on the predicate occurring in hd(G).
Replacement & Unfolding. Replace C in TransfC by a clause D obtained from C by zero or more applications of the goal replacement rule;
if bd(D) is failing in Prog
then Remove D from TransfC
else if bd(D) = ∅
then Add D to TransfProg and delete D from TransfC
else *begin* Select an atom A in bd(D) which is unifiable with the head of at least one clause in Prog, and replace D in TransfC by the set of all clauses which can be derived by unfolding D w.r.t. A using clauses of Prog;
Apply zero or more times the goal replacement rule to the clauses of TransfC and delete from TransfC each clause whose body is failing in Prog
end;
Definition. For every clause E in TransfC and for every block B ∈ Part(bd(E)) consider a clause NewB such that:
1) NewB is a block-generalization of E,
2) B is an instance of bd(NewB)
and if a block-generalization of NewB does not exists in Defs then add NewB to InClauses and Defs;
Folding. For every clause E in TransfC add to TransfProg the clause derived from E by folding every block B of Part(bd(E)) (in particular, if Part(bd(E)) is empty, then add E to TransfProg).■

The reader should notice that in the Definition phase of this strategy we have considered the new clause NewB for any block (with or without unnecessary variables) in Part(bd(E)).

This is due to the fact that the elimination of the unnecessary variables of a block in the body of a clause E may take place because a different block (with or without unnecessary variables) of bd(E) is failing in Prog. Therefore, when clause deletions are possible, we need to process *all* blocks for ensuring the eliminations of *all* unnecessary variables via the strategy Σ_{RC}.

REMARK. The strategy Σ_{RC} can be further optimized. For instance, the Clause Deletion transformation phase can be performed by avoiding, at each execution, the visit of the whole program TransfProg ∪ InClauses generated so far. In particular, if no clause with failing body is found during the Replacement & Unfolding transformation phase, then no clause will be discarded by the next application of Clause Deletion.

THEOREM 10. The strategy Σ_{RC} is partially correct. If every law in L is a block-replacement law, no top predicate of Prog occurs in L, and preds(L) ⊆ preds(Prog), then the strategy Σ_{RC} is complete w.r.t. the set of rules UFDRC. ■

7. Conclusions

We have presented some strategies of transforming logic programs for eliminating the unnecessary

variables. We have shown the completeness of our strategies w.r.t. several sets of transformation rules. Unfortunately, due to space limitations, we were not able to present in detail the proofs of Theorems 5, 8, and 10.

Although we have not proved any formal result which relates the efficiency of the initial program to the one of the transformed program, our practical experience in program transformation shows that, by eliminating unnecessary variables, many redundant computations can be avoided and the program performances can be improved. This improvement realized depends very much on the different sets of replacement laws and the properties of the given initial program.

The interest of the results presented in this paper is also given by the fact that many transformation strategies described in the literature may be rephrased in terms of the strategies for eliminating unnecessary variables. In particular, this is the case for the compiling control technique [5], the techniques for getting tail-recursive programs [1,8], and those for improving the recursive structure of the programs [9].

Finally, we would like to mention that the elimination of unnecessary variables may also be used as a useful transformation method for producing efficient object code. Two examples of this use of the elimination of unnecessary variables are the Bin-Prolog compiler [25] and the compilers supporting AND-parallelism [10].

Bin-Prolog compiles Prolog programs by first transforming them into binary programs, that is, programs with at most one call in the body of every clause, and then using a simplified set of WAM instructions. In [11] it is shown that the Bin-Prolog compiler produces very efficient code if the binary program derived during the initial phase of compilation does not contain unnecessary variables.

The AND-parallel execution of logic programs is particularly simple when, at runtime, no two calls share unbound variables. The strength of our transformation strategies for parallel programs derives from the fact that every program without unnecessary variables during the evaluation of any initial goal without multiple variables does not produce a goal with shared variables.

8. Acknowledgements

Our gratitude goes to Prof. Annalisa Bossi, Prof. Gilberto Filé, and their colleagues at the University of Padua (Italy), and the participants in the Compulog II Meeting in Rome (December 92) for helpful comments and discussions.

9. References

[1] N. Azibi, "TREQUASI: Un système pour la transformation automatique de programmes PROLOG récursifs en quasi-itératifs", PhD thesis, Université de Paris-Sud, Centre D'Orsay, France, 1987.

[2] R.S. Bird, "The Promotion and Accumulation Strategies in Transformational Programming", ACM TOPLAS, vol. 6, no. 4, pp. 487-504, October 1984.

[3] A. Bossi, N. Cocco, and S. Etalle, "Transforming Normal Programs by Replacement", In: Proc. Third International Workshop on Metaprogramming in Logic, Meta '92, Uppsala, Sweden, Lecture Notes in Computer Science 649, pp. 265-279, 1992.

[4] R.S. Boyer and J.S. Moore, "Proving Theorems about LISP Functions", Journal of the ACM, vol. 22, no. 1, pp. 129-144, 1974.

[5] M. Bruynooghe, D. De Schreye, and B. Krekels, "Compiling Control", Journal of Logic Programming, vol. 6, no. 1&2, pp. 135-162, 1989.

[6] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs", Journal of the ACM, vol. 24, no. 1, pp. 44-67, January 1977.

[7] W.N. Chin, "Automatic Methods for Program Transformation", PhD Thesis, Univ. of London, Imperial College, Department of Computing, 1990.

[8] S.K. Debray, "Optimizing Almost-Tail-Recursive Prolog Programs", In: Proc. IFIP International Conference on Functional Programming Languages and Computer Architecture, Nancy, France, 1985. Lecture Notes in Computer Science 201, pp. 204-219, 1985.

[9] S.K. Debray, "Unfold/Fold Transformations and Loop Optimization of Logic Programs", In: Proceeding of the SIGPLAN '88 Conference on Programming Language Design and Implementation, Atlanta, Georgia, 1988.

[10] D. DeGroot, "Restricted AND-Parallelism", In: Proc. of the International Conference on Fifth Generation Computer Systems 1984, North, Holland, pp. 471-478, 1984.

[11] B. Demoen, "On the Transformation of a Prolog Program to a More Efficient Binary Program", In: K.K. Lau and T. Clement (eds.) Logic Program Synthesis and Transformation, Proc. Lopstr '92, Workshops in Computing, Springer-Verlag, 1993.

[12] M.S. Feather, "A System for Assisting Program Transformation", ACM TOPLAS, vol. 4, no. 1, pp.1-20, January 1982.

[13] M.S. Feather, "A Survey and Classification of Some Program Transformation Techniques", In: Proc. TC2 IFIP Working Conference on Program Specification and Transformation, Bad Tölz, Germany, pp. 165-195, 1986.

[14] P.A. Gardner and J.C. Shepherdson, "Unfold/Fold Transformations of Logic Programs", In: J.-L. Lassez and G. Plotkin (eds.), Computational Logic, Essays in Honor of Alan Robinson. MIT Press, pp. 565-583, 1991.

[15] T. Kawamura and T. Kanamori, "Preservation of Stronger Equivalence in Unfold/Fold Logic Program Transformation", Theoretical Computer Science vol. 75, pp. 139-156, 1990.

[16] J.W. Lloyd, *Foundations of Logic Programming*, Springer-Verlag, 2nd edition, 1987.

[17] J.W. Lloyd and J.C. Shepherdson, "Partial Evaluation in Logic Programming", Journal of Logic Programming vol. 11, pp. 217-242, 1991.

[18] R. Paige and S. Koenig, "Finite Differencing of Computable Expressions", ACM TOPLAS, vol. 4, no. 1, pp. 402-454, July 1982.

[19] A. Pettorossi, "Transformation of Programs and Use of Tupling Strategy", In: Proc. Informatica '77, Bled, Yugoslavia, pp. 1-6, 1977.

[20] M. Proietti and A. Pettorossi, "Unfolding-Definition-Folding, in This Order, for Avoiding Unnecessary Variables in Logic Programs", In: J. Maluszynski and M. Wirsing (eds.), Proc. 3rd International Symposium on Programming Language Implementation and Logic Programming, PLILP '91, Passau, Germany, 1991. Lecture Notes in Computer Science 528, pp. 347-358, 1991.

[21] M. Proietti and A. Pettorossi, "Best-first Strategies for Incremental Transformations of Logic Programs", In: K.K. Lau and T. Clement (eds.) Logic Program Synthesis and Transformation, Proc. Lopstr '92, Workshops in Computing, Springer-Verlag, pp. 82-98, 1993.

[22] T. Sato, "An Equivalence Preserving First Order Unfold/Fold Transformation System", In: Proc. 2nd International Conference on Algebraic and Logic Programming, ALP '90, Nancy, France, 1990. Lecture Notes in Computer Science 463, pp. 175-188, 1990.

[23] H. Seki, "Unfold/Fold Transformation of General Logic Programs for the Well-founded Semantics", To appear in: Journal of Logic Programming, Special Issue on Partial Deduction, 1993.

[24] H. Tamaki and T. Sato, "Unfold/Fold Transformation of Logic Programs", In: S.-Å. Tarnlünd (ed.) Proc. 2nd International Conference on Logic Programming, Uppsala, Sweden, pp. 243-251, 1984.

[25] P. Tarau and M. Boyer, "Elementary Logic Programs", In: Proceedings of PLILP '90, P. Deransart and J. Maluszynski (eds.), Springer Verlag, pp. 159-173, 1990.

[26] P. Wadler, "Deforestation: Transforming Programs to Eliminate Trees", In: Proceeedings ESOP '88, Lecture Notes in Computer Science 300, H. Ganzinger (ed.), pp. 344-358, 1988.

[27] H. Zhu, "How Powerful are Folding/Unfolding Transformations?", Techn. Report CSTR-91-2, Brunel University, U.K., 1991.

SEMANTICA II

(Semantics II)

A New Fixpoint Semantics for Prolog

A. Bossi M. Bugliesi M. Fabris
Dip. di Matematica Pura ed Applicata
Università di Padova
Via Belzoni 7, 35131 Padova – ITALY
e-mail: michele@blues.unipd.it, {bossi,fabris}@pdmat1.unipd.it

Abstract

In this paper we present a fixpoint semantics for Prolog defined as an extension of the S-semantics for logic programming. The semantics we present captures, in a goal independent fashion, both the left-to-right selection rule and the depth-first search strategy underling Prolog's evaluation mechanism. We prove the soundness and the completeness of our approach and we show that our construction is rich enough to capture other important properties of the computational behavior of a program, such as existential and universal termination.

1 Introduction

The elegance of the classical approach to the semantics of logic programming has long been considered one of the most appealing features of this programing paradigm as well as one of the reasons for its popularity. The most important contribution of van Enden and Kowalski's seminal paper on the subject was probably represented by the proof of equivalence of the operational, model-theoretic and fixpoint characterizations of the programs' behavious. The different semantic frameworks provide in fact different – but equivalent – theoretical tools for manipulating programs and reasoning about their properties and their equivalence.

However this very same result exposes the weakness of the classical approach. In fact, the correspondence between the operational and declarative interpretations of a program is achieved by assuming a rather poor definition of the program's computational behaviour based on the notion of success set. It has often been advocated that the inadequacy of the semantics was an acceptable tribute to its elegance and that the operational characterization based on SLD-resolution was always at disposal in case of more accurate definitions would be needed. This argument notwithstanding, it is obvious that since programs compute substitutions, then reasoning about their behaviour in terms of success set (or equivalently in terms of Herbrand models) is inadequate. Furthermore, if we use Prolog, or more precisely its "pure" subset to write programs, then there are two more computational aspects which the classical semantics fails to capture: the answer substitutions computed by a program and the control structure underlying the evaluation of a goal.

A considerable amount of work has been recently devoted to the study of both these aspects.

Answer Substitutions. A first solution to the problem of finding a declarative modeling of the computed answer substitutions of a program was provided by the *S-semantics* approach of [14]. The appealing feature of this proposal is that it inherits the elegance of the classical

semantics and it enjoys the same property of correspondence between the operational and declarative interpretations. The relevant difference is that in this case the result holds by assuming a definition of the operational behaviour of a program stated in terms of the program's computed answers. Later extensions ([8, 9]) of the original framework of [14] have then provided further evidence of the effectiveness of this approach to capture other computational (and compositional) properties of the behaviour of *open* programs.

Control. There are two main aspects involved in the control structure underlying the evaluation of a Prolog program: the *selection rule* which makes the ordering among the atoms in a conjunction relevant to the computed answers, and the *search strategy* which imposes an ordering on the program clauses to be tried to reduce the current resolvent.

The first aspect has recently received great attention and two solutions have been proposed in the literature. Both conceived as extension of the S-semantics, they attempt to capture the left-to-right selection rule of Prolog ([17]) and the mechanisms underlying the evaluation of *built-in* predicates such as *var, nonvar, ground* among others ([2]).

As for the second aspect, the first proposal for a declarative semantics with an embedded notion of search strategy, was due to Fitting. Proving properties of Prolog programs constitutes in [15] one of Fitting's motivating arguments. His semantic framework is defined in terms of a fixpoint construction based on a corresponding immediate consequence operator. Interpretations are sets of functions from goals to sequences of substitutions and, accordingly, a model for a program and a goal is an interpretation which associates with the goal the sequence of its computed answers in the program. Although undoubtedly elegant, this approach has a major weakness in that the reseulting semantics is inherently goal dependent: the notions of interpretation and model are in fact sensitive to set of goals chosen to query the program with.

Several other formulations of the semantics of Prolog have been developed after [15], but they all seem – to a certain extent – to lack the declarative flavour peculiar of the classical construction of the semantics of logic programming. We just recall here the denotational frameworks of [10], [5] and [18], and the proposals based on dynamic algebras and transition systems of [6] and [11] and similarly of [13].

A more recent contribution towards a truly declarative modeling of control in logic programming has been developed in [3]. The solution is inspired by the idea of explicitely distinguishing the logical and the control components of Prolog. The control strategy is modeled in terms of an oracle which defines at each computation step, the set of clauses applicable to rewrite the current resolvent. The logical reading of a program results thus unaffected. The program's semantics is defined parametrically on the oracle and this gives to the approach a quite general flavour. However, generality seems to be achieved at the expense of complexity: when specialized to the case of Prolog the framework becomes burdened with a lot of information which, while needed to handle the general case, is in fact irrelevant in the context of the specialization.

1.1 Goals and Outline

In this paper we present a fixpoint reconstruction of the semantics of Prolog which captures both the left-to-right selection rule and the depth-first search strategy underling Prolog's

evaluation mechanism. Our framework is directly inspired by the S-semantics approach of [14] and it inherits two of the distinguishing features of that proposal: the ability to model the computed answer substitutions and, more importantly, the property of goal independence. The semantics of a program is in fact built without any reference to the goals the program will be queried with.

The approach we follow results in a simpler and more direct semantics for Prolog than the one of [3], even though somewhat less general. The idea is to follow the guidelines of [16] and to assume an enriched notion of interpretations and have them include more complex syntactic objects than atoms. An interpretation for a program is thus defined as a set of sequences over an extended Herbrand base $\mathcal{B}_E$ where each sequence in the interpretation is to be regarded as an abstraction of the ordered set of partial answers computed by the program. Equivalently, this amounts to saying that an interpretation provides an abstraction of the set of *resultants* (as defined in [20]) computed by a depth-first & left-to-right evaluation of the *most general* atomic queries for a program. The idea is similar but simpler than the one described in [3]. Non-unit resultants are in fact abstracted upon resorting to the notion of *divergent* atoms. Each resultant of the form A :- B is represented as the divergent atom $\widehat{A}$. $\widehat{A}$ conveys all the relevant information provided by A :- B — that the associated derivation is partial — and abstracts from the body B, which is in fact irrelevant in this context. This very same idea motivated also the approach followed in [12] for declarative modeling the *setof* and *findall* built-in's of Prolog. Divergent atoms are denoted in [12] by $?A$ and interpretations are defined as subsets of an extended base consisting of divergent as well as standard (non-divergent) atoms. The framework we present in this paper shares in fact several other features with the approach proposed in [12], but the significant difference is that we extend these ideas to fit with Prolog's depth-first search strategy.

To this purpose, we define an immediate-consequence operator Φ_P whose least fixpoint consists of the ordered set of sequences over $\mathcal{B}_E$ which represent all the possible partial computations originating from the most general atoms in the program. We then show that, given any particular goal G, the answer substitutions computed for G in P can be obtained by means of a simple projection on the least fixpoint of Φ_P. This result justifies the soundness and completeness of our fixpoint construction and is to be viewed as a direct extension of the correspoding result proved for the S-semantics, where the answer for a any given goal is obtained by simply attempting to unify the goal with an element in the S-model of the program.

We then show that our construction is rich enough to capture other important properties of the computational behavior of a program, such as existential and universal termination for a query. In this second respect, out approach can be seen as an extension of the results presented in [4] (see section 5 for a detailed discussion on this issue).

Plan of the paper. The next section is dedicated to introducing some preliminary definitions and results needed in the sequel of the paper. In section 3 we introduce our semantic characterization and in section 4 we prove the results of soundness and completeness. Finally in section 5 we discuss the relevance our approach in the context of termination and address some of the extensions planned for the near future.

2 Preliminaries

We assume familiarity with the standard definitions for logic programming ([1] and [19] provide the necessary background on the subject). In the sequel of the paper we also make use of the following notation and terminology.

Sequences. Given any set of symbols S we denote by S^* the set of finite sequences of symbols in S. The concatenation of two sequences s_1 and s_2 is denoted by $s_1 :: s_2$ whereas λ stands for the empty sequence. The notation $e \varepsilon s$ is used to denote that e is a member of s.

Substitutions. With *Subst* we denote the space of substitutions and with ϵ and $\theta_{|V}$ we denote respectively the empty substitution and the restriction of θ to the set of variables V. We say that a substitution θ is *less general* than θ' ($\theta \preceq \theta'$) iff there exists a substitution σ such that $\theta = \theta'\sigma$. A *renaming* substitution is a substitution which is a bijection on variables.

Terms, Atoms and Clauses. We say that two terms (atoms or clauses) t_1 and t_2 are *renaming-equivalent* ($t_1 \sim t_2$) iff there exists a renaming α such that $t_1\alpha = t_2$. Given a set of terms (atoms or clauses) $\mathcal{T}$ and a term t, t is said to be *standardized apart* from $\mathcal{T}$ if it shares no variable with any of the elements of $\mathcal{T}$. With $var(\tilde{t})$ we denote the set of variables occurring in the tuple of terms $\tilde{t}$. Also for any atom A, we use the notation $Pred(A)$ to stand for the predicate symbol of A. Finally, we say that a predicate p is *defined* in a program if the program contains at least one clause whose head's predicate symbol p. A program is assumed to be a sequence of definite clauses of the form H :- $\tilde{B}$ where $\tilde{B}$ denotes a conjunction of positive atoms. Unit clauses, clauses with empty body, are denoted by H.

Monotonic Functions and Fixpoints. Given a complete lattice $(L, \subseteq, \bot, \top, \cup, \cap)$ and a function $f : L \mapsto L$, the ordinal powers of f are defined as usual: $f^0(X) = X$, $f^\alpha(X) = f(f^{\alpha-1}(X))$ for every successor ordinal α and $f^\alpha(X) = \cup_{\beta \subseteq \alpha} f^\beta(X)$ for any limit ordinal. If f is monotonic, then f's least fixpoint $lfp(f)$ is well-defined and if f is also continuous $lfp(f) = f^\omega(\bot)$. The continuity of a monotonic function on a complete lattice L is a consequence of the following property ([21]): for any set X in L and any element y, $y \in f(X) \Rightarrow \exists X'$ finite $X' \subseteq X$ such that $y \in f(X')$.

2.1 Basic Definitions and Lemmata

We start by introducing the notion of *Prolog Computation Tree* (*p-tree* for short) for a clause. Prolog Computation Trees provide the basis for formalizing the computed-answer-substitution semantics of Prolog.

Given a goal G, the p-tree for G is defined as usual in terms of unfolding by assuming a fixed (left-to-right) selection rule in the current resolvent and a fixed (textual) selection of the program clauses attempted during evaluation. We generalize this notion by applying it to clauses rather than to simple goals as follows.

Definition 2.1 (p-tree) Let P be a program and C be a clause. T_C^P is a tree rooted at C whose nodes are labeled by clauses – the resultants of C [20] – and satisfy the following properties. For any node $N \in T_C^P$, if H :- $\tilde{B}$ is the associated label, then:

1. if $\tilde{B}$ is empty then N has no children (N is a *success node*)

2. if $\tilde{B} = B_1, \ldots B_n$, and $C_{i_1} :: \cdots :: C_{i_k}$ is the ordered subsequence of the clauses of P whose head unifies with B_1, then N has k ordered children $N_1, \ldots, N_k$ where, for any $j \in [1..k]$, if $C_{i_j} = A_j \text{:-} \tilde{D}_j$ and $\theta_j = mgu(A_j, B_1)$, then N_j is labeled by the clause (resultant) $(H \text{:-} \tilde{D}_j, B_2 \ldots, B_n)\theta_j$.

3. N has one single leaf-child labeled by FAIL otherwise.

Implicit in the above definition there is the proviso that the clauses used to unfold the current resultant be standardized-apart renamings of the clauses of P. The definition of p-tree for a goal is obtained as a byproduct of the above definition by simply taking $T^P_{G\text{:-}G}$ as the p-tree T^P_G associated with G. With this understanding, we will henceforth use the term p-tree to refer interchangeably to the p-tree for a goal G and for the associated clause $G\text{:-}G$. Also, in view of the tight correspondence between p-trees and SLD-trees, the notions of partial derivation and resultant of [20] will be used similarly in the context of p-trees. The only difference from the standard case is that here we use FAIL as the resultant associated with a failing derivation.

The following lemmata establish two important properties of resultants. Both the results hold for general SLD-trees and therefore for our p-trees. Lemma 2.1 establishes the correspondence between derivations (and resultants) for a goal G and for an instance $G\theta$ of G. The proof is due to [20]. Lemma 2.2 establishes a dual property and its proof uses a version of the lifting lemma proved in [20] as well as the result stated in lemma 2.1.

Given a selection rule, say that two derivations D and D' *correspond* iff they select the same program clauses in the exact same order. Also, denote with $\mathcal{R}(D_G)$ (a standardized apart renaming of) the resultant associated with the non-failing derivation D_G for a goal G.

Lemma 2.1 (Instantiation) Let $\text{:-}A$ be a goal, D_A be a partial derivation for $\text{:-}A$ and let $\mathcal{R}(D_A) = (A\text{:-}\tilde{C})\theta$ be the associated resultant. Let also $A\phi$ be an instance of A and $\sigma = mgu(A\theta, A\phi)$. Then there exists a derivation $D_{A\phi}$ corresponding to D_A such that $\mathcal{R}(D_{A\phi}) \sim \mathcal{R}(D_A)\sigma$. □

Lemma 2.2 (Generalization) Let $\text{:-}A\phi$ be a goal and $D_{A\phi}$ be a partial derivation with associated resultant $\mathcal{R}(D_{A\phi}) = (A\phi\text{:-}\tilde{C})\theta$. Then there exists a derivation D_A for A corresponding to $D_{A\phi}$ with resultant $\mathcal{R}(D_A) = (A\text{:-}\tilde{C})\alpha$ such that there exists $\sigma = mgu(A\phi, A\alpha)$ and $\mathcal{R}(D_{A\phi}) \sim \mathcal{R}(D_A)\sigma$. □

Resultants will play a fundamental role in the fixpoint characterization developed in the sequel of the paper. As already mentioned, we will be interested in computing (an abstract representation of) the sequences of resultants associated with the clauses of our programs. We will also show that such sequences can be computed both in a top-down fashion following the construction of the associated p-trees and bottom-up through a corresponding fixpoint computation.

To this purpose, we now introduce the two related notions of *frontier* and *cut* for a finitary tree (a tree is *finitary* iff each node in the tree has a finite number of children).

Definition 2.2 (Frontier) Let T be a finitary tree. A *frontier* $F(T)$ for T is the (left-to-right ordered) sequence of the leaf-nodes of a finite tree obtained by cutting at a finite depth each path from the root of T.

Cuts are just special cases of frontiers obtained by cutting all the paths in the tree at the same depth.

Definition 2.3 (Cut) Let T be a finitary tree. The cut $\mathcal{C}_k(T)$ of T at level k is the frontier obtained by cutting at depth k each path from the root of length greater than k.

Notice that for any given k, there exists a unique cut $\mathcal{C}_k(T)$ for any tree. Given the p-tree T_c for a clause c, any cut in T_c embeds a corresponding sequence of resultants for c found in a top-down traversal of T_c. Dually, a frontier will be the result of the bottom-up computation of a corresponding sequence. Frontiers (and cuts) for the same tree can be related according to the following definition. Call $d(N)$ the depth of a node N in any finitary tree T.

Definition 2.4 Given a finitary tree T and two frontiers $F_1(T)$ and $F_2(T)$, we say that $F_1(T)$ *is dominated* by $F_2(T)$ (denoted by $F_1(T) \prec F_2(T)$), and dually, $F_2(T)$ *dominates* $F_1(T)$, iff for any two nodes N and M on the same path of T, if $N \in F_1(T)$ and $M \in F_2(T)$ then $d(N) \leq d(M)$.

It follows that, given any cut on a p-tree, we can always define a frontier for the tree which dominates the cut, and vice-versa, for any frontier there exists a dominating cut. This very same result provides the basis for proving the correspondence of the top-down and bottom-up constructions addressed earlier in the section.

In the sequel, we will also use the following property on $\prec$-ordered frontiers of the same p-tree. Let F_1 and F_2 be two frontiers for the p-tree T_G such that $F_1 \prec F_2$. Now consider a path in T_G and two resultants $r_1 \in F_1$, $r_2 \in F_2$ on this path. If r_1 is a non-unit resultant with head $G\theta$, then r_2 is either FAIL or a resultant whose head is an instance of $G\theta$. Otherwise, r_1 is either a unit resultant of FAIL and r_1 and r_2 coincide.

We conclude by introducing the definition of *Depth-First-Leftmost* derivation which provides the basis for characterizing the operational behaviour of Prolog in terms of computed answers substitutions.

Definition 2.5 (DFL-derivation) Given a goal G and a program P, a *DFL-derivation* D_G for G in P, is a path terminated by a success node in the p-tree T_G^P such that every path in T_G^P occurring to the left of D_G is finite (either successful of failing).

We can finally formalize the operational behaviour of a Prolog program in terms of the following notion of *Prolog answer substitution* (*p.a.s*). We say that θ is a p.a.s for a goal G in a program P, and we denote it by $G \overset{\theta}{\leadsto}_P \Box$, iff there exists a DFL-derivation D_G for G in P such that θ is the restriction to the variables of G of the composition of the mgu's computed at each node in D_G. In terms of p-trees, this amounts to saying that there exists a path from the root of T_G to the unit resultant $G\theta$ such that all the paths occurring to its left are finite.

3 Fixpoint Semantics

As already stated, our fixpoint reconstruction of the semantics of a Prolog program is based on the idea of characterizing the sequence of the partial answers of the program. The notion

of resultant introduced in the previous section provides the basis for such characterization. Suppose that $(A\text{:-}\tilde{C})\theta$ is a resultant for a program clause. Then the idea is that, given any goal A' with the same predicate symbol as A, a partial answer for A' is simply obtained by taking the mgu of $A\theta$ and A'. This very same idea motivated the use of resultants to achieve the oracle-based declarative semantics of [3]. The semantic domain of [3] is in fact defined by allowing clauses (as opposed to atoms) to occur in the program's interpretations. Here we follow a similar – but simpler – approach and use a more abstract characterization given in terms of *divergent* atoms rather than clauses. If $(A\text{:-}\tilde{B})\theta$ is a resultant for a program clause, then the semantic object associated with the clause is simply the divergent atom denoted by $\widehat{A\theta}$. The qualification *divergent* for $\widehat{A\theta}$ is used here to emphasize the fact that it represents a partial derivation which could in fact never terminate. Successful derivations will be also represented by their associated resultants (which are simply unit clauses or, equivalently, atoms).

The association between sequences of resultants and their abstractions is formalized by the following definition.

Definition 3.1 (abstraction) For any sequence of resultants s, the abstraction $s^\sharp$ of s is obtained by: (i) replacing each occurrence of a non-unit clause $A\text{:-}\tilde{C}$ with the divergent atom $\widehat{A}$; (ii) stripping off all the occurrences of FAIL in s.

We now proceed by introducing the notion of extended interpretations.

3.1 Extended Interpretations

We start by defining the *extended base* for a program. Let $\mathcal{B}$ denote the set of (the representatives of the $\sim$-equivalence classes of the) non-ground atoms built over the sets Π and Σ of the predicate and function symbols occurring in the program. The extended base $\mathcal{B}_E$ for the program is obtained as the union of $\mathcal{B}$ with the set $\widehat{\mathcal{B}}$ of the divergent atoms corresponding to the elements of $\mathcal{B}$. Formally: $\mathcal{B}_E = \{\widehat{A} \mid A \in \mathcal{B}\} \cup \mathcal{B}$.

We also assume that the elements in $\mathcal{B}_E$ are standardized apart from each other. Interpretations are defined as elements of the power-set of $\mathcal{B}_E^*$: an interpretation I is a set of sequences of elements in $\mathcal{B}_E$ where each sequence defines an abstract representation of the corresponding set of sequences of resultants for the program clauses. Given an interpretation I, a sequence $S \in I$ and an atom A, we denote with $\pi_A(S)$ the subsequence of S which consists of the elements which unify with A. If A is in most general form, say $A = p(\tilde{x})$, then $\pi_p(S)$ will be used as a shorthand for $\pi_{p(\tilde{x})}(S)$.

Example 3.1 Any program can be viewed as a sequence of resultants by simply considering each program clause as the resultant associated with a derivation of length 0 for the clause. Therefore the abstraction function applies as well to programs and produces a corresponding sequence of the program's interpretation. The following is an example of a program and its abstraction:

$$P \equiv p(a) :: p(f(x))\text{:-}q(x) :: q(b).. \qquad P^\sharp \equiv p(a) :: p(\widehat{f(x)}) :: q(b)$$

In this case, the projection $\pi_{p(a)}(P^\sharp)$ yields the sequence consisting of the single element $p(a)$ whereas $\pi_{p(x)}(P^\sharp) = \pi_p(P^\sharp) = p(a) :: p(\widehat{f(x)})$. □

The set $\mathcal{P}(\mathcal{B}_E^*)$ is a complete lattice under the usual inclusion ordering, with top and bottom elements defined respectively as $\mathcal{B}_E^*$ and $\emptyset$. Associated with any program P, we can define an immediate-consequence operator Φ_P over $\mathcal{P}(\mathcal{B}_E^*)$. The definition is given in two steps. We first define a function ϕ_P associated with the program, which maps sequences in $\mathcal{B}_E^*$ to sequences in $\mathcal{B}_E^*$.

Definition 3.2 Let P be the program $c_1 :: \cdots :: c_n$. $\phi_P : \mathcal{B}_E^* \mapsto \mathcal{B}_E^*$ is defined clause-wise as the concatenation $\phi_P(S) = \phi_{c_1}(S) :: \cdots :: \phi_{c_n}(S)$, for any sequence S. Let c be a clause standardized apart from S. We distinguish two cases, for unit and non-unit clauses.

- If c is the unit clause $A.$, then $\phi_{A.}(S) = A$.
- Otherwise, let $c = A \text{:-} B, \tilde{D}$ and $\pi_B(S) = d_1 :: \cdots :: d_k$. Then

$$\phi_c(S) \;=\; \alpha_1 :: \cdots :: \alpha_k \quad \text{where} \quad \alpha_i = \begin{cases} \widehat{A\theta_i} & \text{if } d_i = \widehat{B'}, \\ \phi_{(A \text{:-} \tilde{D})\theta_i}(S) & \text{if } d_i = B'. \end{cases}$$

and $\theta_i = mgu(B, B')$ for any $i \in [1..k]$. □

Given any interpretation I, the immediate consequences of I are then obtained by simply collecting all the sequences produced the component-wise application of ϕ_P to all the sequences in I.

Definition 3.3 (Immediate Consequences) The immediate consequence operator $\Phi_P : \mathcal{P}(\mathcal{B}_E^*) \mapsto \mathcal{P}(\mathcal{B}_E^*)$ is defined in terms of ϕ_P as follows:

$$\Phi_P(I) \;=\; \{\phi_P(S) \mid S \in I\} \;\cup\; \{P^\sharp\}$$

The definition of ϕ_P extends the T_P operator used in [14] in two respects. First, similarly to the operator defined in [12], for any sequence S on $\mathcal{B}_E^*$, it computes the sequence of the atomic and of (the abstract representation of) the *non* atomic consequences of S.. Secondly, and in contrast to [12], it is applied to sequences (as opposed to sets) and it embeds a left-to-right selection rule for the atoms in the body of a clause, so as to capture the control component underlying Prolog's computation rule.

Proposition 1 Φ_P is continuous on $(\mathcal{P}(\mathcal{B}_E^*), \subseteq, \bot, \top, \bigcup, \bigcap)$.

Proof. The proof uses the property on monotonic functions mentioned earlier in the paper. Let I be an interpretation and let S be a sequence in $\Phi_P(I)$. By definition, either $S = P^\sharp$ or there exists a sequence S' in I such that $S = \phi_P(S')$. In the first case take $J = \emptyset$, in the second take $J = \{S'\}$ as a finite subset of I. Clearly, $S \in \Phi_P(J)$ and hence the claim. □

The continuity of Φ_P guarantees the existence of the least fixpoint $\mathcal{S}_{DFL}(P) = \Phi_P^\omega(\emptyset)$ of Φ_P. An interesting characterization of the set $\mathcal{S}_{DFL}(P)$ is provided by the following observation. Consider the interpretations $I_1, I_2, \ldots, I_k, \ldots$ resulting from the iterative computation of the fixpoint. At the first step, $I_1 = \Phi_P(\emptyset) = \{P^\sharp\}$. Similarly, at the second step, $I_2 = \Phi_P(I_1) = \{P^\sharp, \phi_P(P^\sharp)\}$. In general, at step k, I_k will consist of k sequences $\mathcal{S}_1, \ldots, \mathcal{S}_k$ and these sequences can be ordered so as to ensure that, for any $j \in [1..k]$, $\phi_P(\mathcal{S}_{j-1}) = \mathcal{S}_j$. It is also easy to see that if $I_k = \{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, then $I_{k+1} = I_k \cup \{\mathcal{S}_{k+1}\}$ where $\mathcal{S}_{k+1} = \phi_P(\mathcal{S}_k)$ is the only new sequence computed at step $k+1$. The fixpoint $\mathcal{S}_{DFL}(P)$ can then be viewed as the limit interpretation I_ω consisting of the (finite or infinite) ordered set of sequences $\mathcal{S}_1, \ldots, \mathcal{S}_k, \ldots$

Example 3.2 First consider the program P consisting of the following (sequence of) clauses:
$p(b) :: p(X) \text{:-} r(X) :: p(c) :: \text{..}r(a) \text{:-} p(a) :: r(b) \text{:-} q(a)$
$\mathcal{S}_{DFL}(P)$ is computed at the third iteration of $\Phi_P^\omega(\emptyset)$ and consists of the ordered set of the following sequences:

$$\mathcal{S}_1 = p(b) :: \widehat{p(X)} :: \widehat{p(c)} :: \widehat{r(a)} :: \widehat{r(b)}$$
$$\mathcal{S}_2 = p(b) :: \widehat{p(a)} :: \widehat{p(b)} :: \widehat{p(c)} :: \widehat{r(a)}$$
$$\mathcal{S}_3 = p(b) :: \widehat{p(a)} :: \widehat{p(c)} :: \widehat{r(a)}$$

Now consider the new program Q: $p(0) :: p(s(X)) \text{:-} p(X)$..
In this case the computation requires infinitely many iterations. At step k, I_k contains the k sequences $\{\mathcal{S}_1, \ldots, \mathcal{S}_k\}$, where for each j:

$$\mathcal{S}_j = p(0) :: \cdots :: p(s^{j-1}(0)) :: \widehat{p(s^j(X))}$$ □

We will show shortly how the ordering on the sequences of $\mathcal{S}_{DFL}(P)$ has a natural counterpart in the ordering on substitution sequences defined for the fixpoint semantics for Prolog proposed in [4]. Before doing so, however, we now move on to discuss the soundness and completeness of our characterization.

4 Soundness and Completeness

The soundness and completeness of the fixpoint construction given in the previous section is stated in terms of the notion of *reachability* of a substitution for an atom in a sequence of abstract resultants. The idea behind the definition of reachability is to capture both the issues involved in the computation of a Prolog answer substitution (P.a.s.): termination and success. Consider a goal G and its associated p-tree T_G. Let $\mathcal{C}$ be a cut on T_G and consider the sequence $\mathcal{C}^\natural$ of abstract resultants corresponding to $\mathcal{C}$. If $G\theta \in \mathcal{C}^\natural$ and all the elements of $\mathcal{C}^\natural$ on the left of $G\theta$ are non-divergent atoms, then certainly θ is a P.a.s. for G. In this case we say that θ is reachable for G in $\mathcal{C}^\natural$.Conversely, if one such element is a divergent atom, say $\widehat{G\sigma}$, then the corresponding derivation in T_G is potentially infinite,

Definition 4.1 (reachability) Let S be a sequence in $\mathcal{B}_E^*$ and A_E be an element of the extended base $\mathcal{B}_E$. Let also G and G' be two atoms such that $Pred(G) = Pred(G')$. $\rho : \mathcal{B} \times \mathcal{B}_E^* \mapsto Subst^*$ is the function defined inductively as follows:

1. $\rho(G, \lambda) = \lambda$
2. $\rho(G,\ G' :: S) = \theta :: \rho(G,\ S)$ if $\theta = mgu(G, G')_{|vars(G)}$
3. $\rho(G,\ \widehat{G'} :: S) = \lambda$ if there exists an $mgu(G, G')$
4. $\rho(G,\ A_E :: S) = \rho(G,\ S)$ otherwise.

Given a non-divergent atom G, we say that θ is *reachable for G in S* iff θ is one of the substitutions in $\rho(G, S)$, i.e. $\theta\ \varepsilon\ \rho(G, S)$. □

Example 4.1 Consider the sequence $S = p(f(a)) :: p(\widehat{g(X)}) :: p(f(b)) :: \widehat{r(a)} :: r(b)$ and take the goal $\text{:-}p(Y)$. Then $\rho(p(Y), S) = \{Y/f(a)\}$ since $p(Y)$ unifies with the divergent atom $p(\widehat{g(X)})$. Conversely, for the goal $\text{:-}p(f(Y))$, $\rho(p(f(Y)), S) = \{Y/a\} :: \{Y/b\}$. Similarly, $\rho(r(X), S) = \lambda$ and $\rho(r(b), S) = \epsilon$. □

The next theorem shows that the above argument can be generalized in a non trivial way to make it independent of the p-tree for the particular goal G. In fact, for any $G = p(\tilde{t})$, we now show that the reachability of a substitution for G can be tested starting from the p-tree $T_{p(\tilde{x})}$ for the most general form $p(\tilde{x})$ of G.

Theorem 4.1 Let $:\!-p(\tilde{t})$ be a goal and P be a program. θ is a *P.a.s.* for $:\!-p(\tilde{t})$ in P, if and only if there exists m such that θ is reachable for $p(\tilde{t})$ in $\mathcal{C}_m^\sharp$, where $\mathcal{C}_m$ is the cut at level m of $T_{p(\tilde{x})}$. Formally:

$$p(\tilde{t}) \overset{\theta}{\rightsquigarrow}_P \square \quad \text{iff} \quad \exists \mathcal{C}_m \text{ for } T_{p(\tilde{x})} \text{ such that } \theta \in \rho(p(\tilde{t}), \mathcal{C}_m^\sharp)$$

Proof. ($\Rightarrow$) By definition, there exists a successful derivation $D_{p(\tilde{t})}$ whose resultant is $p(\tilde{t})\theta$ and such that all the derivations in $T_{p(\tilde{t})}$ on the left of $D_{p(\tilde{t})}$ are finite (either successful or failing). Let k be the max length of all these derivations (including $D_{p(\tilde{t})}$). By lemma 2.2 there exists in $T_{p(\tilde{x})}$ a successful derivation $D_{p(\tilde{x})}$ corresponding to $D_{p(\tilde{t})}$ with resultant $p(\tilde{x})\alpha$ such that $\sigma = mgu(p(\tilde{t}), p(\tilde{x})\alpha)$ and $p(\tilde{t})\theta \sim p(\tilde{x})\alpha\sigma$. Hence, $p(\tilde{t})\theta \sim p(\tilde{t})\sigma$ and then $\theta = \sigma_{|var(t)}$. Let now $m = k+1$ and let $\mathcal{C}_m$ be the cut at level m in $T_{p(\tilde{x})}$. Clearly $p(\tilde{x})\alpha \in \mathcal{C}_m$. Then, to see that $\theta \in \rho(p(\tilde{t}), \mathcal{C}_m^\sharp)$ we have only to show that $p(\tilde{t})$ does not unify with (the head of) any of the non-unit resultants in $\mathcal{C}_m$ which occur to the left of $p(\tilde{x})\alpha$. By contradiction, assume that one such resultant exists. Since the body is not empty, the resultant occurs also in the cut at level $m = k+1$ in $T_{p(\tilde{x})}$. By lemma 2.1 there exists a corresponding derivation for $p(\tilde{t})$ whose resultant occurs in the cut at level $k+1$ in $T_{p(\tilde{t})}$. But then there exists also a derivation in $T_{p(\tilde{t})}$ whose length is $m > k$ which occurs on the left of $D_p(\tilde{t})$ and this leads to a contradiction.

($\Leftarrow$) From $\theta \in \rho(p(\tilde{t}), \mathcal{C}_m^\sharp)$ it follows that there exists a resultant $\mathcal{R}(D_{p(\tilde{x})}) = p(\tilde{x})\alpha$ in $\mathcal{C}_m$ such that *(i)* $\theta = mgu(p(\tilde{t}), p(\tilde{x})\alpha)_{|vars(\tilde{t})}$; *(ii)* $p(\tilde{t})$ does not unify with the head of any non-unit resultant to the left of $p(\tilde{x})\alpha$.
From lemma 2.1 and *(i)* it follows that there exists in $T_{p(\tilde{t})}$ a derivation $D_{p(\tilde{t})}$ corresponding to $D_{p(\tilde{x})}$ whose associated resultant is $p(\tilde{t})\theta$. Furthermore, from lemma 2.2 and *(ii)* it follows that every derivation in $T_{p(\tilde{t})}$ to the left of $D_{p(\tilde{t})}$ is finite. This proves the claim. $\square$

We now move on to the main result of this section. We have just shown that any Prolog answer substitution for an atomic goal G can be characterized in terms of the reachability of G on (the abstraction of) a cut in the p-tree for the most general form of G. Now we prove that this is equivalent to testing the reachability of G on one of the sequences contained in the fixpoint of the immediate-consequence operator Φ_P. We do this in two steps. We first show that testing reachability on the abstraction of a cut in the p-tree can be equivalently performed on the abstraction of a corresponding frontier of the tree (4.2 below). Then we prove that the abstraction of any such frontier is contained in the fixpoint $\mathcal{S}_{DFL}$ of Φ_P (lemmata 4.3 and 4.4). We first show that $\prec$-ordered frontiers preserve the reachability of any substitution for any goal.

Lemma 4.2 Let $:\!-p(\tilde{t})$ be a goal and let F_1 and F_2 be two frontiers of $T_{p(\tilde{x})}$. If $F_1 \prec F_2$ and θ is reachable for $p(\tilde{t})$ in $F_1^\sharp$ then so is in $F_2^\sharp$.
Proof. (See [7]). $\square$

By virtue of this result, it follows that the reachability of any goal $p(\tilde{t})$ on the abstraction of a cut C in $T_{p(\tilde{x})}$ can be equivalently tested on (the abstraction of) any frontier F which dominates the cut C. We now show the abstraction of any such frontier can be computed in finite number of iterations of Φ_P starting from the empty set. Recall that, for any sequence S, $\pi_p(S)$ denotes the sequence in S of the elements whose predicate symbol is p.

Lemma 4.3 Let S_k be sequence computed at the the k-th step of $\Phi_P^\omega(\oslash)$. Then, for any predicate symbol p defined in P, there exists a frontier F which dominates the k-th cut C_k of $T_{p(\tilde{x})}$ and such that $\pi_p(S_k) = F^\natural$.
Proof. (See [7]). □

The soundness and completeness of the fixpoint semantics follows now as an immediate corollary of the following lemma.

Lemma 4.4 Let C_k be the cut at level k of the p-tree $T_{p(\tilde{x})}$ for $p(\tilde{x})$. Then, for any goal $p(\tilde{t})$, $\theta \in \rho(p(\tilde{t}), C_k^\natural)$ iff there exists $S \in \mathcal{S}_{DFL}(P)$ such that $\theta \in \rho(p(\tilde{t}), S)$.

Proof. Immediate by lemmata 4.3 and 4.2 and the continuity of Φ_P. □

Theorem 4.5 (Soundness and Completeness) Let G be an atomic goal and P be a program. θ is a P.a.s. for G in P if and only if there exists $S \in \mathcal{S}_{DFL}(P)$ such that $\theta \in \rho(G, S)$. □

Example 4.2 Consider the first program of example 3.2. We have already showed that the fixpoint consists of the three sequences $\{\mathcal{S}_1, \mathcal{S}_2, \mathcal{S}_3\}$ where

$$\mathcal{S}_1 = p(b) :: \widehat{p(X)} :: \widehat{p(c)} :: \widehat{r(a)} :: \widehat{r(b)}$$
$$\mathcal{S}_2 = p(b) :: \widehat{p(a)} :: \widehat{p(b)} :: \widehat{p(c)} :: \widehat{r(a)}$$
$$\mathcal{S}_3 = p(b) :: \widehat{p(a)} :: \widehat{p(c)} :: \widehat{r(a)}$$

Now consider the goals $:\text{-}p(X)$ and $:\text{-}p(c)$. $\{X/b\}$ is the only substitution computed for $:\text{-}p(X)$ and, correspondingly, $\rho(p(X), \mathcal{S}_1) = \{X/b\}$. Similarly, ϵ is the only P.a.s. for $:\text{-}p(c)$ and $\rho(p(c), \mathcal{S}_3) = \epsilon$

5 Termination

In this section we discuss and illustrate some further properties of the semantics of a Prolog program given in terms of the fixpoint $\mathcal{S}_{DFL(P)}$. More precisely, we show that the structure of $\mathcal{S}_{DFL}(P)$ is rich enough to characterize two important properties of the evaluation of a goal: existential and universal termination. As a matter of fact, an implicit characterization of existential termination is already embedded in the notion of Prolog answer substitution. Say that a goal existentially terminates with success iff there exists a successful derivation for the goal. From the soundness and completeness results, it immediately follows that:

> an atomic goal G existentially terminates with success in P iff there exists $S \in \mathcal{S}_{DFL}(P)$ such that $\rho(G, S) \neq \lambda$.

Existential termination with success is just one of the aspects of a Prolog computation we would like to reason about. The evaluation of a goal in a program may in fact have three different outcomes: *(i)* it may terminate after producing a finite (possibly empty) sequence of substitutions (*universal termination*); *(ii)* it may produce a finite (possibly empty) sequence of substitutions and then loop infinitely without producing any further answer, and finally *(iii)* it may produce an infinite sequence of answer substitutions.

We now show that these different behaviours can all be captured by our fixpoint construction. We do this by refining the notion of reachability to reflect the distinction between computations which terminate after producing k answers and computations which do not terminate after the first k answers. Similar results were obtained in [4] by means of a denotational characterization. Prolog programs are viewed in [4] as functions from goals to the sequences of answer substitutions generated when the program is queried with the goals under consideration. Infinite computations are modeled by allowing a sequence of substitutions to be either infinite or to contain the special marker $\perp$ (bottom) as its last element. The set of such sequences, denoted by S_{seq}, is formally defined in [4] as: $S_{seq} = Subst^* \cup (Subst^* :: \{\perp\}) \cup Subst^*_\infty$ where $Subst^*_\infty$ is the set of infinite sequences of substitutions. Given a goal G, the associated sequence is finite if and only if the number of computed answers for G is finite; its last element is $\perp$ if, after producing a finite number of answers, the execution enters an infinite loop; it is infinite if evaluating G produces infinitely many answers and it is empty in the case of finite failure. We can characterize the above situations by extending the definition of reachability to have ρ range over the set S_{seq} rather than over $Subst^*$. This allows us to achieve results corresponding to those of [4] but, in contrast to that case, we do this in terms of a goal independent construction.

Definition 5.1 (reachability revised) Let S be a sequence in $\mathcal{B}^*_E$ and A_E be an element of the extended base $\mathcal{B}_E$. Let also G and G' be two atoms such that $Pred(G) = Pred(G')$. The generalization $\rho_\perp : \mathcal{B} \times \mathcal{B}^*_E \mapsto S_{seq}$ is the function defined inductively as follows:

1. $\rho_\perp(G, \lambda) = \lambda$
2. $\rho_\perp(G, G' :: S) = \theta :: \rho_\perp(G, S)$ if $\theta = mgu(G, G')_{|vars(G)}$
3. $\rho_\perp(G, \widehat{G'} :: S) = \perp$ if there exists an $mgu(G, G')$
4. $\rho_\perp(G, A_E :: S) = \rho_\perp(G, S)$ otherwise.

Notice that the only difference between ρ and $\rho_\perp$ is that $\rho_\perp(G, S)$ returns bottom (as opposed to the empty sequence) upon hitting a divergent atom in S which unifies with G. This simple extension has a number of fairly interesting consequences. Consider again the ordered set of sequences $\mathcal{S}_1, \ldots, \mathcal{S}_k, \ldots$ in the fixpoint $\mathcal{S}_{DFL}(P)$. For any atomic goal G, the application of $\rho_\perp$ on G and the $\mathcal{S}_i$s produces a corresponding sequence $\rho_\perp(G, \mathcal{S}_1), \ldots, \rho_\perp(G, \mathcal{S}_K), \ldots$ of elements in S_{seq}. We can show that this sequence forms an increasing chain in the complete partial order $(S_{seq}, \sqsubseteq)$ defined in [4]. The ordering relation $\sqsubseteq$ is formally defined in [4] as follows: for any two sequences $s_1, s_2 \in S_{seq}$, $s_1 \sqsubseteq s_2$ iff one of the following conditions holds:

(i) $(s_1 = s :: \perp)$ and $\exists s' \in S_{seq}$ $(s_2 = s :: s')$

(ii) $s_1 \notin Subst* :: \{\perp\}$ and $s_1 = s_2$

Intuitively we can think of any sequence in S_{seq} as the representation of a specific computation stage. Then, $s_1 \sqsubseteq s_2$ gives a formal account of the fact that s_1 represents an earlier stage

than s_2. Assume for instance to have a finite computation producing three substitution at three different stages. The corresponding representation in terms of sequences over S_{seq} is given by: $(\theta_1 ::\bot) \sqsubseteq (\theta_1 :: \theta_2 ::\bot) \sqsubseteq (\theta_1 :: \theta_2 :: \theta_3)$.

Both finite and infinite sequences of substitutions are maximal elements of $(S_{seq}, \sqsubseteq)$. It can also be proved that every chain in S_{seq} has a least upper bound in S_{seq} and hence, that the limit of an ascending chain in S_{seq} is well defined.

The first result we prove is that the order $\sqsubseteq$ on S_{seq} corresponds, via $\rho_\bot$ and the abstraction function $\sharp$, to the ordering $\prec$ on the frontiers of a p-tree.

Lemma 5.1 Let G be an atomic goal and let F_1 and F_2 be two frontiers of T_G. If $F_1 \prec F_2$ then $\rho_\bot(G, F_1^\sharp) \sqsubseteq \rho_\bot(G, F_2^\sharp)$
Proof. (See [7]). □

Based on this result, it is easy to see that the ordering on the sequences contained in the least fixpoint $\mathcal{S}_{DFL}(P)$ induces a corresponding ordering on S_{seq}.

Theorem 5.2 Let G be an atomic goal and P be a program. For every $\mathcal{S}_k \in \mathcal{S}_{DFL}(P)$, $\rho_\bot(G, \mathcal{S}_k) \sqsubseteq \rho_\bot(G, \mathcal{S}_{k+1})$
Proof. (See [7]). □

As the final step we show how to express the semantics of a "queried" Prolog program in term of the sequence of answer substitutions for the query. The proof of this result derives from theorem 5.2, and from the two following generalizations of theorem 4.5.

Theorem 5.3 Let G be an atomic goal and P a program. $\theta_1 :: \cdots :: \theta_n$ is the sequence of the first n *P.a.s.* for G in P iff there exists $S_k \in \mathcal{S}_{DFL}(P)$ such that $(\theta_1 :: \cdots :: \theta_n ::\bot) \sqsubseteq \rho_\bot(G, S_k)$. □

Theorem 5.4 Let G be an atomic goal and P be a program. Then:

> T_G is finite with *P.a.s.* $\theta_1, \ldots, \theta_n$ iff there exists $S \in \mathcal{S}_{DFL}(P)$ such that $\rho_\bot(G, S) = \theta_1 :: \ldots :: \theta_n$.
>
> T_G is infinite iff every $S \in \mathcal{S}_{DFL}(P)$ is such that $\rho_\bot(G, S) = s ::\bot$ for a suitable choice of a sequence s of *P.a.s* for G.

Example 5.1 Consider again program P of example 3.2 and the associated fixpoint $\mathcal{S}_{DFL}(P)$:

$\mathcal{S}_1 = p(b) :: \widehat{p(X)} :: \widehat{p(c)} :: \widehat{r(a)} :: \widehat{r(b)}$
$\mathcal{S}_2 = p(b) :: \widehat{p(a)} :: \widehat{p(b)} :: \widehat{p(c)} :: \widehat{r(a)}$
$\mathcal{S}_3 = p(b) :: \widehat{p(a)} :: \widehat{p(c)} :: \widehat{r(a)}$

When queried with $:\text{-}p(a)$, P enters an infinite loop without producing any answer. Correspondingly, $\rho_\bot(p(a), \mathcal{S}_i) = \bot$ for all the sequences in $\mathcal{S}_{DFL}(P)$. Similarly, if queried with $:\text{-}p(X)$, P produces the substitution $\{X/b\}$ and then enters an infinite loop. Again, an equivalent result is obtained reasoning on the fixpoint: for any $\mathcal{S}_i \in \mathcal{S}_{DFL}(P)$, $\rho_\bot(p(X), \mathcal{S}_i) = \{X/b\} ::\bot$. Finally, for the two queries $:\text{-}r(b)$ and $:\text{-}p(c)$ there are respectively no answers ($:\text{-}r(b)$ finitely fails) and the empty substitution ϵ. Equivalently, $\rho_\bot(r(b), \mathcal{S}_3) = \lambda$ and $\rho_\bot(p(c), \mathcal{S}_3) = \epsilon$.

Consider now program Q from example 3.2. $\mathcal{S}_{DFL}(Q)$ is reached in ω steps; the ordered set of sequences $\mathcal{S}_1, \ldots, \mathcal{S}_k, \ldots$ provides a sequence of approximations of the program behaviour.

$\rho_\perp(p(s^m(o)), \mathcal{S}_n) = \perp$. *if* $n \leq m$

$\rho_\perp(p(s^m(o)), \mathcal{S}_n) = \epsilon$. *if* $n \geq m+1$

$\rho_\perp(p(X), \mathcal{S}_n) = \{X/o\} :: \cdots :: \{X/s^{n-1}(o)\} ::\perp$.

5.1 Remarks and future extensions

All the above results can be extended to non atomic goals by observing that the answers for any goal G in a program P can be obtained by adding a new clause, $ans(\tilde{X}) :\!\!- G$, where $\tilde{X}$ are all the variables occurring in G and *ans* is a new predicate-symbol. Furthermore, we can derive a goal dependent semantics $\mathcal{S}_{DFL}(G, P)$ from $\mathcal{S}_{DFL}(P)$ as follows:

$$\mathcal{S}_{DFL}(G, P) = \{\phi_{ans(\tilde{X}) :\!\!- G}(S) \mid S \in \mathcal{S}_{DFL}(P)\}$$

Hence, the sequence $\theta_1 :: \cdots :: \theta_n$ provides the the first n answer substitution for G in P iff there exists $S_k \in \mathcal{S}_{DFL}(G, P)$ such that $(ans(\tilde{X})\theta_1 :: \cdots :: ans(\tilde{X})\theta_n)$ is a prefix of S_k.

A further desirable refinement of the approach we have presented in these pages would be to simplify the semantic domain so as to define interpretations as sequences of abstract resultant rather than as sets of such sequences. The problem with this extension is that it requires a richer structure for the Herbrand universe used to construct the extended base for our programs. Reasoning with sequences instead of sets of sequences and computing a *limit* sequence requires in fact to have a syntactic characterization for the terms resulting from infinite computations. Hence the use of a Herbrand universe extended with infinite terms becomes crucial in this context.

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

References

[1] K. R. Apt. Introduction to Logic Programming. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science*, volume B: Formal Models and Semantics. Elsevier, Amsterdam and The MIT Press, Cambridge, 1990.

[2] K. R. Apt, E. Marchiori, and C. Palamidessi. A Theory of first-order built-in's of Prolog. In H. Kirchner and G. Levi, editors, *Proc. Third Int'l Conf. on Algebraic and Logic Programming*, pages 69–83. Springer-Verlag, Berlin, 1992.

[3] R. Barbuti, M. Codish, R. Giacobazzi, and M. Maher. Oracle Semantics for Prolog. In H. Kirchner and G. Levi, editors, *Proc. Third Int'l Conf. on Algebraic and Logic Programming*, pages 100–114. Springer-Verlag, Berlin, 1992.

[4] M. Baudinet. Proving Termination Properties of Prolog Programs: A Semantic Approach. *Journal of Logic Programming*, (14):1–29, 1992.

[5] M. Billaud. Simple Operational and Denotational Semantics for Prolog with Cut. *Theoretical Computer Science*, (71):193–208, 1990.

[6] E. Boerger. A Logical Operational Semantics of Full Prolog. In *Proceedings of CSL'90 3rd Workshop on Computer Science Logic*. Springer-Verlag, Berlin, 1990.

[7] A. Bossi, M. Bugliesi, and M. Fabris. A New Fixpoint Semantics for Prolog. In *Proceedings of ICLP'93*. MIT, 1993.

[8] A. Bossi, M. Bugliesi, G. Gabbrielli, G. Levi, and M. C. Meo. Differential Logic Programs. In *Proceedings of POPL'93*. ACM, 1993.

[9] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, 1992.

[10] S. K. Debray and P. Mishra. Denotational and Operational Semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, 1988.

[11] A. deBruin and E. deVink. Continuation Semantics for Prolog with cuts. In *Proc. TAPSOFT'89*, pages 178–192. Springer-Verlag, Berlin, 1989. Lecture Notes in Computer Science.

[12] G. Delzanno and M. Martelli. Insiemi di Soluzioni di Programmi Logici. In S. Costantini, editor, *Proc. Seventh Italian Conference on Logic Programming*, pages 191–205. CittaStudi, Milano,Italy, 1992.

[13] P. Deransart and G. Ferrand. An Operational Formal Definition of PROLOG: A Specification Method and Its Application. *New Generation Computing*, (10):121–171, 1992.

[14] 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.

[15] M. Fitting. A Deterministic Prolog Fixpoint Semantics. *Journal of Logic Programming*, 2(2):111–118, 1985.

[16] M. Gabbrielli and G. Levi. On the Semantics of Logic Programs. In J. Leach Albert, B. Monien, and M. Rodriguez-Artalejo, editors, *Automata, Languages and Programming, 18th International Colloquium*, volume 510 of *Lecture Notes in Computer Science*, pages 1–19. Springer-Verlag, Berlin, 1991.

[17] M. Gabbrielli and M. C. Meo. Fixpoint Semantics for Partial Computed Answer Substitutions and Call Patterns. In H. Kirchner and G. Levi, editors, *Proc. Third Int'l Conf. on Algebraic and Logic Programming*, pages 84–99. Springer-Verlag, Berlin, 1992.

[18] N.D. Jones and A. Mycroft. Stepwise Development of Operational and Denotational Semantics for Prolog. In Sten-Åke Tarnlund, editor, *Proc. Second Int'l Conf. on Logic Programming*, pages 281–288, 1984.

[19] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[20] J. W. Lloyd and J. C. Shepherdson. Partial Evaluation in Logic Programming. *Journal of Logic Programming*, 11:217–242, 1991.

[21] M. J. Maher. Constraint Logic Programming: Analysis Transformation and Semantics. Lecture Notes at the Intl. Summer School on L. P., Acireale, Italy, June 1992.

An Operational Semantics for CHIP.

Gilberto Filé
Giuseppe Nardiello
Adriano Tirabosco

Dipartimento di Matematica Pura ed Applicata
Università di Padova
Via Belzoni, 7 I-35131 Padova (Italy)
E-mail: {*gilberto* | *giuseppe*}@*zenone.unipd.it*

Abstract

A (low-level) operational semantics for the finite-domains part of the constraint logic language CHIP is described.

1 Introduction

In the last years the Logic Programming (LP) community, prompted by the necessity of showing that the LP scheme can be effective for solving real-life problems and not just toy-problems, has devoted a great research effort for improving the efficiency of LP languages. In particular, Constraint Logic Programming (CLP) is becoming a growing research field. The interest in this area is motivated by the fact that many interesting problems in a wide range of application areas (like artificial intelligence, operations research and computer science) can be viewed as constraint satisfaction problems. A constraint satisfaction problem (CSP) involves a set of problem variables, a domain of potential values for each variable, and a set of constraints specifying which combinations of values are allowed for subsets of these variables [16]. If each domain is a finite set of values then the CSP is a finite constraint satisfaction problem (FCSP). (F)CSPs are in general NP-complete problems.

1.1 CLP

The CLP scheme [8, 9, 10] is an extension of the LP scheme where unification in the Herbrand universe has been replaced by the concept of constraint-solvability in some specified domain(s) of application. It defines a class of languages which share common basic properties. A CLP(X) language, instance of the CLP scheme, is a many-sorted (a sort for each domain of application) first-order language. The intended meaning of some predicate symbols, function symbols, and constant symbols over the domain of computation X is specified by a structure $\Re_X$. Atoms built only from these symbols are called *constraints.* The decision procedure which checks the satisfability of a given set of constraints with respect to $\Re_X$ is called *constraint-solver.*

1.2 CHIP

The CLP language CHIP (Constraint Handling In Prolog) [5, 1, 3] extends Prolog by providing three new computation domains in addition to the Herbrand universe: finite domains

for integers, boolean terms and rational terms. For each computation domain, CHIP provides efficient constraint-solving techniques, namely, local consistency techniques for finite domains, boolean unification for boolean terms and a symbolic simplex-like algorithm for rational terms.

In the following our attention is devoted only to the finite-domain part of CHIP.

CHIP differs in two main points from the ideal CLP scheme: the presence of a general delaying mechanism and the use of a local constraint solver in place of a complete (global) solver.

The delaying mechanism is used in order to pass only "easy-to-handle" constraints to the local constraint-solver. Constraints are selected in order to be checked for consistency only when some conditions on them are satisfied (e.g. when they are sufficiently instantiated), otherwise they are delayed. This data-driven mechanism introduces a form of incompleteness: at some time during the computation it could happen that every atom and constraint in the goal is delayed. In order to avoid such floundering situations, the programmer has to take care of introducing value generators (`indomain` atoms: explained below) in the goal. A value generator instantiates a finite-domain variable, so that some delayed atom or constraint is possibly woken up and, hence, the computation can proceed.

While other CLP languages (like Prolog III [4] and CLP(R) [11, 12]) are provided by explicit delaying mechanisms, the use of a constraint solver based on local consistency tecniques is peculiar of CHIP. Algorithms based on local consistency techniques (see [16, 22, 17] and references therein) consider only a subset of the constraints and not the whole set. They work by "refining domains", i.e. by removing, for every variable, (local) inconsistencies, that is values such that there are no corresponding values for the other variables which could satisfy the constraints (these values cannot be part of any global solution to the CSP). The CHIP arc-consistency algorithm is based on the AC-3 algorithm [15].

In conclusion, the efficency of CHIP is based on a cooperation between testing local consistency and generating values. This cooperation is controlled by the delaying mechanism.

Van Hentenryck [19] provided the theoretical basis for the embedding of local consistency techniques in LP. These consistency techniques provide a uniform paradigm for solving constraints both built-in and user-defined ones [19]. However, for efficiency reason, in the CHIP language (as it is available on the software market) only built-in constraints are provided [3].

An extensive amount of empirical evidence over a wide range of application areas proved the practical value of CHIP as language for solving constrained problems (see [19, 6, 3, 2] and references therein). For example, CHIP has been applied to typical CSPs like operations research problems (cutting-stock, scheduling problems, warehouses location), qualitative physics and hardware design problems (simulation and verification of digital circuits, VSLI design, fault diagnosis and automatic test-pattern generation of digital circuits, microcode label assignment).

Further research is currently carried out for improving the efficiency and the expressive power of CLP languages over finite domains [20, 21, 22, 13, 7]. In particular, a successor of CHIP, called `cc(FD)`, has been designed in the concurrent constraint (`cc`) framework [23].

1.3 Goal and Outline

In the following, we describe a detailed (low-level) operational semantics for CHIP defined along the lines of the operational semantics that was outlined by Van Hentenryck in [19] (see also [20]).

We consider, for simplicity sake, only the kernel of the finite-domains part of CHIP. The predicates that are not treated are the following ones (refer to [19] and [3]): higher order predicates like `minimize` (for combinatorial optimization, by means of branch-and-bound techniques); the predicate `delete` for dynamic orderings of variables (to choose according to a given heuristic which one is better to instantiate); and the non-deterministic predicate `split` (for domain splitting and case analysis).

The motivation for this operational semantics is twofold. First, an (operational) semantics is, in general, useful for understanding a language. Secondly, it is a preliminary step for studying semantics-based optimizations of a language. Indeed, even if CHIP was designed to be efficient, still we believe that there is room for further optimizations.

Plan of the paper. In section 2 we give the syntax of CHIP and some preliminary definitions. In section 3 we describe the operational semantics for the CHIP language. For completeness aske, in appendix we give the modified unification algorithm of [19]. We assume the reader familiar with logic programming (refer to [14]) and with Van Hentenryck's work (refer mainly to [19]).

2 Syntax

The finite-domain part of the language CHIP is a two-sorted (the sorts correspond to its two computational domains: Herbrand universe and finite domains for integers) first order language (see [8]) with a denumerable set of variable *Var* for every sort ($Var = Var_H \cup Var_{FD}$), a set of function symbols Σ (containing the standard arithmetic operators $+$, $-$, $*$, $/$) with their signature, and a set of predicate symbols Π with their signature. Variables belonging to Var_{FD} are called *domain variables*. Predicates belonging to $\Pi_C \subset \Pi$ are the *constraint predicates* of CHIP. They are the arithmetic constraints $=$, $\neq$, $\leq$, $<$, $\geq$, $>$, and the symbolic constraints `element`, `atleast`, `atmost` (see [19, 1, 3]). Predicates belonging to $\Pi_A = \Pi \setminus \Pi_C$ are the *atom predicates* of CHIP. In particular `indomain` $\in \Pi_A$, whose only argument is a domain variable or a integer. The notion of term, finite-domain term, and Herbrand term can be given in the usual way by respecting the sorts. We denote $\Re_{CHIP}$ the structure of CHIP.

We let *Constraint* denote the set of constraints. A *constraint* is an atom $p(t_1, ..., t_l)$ such that $p \in \Pi_C$ (arity l) and every t_i for $i \in \{1, ..., l\}$ is a finite-domain term.

We let *Atom* denote the set of atoms that are not constraint (in the following, *atoms tout court*), that is atoms $p(t_1, ..., t_l)$ such that $p \in \Pi_A$ (arity l) and every t_i for $i \in \{1, ..., l\}$ is a term of the appropriate sort. An atom whose predicate is `indomain` is said to be a *generator*.

We let *Subst* denote the set of idempotent substitutions (with the proviso of respecting the sorts). We let ε denote the empty substitution. Let t be a term. We denote $var(t)$ the variables which occur in t.

Definition 2.1 Let $\theta \in Subst$. We denote $ren(\theta)$ the subset of θ that is a pure variable

substitution:

$$ren(\theta) = \{X/\theta(X) | X \in dom(\theta) \,.\, \theta(X) \in Var\}$$

We denote $gro(\theta)$ the subset of θ that is a pure grounding substitution:

$$gro(\theta) = \{X/\theta(X) | X \in dom(\theta) \,.\, var(\theta(X)) = \emptyset\}$$

As usual we denote $Cl \ll_\Upsilon P$ a variant clause Cl of a program clause of the program P such that $var(Cl) \cap var(\Upsilon) = \emptyset$, where Υ is a syntactic object.

2.1 Finite domains

First we introduce some notions that are useful in the following.

Definition 2.2 [finite-domain set] We let $FDomS$ denote the set of finite-domain sets. A *finite-domain set* is a finite set $\{X_1 \,\mathtt{in}\, d_1, ..., X_n \,\mathtt{in}\, d_n\}$, where $X_i \in Var_{FD}$ (all variables are distinct) and d_i is a finite domain $\forall i \in 1, ..., n$. A *finite domain* d is either the expression $\{a_1, ..., a_m\}$ $(m \geq 1)$, where the a_j's $(j \in \{1, ..., m\})$ are positive integers, or the expression $[l..u]$, where l and u are integers such that $l < u$.

In the first case it means that the variable X ranges over the set $\{a_1, ..., a_m\}$; in the second case it means that X range over the integers $l, l+1, ..., u$.

Definition 2.3 Let $FDS \in FDomS$ be $\{X_1 \,\mathtt{in}\, d_1, ..., X_n \,\mathtt{in}\, d_n\}$. We denote $var(FDS)$ the set $\{X_1, ..., X_n\}$ and we denote $fd(X_i, FDS)$ the corresponding set of values for X_i in FDS.

A finite-domain set could also be expressed as a (finite) set of grounding substitutions for its domain variables.

Definition 2.4 Let $FDS \in FDomS$ be the finite-domain set $\{X_1 \,\mathtt{in}\, d_1, ..., X_n \,\mathtt{in}\, d_n\}$. The set of all the possible substitutions obtained from FDS, denoted $subst(FDS)$, is defined by: $\theta \in subst(FDS) \Leftrightarrow \theta = \{X_1/a_1, ..., X_n/a_n\}$ with $a_i \in d_i \; \forall i \in \{1, ..., n\}$

Definition 2.5 Let $V \subseteq Var_{FD}$. We denote by $FDS|_V$ the restriction of FDS to the variables of V:

$$FDS|_V = \{X \,\mathtt{in}\, d \mid X \,\mathtt{in}\, d \in FDS, X \in V\}$$

Definition 2.6 Let $FDS \in FDomS$. Let $\rho \in Subst$ a pure variable substitution such that $var(FDS) \cap rng(\rho) = \emptyset$; we define the finite-domain set

$$FDS[\rho] = \{X \,\mathtt{in}\, d \mid (\exists Y \in dom(\rho) \cap var(FDS) \,.\, (X = \rho(Y) \wedge d = fd(Y, FDS))) \vee \\ \vee (X \notin dom(\rho) \wedge X \,\mathtt{in}\, d \in FDS)\}$$

Let $\rho \in Subst$ be a pure grounding substitution; we define the finite-domain set

$$FDS[\rho] = \{X \,\mathtt{in}\, \rho(X) \mid X \in dom(FDS) \cap dom(\rho)\}$$

The following definitions formalize, by introducing a order relation between finite-domain sets, the notion of *domain refinement*. They take into account the fact that during the execution of a computation step, domains may be refined (the corresponding variables become "more instantiated") while other variables may be introduced in the actual goal. The strict order relation holds when at least one variable domain is reduced.

Definition 2.7 Let $FDS_1, FDS_2 \in FDomS$. We define the following order relation $\preceq$ on $FDomS$:

$$FDS_1 \preceq FDS_2 \Leftrightarrow var(FDS_2) \subseteq var(FDS_1) \wedge \\ \wedge \forall X \in var(FDS_2) \,.\, fd(X, FDS_1) \subseteq fd(X, FDS_2)$$

We define

$$FDS_1 \prec FDS_2 \Leftrightarrow FDS_1 \preceq FDS_2 \wedge \exists X \in var(FDS_2) \,.\, fd(X, FDS_1) \subset fd(X, FDS_1)$$

Note that $FDS \preceq \emptyset$ for each $FDS \in FDomS$. Coherently with that, we also define $FDS \prec \emptyset$ for each $FDS \in FDomS$.

Proposition 2.8 *The relation $\preceq$ is a partial order over $FDomS$ (i.e. $\langle FDomS, \preceq \rangle$ is a poset).*

2.2 Programs

We let *Program* denote the set of programs. A program P consists of a finite set of clauses and of a finite set of declarations.

Definition 2.9 [clauses] We let *Clause* denote the set of clauses:

$$Clause = (Atom \cup nil) \times (Atom \cup Constraint)^* \times FDomS$$

Let Cl be the clause $H \leftarrow B_1, ..., B_n \;;\; FDS$. The atom H, denoted $head(Cl)$ is the *head* of the clause, the finite sequence of atoms and constraints $B_1, ..., B_n$, denoted $body(Cl)$ is the *body* of the clause. $FDS \in FDomS$, denoted $fdset(Cl)$, is the finite-domain set of the clause[1], and it is such that $\forall B_i \in Constraint (1 \leq i \leq n)\ var(B_i) \subset (FDS)$. We let *Goal* denote the set of *goals*, i.e. clauses with *nil* (the zero length sequence) as head. We assume that there are no variables in the clauses (goals) whose finite-domains are singleton and that are not instantiated. This assumption is obviously not restrictive.

Program declarations are delay declarations for atom predicates. A delay declaration for an atom predicate p/n has the form `delay` $p(a_1, ..., a_n)$, where the a_i's are either `nonvar` or `any` or `ground`. The `indomain` predicate is not submitted to any delay declaration.

3 Operational semantics

The operational semantics is defined as a transition rules system on states.

[1] In CHIP [3], cc(FD) [23] and other languages membership to finite domains is expressed as particular (unary) constraint. Here we prefer, for clarity sake, to single out the finite domains from the body.

3.1 Availability predicate

First we define when an atom or a constraint is available for being selected by the computation rule. This notion is given with respect to the current finite-domain set and the given program.

Definition 3.1 [availability predicate] Define the *availability predicate* for P,

$$available_P : (Atom \cup Constraint) \times FDomS \rightarrow \{true, false\}$$

as the predicate such that:

- if $B = p(t_1, ..., t_l) \in Atom$ then $available_P(B, FDS) = true$ iff either
  1. B is not submitted to the delay declaration; or
  2. B is submitted to the delay declaration `delay` $p(\tau_1, ..., \tau_l)$ in P, and for each h ($h \in \{1, ..., l\}$) such that τ_h is `ground`, t_h is a ground term and for each h ($h \in \{1, ..., l\}$) such that τ_h is `nonvar`, t_h is not a variable;
- if $B \in Constraint$ then $available_P(B, FDS) = true$ iff

  B is $\mathtt{X} = \mathtt{Y}$ or $\mathtt{X} \leq \mathtt{Y}$ or $\mathtt{X} < \mathtt{Y}$ or $\mathtt{X} \geq \mathtt{Y}$ or $\mathtt{X} > \mathtt{Y}$, where `X` and `Y` are linear terms[2];

  B is $\mathtt{X} \neq \mathtt{Y} + \mathtt{c}$, where `c` is an integer and, either `X` is a domain variable and `Y` is an integer or `Y` is a domain variable and `X` is an integer;

  B is `element(I,L,X)`, where `L` is a list of positive integers, `I` is a domain variable or a positive integer and `X` is a variable or a positive integer[3].

3.2 States

A *state* is a goal with the atoms and the constraints of his body labelled as *true* or as *false*. The intended meaning is that atoms and constraints labelled as *false* are delayed, while the others can be selected.

Definition 3.2 [states] We let *State* be the set of states:

$$State = ((Atom \cup Constraint) \times \{true, false\})^* \times FDomS$$

We denote $ebody(S)$ and $fdset(S)$ respectively the labelled body and the finite-domain set of the state S. We denote $label_S(B_k)$ the label of the atom or constraint B_k in the body of S.

Definition 3.3 [goal-state mapping] Let $S \in State$ be $\langle B_1 \diamond E_1, ..., B_n \diamond E_n \ ; \ FDS \rangle$. We denote $G(S)$ the goal $\leftarrow B_1, ..., B_n \ ; \ FDS$. Let G be the goal $\leftarrow B_1, ..., B_n \ ; \ FDS$. We denote $S(G)$ the state $\langle B_1 \diamond E_1, ..., B_n \diamond E_n \ ; \ FDS \rangle$, with $E_i = available_P(B_i, FDS)$ ($i \in \{1, ..., n\}$).

[2] A linear term over domain variables is either: (i) a positive integer; or (ii) a domain variable; or (iii) $\mathtt{X} * \mathtt{c}$, where $\mathtt{X} \in Var_{FD}$ and `c` is a positive integer; or (iv) $\mathtt{t} + \mathtt{s}$, where `t` and `s` are linear terms.

[3] We recall that `element(I,L,X)` means that `X` is the `I`-th element of `L`. When this constraint is considered, `X` is instantiated to a domain variable ranging over the elements of `L`.

The fact that is available is a necessary but not sufficient condition for being labelled as *true*. Indeed there are constraints (like `element`: see example 3.10) that, even when available, could not give any new information if selected. Therefore, labels are updated during the computation in order to assure that only possibly useful constraints are selected (see later).

3.3 Computation rule

Definition 3.4 [collect function] Let $P \in Program$ and $S \in State$. We define the *collecting function*

$$collect_P : State \rightarrow (Atom^* \cup Constraint^*)$$

as the function that extracts from the sequence of atoms and constraints of the state the ordered subsequence of all the atoms and constraints labelled as *true*.

Definition 3.5 [computation rules] A *computation rule* for a program P is a function

$$\mathcal{R} : State \rightarrow (Atom \cup Constraint)$$

such that:

- $\mathcal{R}(S)$ is defined iff $collect_P(S) \neq nil$
- $\mathcal{R}(S) \in collect_P(S)$

Let S be state $\langle B_1 \diamond E_1, ..., B_n \diamond E_n \ ; \ FDS \rangle$. The atom or constraint $\mathcal{R}(S) = B_k$ is selected in S by $\mathcal{R}$.

The CHIP computation rule $\mathcal{R}_{CHIP}$ is as follows: from $collect_P(S)$ the disequalities ($\neq$) constraints are scheduled first, secondly the other kinds of constraints, and finally the atoms. In the case of ties the left-most occurrence is chosen.

3.4 State to state transitions

There are three different kinds of state to state transitions: (i) *by value generation*; (ii) *by clause selection*; and (iii) *by constraint propagation*. In the following we always refer to a given computation rule $\mathcal{R}$.

Transitions by value generation take place when a generator atom `indomain`(X) is selected: the effect is the instantiation of the finite-domain variable X.

Definition 3.6 [transition by value generation] Let $P \in Program$ and $S' \in State$ be the state

$$\langle B_1 \diamond E_1, ..., B_n \diamond E_n \ ; \ FDS \rangle$$

There is a *transition by value generation* from S' to the state S'' via $\mathcal{R}$ if the following conditions hold:

1. $\mathcal{R}(S') = B_k$ where B_k is a generator atom, i.e. B_k is `indomain`(t);
2. one of the following two cases holds:

(a) either B_k is `indomain`(a) where a is an integer; then $\theta = \varepsilon$;

(b) or B_k is `indomain`(V) where $V \in var(FDS)$;

3. Let θ be the following substitution: in case 2(a) $\theta = \varepsilon$, in case 2(b) $\theta = \{V/a\}$, where $a \in fd(V, FDS)$ called the selected value; S'' is the state

$$\langle B_1\theta \diamond E'_1, ..., B_{k-1}\theta \diamond E'_{k-1}, B_{k+1}\theta \diamond E'_{k+1}, ..., B_n\theta \diamond E'_n \ ; \ FDS'' \rangle$$

where

$$E'_i = E_i \vee (available_P(B_i\theta, FDS'') \wedge B_i\theta \neq B_i) \ \ (i \in \{1, ..., k-1, k+1, ..., n\})$$

and where

$$FDS'' = FDS|_{var(FDS)\setminus\{V\}}$$

Example 3.7 Let P be a program with delay declaration `delay p(ground)`. Let S' be the state (we indicate the selected atom or constraint by underlining it)

$$\langle \mathtt{Z} \neq \mathtt{X} \diamond \mathit{false}, \underline{\mathtt{indomain(X)} \diamond \mathit{true}}, \mathtt{p(Z)} \diamond \mathit{false} \ ; \ \{\mathtt{X\ in\ [1..5]}, \mathtt{Z\ in\ [1..10]}\} \rangle$$

and S'' the state

$$\langle \mathtt{Z} \neq 2 \diamond \mathit{true}, \mathtt{p(Z)} \diamond \mathit{false} \ ; \ \{\mathtt{Z\ in\ [1..10]}\} \rangle$$

Then there is transition by value generation from S' to S'' in P via $\mathcal{R}_{CHIP}$. The computed substitution is $\theta = \{\mathtt{X}/2\}$.

Transitions by clause selection are the generalization of standard Prolog transitions. The most general unifier function has to be modified in order to deal with finite-domain variables.

Definition 3.8 [*fdmgu* function] Define the *finite-domain mgu function* [19]

$$fdmgu : Atom^2 \times FDomS^2 \rightarrow (Subst \times FDomS) \cup fail$$

as the function that, given two atoms B and H and two finite-domain sets FDS and FDS', returns the result of the execution of the modified unification algorithm (see the appendix at the end of the paper) on the set of expressions $\mathcal{E} = \{B, H\}$ and on the finite-domain set $FDS \cup FDS'$[4]. This algorithm [19] is a modification of the usual unification algorithm enhanced with type checking. It returns an idempotent most general unifier and a finite-domain set.

Let $P \in Program$, $B \in Atom$, $FDS \in FDomS$ and $Cl \ll_B P$. We say that B *unifies (w.r.t. FDS and* $fdset(Cl)$*) with* $head(Cl)$ iff $fdmgu(B, head(Cl), FDS, fdset(Cl)) \neq fail$.

Definition 3.9 [transition by clause selection] Let $P \in Program$, $S' \in State$ be the state

$$\langle B_1 \diamond E_1, ..., B_n \diamond E_n \ ; \ FDS \rangle$$

and let $Cl \ll_{S'} P$ be the clause

$$H \leftarrow B'_1, ..., B'_m \ ; \ FDS'.$$

There is a *transition by clause selection* from S' to S'' via $\mathcal{R}$ if the following conditions hold:

[4]Because of the standardization apart condition this union is simply the set union.

1. $\mathcal{R}(S') = B_k$;
2. $fdmgu(B_k, H, FDS, FDS') = \langle \theta, FDS'' \rangle$;
3. S'' is the state

$\langle B_1\theta \diamond E'_1, ..., B_{k-1}\theta \diamond E'_{k-1}, B'_1\theta \diamond F_1, ..., B'_m\theta \diamond F_m, B_{k+1}\theta \diamond E'_{k+1}, ..., B_n\theta \diamond E'_n \ ; \ FDS'' \rangle$

where, letting $U_i = var(B_i) \cap var(B_k)$ $(i \in \{1, ..., k-1, k+1, ..., n\})$

$$change_i = \begin{cases} true \quad if & either \quad \theta|_{U_i} \text{ is not a pure variable substitution} \\ & or \quad FDS'' \cup FDS_{U_i}[gro(\theta)] \prec FDS_{U_i}[ren(\theta)]) \\ false \quad otherwise & \end{cases}$$

$$E'_i = E_i \ \vee (available_P(B_i\theta, FDS'') \ \wedge \ change_i) \qquad (1)$$

and where

$$F_j = available_P(B'_j\theta, FDS'') \quad (j \in \{1, ..., m\})$$

The formula (1) assures that the labels are updated during the computation so that only possibly useful constraints are selected. Note that the boolean value *change* is a necessary condition. Indeed, if a constraint B_i was available before but it was labelled false, then $change_i$ garantees that it is labelled *true* only if there has been a reduction of its variables domains. Moreover, $change_i$ is also *true* if an atom or a constraint B_i that was not, becomes available. In this case, the computed substitution $\theta|_{U_i}$ is not a pure variable substitution. Compare this formula with the one in [19], where the syntactical check $B_i\theta \neq B_i$ is done: this check is not sufficient. See the example3.10.

Example 3.10 Let $P \in$ *Program* be a program with atom predicates p/2, s/1, q/2 not submitted to any delay declaration in P. Let S' be the state

$\langle$element(X, L, K) $\diamond$ *false*, element(X, [8, 6, 0], Y) $\diamond$ *false*, p(X, T, L) $\diamond$ *true* ; {X in {1, 3}, Y in {0, 8}, T in [1..5]}$\rangle$

Cl the clause of P

$$\texttt{p(Z, W, [6, 4, 5])} \leftarrow \texttt{q(Z, W), s(W) ; \{W in [5..7], Z in \{1, 3\}, K in [4..8]\}}$$

and S'' the state

$\langle$element(V, [6, 4, 5], K) $\diamond$ *true*, element(V, [8, 6, 0], Y) $\diamond$ *false*, q(V, 5) $\diamond$ *true*, s(5) $\diamond$ *true* ; {V in {1, 3}, Y in {0, 8}, K in [4..8]}$\rangle$

Then there is a transition by clause selection from S' to S'' in P (via $\mathcal{R}_{CHIP}$). The computed substitution is θ = {X/V, L/[6, 4, 5], Z/V, T/5, W/5}. Note $label_{S''}$(element(V, [6, 4, 5], K)) = *true* even if the domains for X and K are not reduced, while $label_{S''}$(element(V, [8, 6, 0], Y)) = *false*.

Transition by constraint propagation takes place when a constraint is selected.

Definition 3.11 [*cprop* function] Define the *constraint propagation function*,

$$cprop_P : Constraint \times Var \times FDomS \rightarrow (Subst \times FDomS) \cup fail$$

as the function such that:

- $cprop_P$ is defined for $B \in Constraint$, $V \subseteq Var_{FD}$ and $FDS \in FDomS$ only if $V \subseteq var(FDS)$ and $available_P(B, FDS) = true$,
- if B is ground then it returns $\langle \varepsilon, \emptyset \rangle$ if $\Re_{CHIP} \models B$, *fail*, otherwise.
- if B is not ground it returns the result of the execution of the constraint propagation algorithm (3.1, given below) on B, V and FDS.

The constraint propagation algorithm [19] performes the domain refinement by computing a substitution and a finite-domain set.

Algorithm 3.1 [constraint propagation algorithm] (from [19]) Let B be an available constraint (w.r.t. FDS) in P, $\{X_1, ..., X_r\}$ the set of the domain variables of B and $FDS \in FDomS$. The value of $cprop_P(B, \{X_1, ..., X_r\}, FDS)$ is computed as follows:

1. set h to 0, θ_0 to ε and FDS_0 to $\emptyset$;
2. if $h = r$, then stop: θ_h and FDS_h is the result of the algorithm; otherwise increment h;
3. let[5]

$$\Omega_h = \{a_h \in fd(X_h, FDS) \mid \exists\, (a_1 \in fd(X_1, FDS), ..., a_{h-1} \in fd(X_{h-1}, FDS), a_{h+1} \in fd(X_{h+1}, FDS), ..., a_r \in fd(X_r, FDS)) . \Re_{CHIP} \models B\{X_1/a_1, ..., X_r/a_r\}\};$$

4. if $\Omega_h = \emptyset$ then go to step 6; otherwise go to step 5;
5. if $\Omega_h = \{c\}$, then put $\theta_h = \theta_{h-1} \cup \{X_h/c\}$ and $FDS_h = FDS_{h-1}$; otherwise put $\theta_h = \theta_{h-1} \cup \{X_h/Z_h\}$, with Z_h new variable, and $FDS_h = FDS_{h-1} \cup \{Z_h \text{ in } \Omega_h\}$; go to step 2;
6. stop: return *fail*.

For example, disequations $\mathtt{X} \neq \mathtt{Y}$ are handled as follows: if $\mathtt{X}$ is a domain variable and $\mathtt{Y}$ is a positive integer, then the set Ω in the step 3 is the set $\{a \in fd(\mathtt{X}, FDS) \mid a \neq \mathtt{Y}\}$.

Definition 3.12 [transition by constraint propagation] Let $P \in Program$ and $S' \in State$ be the state

$$\langle B_1 \diamond E_1, ..., B_n \diamond E_n \ ; \ FDS \rangle$$

There is a transition *by constraint propagation* from S' to the state S'' via $\mathcal{R}$ if the following conditions hold:

[5] In the case of linear (in)equality constraints the algorithm computes sets Ω'_h such that $fd(X_h, FDS) \supseteq \Omega'_h \supseteq \Omega_h$, by using an idempotent *sup-inf* procedure [19]. This method is *partial*, that is it works by reasoning only about the lower and upper bounds of domains, so that not *all* the inconsistent values are removed from the domains of the variables.

1. $\mathcal{R}(S') = B_k$;
2. $cprop_P(B_k, V, FDS) = \langle \theta, FDS' \rangle$, where $V = var(B_k) \cap Var_{FD}$;
3. S'' is the state

$$S'' = \begin{cases} \langle B_1\theta \diamond E'_1, ..., B_{k-1}\theta \diamond E'_{k-1}, B_{k+1}\theta \diamond E'_{k+1}, ..., B_n\theta \diamond E'_n \ ; \ FDS'' \rangle \\ \qquad \text{if there is at most one not-ground variable in } FDS' \\ \langle B_1\theta \diamond E'_1, ..., B_{k-1}\theta \diamond E'_{k-1}, B_k\theta \diamond E'_k, B_{k+1}\theta \diamond E'_{k+1}, ..., B_n\theta \diamond E'_n \ ; \ FDS'' \rangle \\ \qquad \text{otherwise} \end{cases}$$

where

$$FDS'' = FDS' \cup FDS|_{var(FDS) \setminus dom(\theta)}$$

and where, letting $U_i = var(B_i) \cap var(B_k)$ $(i \in \{1, ..., k-1, k+1, ..., n\})$

$$change_i = \begin{cases} true \quad if & either \quad \theta|_{U_i} \text{ is not a pure variable substitution} \\ & or \quad FDS'' \cup FDS_{U_i}[gro(\theta)] \prec FDS_{U_i}[ren(\theta)]) \\ false \quad otherwise \end{cases}$$

$$E'_i = E_i \ \vee (available_P(B_i\theta, FDS'') \ \wedge \ change_i)$$

$$E'_k = false \qquad \text{(if still present)}$$

Example 3.13 Let $P \in Program$ be a program. Let S' be the state

$$\langle \texttt{element(T,[10,20,30],X} \diamond \textit{false}, \underline{\texttt{1} \neq \texttt{T} \diamond \textit{true}} \ ; \ \{\texttt{X in \{10,20,30\}, T in [1..3]}\}\rangle$$

S'' be the state

$$\langle \underline{\texttt{element(T1,[10,20,30],X} \diamond \textit{true}} \ ; \ \{\texttt{X in \{10,20,30\}, T1 in [2..3]}\}\rangle$$

and S''' be the state

$$\langle \texttt{element(T2,[10,20,30],X2} \diamond \textit{false} \ ; \ \{\texttt{X1 in \{20,30\}, T2 in [2..3]}\}\rangle$$

Then there is a transition by constraint selection from S' to S'' and from S'' to S''' in P (via $\mathcal{R}_{CHIP}$). The computed substitution is $\theta = \{\texttt{X/X1, T/T2}\}$.

Therefore we have the following definition.

Definition 3.14 [state to state transitions] Let $S', S'' \in State$. There is a transition from S' to S'' via $\mathcal{R}$ in P (using θ), denote $S' \xrightarrow[P,\mathcal{R}]{\theta} S''$, if either (i) there is a transition by value generation from S' to S'' via $\mathcal{R}$; or (ii) there is a transition by clause selection from S' to S'' via $\mathcal{R}$; or (iii) there is a transition by constraint propagation from S' to S'' via $\mathcal{R}$.

Proposition 3.15 *Let $P \in Program$ and $S', S'' \in State$. Then*

$$S' \xrightarrow[P,\mathcal{R}]{\theta} S'' \Rightarrow fdset(S'') \cup fdset(S')[gro(\theta)] \preceq fdset(S')[ren(\theta)]^6$$

[6]Note that the strict inequality does *not* necessarily hold (consider for example the stae S'' of the previous example as initial state).

3.5 Computations

When a selection rule $\mathcal{R}$ is given, there are two non-deterministic aspects: one pertaining to the choice of clauses whose head unifies with the atom selected by $\mathcal{R}$, and the other pertaining to the choice of values for the not-ground generator atom selected by $\mathcal{R}$.

Definition 3.16 [computations] Let $P \in \mathit{Program}$ and $G \in \mathit{Goal}$. A *computation* $\mathcal{C}$ for $P \cup \{G\}$ (via $\mathcal{R}$) is a (possibly infinite) sequence of states $S_0, S_1, ..., S_n, ...$, where $S_0 = S(G)$ is the initial state and $S_i \xrightarrow[P,\mathcal{R}]{} S_{i+1}$ $(i \geq 0)$.

As usual, we denote $\xrightarrow[P,\mathcal{R}]{*}$ the transitive closure of the $\xrightarrow[P,\mathcal{R}]{}$ relation.

Referring to a program $P \in \mathit{Program}$, we define the following notions.

Definition 3.17 [success states] $S \in \mathit{State}$ is a *success state* iff $\mathit{ebody}(S) = \mathit{nil}$.

Definition 3.18 [floundering states] $S \in \mathit{State}$ is a *floundering state* if it is not a success state and it is such that $\mathit{collect}_P(S) = \mathit{nil}$.

Definition 3.19 [failure states] $S \in \mathit{State}$ is a *failure state* iff one of the followings holds:

(a) if $\mathcal{R}(S) = B \in \mathit{Atom}$, then B is not a generator atom and it does not unify with the head of any clause of P;

(b) if $\mathcal{R}(S) = B \in \mathit{Constraint}$, then $\mathit{cprop}_P(B, V, \mathit{fdset}(S)) = \mathit{fail}$, V being the (possibly empty) set of the variables of B.

Computations may be finite or infinite. Finite computations may be successful or failed or they may flounder. A *successful computation* is one that ends with a success state. A *finitely failed computation* is one that ends with a fail state. A *floudering computation* is one that ends with a floundering state[7].

The following definition, capture the notion of result of a computation.

Definition 3.20 [computed answer substitutions] Let $P \in \mathit{Program}$ and $G \in \mathit{Goal}$ be a goal such that there is a successful computation $\mathcal{C}$ for $P \cup \{G\}$. A *computed answer substitution* θ for $P \cup \{G\}$ is the substitution obtained by restricting the composition of the substitution used in $\mathcal{C}$ to the variables of G.

Definition 3.21 [computed answer finite-domain set] Let $P \in \mathit{Program}$ and $G \in \mathit{Goal}$ be a goal such that there is a successful computation $S_0 = S(G), ..., S_n$ for $P \cup \{G\}$ with computed answer substitution θ. The finite-domain set

$$\{X \text{ in } \{X\theta\} \mid X \in \mathit{var}(\mathit{fdset}(G)) \cap \mathit{dom}(\mathit{gro}(\theta))\} \cup \mathit{fdset}(S_n)|_{\mathit{var}(G\theta)}$$

is said the *computed answer finite-domain set* for $P \cup \{G\}$.

[7]Remember that in CHIP the burden of providing generators that garantee that sooner or later any constraint will be selected is left to the programmer [19].

The computed answer finite-domain is the union of the set of the finite-domains corresponding to the domain variables of the initial goal that have been instantiated during the computation, together with the set of the (possibly reduced) finite-domains corresponding to the other (not instantiated) initial variables. .

Definition 3.22 [operational semantics] The operational semantics $\mathcal{O}[P]$ of a program P is given in terms of the set $\mathcal{O}[P] \in (Atom \cup Constraint)^*$ for P:

$$\begin{aligned}\mathcal{O}[P] = \{B\theta\sigma \mid\ & B \in Atom \cup Constraint, \theta \in Subst, FDS, FDS' \in FDomS, \sigma \in Subst\ . \\ & var(FDS) = var(B) \cap Var_{FD}, \\ & \langle B; FDS\rangle \xrightarrow[P,\mathcal{R}]{*} \langle \varepsilon, FDS'\rangle \\ & \text{with computed answer substitution } \theta, \text{ and} \\ & \text{with } \sigma \in subst(FDS)\}.\end{aligned}$$

4 Appendix

The modified unification algorithm [19] takes as input a set of expressions $\mathcal{E}$ and a finite-domain set $FDS \in FDomS$ and gives back as output a substitution and a finite-domain set. If $\mathcal{E}$ is unifiable w.r.t. FDS, then the unification algorithm terminates and gives an mgu for $\mathcal{E}$ and a new finite-domains set. If $\mathcal{E}$ is not unifiable w.r.t. FDS, then the unification algorithm terminates and reports this fact (see [19]).

Algorithm 4.1 [modified unification algorithm](from [19]) Let $\mathcal{E}$ be a finite set of simple expressions and let $FDS \in FDomS$. The *modified unification algorithm* is the following:

1. set k to 0, θ_0 to ε and FDS_0 to FDS;
2. if $\mathcal{E}\theta_k$ is a singleton, then stop: θ_k is the substitution and FDS_k is the finite-domains set returned by the algorithm; otherwise find the disagreement set D_k of $\mathcal{E}\theta_k$;
3. if there exist a variable $V \notin Var_{FD}$ and t in D_k such that V does not occur it t, then put $\theta_{k+1} = \theta_k\{V/t\}$, $FDS_{k+1} = FDS_k$, increment k and go to step 2;
4. if there exist a domain variable $V \in Var_{FD}$ and a constant c in D_k such that $c \in fd(V, FDS)$, then put $\theta_{k+1} = \theta_k\{V/c\}$, $FDS_{k+1} = FDS_k|_{var(FDS_k)\setminus\{V\}}$, increment k and go to step 2;
5. if there exist $V, U \in var(FDS_k)$ in D_k such that $FDS' = fd(V, FDS_k) \cap fd(U, FDS_k) \neq \emptyset$, then[8]: if $FDS' = \{c\}$ then put $FDS_{k+1} = FDS_k|_{var(FDS_k)\setminus\{V,U\}}$ and $\theta_{k+1} = \theta_k \cup \{V/c\} \cup \{U/c\}$; else put $\theta_{k+1} = \theta_k \cup \{V/Z, U/Z\}$, $FDS_{k+1} = FDS_k|_{var(FDS_k)\setminus\{V,U\}} \cup \{Z \text{ in } FDS'\}$, where Z is a new variable[9], increment k and go to step 2;
6. otherwise stop: $\mathcal{E}$ is not unifiable.

[8]This is a slight modification of the original algorithm (see [19]): it ensures that new variables whose domain is a singleton are instantiated.

[9]By new variable for a program and a states sequence, that is the current partial computation, we mean a variable not used elsewhere in the program or in the states of the sequence.

5 Conclusions

We have described a detailed (low-level) operational semantics for the kernel of the finite-domains part of the CHIP language. This semantics completes and corrects the one outlined by Van Hentenryck in [19].

As mentioned in the introduction, the definition of this semantics is intended to constitute a first preliminary step in the direction of the development of semantics-based optimizations of CHIP.

Acknowledgments We would like to thank ICON S.r.l. for the access to their version of CHIP.

References

[1] A. Aggoun and N. Beldiceanu. Overview of the CHIP Compiler System. In Proc. Eighth Int. Conf. on Logic Programming (ICLP'91), MIT Press, pp. 775-789, Paris, France (1991).

[2] A. Aggoun and N. Beldiceanu. La Technologie CHIP. Tech. Rep., COSYTEC SA, France (1992).

[3] CHIP User's Guide. Version 1.1 Revision B. COSYTEC SA, France (1991).

[4] A. Colmerauer. An Introduction to Prolog III. ACM Comm., 33(7), pp. 70-90, July (1990).

[5] M. Dincbas, P. Van Hentenryck, M. Simonis, A. Aggoun, T. Graf, F. Berthier. The Constraint Logic Programming Language CHIP. In Proc. Int. Conf. on Fifth Generation Computer System (FGCS'88), pp. 693-702, Tokyo, Japan, December (1988).

[6] M. Dincbas, M. Simonis, and P. Van Hentenryck. Solving Large Combinatorial Problems in Constraint Logic Programming. Journal of Logic Programming, 8(1-2), pp. 75-93, (1990).

[7] T. Frühwirth. Constraint Simplification Rules. Tech. Rep. ECRC, Munich, Germany (1992)

[8] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. Tech. Rep. 86/73, Monash University, Victoria, Australia, June (1986).

[9] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. In Proc. 14th ACM Conf. on Priciples of Programming Languages (POPL'87), pp. 111-119, Munich, January (1987).

[10] J. Jaffar and J.-L. Lassez. From Unification to Constraint. In Proc. 6th Japanise Logic Programming Conf., Tokyo, Japan, June, (1987). Lecture Notes in Computer Science No. 315, Springer-Verlag, Berlin, pp. 12-24, (1987).

[11] J. Jaffar and S. Michaylov. Methodology and Implementation of CLP System. In Proc. Fourth Int. Conf. on Logic Programming (ICLP'87), pp. 196-218, Melbourne, May (1987).

[12] J. Jaffar, S. Michaylov, P.J. Stuckey and R.H.C. Yap. The CLP($\mathcal{R}$) Language and System. ACM Trans. on Prog. Lang. and Systems, vol. 14(3), pp. 339-395, (1992).

[13] T. Le Provost and M. Wallace. Generalised Constraint Propagation Over the CLP Scheme. Tech. Rep. ECRC, Munich, Germany (to appear in the Journal of Logic Programming) (1992)

[14] J.W. Lloyd. *Foundations of Logic programming.* 2nd edition, Springer-Verlag, (1987).

[15] A.K. Mackworth. Consistency in Network of Relations. AI Journal, 8(1), pp.99-118, (1977).

[16] A.K. Mackworth. The logic of constraint satisfaction. Art. Int., 58 (1-3), pp.3-20, (1992).

[17] U. Montanari and F. Rossi. Finite Domain Constraint Solving and Constraint Logic Programming. In *CLP: Constraint Logic Programming: Stated Research* (A. Colmerauer and F. Benhamou eds.), MIT Press, (1992).

[18] P. Van Hentenryck. A Theoretical Framework for Consistency Techniques in Logic Programming. In Proc. of Int Joint Conf. on Artificial Intelligence (IJCAI-87), pp. 2-8, Milan, Italy, August (1987).

[19] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, (1989).

[20] P. Van Hentenryck and Y. Deville. Operational Semantics for Constraint Logic Programming over Finite Domains. In Proc. of PLILP'91, Passau, Germany, August, vol 528 of Lecture Notes in Computer Science, pp. 395-406, Springer-Verlag, Berlin (1991).

[21] P. Van Hentenryck and Y. Deville. The Cardinality Operator: A new Logical Connective for Constraint Logic Programming. In Proc. Eighth Int. Conf. on Logic Programming (ICLP'91), MIT Press, Paris, France (1991).

[22] P. Van Hentenryck, Y. Deville and C.-M. Teng. A generic arc-consistency algorithm and its specializations. Art. Int. 57, pp 291-321, (1992).

[23] P. Van Hentenryck, V.A. Saraswat and Y. Deville. Design, Implementations, and Evaluation of the Constraint Language cc(FD). Tech. Rept., Brown Univ., Providence, RI, January (1993).

A Note on Logic Programming Fixed-Point Semantics

Vladimiro Sassone

Dipartimento di Informatica – Università di Pisa
e-mail: vladi@di.unipi.it

Abstract

In this paper, we present an account of classical Logic Programming fixed-point semantics in terms of two standard categorical constructions in which the least Herbrand model is characterized by properties of universality.

In particular, we show that, given a program P, the category of models of P is reflective in the category of interpretations for P. In addition, we show that the immediate consequence operator gives rise to an endofunctor T_P on the category of Herbrand interpretations for P such that category of algebras for T_P is the category of Herbrand models of P.

As consequences, we have that the least Herbrand model of P is the least fixed-point of T_P and is the reflection of the empty Herbrand interpretation.

Introduction

Logic Programming, arisen in the early seventies from the work on automatic theorem proving, is a very simple formalism based on the rigorous mathematical framework of first-order theories. The revolutionary idea introduced in Kowalski's work [13] is that logic has a *computational interpretation* and, therefore, logic can be used as a programming language.

Generally speaking, following [17], a logic system is a formal system consisting of the set of sentences over a certain alphabet equipped with an *entailment relation*—informally speaking a proof system capable to "calculate" consequences of sets of sentences—and a *satisfaction relation* related by a condition of soundness, but not necessarily of completeness, of entailment with respect to satisfaction. From the Logic Programming point of view, any such system is a programming language whose operational semantics is given by the entailment relation.

The classical theory of Logic Programming is devoted to the study of the fragment of first-order logic consisting of *Horn clauses*. Although Horn clauses reduce the expressive power of first-order logic, the choice of such a fragment presents some advantages both from the computational and from the model theoretic point of view. In fact, on one hand it has a simple, efficient goal-oriented deduction system (*SLD-resolution*) while on the other hand it is the largest fragment of first-order logic such that every set of formulas (program) admits an *initial model*, i.e., a model in which the facts entailed by the program are true and everything else is false. Moreover, Horn clauses have a nice computational interpretation as function calls in functional programming.

The latter fact led to the development of a *functional* (or fixed-point) *semantics* for logic programs with which we are concerned in this note and that we recall in Section 1.

In recent years, *algebraic* ([10, 11]) and *categorical* ([2, 3, 8, 17]) methods have been applied in the study of logic programs. The work here lies between these approaches in the sense that we follow the algebraic style of Goguen and Meseguer and study how well-known categorical constructions relate to standard logic programs fixed-point semantics. Doing that, we invariantly find the *least Herbrand model* as a distinguished element in those constructions.

Among the most studied categorical constructions, there are *adjunctions* and *algebras for endofunctors* ([12, 15, 4, 5]).

Algebras for endofunctors are a translation to the categorical language and generalization to arbitrary categories of the classical notion of algebra built over a set of elements, where the role of the signature is played by functors. Equipped with a sensible notion of morphism, the algebras for a given endofunctor $\mathcal{T}$ form a category which provides the mechanism to define the concepts of fixed-point and least fixed-point for $\mathcal{T}$.

In Section 2.1, we show that, fixed a program P, the immediate consequence operator can be lifted to an endofunctor $\mathcal{T}_P$ on the category of Herbrand interpretations, in such a way that the category of algebras for such a functor is exactly the category of Herbrand models of P. Moreover, we show that the least Herbrand model of P is the least fixed-point of $\mathcal{T}_P$.

Universal and free constructions appear everywhere in Mathematics and Computer Science and the relevance of adjunctions follows exactly from the fact that they elegantly describe such situations. Particular forms of adjunctions are the *reflections*. Very informally, a reflection of a category $\mathbf{B}$ to a subcategory $\mathbf{A}$ guarantees the existence of a canonical representative in $\mathbf{A}$ for each object in $\mathbf{B}$. In Section 2.2, we show that the category of models of a program is *reflective* in the category of interpretations for that program and that the least Herbrand model is the *reflection* of the empty Herbrand interpretation.

1 Fixed-Point Semantics for Logic Programs

In this section, we briefly recall the basic definitions and results of fixed-point semantics for logic programs ([1, 14, 16], among the others) in the algebraic style of [10, 11].

1.1 Logic Programs

Syntactically, logic programs are terms of languages in which at least two different kinds of entities can be recognized: operators (or constructors) and predicates. Such a nature is faithfully taken into account by signatures with predicates.

Definition 1.1 *(Signatures with Predicates)*
A (one-sorted) signature with predicates *is a pair* $\langle \Sigma, \Pi \rangle$ *where* Σ *and* Π *are disjoint families of disjoint sets of, respectively, symbols for operations* $\{\Sigma_n \mid n \in \mathbb{N}\}$ *and symbols for predicates* $\{\Pi_n \mid n \in \mathbb{N}\}$. Σ_n *and* Π_n *contain the symbols of arity* n. ✓

Given a signature $\langle \Sigma, \Pi \rangle$ and a set of symbols for variables X, that without loss of generality we will suppose disjoint from any Σ_i, terms built up from constructors and variables represent individuals to whom predicates may be applied.

Definition 1.2 *(Terms)*
The set $T_{\Sigma,\Pi}(X)$ *of* terms *with variables* X *on the signature* $\langle \Sigma, \Pi \rangle$ *is the smallest set such that:*

(i) $\Sigma_0 \cup X \subseteq T_{\Sigma,\Pi}(X)$;

(ii) $\forall t_1, \ldots, t_n \in T_{\Sigma,\Pi}(X)$ *and* $\forall \sigma \in \Sigma_n$, $\sigma(t_1, \ldots, t_n) \in T_{\Sigma,\Pi}(X)$ ✓

Definition 1.3 *(Atoms)*
The set of atoms *with variables* X *on* (Σ, Π) *is the set of formulas*

$$B_{\Sigma,\Pi}(X) = \left\{ \rho(t_1, \ldots, t_n) \;\middle|\; \rho \in \Pi_n \text{ and } t_1, \ldots, t_n \in T_{\Sigma,\Pi}(X) \right\}.$$ ✓

Given $B \subseteq B_{\Sigma,\Pi}(X)$, we will denote by $[\![B]\!]_\rho$ the set $\left\{ (t_1, \ldots, t_n) \;\middle|\; \rho(t_1, \ldots, t_n) \in B \right\}$.

When the set of variables is the empty, the previous constructions give the set of *ground terms* $T_{\Sigma,\Pi}(\emptyset)$, denoted by $T_{\Sigma,\Pi}$ and called *Herbrand universe* for $\langle \Sigma, \Pi \rangle$, and the set of *ground atoms* $B_{\Sigma,\Pi}(\emptyset)$, denoted by $B_{\Sigma,\Pi}$ and called *Herbrand base* for $\langle \Sigma, \Pi \rangle$.

Definition 1.4 *(Horn Clauses, Goals and Programs)*
Fixed a signature (Σ, Π) *and a set of variables* X, definite clauses, goals *or* queries *and* definite programs *on* (Σ, Π) *with variables* X *are, respectively, formulas of the type* $A \leftarrow B_1, \ldots, B_n$, *formulas of the type* $\leftarrow B_1, \ldots, B_n$, *where* $A, B_1, \ldots, B_n$ *are atoms in* $B_{\Sigma,\Pi}(X)$, *and sets of definite clauses.* ✓

1.2 Interpretations

In the previous section, we have defined in a purely syntactic way what a logic program is. The first thing we need to start talking about semantics is an interpretation for the symbols which constitute the program. We will identify interpretations with the category of the algebras ([6, 7], for an excellent survey see [9]) whose signature is program's one. Let us begin by recalling the basic definitions of Category Theory ([15, 5]).

Definition 1.5 *(Graphs)*
A graph *is a structure* $(dom, cod: A \to O)$, *where* A *and* O *are classes*[1] *of, respectively, arrows and objects, and dom and cod are functions which associate to each arrow, respectively, a domain and a codomain.* ✓

Given a graph G, the class of its *composable arrows* is

$$A \times_O A = \left\{ \langle g, f \rangle \;\middle|\; g, f \in A \text{ and } dom(g) = cod(f) \right\}.$$

A category is a graph where each object has an identity arrow and arrows are closed under a given operation of composition.

[1] We shall not worry about foundational problems. We suppose to be working in a given Grothendieck universe.

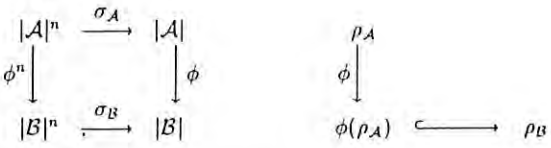

Figure 1: (Σ, Π)-homomorphisms

Definition 1.6 *(Categories)*
A category $\mathbf{C}$ *is a graph together with two additional functions*

$$id: O \to A \text{ and } \circ: A \times_O A \to A,$$

called, respectively, identity and composition, such that

$$\forall \mathcal{A} \in O,\ cod\big(id(\mathcal{A})\big) = \mathcal{A} = dom\big(id(\mathcal{A})\big),$$

$$\forall \langle g, f \rangle \in A \times_O A,\ cod\big(g \circ f\big) = cod(g) \text{ and } dom\big(g \circ f\big) = dom(f).$$

Moreover, $\circ$ *is associative and for all* $f \in A$*, given* $\mathcal{A} = dom(f)$ *and* $\mathcal{B} = cod(f)$*, we have* $f \circ id(\mathcal{A}) = f = id(\mathcal{B}) \circ f$. ✓

Usually, the arrows of a category, also called *morphisms*, are denoted by $f: \mathcal{A} \to \mathcal{B}$, where $\mathcal{A} = dom(f)$ and $\mathcal{B} = cod(f)$ and identities by $id_{\mathcal{A}}$. The class of arrows $f: \mathcal{A} \to \mathcal{B}$ in $\mathbf{C}$ is indicated as $\mathbf{C}[\mathcal{A}, \mathcal{B}]$. Moreover, in dealing with a category $\mathbf{C}$ the actual classes A and O are never mentioned: we write $\mathcal{A} \in \mathbf{C}$ for objects and f in $\mathbf{C}$ for arrows.

A *subcategory* $\mathbf{B}$ of $\mathbf{C}$ is a category whose classes of objects and arrows are contained in the respective classes of $\mathbf{C}$; $\mathbf{B}$ is *full* when for each $\mathcal{A}, \mathcal{B} \in \mathbf{B}$ we have $\mathbf{B}[\mathcal{A}, \mathcal{B}] = \mathbf{C}[\mathcal{A}, \mathcal{B}]$.

Definition 1.7 *(*$\mathbf{Alg}_{\Sigma,\Pi}$*)*
A (Σ, Π)-algebra $\mathcal{A}$ *consists of*

(i) *a set* A*, called* carrier *of the algebra and denoted by* $|\mathcal{A}|$*;*

(ii) *for each* $n \in \mathbb{N}$ *and* $\sigma \in \Sigma_n$ *an operation* $\sigma_{\mathcal{A}}: A^n \to A$*;*

(iii) *for each* $n \in \mathbb{N}$ *and* $\rho \in \Pi_n$ *a predicate* $\rho_{\mathcal{A}} \subseteq A^n$.

A (Σ, Π)-homomorphism *between the* (Σ, Π)*-algebras* $\mathcal{A}$ *and* $\mathcal{B}$ *is a function* $\phi: |\mathcal{A}| \to |\mathcal{B}|$ *which respects operations and predicates (see Figure 1), i.e., such that:*

(i) *for each* $n \in \mathbb{N}$ *and* $\sigma \in \Sigma_n$, $\phi\big(\sigma_{\mathcal{A}}(a_1, \ldots, a_n)\big) = \sigma_{\mathcal{B}}\big(\phi(a_1), \ldots, \phi(a_n)\big)$*;*

(ii) *for each* $n \in \mathbb{N}$ *and* $\rho \in \Pi_n$, $\phi^n(\rho_{\mathcal{A}}) \subseteq \rho_{\mathcal{B}}$.

where ϕ^n *is the cartesian product of* n *copies of* ϕ.

This, with the usual notion of composition for homomorphisms, gives the category $\mathbf{Alg}_{\Sigma,\Pi}$. ✓

In order to simplify notation, we will denote ϕ^n by ϕ itself. Moreover, we will use $\pi\mathcal{A}$ to indicate the set $\{\rho(e_1, \ldots, e_n) \mid (e_1, \ldots, e_n) \in \rho_{\mathcal{A}},\ \rho \in \Pi\}$ and we extend the notation $[\![\_]\!]_\rho$ to subsets of $\pi\mathcal{A}$ for $\mathcal{A} \in \mathbf{Alg}_{\Sigma,\Pi}$.

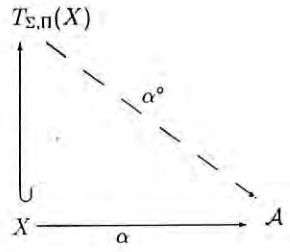

Figure 2: Free algebra on a set of generators X

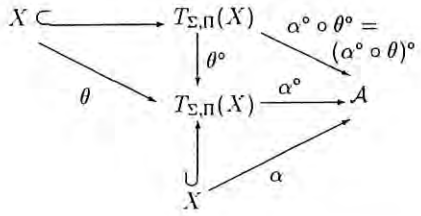

Figure 3: Composition of Substitutions

The set $T_{\Sigma,\Pi}(X)$ given in the previous section can be given the structure of a (Σ,Π)-algebra by defining operations and predicates as follows:

$$\text{for } \sigma \in \Sigma_n,\ \sigma_{T_{\Sigma,\Pi}(X)}(t_1,\ldots,t_n) = \sigma(t_1,\ldots,t_n) \quad \text{and} \quad \text{for } \rho \in \Pi,\ \rho_{T_{\Sigma,\Pi}(X)} = \emptyset.$$

The same construction makes $T_{\Sigma,\Pi}$ be a (Σ,Π)-algebra. Observe that variables in $T_{\Sigma,\Pi}(X)$ are not considered operations of arity zero, but just elements of the algebra. This allows (Σ,Π)-homomorphisms to map variables to any element of the target algebras and, therefore, to capture the notions of *substitutions* and *evaluations*.

Proposition 1.8 *($T_{\Sigma,\Pi}(X)$ is the Free Algebra on X)*
Let $\mathcal{A}$ be a (Σ,Π)-algebra. Given an assignment *of values in $\mathcal{A}$ to the variables in X, i.e., a function $\alpha: X \to \mathcal{A}$, there exists a unique (Σ,Π)-homomorphism $\alpha^\circ: T_{\Sigma,\Pi}(X) \to \mathcal{A}$ which extends α (see Figure 2), i.e., such that $(\alpha^\circ)_{|X} = \alpha$.* ✓

The (Σ,Π)-homomorphisms obtained as liftings of assignments are called *evaluation*. An assignment $\theta: X \to T_{\Sigma,\Pi}(X)$ and the correspondent θ° are what is usually called substitution.

Evaluations can be composed by composing their liftings as homomorphism. For instance, the composition of $\theta: X \to T_{\Sigma,\Pi}(X)$ and $\alpha: X \to \mathcal{A}$ gives rise to the assignment $(\alpha^\circ \circ \theta^\circ)_{|X} = \alpha^\circ \circ \theta: X \to \mathcal{A}$, as can be seen in Figure 3. In the following, with abuse of notation, we will forget any difference between α and α°. We will write αt to denote the evaluation of t under the assignment α, i.e., $\alpha^\circ(t)$. Moreover, composition of evaluations will be denoted by left juxtaposition. Therefore, for instance, we will write $\alpha\theta(t_1,\ldots,t_n)$ for

$$\big(\alpha^\circ \circ \theta^\circ(t_1),\ldots,\alpha^\circ \circ \theta^\circ(t_n)\big) = \big((\alpha^\circ \circ \theta)^\circ(t_1),\ldots,(\alpha^\circ \circ \theta)^\circ(t_n)\big).$$

We need a formal definition to establish whether a formula holds under an interpretation $\mathcal{A}$. This is the purpose of the next definition, in which $\mathcal{A}$ denotes a (Σ,Π)-algebra, α a generic assignment $T_{\Sigma,\Pi}(X) \to \mathcal{A}$, $A, B_1, \ldots, B_n$ are atoms, C ranges over clauses and P over programs.

Definition 1.9 *(Satisfaction Relation)*
The satisfaction relation $\models$ *between* (Σ,Π)*-algebras, and clauses, programs and goals is the smallest relation such that:*

$$\begin{array}{lll}
\mathcal{A} \models_\alpha \rho(t_1,\ldots,t_n) & \text{if and only if} & (\alpha t_1,\ldots,\alpha t_n) \in \rho_{\mathcal{A}};\\
\mathcal{A} \models_\alpha B_1,\ldots,B_n & \text{if and only if} & \mathcal{A} \models_\alpha B_i \text{ for } i = 1,\ldots,n;\\
\mathcal{A} \models_\alpha A \leftarrow B_1,\ldots,B_n & \text{if and only if} & \mathcal{A} \models_\alpha B_1,\ldots,B_n \Rightarrow \mathcal{A} \models_\alpha A;\\
\mathcal{A} \models A \leftarrow B_1,\ldots,B_n & \text{if and only if} & \mathcal{A} \models_\alpha A \leftarrow B_1,\ldots,B_n \text{ for each } \alpha;\\
\mathcal{A} \models P & \text{if and only if} & \mathcal{A} \models C \text{ for each } C \in P;\\
\mathcal{A} \models \leftarrow B_1,\ldots,B_n & \text{if and only if} & \mathcal{A} \models_\alpha B_1,\ldots,B_n \text{ for some } \alpha.
\end{array}$$

✓

Models are those interpretations under which the logical implications specified by the clauses in the program are all realized. Some of the facts holding in a model of P will be logical consequences of the program, while other will just depend on the nature of the particular model. Hence, the consequences of a program, i.e., its semantics, are defined to be the set of facts which hold under every model of P. That is formally stated in the next definitions.

Definition 1.10 *(Models of Programs)*
The category of the models *of a program P, denoted by* $\mathbf{Mod}(P)$*, is the full subcategory of* $\mathbf{Alg}_{\Sigma,\Pi}$ *consisting of those algebras which satisfy P, i.e., the algebras $\mathcal{M}$ such that $\mathcal{M} \models P$.* ✓

Definition 1.11 *(Logical Consequences)*
An atom A is a logical consequence *of a program P, written $P \models A$, if and only if $\mathcal{M} \models A$ for any $\mathcal{M} \in \mathbf{Mod}(P)$.* ✓

1.3 Fixed-Point Semantics

Classical Logic Programming theory is concerned with the study of different characterizations of the set of logical consequences of programs. One of the classical approaches is the so-called fixed-point semantics, in which such a set is constructed as least fixed-point of a endofunction on the set of subsets of ground atoms. In this section we recall this approach.

Definition 1.12 *(Consequence Operators)*
Given a program P, the family of the immediate consequence operators *is*

$$\mathcal{T}_P = \left\{\mathcal{T}_{\mathcal{A}} : 2^{\pi\mathcal{A}} \to 2^{\pi\mathcal{A}} \mid \mathcal{A} \in \mathbf{Alg}_{\Sigma,\Pi}\right\},$$

where $\mathcal{T}_{\mathcal{A}}$ is the function which maps $B \subseteq \pi\mathcal{A}$ to the set

$$\left\{\rho(\theta t_1,\ldots,\theta t_n) \;\middle|\; \begin{array}{l}\theta : X \to \mathcal{A},\ \rho'(\theta t'_1,\ldots,\theta t'_{n_i}) \in B,\ i = 1,\ldots,k,\\ \rho(t_1,\ldots,t_n) \leftarrow \rho(t'_1,\ldots,t'_{n_1}),\ldots,\rho(t^k_1,\ldots,t^k_{n_k}) \in P\end{array}\right\} \cup B.$$

Since $\bigcup_n \mathcal{T}^n_{\mathcal{A}}(B)$ exists for each $B \subseteq \pi\mathcal{A}$, where $\mathcal{T}^n_{\mathcal{A}}(B)$ denotes n nested applications of $\mathcal{T}_{\mathcal{A}}$ to B and $\bigcup$ is the union of sets, we can define the function $\mathcal{T}^\omega_{\mathcal{A}}$ which maps B to $\bigcup_n \mathcal{T}^n_{\mathcal{A}}(B)$ and the family of consequence operators $\mathcal{T}^\omega_P = \left\{\mathcal{T}^\omega_{\mathcal{A}} \;\middle|\; \mathcal{A} \in \mathbf{Alg}_{\Sigma,\Pi}\right\}$. ✓

Models built from term algebras are called Herbrand models. The following is a well-known fact about Herbrand models which does not hold for general models.

Proposition 1.13 *(Characterization of Herbrand Models)*
A term algebra $\mathcal{A}$ is a model of P if and only if $\mathcal{T}_{\mathcal{A}}(\pi\mathcal{A}) \subseteq \pi\mathcal{A}$. ✓

Definition 1.14 *(Least Herbrand Model)*
The least Herbrand model *of a program P is the algebra $T_{\Sigma,\Pi,P}$ obtained from $T_{\Sigma,\Pi}$ by enforcing $(t_1,\ldots,t_n) \in \rho_{T_{\Sigma,\Pi,P}}$ if and only if $\rho(t_1,\ldots,t_n) \in \mathcal{T}^{\omega}_{T_{\Sigma,\Pi}}(\emptyset)$.* ✓

The least Herbrand model is the most relevant model in that it completely characterizes the semantics of a program: the facts true in $T_{\Sigma,\Pi,P}$ are exactly the consequences of P, i.e., the facts true in every model. In this sense the least Herbrand model is universal among the models P. In the next section, we will see that it is universal also in some precise algebraic senses. The standard results concerning $T_{\Sigma,\Pi,P}$ are listed in the following propositions.

Proposition 1.15 *($T_{\Sigma,\Pi,P}$ is Universal)*
$T_{\Sigma,\Pi,P}$ is a model of P and $T_{\Sigma,\Pi,P} \models A \Leftrightarrow P \models A$. ✓

Proposition 1.16 *(Herbrand's Theorem)*
$T_{\Sigma,\Pi,P} \models \leftarrow B_1,\ldots,B_n$ if and only if $\mathcal{M} \models \leftarrow B_1,\ldots,B_n$ for each $\mathcal{M} \in \mathbf{Mod}(P)$. ✓

2 Categorical Semantics

In this section, we give two categorical characterizations of the least Herbrand model. In particular, we show that the classical notion of algebraic structure over a category can be applied to case of Logic Programming getting the category of Herbrand models from the category of Herbrand interpretations via the standard construction of *algebras for endofunctors* (see [15, 4, 5]). As a consequence, we find $T_{\Sigma,\Pi,P}$ as the least fixed-point of an endofunctor.

Moreover, we show that the category of (general) models is *reflective* in the category of interpretations and we find again the least Herbrand model as reflection of the Herbrand interpretation under which no fact holds.

We start by recalling that $T_{\Sigma,\Pi,P}$ is the initial object in $\mathbf{Mod}(P)$. For a discussion about the relevance of concept of initiality see, for instance, [9].

Definition 2.1 *(Initiality)*
An object $\mathfrak{I}$ in a category C *is* initial *in* C *if $\mathfrak{I}$ belongs to* C *and for any $c \in$ C there exists a unique morphism from $\mathfrak{I}$ to c.* ✓

It is worthwhile observing that the initial object in $\mathbf{Alg}_{\Sigma,\Pi}$ is $T_{\Sigma,\Pi}$, the free algebra over the empty set of generators.

The universality of the initial object is reflected in the following standard result from Category Theory.

Proposition 2.2 *(Uniqueness of the Initial Object)*
The initial object of a category C*, if it exists, is unique up to isomorphisms.* ✓

Proposition 2.3 *(Initiality of $T_{\Sigma,\Pi,P}$)*
$T_{\Sigma,\Pi,P}$ is initial in $\mathbf{Mod}(P)$.

Proof. Since $T_{\Sigma,\Pi,P}$, when we forget about predicates, coincides with the initial (Σ,Π)-algebra $T_{\Sigma,\Pi}$, given any $\mathcal{M} \in \mathbf{Mod}$, it exists exactly one function ϕ from $T_{\Sigma,\Pi,P}$ to $\mathcal{M}$ which respects condition *(i)* of Definition 1.7. From Proposition 1.15, it is immediate to see that ϕ respects also condition *(ii)* and that, therefore, is the unique (Σ,Π)-homomorphism from $T_{\Sigma,\Pi,P}$ to $\mathcal{M}$. ✓

2.1 $T_{\Sigma,\Pi,P}$ as Endofunctor Fixed-Point

Definition 2.4 *(Endofunctors)*
Given two categories C *and* D, *a* functor $\mathcal{F}: \mathrm{C} \to \mathrm{D}$ *is a function which maps objects in* C *to objects in* D *and morphisms in* C *to morphisms in* D *in such a way that:*

(i) $\mathcal{F}(h): \mathcal{F}(\mathcal{A}) \to \mathcal{F}(\mathcal{B})$, *for each* $h: \mathcal{A} \to \mathcal{B}$ *in* C;
(ii) $\mathcal{F}(id_{\mathcal{A}}) = id_{\mathcal{F}(\mathcal{A})}$;
(iii) $\mathcal{F}(k \circ h) = \mathcal{F}(k) \circ \mathcal{F}(h)$.

An endofunctor *is a functor from a category to the category itself.* ✓

Given a category C, the function which is the identity both on objects and arrows is clearly an endofunctor. It will be denoted by $\mathbf{1}_{\mathrm{C}}$. Composition of functors is defined in the obvious way and denoted by left juxtaposition.

Definition 2.5 *($\mathcal{H}\mathbf{Alg}_{\Sigma,\Pi}$)*
Let $\mathcal{H}\mathbf{Alg}_{\Sigma,\Pi}$ be the full subcategory of $\mathbf{Alg}_{\Sigma,\Pi}$ consisting of those algebras which are Herbrand interpretations, i.e., the (Σ,Π)-algebras of terms. ✓

In the following, we will denote by $\mathcal{H}\mathbf{Mod}(P)$ the full subcategory of $\mathcal{H}\mathbf{Alg}_{\Sigma,\Pi}$ consisting of the (Herbrand) models of P.

Now, we can see that the immediate consequence operator $\mathcal{T}_{T_{\Sigma,\Pi}}$ gives an endofunctor on the category of Herbrand interpretations.

Proposition 2.6 *(Functor $\mathcal{T}_P$)*
Given a program P on $\langle\Sigma,\Pi\rangle$, let us consider the function $\mathcal{T}_P: \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi} \to \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi}$ defined by:

(i) $\mathcal{T}_P(\mathcal{A})$ *has the same elements of* $\mathcal{A}$;
(ii) $\sigma_{\mathcal{T}_P(\mathcal{A})} = \sigma_{\mathcal{A}}$ *for each* $\sigma \in \Sigma$;
(iii) $\rho_{\mathcal{T}_P(\mathcal{A})} = [\![\mathcal{T}_{T_{\Sigma,\Pi}}(\pi\mathcal{A})]\!]_\rho$ *for each* $\rho \in \Pi$;
(iv) $\mathcal{T}_P$ *is the identity on homomorphisms.*

Then $\mathcal{T}_P$ is an endofunctor.

Proof. Properties *(ii)* and *(iii)* in Definition 2.4 obviously hold. In order to show *(i)*, since h is necessarily the identity,it is enough to show that

$$\left(\forall \rho \in \Pi,\ \rho_{\mathcal{A}} \subseteq \rho_{\mathcal{B}}\right) \Rightarrow \left(\forall \rho \in \Pi,\ \rho_{\mathcal{T}_P(\mathcal{A})} \subseteq \rho_{\mathcal{T}_P(\mathcal{B})}\right).$$

Let $(t_1,\ldots,t_n) \in \rho_{\mathcal{T}_P(\mathcal{A})}$. Then, there exist $(t_1^i,\ldots,t_{n_i}^i) \in \rho_{\mathcal{A}}^i$, $i = 1,\ldots,k$ and

$$\rho(T_1,\ldots,T_n) \leftarrow \rho^1(T_1^1,\ldots,T_{n_1}^1),\ldots,\rho^k(T_1^k,\ldots,T_{n_k}^k) \in P$$

with $\theta: X \to \mathcal{A}$ such that $\theta T_i = t_i$ and $\theta T_i^k = t_i^k$, $i = 1,\ldots,k$.
Since by hypothesis $h(\rho_{\mathcal{A}}^i) \subseteq \rho_{\mathcal{B}}^i$, we have $h\theta(T_1^i,\ldots,T_{n_i}^i) \in \rho_{\mathcal{B}}^i$, $i = 1,\ldots,k$ and, therefore, $h\theta(T_1,\ldots,T_n) = (t_1,\ldots,t_n) \in \rho_{\mathcal{T}_P(\mathcal{B})}$. ✓

Now, let us recall the basic definitions about algebras for endofunctors as stated in [5].

Definition 2.7 *($(\mathcal{T}:\mathrm{C})$)*
Given a category C *and an endofunctor* $\mathcal{T}:\mathrm{C} \to \mathrm{C}$, *a* $\mathcal{T}$-algebra *on* C *is a pair* $(\mathcal{A},a)$, *where* $\mathcal{A}$ *is an object in* C *and* $a:\mathcal{T}(\mathcal{A}) \to \mathcal{A}$ *is a morphism in* C.

A $\mathcal{T}$-homomorphism $\phi:(\mathcal{A},a) \to (\mathcal{B},b)$ *is a morphism* $\phi:\mathcal{A} \to \mathcal{B}$ *in* C *such that* $\phi \circ a = b \circ \mathcal{T}(\phi)$, *i.e., such that the following diagram commutes.*

$$\begin{array}{ccc} \mathcal{T}(\mathcal{A}) & \xrightarrow{a} & \mathcal{A} \\ {\scriptstyle \mathcal{T}(\phi)}\downarrow & & \downarrow{\scriptstyle \phi} \\ \mathcal{T}(\mathcal{B}) & \xrightarrow{b} & \mathcal{B} \end{array}$$

This defines $(\mathcal{T}:\mathrm{C})$, *the category of* $\mathcal{T}$*-algebras on* C. ✓

Definition 2.8 *(Fixed-Points)*
A fixed-point *for* $\mathcal{T}$ *is a* $\mathcal{T}$*-algebra* $(\mathcal{A},a) \in (\mathcal{T}:\mathrm{C})$ *such that* $a:\mathcal{T}(\mathcal{A}) \to \mathcal{A}$ *is an isomorphism.* ✓

Hence, a fixed-point for an endofunctor is a $\mathcal{T}$-algebra $\mathcal{A}$ such that $\mathcal{A} \cong \mathcal{T}(\mathcal{A})$. It is therefore clear as this represents a generalization of the concept of fixed-point in the categorical language, where everything is treated up to isomorphisms.

Proposition 2.9 *(lfp(T))*
If $(\mathcal{T}:\mathrm{C})$ *admits an initial object* $(\mathcal{A},a)$, *then* $(\mathcal{A},a)$ *is a fixed-point for* $\mathcal{T}$.

Proof. If $(\mathcal{A},a)$ is initial, then $\exists!\phi:(\mathcal{A},a) \to (\mathcal{T}(\mathcal{A}),\mathcal{T}(a))$ and $id_{\mathcal{A}}$ is the unique morphism from $(\mathcal{A},a)$ to itself. Then the following diagram commutes.

$$\begin{array}{ccc} \mathcal{T}(\mathcal{A}) & \xrightarrow{a} & \mathcal{A} \\ {\scriptstyle \mathcal{T}(\phi)}\downarrow & & \downarrow{\scriptstyle \phi} \\ \mathcal{T}^2(\mathcal{A}) & \xrightarrow{\mathcal{T}(a)} & \mathcal{T}(\mathcal{A}) \\ {\scriptstyle \mathcal{T}(a)}\downarrow & & \downarrow{\scriptstyle a} \\ \mathcal{T}(\mathcal{A}) & \xrightarrow{a} & \mathcal{A} \end{array}$$

The upper square commutes by definition of morphism in $(\mathcal{T}:\mathrm{C})$. Reading the whole rectangle, we have that $a \circ \phi$ is a $\mathcal{T}$-homomorphism from $(\mathcal{A},a)$ to itself. Therefore, $a \circ \phi = id_{\mathcal{A}}$ and so $\mathcal{T}(a \circ \phi) = \mathcal{T}(a) \circ \mathcal{T}(\phi) = id_{\mathcal{T}(\mathcal{A})}$. On the other hand, from the upper square we have that $\mathcal{T}(a) \circ \mathcal{T}(\phi) = \phi \circ a$, from which $\phi \circ a = id_{\mathcal{T}(\mathcal{A})}$. Hence ϕ is an isomorphism. ✓

The following definition is now completely natural. The reader can find in [5] some examples which further justify it.

Definition 2.10 *(Least Fixed-Point)*
If $(\mathcal{T}: \mathbf{C})$ has an initial object, then it is called least fixed-point *of $\mathcal{T}$.* ✓

Let us consider now $(\mathcal{T}_P: \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi})$. We will show that it has an initial object and that it coincides with $T_{\Sigma,\Pi,P}$. In other words, the (categorical) fixed-point of $\mathcal{T}_P$ is the least Herbrand model of P.

Proposition 2.11 *(Herbrand models vs $\mathcal{T}_P$-algebras, part 1)*
$\mathcal{A}$ is an Herbrand model of P if and only if $(\mathcal{A}, id_{\mathcal{A}}) \in (\mathcal{T}_P: \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi})$.

Proof. Trivially from Proposition 1.13, since both conditions are equivalent to $\rho_{\mathcal{T}_P(\mathcal{A})} \subseteq \rho_{\mathcal{A}}$ for each $\rho \in \Pi$. ✓

Proposition 2.12 *(Herbrand models vs $\mathcal{T}_P$-algebras, part 2)*
Let $(\mathcal{A}, a)$, $(\mathcal{B}, b)$ be in $(\mathcal{T}_P: \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi})$. Then φ is a (Σ,Π)-homomorphism from $\mathcal{A}$ to $\mathcal{B}$ if and only if it is a $\mathcal{T}_P$-homomorphism from $(\mathcal{A}, a)$ to $(\mathcal{B}, b)$.

Proof. ($\Leftarrow$). Obvious.
($\Rightarrow$). It is evident that the diagram in Definition 2.7 commutes, because $\mathcal{T}_P(\mathcal{A})$ and $\mathcal{A}$ are term algebras obtained from $T_{\Sigma,\Pi,P}$, and therefore the unique (Σ,Π)-homomorphism between them, if any, is the identity on terms. ✓

Two categories $\mathbf{C}$ and $\mathbf{D}$ are said isomorphic if there exists a pair of functors $\mathcal{F}: \mathbf{C} \to \mathbf{D}$ and $\mathcal{G}: \mathbf{D} \to \mathbf{C}$ such that $\mathcal{G}\mathcal{F} = \mathbf{1}_{\mathbf{C}}$ and $\mathcal{F}\mathcal{G} = \mathbf{1}_{\mathbf{D}}$. Clearly, isomorphic categories are essentially the same.

Corollary 2.13 *($\mathcal{T}_P$-algebras are models)*
$(\mathcal{T}_P: \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi}) \cong \mathcal{H}\mathbf{Mod}(P)$.

Proof. It follows immediately from Proposition 2.12 that the functor $\mathcal{F}: \mathcal{H}\mathbf{Mod}(P) \to (\mathcal{T}_P: \mathcal{H}\mathbf{Alg}_{\Sigma,\Pi})$ which sends $\mathcal{M}$ in $(\mathcal{M}, id_{\mathcal{M}})$ and is the identity on morphisms is an isomorphism of categories. ✓

Since isomorphisms of categories in particular are isomorphisms of classes of arrows between correspondent objects, it is immediate to see that they send initial objects to initial objects. The next corollary follows from this observation.

Corollary 2.14 *($T_{\Sigma,\Pi,P}$ is $lfp(\mathcal{T}_P)$)*
$T_{\Sigma,\Pi,P}$ is the least fixed-point of $\mathcal{T}_P$. ✓

2.2 Models are reflective in Interpretations

In this section, we present a characterization of the least Herbrand model based on an *adjunction* between interpretations and models.

Adjunctions, introduced in [12], provide an elegant way to formulate properties of free objects and universal constructions.

Definition 2.15 *(Adjunctions)*
An adjunction *from* $\mathbf{C}$ *to* $\mathbf{D}$ *is a triple* $\langle \mathcal{F}, \mathcal{G}, \varphi \rangle$: $\mathbf{C} \rightharpoonup \mathbf{D}$, *where* $\mathcal{F}$: $\mathbf{C} \to \mathbf{D}$ *and* $\mathcal{G}$: $\mathbf{D} \to \mathbf{C}$ *are functors and* φ *is a function which assigns to each pair of objects* $\mathcal{A} \in \mathbf{C}$ *and* $\mathcal{B} \in \mathbf{D}$ *a bijection*

$$\varphi_{\mathcal{A},\mathcal{B}}: \mathbf{D}[\mathcal{F}(\mathcal{A}), \mathcal{B}] \cong \mathbf{C}[\mathcal{A}, \mathcal{G}(\mathcal{B})],$$

which is natural both in $\mathcal{A}$ *and* $\mathcal{B}$, *i.e., such that for all* k: $\mathcal{A}' \to \mathcal{A}$ *and* h: $\mathcal{B} \to \mathcal{B}'$ *the following diagrams commute.*

$$\begin{array}{ccc} \mathbf{D}[\mathcal{F}(\mathcal{A}), \mathcal{B}] & \xrightarrow{\varphi_{\mathcal{A},\mathcal{B}}} & \mathbf{C}[\mathcal{A}, \mathcal{G}(\mathcal{B})] \\ {\scriptstyle - \circ \mathcal{F}(k)}\downarrow & & \downarrow{\scriptstyle - \circ k} \\ \mathbf{D}[\mathcal{F}(\mathcal{A}'), \mathcal{B}] & \xrightarrow{\varphi_{\mathcal{A}',\mathcal{B}}} & \mathbf{C}[\mathcal{A}', \mathcal{G}(\mathcal{B})] \end{array} \qquad \begin{array}{ccc} D[\mathcal{F}(\mathcal{A}), \mathcal{B}] & \xrightarrow{\varphi_{\mathcal{A},\mathcal{B}}} & \mathbf{C}[\mathcal{A}, \mathcal{G}(\mathcal{B})] \\ {\scriptstyle h \circ -}\downarrow & & \downarrow{\scriptstyle \mathcal{G}(h) \circ -} \\ \mathbf{D}[\mathcal{F}(\mathcal{A}), \mathcal{B}'] & \xrightarrow{\varphi_{\mathcal{A},\mathcal{B}'}} & \mathbf{C}[\mathcal{A}, \mathcal{G}(\mathcal{B}')] \end{array}$$ ✓

If $\langle \mathcal{F}, \mathcal{G}, \varphi \rangle$: $\mathbf{C} \rightharpoonup \mathbf{D}$ is an adjunction, $\mathcal{F}$ is called *left adjoint* to $\mathcal{G}$ and, viceversa, $\mathcal{G}$ is called *right adjoint* to $\mathcal{F}$.

A well-know result in Category Theory is that left adjoints preserve initiality, i.e., they map initial objects to initial objects.

Definition 2.16 *(Reflections)*
A subcategory $\mathbf{C}$ *of* $\mathbf{D}$ *is said* reflective *in* $\mathbf{D}$ *if there exists a left adjoint, said* reflector *of* $\mathbf{D}$ *in* $\mathbf{C}$, *for the inclusion functor* $\mathbf{C} \hookrightarrow \mathbf{D}$. ✓

In the following, we will show the family of functions $\mathcal{T}_P^\omega$, defined in Section 1.3, defines a reflector of $\mathbf{Alg}_{\Sigma,\Pi}$ in $\mathbf{Mod}(P)$. Informally speaking, in our setting it means that any interpretation can be "completed" to be a model in a universal way, i.e., that any interpretation has a canonical representative—its reflection—in the category of models. The reader is referred to [15] for further considerations on the relevance the concept of reflection. By exploiting standard results from Category Theory, we show that the $T_{\Sigma,\Pi,P}$ is the canonical object linked by the adjunction to the empty Herbrand interpretation, i.e., $T_{\Sigma,\Pi}$.

Definition 2.17 *(Functor $\mathcal{T}_P^\omega$)*
Given a program P *on* $\langle \Sigma, \Pi \rangle$, *let us consider the function* $\mathcal{T}_P^\omega$: $\mathbf{Alg}_{\Sigma,\Pi} \to \mathbf{Alg}_{\Sigma,\Pi}$ *defined by:*

(i) $\mathcal{T}_P^\omega(\mathcal{A})$ *has the same elements of* $\mathcal{A}$*:*

(ii) $\sigma_{\mathcal{T}_P^\omega(\mathcal{A})} = \sigma_{\mathcal{A}}$ *for each* $\sigma \in \Sigma$;

(iii) $\rho_{\mathcal{T}_P^\omega(\mathcal{A})} = [\![\mathcal{T}_{\mathcal{A}}^\omega(\pi\mathcal{A})]\!]_\rho$ *for each* $\rho \in \Pi$;

(iv) $\mathcal{T}_P^\omega$ *is the identity on homomorphisms.* ✓

Proposition 2.18 *($\mathcal{T}_P^\omega$: $\mathbf{Alg}_{\Sigma,\Pi} \to \mathbf{Mod}(P)$)*
For each $\mathcal{A} \in \mathbf{Alg}_{\Sigma,\Pi}$, $\mathcal{T}_P^\omega(\mathcal{A})$ *is a model. Moreover,* ϕ: $\mathcal{A} \to \mathcal{B}$ *in* $\mathbf{Alg}_{\Sigma,\Pi}$ *if and only if* ϕ: $\mathcal{T}_P^\omega(\mathcal{A}) \to \mathcal{T}_P^\omega(\mathcal{B})$ *is in* $\mathbf{Mod}(P)$. ✓

Proof. Obviously, since $\mathcal{T}_{\mathcal{A}}^\omega(\pi\mathcal{A}) = \mathcal{T}_{\mathcal{A}}^\omega(\mathcal{T}_{\mathcal{A}}^\omega(\pi\mathcal{A}))$, we have that $\mathcal{T}_P^\omega(\mathcal{A}) = \mathcal{T}_P^{(\omega)}(\mathcal{T}_P^{(\omega)}(\mathcal{A}))$ which, therefore, is a model.

Concerning the claim about morphisms, the left implication is clearly true. In order to show the converse implication, suppose now that $(e_1, \ldots, e_n) \in \rho_{\mathcal{T}_P^\omega(\mathcal{A})}$. We show by induction on the least $k \in \mathbb{N}$ such that $\rho(e_1, \ldots, e_n) \in \mathcal{T}_{\mathcal{A}}^k(\pi\mathcal{A})$ that $\phi(e_1, \ldots, e_n) \in \rho_{\mathcal{T}_P^\omega(\mathcal{B})}$.

Induction base. In this case we have that $\rho(e_1, \ldots, e_n) \in \pi\mathcal{A}$.
Then we have that $\phi(e_1, \ldots, e_n) \in \rho_{\mathcal{T}_P^\omega(\mathcal{B})}$, since ϕ is a (Σ, Π)-homomorphism and $\pi\mathcal{B} \subseteq \pi\mathcal{T}_P^\omega(\mathcal{B})$.

Inductive step. There exists $k \in \mathbb{N} \setminus \{0\}$ such that for some $\theta: X \to \mathcal{A}$ and for some $\rho(t_1, \ldots, t_n) \leftarrow \rho^1(t_1^1, \ldots, t_{n_1}^1), \ldots, \rho^h(t_1^h, \ldots, t_{n_h}^h) \in P$, we have that $\theta t_i = e_i$ for $i = 1, \ldots, n$ and $\rho^1(t_1^1, \ldots, t_{n_1}^1), \ldots, \rho^h(t_1^h, \ldots, t_{n_h}^h) \in \mathcal{T}_{\mathcal{A}}^{(k-1)}(\pi\mathcal{A})$. Therefore, by a straightforward application of the inductive hypothesis, the proof is concluded. ✓

As a corollary to the previous proposition we have that $\mathcal{T}_P^{(\omega)}: \mathbf{Alg}_{\Sigma,\Pi} \to \mathbf{Mod}(P)$ is a functor.

Proposition 2.19 *($\langle \mathcal{T}_P^\omega, \hookrightarrow \rangle: \mathbf{Alg}_{\Sigma,\Pi} - \mathbf{Mod}(P)$)*
$\mathcal{T}_P^{(\omega)}$ is the left adjoint to the inclusion functor of $\mathbf{Mod}(P)$ *in* $\mathbf{Alg}_{\Sigma,\Pi}$.

Proof. Given $\mathcal{A} \in \mathbf{Alg}_{\Sigma,\Pi}$ and $\mathcal{M} \in \mathbf{Mod}(P)$ we have that $\phi: \mathcal{A} \to \mathcal{M}$ if and only if $\phi: \mathcal{T}_P^{(\omega)}(\mathcal{A}) \to \mathcal{T}_P^{(\omega)}(\mathcal{M}) = \mathcal{M}$. Therefore the natural isomorphism we are seeking is the identity $\mathbf{Alg}_{\Sigma,\Pi}[\mathcal{A}, \mathcal{M}] = \mathbf{Mod}(P)[\mathcal{T}_P^\omega(\mathcal{A}), \mathcal{T}_P^\omega(\mathcal{M})] = \mathbf{Mod}(P)[\mathcal{T}_P^\omega(\mathcal{A}), \mathcal{M}]$. ✓

Finally, we find again the least Herbrand model as the image of the initial (Σ, Π)-algebra via $\mathcal{T}_P^\omega$.

Corollary 2.20 *($T_{\Sigma,\Pi,P}$ is the reflection of $T_{\Sigma,\Pi}$)*
$T_{\Sigma,\Pi,P} = \mathcal{T}_P^{(\omega)}(T_{\Sigma,\Pi})$.

Proof. $T_{\Sigma,\Pi}$ is initial in $\mathbf{Alg}_{\Sigma,\Pi}$ and $\mathcal{T}_P^\omega$ preserves initiality. ✓

ACKNOWLEDGMENTS. *I wish to thank Roberto Di Meglio and Maurizio Gabbrielli for their valuable comments. Giorgio Levi deserves special thanks for his encouragement to write this note.*

References

[1] K.R. Apt. Introduction to Logic Programming. In *Handbook of Theoretical Computer Science*, vol. B, Elsevier and MIT Press, 1990.

[2] A. Asperti and A. Corradini. A categorical model for logic programs: indexed monoidal categories. In *Proceedings of REX '92*. To appear in the LNCS series, Springer-Verlag, 1993.

[3] A. Asperti and S. Martini. Projections instead of Variables, A Category Theoretic Interpretation of Logic Programs. In *Proceedings of the 6th International Conference on Logic Programming*, 1989.

[4] M. Barr and C. Wells. *Toposes, Triples and Theories*, Springer-Verlag, 1985.

[5] M. Barr and C. Wells. *Category Theory for Computing Science*, Prentice-Hall, 1990.

[6] G. Birkhoff. On the Structure of Abstract Algebras, in *Proceedings of the Cambridge Philosophical Society*, n. 31, 1935.

[7] G. Birkhoff and J. Lipson. Heterogeneous Algebras. *Journal of Combinatorial Theory*, n. 8, 1970.

[8] A. Corradini and U. Montanari. An Algebraic Semantics for Structured Transitions Systems and its application to Logic Programming. *Theoretical Computer Science*, n. 103, 1992.

[9] J. Goguen and J. Meseguer. Initiality, Induction, and Computability, in *Algebraic Methods in Semantics*, M. Nivat and J. Reynolds, Eds., Cambridge University Press, 1985.

[10] J. Goguen and J. Meseguer. Eqlog: Equality, Types, and Generic Modules for Logic Programming, in *Functional and Logic Programming*, Douglas DeGroot and Gary Lindstrom, Eds., Prentice-Hall, 1986.

[11] J. Goguen and J. Meseguer. Models and Equality for Logiçal Programming, in *Proceedings of TAPSOFT '87*, LNCS 250, Springer-Verlag, 1987.

[12] D.M. Kan. Adjoint Functors. *Transactions of the American Mathematical Society*, n. 87, 1958.

[13] R.A. Kowalski. Predicate Logic as a Programming Language, in *Proceedings of IFIP '74*, North-Holland, 1974.

[14] J.W. Lloyd. *Foundations of Logic Programming*, Second Edition, Springer-Verlag, 1987.

[15] S. MacLane. *Categories for the Working Mathematician*, Graduate Text in Mathematics, Springer-Verlag, 1971.

[16] M.J. Maher. *Semantics of Logic Programs*, Technical Report 85/14, Department of Computer Science, University of Melbourne, Australia. 1985.

[17] J. Meseguer. General Logics, in *Logical Colloquium '87*, North-Holland, 1987.

Modular Logic Programs over Finite Domains

Maurizio Gabbrielli Roberto Giacobazzi Danilo Montesi

CWI,
P.O. Box 4079,
1009 AB Amsterdam,
The Netherlands.
gabbri@cwi.nl

Dipartimento di Informatica
Università di Pisa
Corso Italia 40, 56125 Pisa
Italy
(giaco,montesi)@di.unipi.it

Abstract

In this paper we study the properties of a compositional semantics for logic programs and its applications to modular analysis and logic-based databases, focusing on programs defined on a finite domain (i.e. on a function free finite signature). By this restriction we obtain a finite characterization of the compositional semantics which has the same correctness and compositionality properties of the original (fixpoint-based) definition, even if the latter could introduce infinite denotations also for finite domains programs. This property is particularly suitable for semantics-based program analysis, since it allows to identify a class of abstract domains for which the same finite characterization of the (abstract) compositional semantics holds. By combining this result with an existing framework for the modular analysis of logic programs, we can obtain a compositional analysis without requiring an additional level of abstraction. The finiteness of our characterization can also be useful for the modular construction and the uniform compilation of Datalog programs.

1 Introduction

Module systems and data abstraction are powerful methods for managing the complexity of large programs. Recently, a new semantics for modular logic programs has been introduced in [5]. The essential idea is to enhance the semantics proposed in [14] by allowing clauses in the semantic domain. The meaning of a module is given by the least fixpoint $\mathcal{F}_\Omega(P)$ of a suitable operator T_P^Ω. This provides a semantics which is compositional wrt union of modules, still maintaining the simplicity of the standard T_P-based semantics introduced by van Emden and Kowalski. From the point of view of applications, a drawback of such a fixpoint semantics is that since clauses are used as semantics objects, $\mathcal{F}_\Omega(P)$ can be an infinite set even if considering programs defined over a *finite domain* (i.e. a function-free finite signature).

In this paper we notice that this restriction allows to obtain a *finite* characterization of the compositional semantics of a program. This is obtained by taking as semantics of P the result of a finite number of iterations of the operator T_P^Ω, without reaching the fixpoint. While this would not be correct in general, we show that for finite domain programs the resulting denotation has the same compositionality and correctness properties of the fixpoint

semantics. This is particularly suitable for semantics-based program analysis and logic-based databases. In fact, when considering abstract interpretation based on $\mathcal{F}_\Omega(P)$, there are two independent dimensions along which we need finite descriptions in the abstract domain. The first one is the usual finite abstraction on the (possibly) infinite set of descriptions of the concrete computations which depends on the particular property analyzed. For the case of logic programs, concrete observables can be expressed by (infinite sets of) substitutions and their abstraction is provided by the standard approaches to logic program analysis ([8,3,6,12]). The second one is a finite description of sequences of atoms. Indeed, even if the set of abstract substitutions is finite, the abstract version of the compositional fixpoint semantics may introduce arbitrary large clauses in the abstract semantics, so that the analysis may not terminate. In [9] this problem has been tackled by introducing an additional layer of abstraction which is obtained by applying fixpont acceleration techniques such as the so called $\star$-*abstraction*. This is applied to provide finitary descriptions for arbitrary large clauses, thus introducing a further approximation which make the analysis less precise. While this is needed to handle abstractions over generic (possibly infinite) abstract domains, we will show that there exists a wide class of abstract domains for which a finite description of the semantics is obtainable without a further level of abstraction. This corresponds precisely to consider compositional semantics over finite domains.

The finite characterization of the compositional semantics can be useful also for Datalog programs. A Datalog program ([7]) is a particular logic program over a function-free signature which consists of two components: a set of Horn clauses (the *intensional database*) which are used to derive new facts and the *extensional database* storing the information in terms of ground unit clauses. Compilation techniques are often used to optimize the *query answering process*. Since there are no uniform compilation methods, usually the intensional database is partitioned into syntactic classes of clauses (e.g. linear, regular, etc.) and each class is compiled by using a specific technique. The compositional semantics $\mathcal{F}_\Omega(P)$ provides a way to translate a recursive program into a recursion-free one. Our results give a finite characterization of this process for Datalog programs (on a finite domain), thus providing a *uniform* modular compilation technique for intensional databases. The mentioned compositionality furnishes also the semantic base to develop deductive databases in an incremental and modular way.

The remaining of this paper is organized as follows. After some preliminaries, in the next section we introduce the compositional semantics which we are referring to. Section 3 shows the formal results that set up our finite characterization for the semantics and the related correctness and compositionality properties. In section 4 we first apply these results to static program analysis, by identifying a suitable class of abstract domains which allows the applicability of the previous results to the abstract semantics. Then we consider the case of Datalog programs.

2 Preliminaries

In the following we assume familiarity with the standard definitions and notation for logic programs. The standard reference works by Apt [1] and Lloyd [22] provide the necessary background material.

Throughout we denote by (Π, Σ, Var) a first-order language defined over a signature Σ (a set of function symbols with their arity), a set of predicate symbols Π and a denumerable set of variables Var. The non-ground term algebra over Σ and Var is denoted by $Term(\Sigma, Var)$ or $Term$ for short. The set of atoms constructed from predicate symbols in Π and terms from

Term is denoted by $\mathcal{A}^{\Pi}(\Sigma, Var)$ or $\mathcal{A}^{\Pi}$ for short. A goal is a sequence of atoms. The *length* of a sequence b is denoted by $|b|$. A clause is an object of the form $h : -B$ where h is an atom, called the *head*, and B is a goal, called the *body*. A clause with an empty body is a *unit clause*. The set of clauses constructed from elements of $\mathcal{A}^{\Pi}(\Sigma, Var)$ is denoted by $\mathcal{C}(\Pi, \Sigma, Var)$ or $\mathcal{C}$ for short. We will abuse by considering unit clauses as atoms. A logic program P is a finite set of clauses. A *substitution* ϑ is a mapping from *Var* to *Term* which acts as the identity almost everywhere, i.e. such that its domain $dom(\vartheta) = \{x \in Var \mid \vartheta(x) \neq x\}$ is finite. It extends to apply to any syntactic object (e.g. terms, atoms, clauses, *etc.*) in the usual way. The set $range(\vartheta)$ is defined as $range(\vartheta) = \cup_{x \in dom(\vartheta)} var(\vartheta(x))$. We will use the set theoretic notation $\{ x \mapsto t \mid x \in dom(\theta),\ \theta(x) = t \}$ to represent θ. Given a syntactic object s, we denote by $\vartheta_{|s}$ the restriction of ϑ to the variables of s and by $S\theta$ the application of ϑ to s. Composition of substitutions ρ and σ is defined as usual and denoted by $\rho\sigma$. The set of substitutions is denoted by *Sub*. We fix a partial function *mgu* which maps a pair of syntactic objects to an idempotent most general unifier of the objects, if such exists. Thus, a statement $\theta = mgu(s, t)$ implies that s and t are unifiable. A *variable renaming* is a substitution that is a bijection on *Var*. Syntactic objects t_1 and t_2 are *equivalent up to renaming*, denoted by $t_1 \sim t_2$, if, for some variable renaming ρ, $t_1\rho = t_2$. Moreover we denote by $pred(t)$ the set of predicate symbols occurring in the syntactic object t. The set of variables that occur in a syntactic object t is denoted by $var(t)$. If G is a goal, $G \overset{\vartheta}{\leadsto}_P \square$ denotes a SLD refutation of G in the program P with computed answer ϑ. A *parallel derivation* for a goal G in a program P is obtained by simultaneously replacing each atom in G with the body of clauses in P, provided that G and the corresponding clause heads are unifiable. The notion of ordinal powers of a function f on a complete lattice L is defined as usual, namely $f \uparrow 0 = \bot$, $f \uparrow (n+1) = f(f \uparrow n)$ and $f \uparrow \omega = \sqcup_{n<\omega} f \uparrow n$ where $\bot$ is the bottom and $\sqcup$ is the *lub* of the lattice. The notations $\tilde{t}$, $\tilde{x}$ will be used to denote tuples of terms and variables respectively, while $\tilde{B}$ denotes a (possibly empty) conjunction of atoms. Moreover, we denote the power set of X by $\wp(X)$. Finally, to specify function parameters, we often make use of Church's lambda notation.

2.1 Semantics for Composition

In this section we will introduce a semantics which is compositional wrt the union of (logic) programs. Compositionality has to do with a (syntactic) program composition operator $\circ$, and holds when the semantics of the compound construct $C_1 \circ C_2$ is defined by (semantically) composing the semantics of the constituents C_1 and C_2. In the case of logic programs, the problematic syntactic construct is the *union* of clauses. Indeed, it is easy to see that the usual semantics based on atoms, such as the least Herbrand model and the *s-semantics* [14] are not compositional wrt $\cup$. For example, for $P = \{p(X) : -q(X),\ r(a)\}$ and $Q = \{q(a)\}$, the least Herbrand model $M(P \cup Q) = \{p(a), q(a), r(a)\}$ of the union cannot be obtained from $M(P) = \{r(a)\}$ and $M(Q) = \{q(a)\}$.

This kind of compositionality is a desirable property since it allows an incremental and modular definition of programs and of their semantics, and, as shown in [9], it provides a semantic base for modular program analysis. In the case of logic languages, a typical partially defined program could be a program where the intensional definitions are completely known while extensional definitions are only partially known and can be further specified by adding new clauses. Partially defined programs have been called Ω-*open* programs. An Ω-open program (open program for short) is a logic program P together with a set Ω of predicate symbols. A predicate symbol occurring in Ω is considered to be only partially defined in P and can be further specified by composing P with other programs. As specified by the following definition, the composition of open programs is simply union of clauses modified in

order to take into account the "interface" described by the set Ω.

Definition 2.1 (Ω-union) *[5] Let P_1 be an Ω_1-program and P_2 be an Ω_2-program. If $\Omega \subseteq \Omega_1 \cup \Omega_2$ and $(pred(P_1) \cap pred(P_2)) \subseteq (\Omega_1 \cap \Omega_2)$ then $P_1 \cup_\Omega P_2$ is the Ω-open program $P_1 \cup P_2$. Otherwise $P_1 \cup_\Omega P_2$ is not defined.*

A semantics which is compositional wrt the $\cup_\Omega$ operator is obtained in [5] by using a domain based on clauses. This semantics is formalized in terms of unfolding of clauses. The unfolding operator *unf* specifies the result of unfolding clauses from a program P_1 with clauses from a program P_2.

Definition 2.2 (Unfolding)
The unfolding operator $unf : \wp(C) \times \wp(C) \to \wp(C)$ is defined as

$$unf_P(Q) = \left\{ \hat{c} \;\middle|\; \begin{array}{l} \hat{c} = (h : -\tilde{b}_1, \ \cdots \ , \tilde{b}_n)\sigma, \\ c = h : -g_1, \ldots, g_n \in P, \ (n \geq 0), \ h_1 : -\tilde{b}_1, \ldots, h_n : -\tilde{b}_n \in Q \\ \textit{renamed apart and } \sigma = mgu(\langle g_1, \ldots, g_n \rangle, \langle h_1, \ldots, h_n \rangle) \end{array} \right\}.$$

With abuse of notation, in the following we denote by $\sim$ also the equivalence on clauses defined by $H : -A_1, \ldots, A_n \sim K : -B_1, \ldots, B_n$ iff there exists a permutation $\{i_1, \ldots, i_n\}$ such that $H : -A_1, \ldots, A_n$ and $K : -B_{i_1}, \ldots, B_{i_n}$ are variant. Moreover C is also the set of the $\sim$ equivalence classes of clauses (on the given signature) and, given a set of predicates Ω, we define $C^\Omega = \{H : -\tilde{B} \in C \mid pred(\tilde{B}) \subseteq \Omega\}$ and $\mathcal{A}^\Omega = \{A \mid A \in \mathcal{A}^\Pi \text{ and } pred(A) \in \Omega\}$.

Definition 2.3 (Interpretations) *[5]*
An interpretation *is any element in $\wp(C)$.*

Remark
To simplify the notation we will denote both a clause c and its $\sim$ equivalence class by c and we will consider a program P both as a set of clauses and as an interpretation. We assume that, when considering a syntactic element of an $\sim$ equivalence class, such an element is implicitly renamed apart, i.e. it does not share any variable with any other expression in the given context. Moreover, given any syntactic operator f on sets of clauses, f will denote also the operator on sets of $\sim$ equivalence classes obtained in the obvious way. This is correct since all the syntactic operators use clauses which are renamed apart.

Note that, according to previous remark, *unf* denotes also a binary (semantic) operator on interpretations. The unfolding operator is of interest as it can be applied to formalize both top-down and bottom-up semantics for logic programs [21]. Below we show the bottom-up construction of the semantics for the open-programs. For a set of predicate symbols Ω we denote by id_Ω the set $\{p(\tilde{x}) : -p(\tilde{x}) \in C \mid p \in \Omega\}$.

Definition 2.4 (Fixpoint semantics [5])
The fixpoint semantics of an Ω-open program P is given by the function $\mathcal{F}_\Omega : \wp(C) \to \wp(C)$, defined as $\mathcal{F}_\Omega(P) = lfp(T_P^\Omega)$, where $T_P^\Omega : \wp(C) \to \wp(C)$ is defined as $T_P^\Omega(I) = unf_P(I \cup id_\Omega)$.

By T_P^Ω continuity we have $lfp(T_P^\Omega) = T_P^\Omega \uparrow \omega$. In order to obtain a compositional semantics where open predicates in Ω can be defined from outside P, the T_P^Ω operator generates interpretations which contain clauses. Technically, this is accomplished by letting id_Ω to add tautological clauses for the open predicates and by unfolding all predicates in the body of a clause. Notice that the unfolding of a predicate with a tautological clause is basically a

"no-op" and that the clauses in $\mathcal{F}_\Omega(P)$ have bodies with only predicates in Ω. Moreover, note that for $\Omega = \emptyset$, since $id_\emptyset$ is empty, $T_P^\emptyset(I) = unf_P(I)$. Therefore when I consists of unit clauses (facts), $T_P^\emptyset$ is exactly the (non-ground) immediate consequence operator whose least fixpoint $\mathcal{F}_\emptyset(P)$ is the s-semantics of P ([14]).

Considering *computed answers* as "observable" and $\cup_\Omega$ as program composition operator, we obtain an "observational congruence" on programs which can be defined as follows.

Definition 2.5 *Let P_1, P_2 be Ω-open programs. Let G be a goal and Q be a program such that $P_i \cup_\Omega Q$ is defined, for $i = 1, 2$. Then we define $P_1 \approx_\Omega P_2$ iff for any G, Q we have that $G \overset{\vartheta}{\leadsto}_{P_1 \cup_\Omega Q} \square$ iff $G \overset{\vartheta'}{\leadsto}_{P_2 \cup_\Omega Q} \square$ and $G\vartheta \sim G\vartheta'$.*

$\mathcal{F}_\Omega$ is proved to be compositional wrt $\cup_\Omega$ and correct wrt $\approx_\Omega$. In other words $\mathcal{F}_\Omega(P)$ contains the information necessary to describe the behaviour of P in terms of computed answers, compositionally wrt union of programs. Moreover, each successful derivation for the goal G in $P \cup_\Omega Q$, where $Q \subseteq \mathcal{A}^\Omega$, can equivalently be performed by using parallel derivation steps in $\mathcal{F}_\Omega(P \cup_\Omega Q)$. The following are the formal results.

Theorem 2.6 (Compositionality [5])
Let P_1 be an Ω_1-open program and P_2 be an Ω_2-open program such that, for a set of predicates Ω, $P_1 \cup_\Omega P_2$ is defined. Then $\mathcal{F}_\Omega(P_1 \cup_\Omega P_2) = \mathcal{F}_\Omega(\mathcal{F}_{\Omega_1}(P_1) \cup_\Omega \mathcal{F}_{\Omega_2}(P_2))$.

Theorem 2.7 (Correctness [5])
Let P, P' be Ω-open programs. Then

1. *$P \approx_\Omega \mathcal{F}_\Omega(P)$.*
2. *if $\mathcal{F}_\Omega(P) = \mathcal{F}_\Omega(P')$ then $P \approx_\Omega P'$*

Moreover for $Q \subseteq \mathcal{A}^\Omega$ and for any goal $G = A_1, \ldots, A_n$, $G \overset{\vartheta}{\leadsto}_{P \cup_\Omega Q} \square$ iff there exists $B_1, \ldots, B_n \in T^\vartheta_{\mathcal{F}_\Omega(P)}(Q) \cup Q$ such that $\gamma = mgu((A_1, \ldots, A_n)(B_1, \ldots, B_n))$ and $\gamma_{|G} = \vartheta_{|G}$.

For the particular case of $\Omega = \emptyset$ we have the following version of previous theorem which shows the full abstraction of the s-semantics.

Theorem 2.8 *[14] Let P_1, P_2 be programs. Then $P_1 \approx_\emptyset P_2$ iff $\mathcal{F}_\emptyset(P_1) = \mathcal{F}_\emptyset(P_2)$*

Finally note that, as shown by the following theorem (which generalize a theorem given in [17] for the case $\Omega = \Pi$), the composition $\cup_\Omega$ of two Ω-open programs induce the same equivalence $\approx_\Omega$ on programs as the union of an Ω-open program with set of atoms whose predicates are in Ω.

Theorem 2.9 *[16] Let P_1, P_2 be programs. Then $P_1 \not\approx_\Omega P_2$ iff there exists a set of atoms $Q \subseteq \mathcal{A}^\Omega$ such that $P_1 \cup Q \not\approx_\emptyset P_2 \cup Q$.*

3 Compositional Semantics over Finite Domains

In this section we will always consider programs and goals defined on a *finite signature* Σ which contains only constant symbols. We will equivalently refer to this assumption by saying that we consider programs on a *finite domain*.

Given an Ω-open program P, we define $P_n = T_P^\Omega \uparrow n$. P_n can be considered a kind of *partially* correct semantics of P. Indeed, when considering a signature Σ containing a function

symbol (or infinite constant symbols), it is always possible to find a program P such that, for any finite n, $P \not\approx_\Omega P_n$. This is not the case if we consider a finite domain. In fact with this assumption we can obtain an $n^* < \omega$ such that the finite denotation $P_{n^*} = T_P^\Omega \uparrow n^*$ has the properties stated for $\mathcal{F}_\Omega$ in Theorem 2.7. Even if P_{n^*} is not a fixpoint of T_P^Ω (wrt set inclusion), P_{n^*} can be considered the *finite* compositional semantics of the program. Note that even for finite domains $\mathcal{F}_\Omega(P)$ may be infinite. For example, for the Ω-program $P = \{p, p : -p, q\}$, with $\Omega = \{q\}$, we obtain the infinite semantics $\mathcal{F}_\Omega(P) = \{p : -q,\ p : -q, q,\ p : -q, q, q \ldots\}$. In the following we show that P_{n^*} is the first interpretation in the chain $\{P_n\}_{n<\omega}$ such that, for any $Q \subseteq \mathcal{A}^\Omega$, $(T^\emptyset_{P_{n^*}} + id)(Q) = (T^\emptyset_{P_{n^*+1}} + id)(Q)$. This property is shared by all of the partial interpretations P_n in the chain, for $n \geq n^*$, including $lub(\{P_n\}_{n<\omega})$ (which is precisely the $\mathcal{F}_\Omega(P)$ semantics). This implies (essentially because of theorem 2.9) that we have obtained the same compositional behaviour as P. In order to formally prove these statements, let us first define the notion of T-stable partial semantics P_n, and let show that there exists a T-stable P_n for any program P.

Definition 3.1 *Let P be an Ω-open program. We say that P_n is T-stable iff for any $Q \subseteq \mathcal{A}^\Omega$, $(T^\emptyset_{P_n} + id)(Q) = (T^\emptyset_{P_{n+1}} + id)(Q)$.*

Proposition 3.2
Let P be an Ω-open program. There exists $n < \omega$ such that P_n is T-stable.

Lemma 3.3 shows that, as one would expect, when ascending in the chain $\{P_n\}_{n<\omega}$, T-stability is always preserved. The proof can be carried out easily by using the definition $T_P^\Omega(Q) = unf_P(Q \cup Id_\Omega)$ and by the associativity of the unfolding.

Lemma 3.3
Let P be an Ω-open program and let $Q \subseteq \mathcal{A}^\Omega$. If $(T^\emptyset_{P_n} + id)(Q) = (T^\emptyset_{P_{n+1}} + id)(Q)$ then for any $k \geq n$ and for any $Q \subseteq \mathcal{A}^\Omega$, $(T^\emptyset_{P_n} + id)(Q) = (T^\emptyset_{P_k} + id)(Q)$.

Remark
Note that when considering programs on a finite domain, the set of (variance) equivalence classes of atoms $\mathcal{A}^\Pi$ is finite. Then for any program P and for any $Q \subseteq \mathcal{A}^\Pi$, $\mathcal{F}_\emptyset(P \cup Q)$ is a finite set. By theorem 2.9, $P_1 \approx_\Omega P_2$ iff for any $Q \subseteq \mathcal{A}^\Omega \subseteq \mathcal{A}$, $P_1 \cup Q \approx_\emptyset P_2 \cup Q$ iff (by Theorem 2.8) for any $Q \subseteq \mathcal{A}^\Omega$, $\mathcal{F}_\emptyset(P_1 \cup Q) = \mathcal{F}_\emptyset(P_2 \cup Q)$. Therefore for any Ω and any P_1, P_2 under previous assumption it is decidable if $P_1 \approx_\Omega P_2$.
Moreover, for any program P and for any n, P_n, $(T_P^\emptyset + id)(Q)$ and $(T^\emptyset_{P_n} + id)(Q)$ are finite sets. Therefore T-stability is decidable.

Let us now show the desired result for T-stable interpretations.

Proposition 3.4
Let P be an Ω-open program. Then

1. *if P_n is T-stable then $P \approx_\Omega P_n$. The converse does not hold,*
2. *P_n is T-stable iff for any goal $G = A_1, \ldots, A_m$ and for any set of atoms $Q \subseteq \mathcal{A}^\Omega$:*

$$G \overset{\vartheta}{\rightsquigarrow}_{P \cup Q} \square \quad \Leftrightarrow \quad \text{there exist } B_1, \ldots, B_m \in T^\emptyset_{P_n}(Q) \cup Q \text{ such that } \vartheta_{|G} = \gamma_{|G},\ \text{where } \gamma = mgu((A_1, \ldots, A_m)(B_1, \ldots, B_m)).$$

Example 3.1 Let us consider a signature $\Sigma = \{a, b\}$, a set of open predicates $\Omega = \{p, q\}$ and a program $P = \{p(X, Y) : -p(X, Y), q(X, Y)\}$. Then $P_1 \approx_\Omega P$ since, $T_P^\Omega \uparrow 1 = P_1 = P$. However P_1 is not T-stable since, for $Q = \{q(a, Y), q(X, b), p(X, Y)\}$, $T_{P_1}^\emptyset(Q) \neq T_{P_2}^\emptyset(Q)$. Indeed $p(a, b) \notin T_{P_1}^\emptyset(Q)$ and $p(a, b) \in T_{P_1}^\emptyset(Q)$ because $p(X, Y) : -p(X, Y), q(X, Y), q(X, Y) \in P_2 = T_P^\Omega(P_1)$. □

The previous results show that there exists a T-stable denotation P_n for program P, and that each T-stable denotation P_n satisfies the properties of $\mathcal{F}_\Omega(P)$, i.e. the compositional behaviour of a program P in terms of computed answers is completely specified by P_n. Indeed, further applications of the T^Ω operator for a T-stable interpretation may add new clauses, but these clauses do not add any possibility of computing new answers when composed with other program modules. We can now give the following definition of the finite compositional semantics P_{n^*}. It is clear from the definition and from proposition 3.4 that P_{n^*} is finite and has the required correctness and compositionality properties. In the next section we will show how P_{n^*} can be used to develop a modular analysis.

Definition 3.5 *Let P be an Ω-open program. Let n^* be the least index such that P_{n^*} is T-stable. Then P_{n^*} is the* finite *open semantics of the program P.*

We notice that the test for T-stability can be perfomed incrementally. Indeed for any $Q \subseteq \mathcal{A}^\Omega$, clearly $T_{P_{n+1}}^\emptyset(Q) = T_{P_n}^\emptyset(Q)$ iff $T_{\Delta n}^\emptyset(Q) \subseteq T_{P_n}^\emptyset(Q)$ where $\Delta 0 = \emptyset$ and $\Delta n = P_{n+1} \setminus P_n$ for $n \geq 1$. Moreover, we do not need to consider the whole $\wp(\mathcal{A}^\Omega)$ space for any P_n to decide T-stability. In fact, while computing the (T_P^Ω) fixpoint we can reduce the number of the elements in $\wp(\mathcal{A}^\Omega)$ to be checked by observing that, for a given $Q \in \wp(\mathcal{A}^\Omega)$, if $(T_{P_{n+1}}^\emptyset + id)(Q) = (T_{P_n}^\emptyset + id)(Q)$ then for any $k \geq n$ (by Lemma 3.3) $(T_{P_{k+1}}^\emptyset(Q) + id) = (T_{P_k}^\emptyset + id)(Q)$. A simple algorithm to compute the compositional T-stable semantics for a program module P can be defined as in Table 1. The procedure is terminating because the set D decreases during the computation (a straightforward consequence of Proposition 3.2).

For $\Omega \neq \Pi$, there exists P and n such that $P \not\approx_\Omega P_n$. For example, if $P = \{q(x), p(X) : -q(X)\}$ and $\Omega = \emptyset$, then $P \not\approx_\emptyset P_n$ for $n = 0, 1$ and $P \approx_\emptyset P_2$. Moreover, as shown in Example 3.1, in general P_{n^*} is not the first interpretation in the chain $\{P_n\}_{n<\omega}$ which is $\approx_\Omega$ equivalent to P.

We can alternatively consider as finite compositional semantics for P the denotation $P_{\overline{n}}$ where $\overline{n}$ is the least n such that $P \approx_\Omega P_n$. By Proposition 3.4 $P_{\overline{n}}$ is a subset of P_{n^*} and it is easy to see that $P_{\overline{n}}$ has the same properties of P_{n^*}, with the only difference that a derivation in $P_{\overline{n}} \cup Q$ may require more than two (parallel) derivation steps. For some applications in data-flow analysis, where it is more important to maintain a small dimension of the semantics than to have an efficient analysis, the semantics $P_{\overline{n}}$ can provide better results than P_{n^*}. It is worth noting that, as shown by the following result, there exists an interesting class of programs for which n^* and $\overline{n}$ coincide, namely *predicate disjoint programs*. We say that the predicate p is defined in the program P if P contain a clause such that p appear in the head.

Definition 3.6 *An Ω-open program P is* predicate disjoint *if any predicate symbol defined in P does not belong to Ω.*

According to previous definition, if we consider a $\cup_\Omega$ composition of *predicate disjoint* modules, predicates defined in each module are disjoint from those defined in the others. From a practical point of view the notion of *predicate disjoint* module is relevant since often module based implementations of logic programming languages assume this kind of separation. A

Input: A Ω-open program P.
Output: The T-stable semantics Pn for P.

begin
$\{n = 0\}$
$D := \wp(\mathcal{A}^\Omega)$;
$Pn := \emptyset$;
$\Delta n := T_P^\Omega(\emptyset)$;
while $D \neq \emptyset$ and $\Delta n \neq \emptyset$ **do**
$\left\{ Pn = T_P^\Omega \uparrow n \ \wedge \ \Delta n = T_P^\Omega \uparrow (n+1) \setminus Pn \ \wedge \ \left((T_{Pn}^\emptyset + id)(X) \neq (T_{T_P^\Omega(Pn)}^\emptyset + id)(X) \Rightarrow X \in D \right) \right\}$
for each $X \in D$ **do**
if $(T_{\Delta n}^\emptyset + id)(X) \subseteq (T_{Pn}^\emptyset + id)(X)$ **then** $D := D \setminus X$ **fi**;
$Pn := Pn \cup \Delta n$;
$\Delta n := T_P^\Omega(Pn) \setminus Pn$
$\{n := n + 1\}$
endw $\{Pn = T_P^\Omega \uparrow n$ is a T-stable interpretation in $\{P_n\}_{n<\omega} \ \wedge \ \Delta n = P_{n+1} \setminus P_n\}$
end

Table 1: An algorithm for T-stable semantics.

similar restriction is also considered in [17] and in [9]. A typical example of *predicate disjoint* program is provided by a Datalog program, where intensional and extensional predicates are disjoint, and where the extensional predicates are considered partially defined (see Section 4.2). Let us now prove the mentioned equality.

Proposition 3.7
Let P be an Ω-open predicate disjoint program. Let $\overline{n}$ be the least n such that $P_n \approx_\Omega P$ and let n^* be defined as in Definition 3.5. Then $n^* = \overline{n}$.

4 Applications

In the following we describe two applications of the previous results in the context of compositional data-flow analysis and logic-based databases.

4.1 Compositional Analysis over Finite Domains

It is well known that modularity helps in reducing the complexity of designing and proving correctness of programs and provides an easy approach to develop adaptable software. However, most of the already existing frameworks for analysis ([8,3,6,12]) require the entire program to be available at the time of analysis. This approach is often unpractical for large programs or for software developed by teams, either because the resource requirements are prohibitively high, or because not all program components are available at the same time for analysis.

The relevance of compositionality in logic program analysis has been firstly addressed in [9] by introducing a framework which is based on the compositional semantics $\mathcal{F}_\Omega$ and

which formalizes the analysis in terms of the standard theory of abstract interpretation ([11]). The abstract meaning of a module corresponds to its analysis and composition of abstract semantics corresponds to composition of analyses. In the following we will briefly describe this framework. Then, using the results of previous section, we will show how for a wide class of abstract domains the analysis can be performed by avoiding a further level of abstraction on clauses which was introduced in [9] in order to treat any generic (possibly infinite) abstract domain.

An abstract semantics typically associate programs with entities which capture the essence of "what they do" while abstracting away from the *concrete* details related with its execution model. This abstraction should provide a finite characterization of the (concrete) semantics, which is useful for static data-flow analysis of programs. Let (Π, Σ, Var) be a first-order language. To handle compositionality, the abstract semantic objects are descriptions of clauses. They are defined by describing sets of substitutions in $\wp(Sub)$ by (abstract) descriptions of substitutions in a lattice $ASub$. We denote by $\sqcup_S$ the corresponding *lub*. As the standard theory of abstract interpretation is formalized in terms *Galois insertions* [11], specifying the relation between abstract and concrete semantics objects, we assume $(\wp(Sub), \alpha_S, \gamma_S, ASub)$ be a Galois insertion[1]. *Abstract clauses* are pairs $\mathcal{LC} \times ASub$, where $\mathcal{LC} \subseteq \mathcal{C}(\Pi, \emptyset, Var)$ is the set of *flat clauses*, i.e. where no variable occurs twice. Abstract interpretations are sets of abstract clauses. Let I^a be an abstract interpretation. The meaning of I^a is specified by the following concretization function γ:

$$\gamma(I^a) = \left\{ c\theta \;\middle|\; \begin{array}{l} \langle c; \kappa \rangle \in I^a \\ \theta \in \gamma_S(\kappa) \end{array} \right\}.$$

Several abstract objects may represent the same concrete interpretation (i.e. γ may not be injective). This problem is solved in [9] by defining an equivalence relation $\sim_\gamma$ on $\wp(\mathcal{LC} \times ASub)$ such that $I_1^a \sim_\gamma I_2^a$ iff $\gamma(I_1^a) = \gamma(I_2^a)$. Notice that the equivalence $\sim_\gamma$ provides variable restriction and equivalence up to renaming.

Let $AInt = \wp(\mathcal{LC} \times ASub)/_{\sim_\gamma}$ be a complete lattice with $\sqcup$ as *lub*. This approach induces a Galois insertion between concrete interpretations $\wp(\mathcal{C})$ and abstract interpretations $AInt$: $(\wp(\mathcal{C}), \alpha, \gamma, AInt)$ (α is induced by γ in the usual way).

As in the concrete case, the abstract fixpoint operator is based on iterating *abstract unfolding*. Let $P^a, Q^a \in AInt$, and $unf^a : AInt \times AInt \to AInt$ such that

$$unf^a_{P^a}(Q^a) = \bigsqcup \left\{ \bar{c} \;\middle|\; \begin{array}{l} \bar{c} = \langle h : -\tilde{b}_1 :: \quad \cdots \quad :: \tilde{b}_n; \hat{\kappa} \rangle, \\ c = \langle h : -g_1, \ldots, g_n; \kappa \rangle \in P^a, \\ \langle h_1 : -\tilde{b}_1; \kappa_1 \rangle, \ldots, \langle h_n : -\tilde{b}_n; \kappa_n \rangle \in Q^a \text{ renamed apart}, \\ \hat{\kappa} = mgu^a\left(\langle \langle g_1, \ldots, g_n \rangle; \kappa \rangle, \langle \langle h_1, \ldots, h_n \rangle; \kappa_1 \ldots \kappa_n \rangle\right) \end{array} \right\}.$$

where mgu^a represent the *abstract unification* and, for any $\vartheta \in \gamma_S(\kappa_1)$ and $\sigma \in \gamma_S(\kappa_2)$, if $dom(\vartheta) \cap dom(\sigma) = \emptyset$ and $range(\sigma) \cap range(\vartheta) = \emptyset$, then $\vartheta\sigma \in \gamma_S(\kappa_1\kappa_2)$.

Given a program (module) P let us define $P^a = \alpha(P)$ and $T_P^\Omega : AInt \to AInt$ such that $T^\Omega_{P^a}(I^a) = unf^a_{P^a}(I^a \cup \alpha(id_\Omega))$. By unf^a definition, $T^\Omega_{P^a}$ is continuous. Moreover, by assuming the correctness of the abstract unfolding (i.e. $unf_P(Q) \subseteq \gamma(unf^a_{\alpha(P)}(\alpha(Q)))$ it follows that for each logic program P and interpretation I:

$$T_P^\Omega(I) \subseteq \gamma(T^\Omega_{\alpha(P)}(\alpha(I))).$$

Based on this observations, the following *abstract composition* theorem guarantees the correctness of the analysis. We denote $\sqcup_\Omega$ the abstract version of $\cup_\Omega$.

[1] i.e. α_S and γ_S are monotonic functions, $\alpha_S \circ \gamma_S = id$ and $\gamma_S \circ \alpha_S \supseteq id$ [11].

Theorem 4.1 (Correctness [9])
Let P_1 and P_2 be Ω_1 and Ω_2-open programs respectively and $\Omega \subseteq \Pi$ such that $P_1 \cup_\Omega P_2$ is defined. Let also $(\wp(C), \alpha, \gamma, AInt)$ be a Galois insertion such that

$$unf_P(Q) \subseteq \gamma(unf^a_{\alpha(P)}(\alpha(Q))).$$

Then $\mathcal{F}_\Omega(P_1 \cup_\Omega P_2) \subseteq \gamma(\mathcal{F}^a_\Omega(\mathcal{F}^a_{\Omega_1}(\alpha(P_1)) \sqcup_\Omega \mathcal{F}^a_{\Omega_2}(\alpha(P_2))))$.

The correctness specifies the meaning of the approximation introduced to let the analysis be effective. In this case, the concrete meaning of the composition of two programs is described by the composition of abstract meaning of the two programs, i.e. everything which is computed is described.

While the standard correctness property for abstract unfolding is based on the correctness of the abstract unification mgu^a, several technical properties of the abstract unification are required to allow the abstract unfolding to correctly mimic the concrete one. In particular, if $\mathcal{K} \subseteq ASub$, $\kappa, \kappa_1, \kappa_2, \kappa_3 \in ASub$, and $\bar{a}, \bar{b}, \bar{c}, \bar{d}$ be renamed apart tuples of flat atoms such that $|\bar{a}| = |\bar{c}|$ and $|\bar{b}| = |\bar{d}|$, we assume:

- $mgu^a(\langle \bar{a}; \sqcup_S \mathcal{K} \rangle, \langle \bar{c}; \kappa \rangle) = \sqcup_S \left\{ mgu^a(\langle \bar{a}; \kappa' \rangle, \langle \bar{c}; \kappa \rangle) \,\middle|\, \kappa' \in \mathcal{K} \right\}$ *additivity*
- $mgu^a(\langle \bar{a}; \kappa_1 \rangle, \langle \bar{c}; \kappa_2 \rangle) = mgu^a(\langle \bar{c}; \kappa_2 \rangle, \langle \bar{a}; \kappa_1 \rangle)$ *commutativity*
- $mgu^a(\langle \bar{b}; mgu^a(\langle \bar{a}; \kappa_1 \rangle, \langle \bar{c}; \kappa_2 \rangle) \rangle, \langle \bar{d}; \kappa_3 \rangle) = mgu^a(\langle \bar{a}; \kappa_1 \rangle, \langle \bar{c}; mgu^a(\langle \bar{b}; \kappa_2 \rangle, \langle \bar{d}; \kappa_3 \rangle) \rangle)$

  associativity

To extend the results of the previous section to the abstract semantics for composition, we need an associative and additive abstract unfolding operator. As shown by the following lemma, the associativity of the abstract unification implies the associativity of the abstract unfolding. This property allow us to repeat essentially the same proofs of previous section also for the abstract semantics.

Lemma 4.2
If mgu^a is associative and commutative then unf^a is associative.

To provide convergence in solving semantic fixpoint equations, the abstract domain is usually required to satisfy the *ascending chain condition*. Most of the existing abstract domains for substitutions are studied to satisfy this condition (e.g. *Prop* [10] [2], *affine relations* [20] [3] and *Sharing* [19] which is discussed later in the paper). Even if $ASub$ satisfies this condition, the induced abstract domain of interpretations does not guarantee termination, as it is always possible to build (by unfolding) clauses with arbitrary long bodies. Thus, due to

[2] The abstract domain *Prop* provides a concise representation for abstract substitutions able to describe ground dependency relations among their arguments. A propositional formula with connectives $\{\leftrightarrow, \vee, \wedge\}$ is associated with each substitution. For example, the formula x represents a substitution ϑ such that $\vartheta(x)$ is ground, while $x \leftrightarrow y \wedge z$ represents a substitution ϑ such that $var(\vartheta(x)) = var(\vartheta(y)) \cup var(\vartheta(z))$, i.e. x is ground iff both y and z are ground.

[3] The domain of *affine relations* has been introduced to derive linear size relations among variables in a program. This analysis is useful for compile-time overflow and array bound checking. Let V be a finite set of variables and $\delta : Term \to \omega$ be a norm. In the logic programming case an abstract substitution is an *affine subspace* of $\Re^{|V|}$, i.e. a point, line, plane *etc.* possibly not including the origin. An abstract substitution κ represents any concrete substitution ϑ such that $dom(\vartheta) = V$ and the vector $\langle \delta(\vartheta(x)) \rangle_{x \in V}$ satisfies κ. This abstract domain satisfies the ascending chain condition because in any properly ascending chain of affine subspaces $\kappa_1 \subseteq \kappa_2 \subseteq \ldots$ the subspace κ_i must have a dimension of at least one greater than κ_{i-1}, and $|V| < \omega$.

the base semantics construction, an additional layer of abstraction or the use of *extrapolation techniques* like *widening/narrowing* ([11]) is required. The solution adopted in [9] is a further abstraction performed by using these techniques (i.e. $\star$-abstraction). Abstract clauses are allowed to contain at most one occurrence of the same predicate symbol. Multiple occurrences of atoms for the same predicate in a body are then "compressed" to a single atom. In general, such a compression introduces à further approximation which makes the analysis less precise.

In view of the previous results, we can identify a wide class of abstract domains for which finiteness of the semantics, and hence termination, can be guaranteed independently on the finiteness of *AInt*. Therefore, using these domains the (compositional) analysis can be performed without the mentioned further loss of precision. We classify these domains as *compositionally tractable* abstract domains. From the abstract interpretation viewpoint, they play the same role as *finite domains* in (concrete) logic programming.

Definition 4.3
The abstract domain of interpretations AInt is compositionally tractable iff for each flat clause c, the set $\left\{ [\{\langle c;\kappa\rangle\}]_{\sim_\gamma} \,\middle|\, \kappa \in ASub \right\}$ *is finite.*

As in the concrete case, we define $P^a_n = \mathcal{T}^{\Omega}_{P^a} \uparrow n$, for $P^a \in AInt$. Moreover, we extend the notion of T-stability from concrete to abstract interpretations in the obvious way. Notice that, by correctness: $\gamma(\alpha(P_n)) \subseteq \gamma(\alpha(P)_n)$. The proof of the following result is analogous to that of Proposition 3.2, since by mgu^a additivity $\lambda I.unf^a_I(P^a)$ is additive.

Proposition 4.4
Let P be an Ω-open program and AInt be a compositionally tractable abstract domain. There exists $n < \omega$ such that $\alpha(P)_n$ is T-stable.

A compositional abstract interpretation can be provided by a finite iteration of $\mathcal{T}$. We call this interpretation a *T-stable abstract interpretation*. A characterization of n^* can be obtained by mimicing the results of the previous section (Lemma 3.3 and Proposition 3.7). The algorithm in Table 1 provides a compositional abstract interpretation on a compositionally tractable domain for any Ω open program (module). The followin theorem states the correctness of a T-stable abstract interpretation wrt computed answers.

Theorem 4.5
Let $G = A_1, ..., A_m$ be a goal, $Q \subseteq \mathcal{A}^{\Omega}$ and P be a Ω-open program and $\alpha(P)_n$ be T-stable. If $G \overset{\vartheta}{\leadsto}_{P \cup Q} \square$ then there exist $B_1, ..., B_m \in \gamma\left(\mathcal{T}^{\vartheta}_{\alpha(P)_n}(\alpha(Q)) \sqcup \alpha(Q)\right)$ renamed apart such that $\beta = \dot{m}gu((A_1, \ldots, A_m)(B_1, \ldots, B_m))$ and $\vartheta_{|G} = \beta_{|G}$.

Several abstract domains for logic program are compositionally tractable. As an example of a domain of abstract substitutions, in the following we consider the domain *Sharing*, proposed by Jacobs and Langen in [19]. *Sharing* has been introduced to identify aliasing between variables with a great deal of accuracy. Let σ be a (concrete) substitution: x and y *share* the variable w if $w \in var(\sigma(x)) \cap var(\sigma(y))$. The sharing information description is based on the notion of *sharing group*. The sharing group of a variable w (denoted by $sg(\sigma, w)$) is the set of variables in the domain of σ that share w [19]. A suitable representation for aliasing is provided by defining $Sharing = \wp(\wp(Var))$. Intuitively, if $S \in Sharing$, each of the $X \in S$ represents a sharing group for a given variable name. For example, the substitution

$$\sigma = \{x_1 \mapsto [A \mid L], x_2 \mapsto M, x_3 \mapsto [A \mid N], x_4 \mapsto L, x_5 \mapsto M, x_6 \mapsto N\}$$

is represented by the abstract substitution: $\kappa = \{\{x_1, x_4\}, \{x_1, x_3\}, \{x_2, x_5\}, \{x_3, x_6\}\}$, where $\{x_1, x_4\}$ is the sharing group for L in σ. Groundness and independence can also be derived from *Sharing*. Let $S \in$ *Sharing*; a variable is ground in S if it does not appear in any sharing group of S. Two variables x and y are *independent* iff they do not share a common variable. By the definition, x and y are independent in S iff they do not appear together in the same sharing group. Ground dependences can also be captured: in $\{\{x_1, x_3\}, \{x_2, x_3\}\}$ x_3 is ground iff both x_2 and x_3 are ground.

Proposition 4.6
Sharing is compositionally tractable.

The same holds for the abstract domain *Prop* ([10]) and for depth-k abstractions ([3]). On the other hand, the abstract domain of *affine relations* ([20]) is not compositionally tractable because for $n < \omega$, there are infinitely many affine subspaces in $\Re^n$.

Because of the use of flat clauses as semantic objects, abstract unification is much simpler than the one defined in [19]. We follow Jacobs and Langen in the definition of mgu^a. Let $S \in$ *Sharing*, $x \in$ *Var* and $in(x, S) = \left\{ X \in S \,\middle|\, x \in X \right\}$. The *closure under union* of S, denoted by S^* is the smallest superset of S such that $X \in S^* \wedge Y \in S^* \Rightarrow X \cup Y \in S^*$. The abstract unification requires an auxiliary binary operator $\oplus$ on *Sharing*, such that $S \oplus T = \left\{ X \cup Y \,\middle|\, X \in S, Y \in T \right\}$ for $S, T \in$ *Sharing*. Let $p(x_1, ..., x_n)$ and $p(x'_1, ..., x'_n)$ be (flat) atoms, and $\kappa, \kappa' \in$ *Sharing*:

$$mgu^a(\langle p(x_1, .., x_n); \kappa\rangle, \langle p(x'_1, .., x'_n); \kappa'\rangle) = unify(x_1, .., x_n; x'_1, .., x'_n; \kappa \cup \kappa') \quad \text{where}$$

$$unify(x_1, ..., x_n; x'_1, ..., x'_n; S) = \begin{cases} (S \setminus in(x_1, S) \setminus in(x'_1, S)) \cup \\ \qquad (in(x_1, S) \oplus in(x'_1, S))^* & \text{if } n = 1 \\ unify(x_2, ..., x_n; x'_2, ..., x'_n; unify(x_1; x'_1; S)) & \text{if } n > 1 \end{cases}$$

The correctness of mgu^a is a consequence of the correctness condition proved in [19]. The same holds for additivity, commutativity and associativity of mgu^a. The relevance of *Sharing* in data-flow analysis of logic programs has been considered in [23] to exploit *independent AND-parallelism*. Other important optimizations can be obtained from the ground dependency information. In the following we consider an example for compositional sharing analysis defined in terms of T-stable abstract interpretations. It shows a practical case where, once a module has been changed, the semantics of the composition may be affected.

Example 4.1 Consider an "insertion sort" routine defined in terms of three modules:

P_{sort}
```
sort([],[]).
sort([X|Xs],Ys) :- sort(Xs,Zs),insert(X,Zs,Ys).
```

P_{ins}
```
insert(X,[],X).
insert(X,[Y|Ys],[Y|Zs]) :- gt(X,Y),insert(X,Ys,Zs).
insert(X,[Y|Ys],[X,Y|Zs]) :- le(X,Y).
```

P_{gl}
```
gt(s(X), X).                    le(X, X).
gt(s(X), Y) :- gt(X, Y).        le(X, s(Y)) :- le(X,Y).
gt(s(X), s(Y)) :- gt(X, Y).     le(s(X), s(Y)) :- le(X, Y).
```

Assume `insert` be open in P_{sort}. Then we obtain:

$I^a_{sort_1}$: $\langle sort(x_1, x_2) : -; \emptyset \rangle$

$I^a_{sort_2}$: $\langle sort(x_1, x_2) : -insert(x_5, x_6, x_7); \{\{x_1, x_5\}, \{x_2, x_7\}\} \rangle$

$I^a_{sort_3}$: $\left\langle \begin{array}{r} sort(x_1, x_2) : -insert(x'_5, x'_6, x'_7), insert(x_5, x_6, x_7); \\ \{\{x_1, x_5\}, \{x_2, x_7\}, \{x_6, x'_7\}\} \end{array} \right\rangle$

$\vdots$

Assume also `le` and `gt` be open in P_{ins}. Thus we have:

$I^a_{ins_1}$: $\langle insert(x_1, x_2, x_3) : -; \{\{x_1, x_3\}\} \rangle$

$\left\langle \begin{array}{r} insert(x_1, x_2, x_3) : -le(x_4, x_5); \\ \{\{x_1, x_4, x_3\}, \{x_2, x_3, x_5\}, \{x_2, x_3\}\} \end{array} \right\rangle$

$I^a_{ins_2}$: $\left\langle \; insert(x_1, x_2, x_3) : -gt(x_4, x_5); \{\{x_2, x_3, x_5\}, \{x_1, x_4, x_3\}\} \; \right\rangle$

$\left\langle \begin{array}{r} insert(x_1, x_2, x_3) : -gt(x_4, x_5), le(x'_4, x'_5); \\ \{\{x_1, x_4, x'_4\}, \{x_2, x_3, x'_5\}, \{x_2, x_3\}, \{x_2, x_3, x_5\}\} \end{array} \right\rangle$

$\vdots$

P_{gl} is closed (i.e. completely defined) and has the following abstract semantics $\alpha(P_{gl})_1$:

$\langle gt(x_1, x_2) : -; \{\{x_1, x_2\}\} \rangle \;\; \langle le(x_1, x_2) : -; \{\{x_1, x_2\}\} \rangle$

The T-stable abstract semantics for `insert` is obtained in 2 steps and hence is $\alpha(P_{ins})_2 = I^a_{ins_1} \cup I^a_{ins_2}$. Notice that the test for T-stability of $\alpha(P_{ins})_2$ can be performed by unfolding $\alpha(P_{ins})_3 \backslash \alpha(P_{ins})_2$ with any possible sharing-pattern for `gt` and `le`. The abstract interpretation for $P_{ins} \cup P_{gl}$ is given by composing the interpretations for each single module:

$$\left\{ \begin{array}{l} \langle gt(x_1, x_2) : -; \{\{x_1, x_2\}\} \rangle \;\; \langle le(x_1, x_2) : -; \{\{x_1, x_2\}\} \rangle \\ \langle insert(x_1, x_2, x_3) : -; \{\{x_1, x_3\}\} \rangle \\ \langle insert(x_1, x_2, x_3) : -; \{\{x_1, x_2, x_3\}, \{x_2, x_3\}\} \rangle \\ \langle insert(x_1, x_2, x_3) : -; \{\{x_1, x_2, x_3\}\} \rangle \end{array} \right\}.$$

A different definition for the predicates `gt` and `le`, like the following:

P'_{gl}

```
gt(s(0), 0).                    le(0, 0).
gt(s(X), Y) :- gt(X, Y).        le(X, s(Y)) :- le(X,Y).
gt(s(X), s(Y)) :- gt(X, Y).     le(s(X), s(Y)) :- le(X, Y).
```

provides a different meaning for $P_{ins} \cup P'_{gl}$ wrt the sharing information:

$$\left\{ \begin{array}{l} \langle gt(x_1, x_2) : -; \emptyset \rangle \;\; \langle le(x_1, x_2) : -; \emptyset \rangle \\ \langle insert(x_1, x_2, x_3) : -; \{\{x_1, x_3\}\} \rangle \\ \langle insert(x_1, x_2, x_3) : -; \{\{x_2, x_3\}\} \rangle \\ \langle insert(x_1, x_2, x_3) : -; \emptyset \rangle \end{array} \right\}.$$

We do not need to recompile the analysis for P_{ins} and P_{sort} once P_{gl} has been changed. The abstract meaning of P_{sort} does not result affected by modifying P_{gl}. For P_{sort}, in both the cases, we obtain: $\{\langle sort(x_1, x_2) : -; \emptyset \rangle \;,\; \langle sort(x_1, x_2) : -; \{\{x_1, x_2\}\} \rangle\}$. □

4.2 Databases

A Datalog program is a function-free logic program consisting of two components [7]: the extensional database (*EDB*) and the intensional database (*IDB*). The set Π of predicate symbols is partitioned into Π_{EDB} and Π_{IDB}. *EDB* uses only predicate symbols of Π_{EDB} and defines *extensional* relations. Clauses of *IDB* are built using symbols in Π_{IDB} for the head and symbols in Π for the body, and define *intensional* relations. A logical query is a goal. Two basic methods of logical query processing can be identified. Using the *interpretation method*, a query is directly evaluated against the set of clauses (*IDB*) and the base relations (*EDB*). The query interpreter receives as input a set of base relations, a set of clauses, a query, and returns a result. This method does not, in essence, distinguish between the *data extension*, in the form of base relations, and the *data intension*, in the form of a set of clauses. On the other side, the *compilation method* makes a distinction between the data intension and the data extension by considering the intensional database as a *mapping* between intensional relations and extensional one. It divides query processing into the following three phases:

1. A *database translation phase*, which receives as input the set of clauses of the *IDB*, and transform it into a recursion-free one.
2. A *query translation phase*, which receive as input the translated database, and a query, possibly involving one or more intensional relations, and produces the equivalent query that involves only extensional relations.
3. An *execution phase* which receives as input the translated query, evaluates this query against the stored data and returns the results.

The *standard fixpoint semantics* for logic programs ([1,22]) as introduced by van Emden and Kowalski, provides a bottom-up execution model which is strongly related to the interpretation view. Several evaluation strategies have been developed to improve the naive strategies based on the immediate consequence operator [2]. However, interpretation methods are very inefficient and therefore most systems approach the problem of query processing by means of compilation. The main problem for compilation methods is the treatment of recursive clauses. Several specialized techniques have been proposed for specific classes of clauses (such as linear, regular etc.) and there is not a general compilation framework which can treat all the types of rules. The standard fixpoint semantics in this case is not useful since it does not distinguish between the data intension and the data extension.

The compositional fixpoint semantics introduced in Definition 2.4 allows to consider the data extension and data intension as separate modules. The set of extensional predicates Π_{EDB} can be seen as the set of open predicate symbols Ω. *IDB* and *EDB* are then two predicate disjoint Π_{EDB}-open programs. Intensional relations are hypothetically defined in $\mathcal{F}_{\Pi_{EDB}}$ by means of non recursive clauses. For Datalog programs defined over finite domains, according to the results of Section 3, we can then obtain a finite semantics (the T-stable IDB_n) with the mentioned correctness and compositionality features. This provides the basis for an *uniform technique* to translate the intensional database *IDB* into an equivalent one which is recursion-free (and which maps intensional relations into extensional one). The translation is obtained essentially by the algorithm in Table 1. Note that, according to proposition 3.4 the translated *IDB* allows to answer to any query (with intensional relations) using at most two (parallel) derivation steps.

Example 4.2
Consider Σ = {henry, peter, john}, Ω = {parent},
EDB = {parent(henry, peter), parent(peter, john) and

$$IDB = \{\ \texttt{ancestor(X,Y)} :- \texttt{parent(X,Y)}.\quad \texttt{ancestor(X,Z)} :- \texttt{parent(X,Y)},\ \texttt{ancestor(Y,Z)}.\ \}$$

Then

$$T^{\Omega}_{IDB} \uparrow \omega = \{\ ancestor(X,Y) :- parent(X,Y),\ ancestor(X,Z) :- parent(X,Y), parent(Y,Z),\ \vdots\ ancestor(X,Z) :- parent(X,Y_1), \ldots, parent(Y_t,Z),\ \vdots\ \}$$

However, since the domain is finite, we can obtain the following finite semantics $T^{\Omega}_{IDB} \uparrow 2$ such that $T^{\Omega}_{IDB} \uparrow 2 \approx_{\Omega} IDB$. The query `ancestor(henry,john)` is translated by means of $T^{\Omega}_{IDB} \uparrow 2$ into the extensional query `parent(henry,Y),parent(Y,john)` which is evaluated against EDB.

$$T^{\Omega}_{IDB} \uparrow 2 = \{\ ancestor(X,Y) :- parent(X,Y),\ ancestor(X,Z) :- parent(X,Y), parent(Y,Z)\ \}$$

□

Because of compositionality, the mentioned translation technique can be carried out in an incremental and modular way, with obvious advantages. Moreover, the (finite) compositional semantics can be used to support modular construction of Datalog programs where also the IDB is partitioned in several modules. As illustrated by the previous examples, each time a new module is added, we do not need to recompute the semantics of (and we do not need to recompile) the whole system but we can compose the semantic of the new module with that of the old one. To the best of our knowledge, the only existing approach to modular databases is in [15], but it does not support a modular bottom-up execution model such as the one considered above.

References

[1] 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.

[2] F. Bancilhon and R. Ramakrishnan. Performance Evaluation of Data Intensive Logic Programs. In J. Minker, editor, *Foundation of Deductive Databases and Logic Programming*, pages 439–519. Morgan-Kaufmann, 1987.

[3] R. Barbuti, R. Giacobazzi, and G. Levi. A General Framework for Semantics-based Bottom-up Abstract Interpretation of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(1):133–181, 1993.

[4] E. Bertino, M. Martelli, and D. Montesi. An Incremental Semantics for CLP(AD). In A. Marchetti, Spaccamela, P. Mentrasti, and M. Venturini Zilli, editors, *Proc. Fourth Italian Conference on Theoretical Computer Science*, pages 53–67. World Scientific, 1992.

[5] A. Bossi, M. Gabbrielli, G. Levi, and M. C. Meo. Contributions to the Semantics of Open Logic Programs. In *Proceedings of the International Conference on Fifth Generation Computer Systems 1992*, pages 570–580, 1992.

[6] M. Bruynooghe. A Practical Framework for the Abstract Interpretations of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[7] S. Ceri, G. Gottlob, and L. Tanca. What You Always Wanted to Know About Datalog - And Never Dared to Ask. *IEEE Tran. on Knowledge and Data Eng.*, 1(1):146–164, March 1989.

[8] M. Codish, D. Dams, and E. Yardeni. Bottom-up Abstract Interpretation of Logic Programs. Technical report, Dept. of Computer Science, The Weizmann Institute, Rehovot, 1990. To appear in Theoretical Computer Science.

[9] M. Codish, S. K. Debray, and R. Giacobazzi. Compositional Analysis of Modular Logic Programs. In *Proc. Twentieth Annual ACM Symp. on Principles of Programming Languages*, pages 451–464. ACM Press, 1993.

[10] A. Cortesi, G. Filè, and W. Winsborough. *Prop* revisited: Propositional Formula as Abstract Domain for Groundness Analysis. In *Proc. Sixth IEEE Symp. on Logic In Computer Science*, pages 322–327. IEEE Computer Society Press, 1991.

[11] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Proc. Fourth ACM Symp. Principles of Programming Languages*, pages 238–252, 1977.

[12] S. K. Debray. Efficient Dataflow Analysis of Logic Programs. *Journal of the ACM*, 39(4):949–984, 1992.

[13] F. Denis and J.-P. Delahaye. Unfolding, Procedural and Fixpoint Semantics of Logic Programs. In C. Choffrut and M. Jantzen, editors, *STACS 91*, volume 480 of *Lecture Notes in Computer Science*, pages 511–522. Springer-Verlag, Berlin, 1991.

[14] 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.

[15] B. Freitag. A Deductive Database Language Supporting Modules. In *Proc. Second Int'l Computer Science Conference*, 1992. To appear.

[16] M. Gabbrielli, G. Levi, and M. C. Meo. Observational Equivalences for Logic Programs. In K. Apt, editor, *Proc. Joint Int'l Conf. and Symposium on Logic Programming*, pages 131–145. The MIT Press, Cambridge, Mass., 1992.

[17] 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.

[18] H. Gaifman and E. Shapiro. Proof theory and semantics of logic programs. In *Proc. Fourth IEEE Symp. on Logic In Computer Science*, pages 50–62. IEEE Computer Society Press, 1989.

[19] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent AND Parallelism. *Journal of Logic Programming*, 13(2 & 3):291–314, 1992.

[20] M. Karr. Affine Relationships Among Variables of a Program. *Acta Informatica*, 6:133–151, 1976.

[21] G. Levi. Models, Unfolding Rules and Fixpoint Semantics. In R. A. Kowalski and K. A. Bowen, editors, *Proc. Fifth Int'l Conf. on Logic Programming*, pages 1649–1665. The MIT Press, Cambridge, Mass., 1988.

[22] J. W. Lloyd. *Foundations of Logic Programming*. Springer-Verlag, Berlin, 1987. Second edition.

[23] K. Muthukumar and M. Hermenegildo. Compile-time derivation of variable dependency using abstract interpretation. *Journal of Logic Programming*, 13(2 & 3):315–347, 1992.

Finito di stampare nel mese di maggio 1993
presso LITOGRAF
COSENZA