INFORMATIQUE THÉORIQUE ET APPLICATIONS

M. BELLIA

# Logic and functional programming by retractions

*Informatique théorique et applications*, tome 22, nº 3 (1988), p. 271-310.

<http://www.numdam.org/item?id=ITA_1988__22_3_271_0>

Informatique théorique et Applications/Theoretical Informatics and Applications
(vol. 22, n° 3, 1988, p. 271 à 310)

# LOGIC AND FUNCTIONAL PROGRAMMING BY RETRACTIONS (*)

by M. BELLIA [1]

Communicated by G. LONGO

Abstract. – *The paper applies the concept of* retraction *to represent computable relations. A strict subclass of retractions is defined. This class of functions is then discussed in a set theory which is built according to the structure of the Herbrand Universe in clausal logic. The set theory allows a set theoretic interpretation of the Herbrand terms and supplies them with a combinatory formulation. The approach provides a functional programming paradigm with almost all the features of logic programming. According to it, predicates, defined by a set of Horn clauses, are reformulated in terms of retractions, while queries in terms of function invocations. Existentially quantified variables, logic variables, which occur in a query are mapped into combinatory forms. Relations between inference in logic formulas and reduction of combinatory forms are finally discussed. Topics related to the definition of reduction systems for our combinatory forms are deferred to a separate companion paper.*

Résumé. – *L'article applique la notion de* rétraction *pour représenter les relations calculables. Une sous-classe stricte de rétraction est définie. Cette classe de fonctions est alors discutée dans une théorie des ensembles qui est construite en accord avec la structure de l'Univers d'Herbrand en logique clausale. La théorie des ensembles permet une interprétation théoretique des termes d'Herbrand en fournissant une formulation combinatoire. Cette approche fournit un paradigme fonctionnel de programmation avec presque tous les aspects de la programmation logique. En accord avec ceci, les prédicats définis par un ensemble de clauses de Horn, sont reformulés en termes des rétractions, et les interrogations en termes d'invocations de fonctions. Les variables quantifiées existentiellement, variables logiques, qui interviennent dans une interrogation, sont transformées en formes combinatoires. Les relations entre l'inférence dans les formules logiques et la réduction de formes combinatoires sont finalement discutées.*

## INTRODUCTION 1.

In the last few years languages based on first order logic [Lloyd84] have become very popular declarative programming languages [Shapiro86]. [Robinson83] analyzes the historical framework and the main motivations which

(*) Received in . . .
This work was partially supported by the European Community under ESPRIT Project 415.
(1) Dipartimento di Informatica, Università di Pisa, Corso Italia, 40 - 56100 Pisa, Italy.


0296-1598 88/03 271 40/$6.00/© Gauthier-Villars

make this class of languages so innovative, attractive and potentially relevant in several applications [vanCaneghem84]. The starting idea [Kowalski74] was the procedural interpretation of linear resolution systems [Kowalski71], restricted to Horn clause logic. In this framework, resolution [Robinson65] results quick and simple to apply and its proof search space is reduced to a tree. Procedural interpretation consists in interpreting each step in the resolution process as a step in a program computation process.

The resolution process becomes a programming language interpreter. From theorem provers, logic languages inherit mechanisms which are different from those of programming languages. We mention *unification* which is used as the *parameters passing and value return* mechanism, and the *goal directed computation* which is used as the *expression evaluation rule.*

Horn clause logic theories become programs. From logic, programs inherit features which are unusual. We mention: the *operational semantics*, according to which procedure invocations can be composed in (a goal or) clause rigth part and procedures can be abstractly (or more intentionally) defined, and *relations* which become the mathematical objects that predicative procedures denote.

Several proposals are currently pursuing the integration of logic and functional languages to obtain a super-language with the features of both languages [Abramson84, Lindstrom85, Kahn81, Robinson82, Komorowski82, Mellish84, Srivastava85, Bellia84, Barbuti85, Clark83, Fribourg85, Reddy85, Goguen84, Subrahmanyam84, Hsiang83, Dershowitz84, Dershowitz85, Kornfeld83, Sato84, Rety85, Berkling82, Darlington85, Bowen85]. The aim is the definition of a super-language with the features of both which, on one hand is adequate to the need of intelligent applications and, on the other hand can efficiently be executed by machines [Moto-Oka82]. In spite of the different mechanisms and constructs, logic and functional languages share various features: both are applicatives, i. e. the language main construct is *application* and computations are manipulations of values, and adequate for symbolic computations, i. e. *symbolic data* can be expressed and manipulated as values.

Though substantially different in the techniques [Bellia86], all the current proposals are based on extensions and/or on merges of the mechanisms on which the two classes of languages are based.

The resulting mechanisms provide the super-language with a programming paradigm which copes with almost all the features of both logic and functional programming. However, logic and functional features, even if present in the resulting super-language, cannot be equally made in use, and sometimes, combinations of them are obscure or meaningless. Moreover, the resulting language is not as simple, semantically clean and well machine supported as the original languages are.

Common to the above mentioned proposals is that in order to combine logic with functional programming features, logical variables (i. e. existentially quantified variables) and functional expressions have to live together. We claim that logic programming features can be equally *expressed* in a purely functional programming paradigm, without changes on the basic mechanisms of functional languages, and without the introduction of any additional construct or mechanism, e. g. logical variables and, narrowing or unification. Inside this functional paradigm, logic programming as well as functional programming can be formulated by combinatory formulas for which promising realizations of reduction machines are in progress [Berkling75, Clarke80, Darlington81, Kluge80, Mago80, Hankin85], and they could be the natural machine support for our super-language. Functional languages are potential super-languages in which logic and functional programming features can be combined in a natural and semantically clean way.

The approach is based on a combination of the computable function theory and of a set theory [Bellia87] suitable for computations on sets of symbolic data. To each predicate, defined by a logic program (Horn clause theory), we univocally associate exactly one function of a class of retractions. The elements of the domain and image sets of the retractions are elements of a set theory (of symbolic data) which is build according to the structure of the terms of the Herbrand Universe. To each query in a logic program, we associate a purely functional expression. The expression is an application of retractions which are associated to predicates in the logic program, and its evaluation corresponds to the *success set* of the query in the logic program (i. e. set of all the instances of the query which can be deduced from the logic program). The nature of the correspondence between resolution of queries and computation of applications of retractions, the features and properties of the class of retractions which is here, associated to the set of predicates, the structure of the set theory, which models domain and image sets of retractions, are the main topics of the present paper. Other related topics are functional programming with sets and the combination of logic and functional programming by using retractions to express predicates and queries.

Section 2 introduces the concept of retraction, relates it to the concept of relation (Definition 2.1) in the theory of computability and then, to the concept of predicate in logic programming. The features and properties of the retractions are formally stated by Propositions 2.1-2.6 and their relevance to the logic programming concepts of program invertibility and of partially evaluated data structures is briefly outlined and exemplified. In particular, Proposition 2.4 states a one-to-one correspondence between a subclass of the retractions on the computable parts of a domain D and the class of computable relations on D, or equally, when $D$ is the Herbrand Universe and relations are expressed by Horn clause theories, the class of predicates. Section 3 concerns the treatment of logic programming predicates as retractions in functional programming and, the structure of the set theory whose elements form the domain and image sets of the retractions. To make the treatment more concrete, Section 3 introduces a functional language to express retractions, and the structure of the set theory is then, discused as the data domain of the language. The language is essentially Church's lambda calculus. Emphasis is placed on the language data domain operators, which are set operators: formal definitions and examples of the use are included. Sections 4 and 5 concern the relations between logical variables and herbrand terms, on one side, and functional expressions, on the other. Then, Proposition 4.1 states the existence of a function which maps (tuples of possibly non-ground) Herbrand terms into *constant expressions*, i. e. a class of combinatory formulas which only contain data and applications of the language operators. Section 4 introduces the concept of *most general instance* of pairs of (tuples of) Herbrand terms and relates it to the concept of unification. Then, Proposition 5.1 states a one-to-one correspondence between the computation of the *most general instance* of a pair of (tuples of) Herbrand terms and the application of set intersection to the corresponding constant expressions. Section 6 compares logic with functional programming. It shows that, when retractions are used to characterize predicates, almost all the (programming) features of the logic paradigm are preserved in the functional programming paradigm and are straightforwardly combined with the functional ones. Section 7 contains concluding remarks and a brief comparison of our approach with some other proposals for the integration of logic and functional programming.

Finally, topics related to computations with the set operators discussed in Sections 3 and 5, and to expression reductions are deferred to a separate companion paper. In [Bellia88] we show that constant expression have *normal form* and we formulate a reduction system to compute normal forms. Such a set of reduction rules toghether with rules $\alpha$, $\beta$, $Y$ for lambda-terms, forms an operational semantics for our calculus with retractions.

## 2. RELATIONS, RETRACTIONS AND PREDICATES

A well known set theoretic representation of relations is based on caracteristic functions. To each *computable relation*, $R_D$, on a (non empty) space

$D=D_1 \times \ldots \times D_n$, we can associate a computable boolean function $\mathbf{f}_R: D \to \{0, 1\}$, such that:

$$\forall x \in D, \quad x \in R_D \quad \Leftrightarrow \quad \mathbf{f}_R(x)=1. \tag{1}$$

The function $\mathbf{f}_R$ is the *characteristic function* of the set denoted by $R_D$. Note that, $\mathbf{f}_R$ is a partially defined function, hence if $x$ is not in $R_D$, $\mathbf{f}_R$ results 0 or is undefined. Furthermore, if $n=1$, $D$ is not a product, and $R_D$ is *degenerate*.

In a sense, $\mathbf{f}_R$ reminds us of the concept of *predicate*, and it behaves as a predicate if $\{0, 1\}$ is interpreted as the truth domain. In order to extend functional with logic programming features, some authors [Hsiang83, Dershowitz84, Dershowitz85], use a generalization of (1) which could be formulated by

$$\forall [X] \in \mathbf{P}(D), \quad X \subseteq R_D \quad \Leftrightarrow \quad \mathbf{f}_R([X])=\text{true} \tag{2}$$

where $\mathbf{P}(D)$ is the *power set* of $D$, $[X]$ is the element of $\mathbf{P}(D)$ which denotes the subset $X \subseteq D$, and $\mathbf{f}_R$ is a function from $\mathbf{P}(D)$ to $\{\text{true, false}\}$. Then, in order to deal with $\mathbf{f}_R$, the functional language has to be extended to handle sets and functions from sets into values. Anyway, (2) is not all the functional language needs in order to gain the full logic programming power. Extensions on the language mechanisms (notably, narrowing and logical variables) have been added in order to give it multi mode use of relations and partially evaluated data, i. e. to "answer" questions like: for which $x$, $\mathbf{f}_R(x)=\text{true}$ holds?

We show that such additional mechanisms can be avoided by using a different characterizaition of relations. We introduce the following.

Definition 2.1: Given a space $D$, let $\mathbf{P}(D)$ be the power-set (i. e. set of the parts) of $D$, and $\{\mathbf{F}_R\}$ be the set of all the functions, $\mathbf{F}_R$, which (3) associates to the set of all computable relations $R_D$ on $D$:

$$\forall [X] \in \mathbf{P}(D), \quad \mathbf{F}_R([X])=[X \cap R_D]. \tag{3}$$

$\mathbf{F}_R: \mathbf{P}(D) \to \mathbf{P}(D)$ is a *set function*, mapping sets into sets.

Note that, given $R_D$, (3) uniquely determines one $\mathbf{F}_R$. We say that $\mathbf{F}_R$ is the *representative* [according to (3)] of $R_D$. To characterize relations on $D$ in terms of functions, (3) uses a class of functions defined in the more complex space $\mathbf{P}(D)$.

$\{\mathbf{F}_R\}$ is a class of *retractions*, or idempotent functions.

Proposition 2.1:

$$\forall \mathbf{F}_R: \quad \mathbf{P}(D) \to \mathbf{P}(D) \in \{\mathbf{F}_R\}, \qquad \mathbf{F}_R \textit{ is a retraction of } \mathbf{P}(D).$$

*Proof:* Note that,

$$\forall [X] \in \mathbf{P}(D), \quad [X \cap \boldsymbol{R}_D] \in \mathbf{P}(D)$$

is a fixed-point of $\mathbf{F}_R$, i. e. $\mathbf{F}_R([X \cap \boldsymbol{R}_D]) = [X \cap \boldsymbol{R}_D]$.

Retractions on sets are widely used in topology, and were used in [Scott76] to model data types in programming languages. The image set of a retraction is called its *retract*. A retraction with retract $U$ is called a retract on $U$.

PROPOSITION 2.2:

$$\forall \mathbf{F}_R : \mathbf{P}(D) \to \mathbf{P}(D) \in \{\mathbf{F}_R\}, \qquad \mathbf{F}_R \textit{ is a retract on } \mathbf{P}([R_D]).$$

*Proof:* By definition of the class $\{\mathbf{F}_R\}$ in (3),

$$\mathbf{F}_R([X]) = [X \cap \boldsymbol{R}_D] \quad \text{then} \quad \mathbf{F}_R([X]) = [X] \qquad \text{iff} \quad X \subseteq \boldsymbol{R}_D.$$

Note that, the elements of $\mathbf{P}(D)$ are partially ordered by set inclusion, $\subseteq$, on $D$. Later on, we will use $\subseteq$ to denote both the set inclusion on the subsets of $D$ and the ordering relation on the elements of $\mathbf{P}(D)$, i. e. if $X \subseteq Y$ then, $[X] \subseteq [Y]$ in $\mathbf{P}(D)$.

$\{\mathbf{F}_R\}$ is a *proper* subclass of the retractions, as immediately follows from Proposition 2.3.

PROPOSITION 2.3:

$$\forall \mathbf{F}_R : \quad \mathbf{P}(D) \to \mathbf{P}(D) \in \{\mathbf{F}_R\}, \qquad \forall [X] \in \mathbf{P}(D), \quad \mathbf{F}_R([X]) \subseteq [X].$$

*Proof:* By definition of the class $\{\mathbf{F}_R\}$ and by the properties of set-intersection.

Constant functions are examples of retractions which are not in $\{\mathbf{F}_R\}$ (unless $D$ has cardinality 1).

For example, let $D$ be the cartesian product $D1 \times D2$, where $D1 = \{a, b\}$ and $D2 = \{c, d\}$, then $\mathbf{P}(D)$ contains $2^4$ elements, for instance $[\{\langle a, c\rangle\}]$, $[\{\langle b, c\rangle, \langle b, d\rangle\}]$ are elements of $\mathbf{P}(D)$. The function $f$ such that:

$$\forall X \in \mathbf{P}(D), \quad f(X) = [\{\langle a, c\rangle\}]$$

is a retraction of $\mathbf{P}(D)$ but does not satisfy Proposition 2.3, and is not a member of $\{\mathbf{F}_R\}$. Thus, we say that there are no relations on $D$ for which $f$ is the representative. In contrast, the function $g$ such that:

$$g([X]) = \begin{cases} [\{\langle a, c\rangle\}] & \text{if } X \text{ contains } \langle a, c\rangle \text{ but does not contain } \langle a, d\rangle \\ [\{\langle a, d\rangle\}] & \text{if } X \text{ contains } \langle a, d\rangle \text{ but does not contain } \langle a, c\rangle \\ [\{\langle a, c\rangle, \langle a, d\rangle\}] & \text{if } X \text{ contains both } \langle a, c\rangle \text{ and } \langle a, d\rangle \\ [\{\quad\}] & \text{otherwise (i. e. } X \text{ neither contains } \langle a, c\rangle \text{ nor } \langle a, d\rangle) \end{cases}$$

is a retraction of $\mathbf{P}(D)$. $g$ satisfies Proposition 2.3 and is the member of $\{\mathbf{F}_R\}$ that (3) associates to the relation $R = \{\langle a, c\rangle, \langle a, d\rangle\}$. Furthermore, according to Proposition 2.2, $g$ is a retraction on $\mathbf{P}(R)$. Note that, though each function $f$ in $\{\mathbf{F}_R\}$ satisfies both Propositions 2.1 and 2.3, the converse does not hold, i. e. a function which satisfies both the above propositions is not necessarily the representative of some relation. For example, consider the following function $g'$:

$$g'([X])=\begin{cases}[\{\langle a, c\rangle\}] & \text{if } X \text{ contains } \langle a, c\rangle\\ [\{\langle a, d\rangle\}] & \text{if } X=\{\langle a, d\rangle\}\\ [\{\ \}] & \text{otherwise}\end{cases}$$

$g'$ is a retraction of $\mathbf{P}(D)$ and satisfies Proposition 2.3. However, Proposition 2.2 does not hold for any $R_D$. In particular, note that for $R$ above,

$$g'([X]) \neq [X \cap R] \text{ for each } X \text{ which has } \{\langle a, d\rangle\} \text{ as a proper subset.}$$

A comparison of $g'$ and $g$ shows that $g'$ is *less defined* than $g$, i. e. $g'=g'\circ g=g\circ g'$ [Scott76] ($\circ$ is function composition), or:

$$\forall X\in\mathbf{P}(D),\quad g'(X) \text{ defined on } X \text{ implies } g'(X) \subseteq g(X).$$

Definition 2.2: Given a retraction $\mathbf{F}_R$ with retract $U \subseteq \mathbf{P}(D)$, we define as *union set* of $\mathbf{F}_R$ the subset $R$ of $D$ such that: $R=\{x\in u \mid [u]\in U\}$.

Note that the union set of each retraction, $\mathbf{F}_R\in\{\mathbf{F}_R\}$, is (the set of points of) the relation $R_D$ of which the retraction is the representative.

Proposition 2.4:

$$\forall f:\quad \mathbf{P}(D)\rightarrow\mathbf{P}(D)$$

$f\in\{\mathbf{F}_R\}$ *iff f is the* greatest *retraction which has union set R, for a subset R of D, and satisfies Proposition* 2.3.

*Proof:* Let $f\in\{\mathbf{F}_R\}$ be the retraction with union set $R$, and $g$ be the greatest function which satisfies Proposition 2.3 and has union set $R$, then:

$$\forall [X]\in\mathbf{P}(D),\quad f([X])=[X\cap R]\subseteq g([X]) \text{ (because } g \text{ is the greatest))}$$

and

$$g([X])\subseteq [X] \text{ and } g([X])\subseteq [R] \text{ (because of Proposition 2.3),}$$

thus $g([X])\subseteq [X\cap R]$.

Proposition 2.4 completely characterizes the class $\{\mathbf{F}_R\}$ of retractions of $\mathbf{P}(D)$. Moreover, it shows how to formulate questions about the behaviour of a relation, in terms of function applications. As an example, let us consider the function $f_{\mathbf{app}}$ which (3) associates to the relation **app**, defined as the least relation which satisfies the following axioms (expressed in Horn clause logic):

$$\mathbf{app}(\mathrm{NIL}, y, y)\leftarrow .,\qquad \mathbf{app}(p.x, y, p.z)\leftarrow\mathbf{app}(x, y, z). \tag{4}$$

$f_{app}$ is a function of $\mathbf{P}(D) \to \mathbf{P}(D)$, where $D$ is the cartesian product List × List × List for some space List. We can assume List to be the space of all the lists of naturals, $p$ to be a variable on naturals and $x$, $y$, $z$ to be variables on List. Let List × K.List × H.K.List be the element of $\mathbf{P}(D)$ which denotes the subset of List × List × List which contains all the triples $\langle u, v, w \rangle$ such that $u$ is any list, $v$ is any list whose *car* is the natural $K$ and $w$ is any list whose car is the natural $H$ and whose *cadr* is the natural $K$, i.e. List × K.List × H.K.List $= [\{\langle u, v, w \rangle \mid v = K.v', w = H.K.w', u, v, w \in \text{List}\}]$. Then the application:

$$f_{\text{app}}(\text{List} \times \text{K.List} \times \text{H.K.List}) \tag{5}$$

computes the element of $\mathbf{P}(D)$ which denotes the subset of List × List × List which is the greatest subset of List × K.List × H.K.List and contains all the triples which make valid in (4) the following query:

$$\mathbf{app}(x, K.y, H.K.z) \tag{6}$$

where $x$, $y$, and $z$ are logical variables which range over List, and $H$ and $K$ are the above defined constants.

A comparison of (5) and (6) shows that the application in (5) corresponds to the query in (6) and, the value List × K.List × H.K.List in (5) corresponds to the triple of *Herbrand terms* in (6). However, List × K.List × H.K.List is merely notation, we will define in Section 3 a structure of sets which allows us to constructively express such values. Moreover, in Section 4 to each tuple of Herbrand terms, $T$, we associate a value (a constant expression), $E$, such that if $f$ is the retraction that (3) associates to the relation which is the minimal model of a predicate $R$ in a (Horn clause) logic theory, then $f(E)$ computes the element of $\mathbf{P}(D)$ which denotes the set of all the values in $D$ which make valid $R(T)$ in the theory.

Finally, if our sets are equipped with suitable operators for product and projection, Proposition 2.4 models in a functional programming paradigm the *program invertibility* feature of predicative languages. For instance, the set of lists to which the variable $x$ in (6) can be instantiated to make valid (6) in (4), can be obtained from (5) by the projection of $f_{\text{app}}(\text{List} \times \text{K.List} \times \text{H.K.List})$ on the first component of the cartesian product List × List × List.

Though $\{\mathbf{F}_R\}$ is only a sub-class of the retracts, it is closed under function composition. Thus, the following propositions hold.

PROPOSITION 2.5:

$$\forall f, g: \quad \mathbf{P}(D) \to \mathbf{P}(D) \in \{\mathbf{F}_R\}, \qquad f \circ g, \quad g \circ f \in \{\mathbf{F}_R\}.$$

*Proof:* Let $f$ and $g$ be the functions that (3) associates to $R_f$ and $R_g$, respectively, then:

$$R_f \cap R_g \text{ is a relation on } D,$$

and

$$\forall [X] \in \mathbf{P}(D), \quad [R_f \cap R_g \cap X] = f \circ g([X]).$$

PROPOSITION 2.6:

$$\forall f, g: \quad \mathbf{P}(D) \to \mathbf{P}(D) \in \{\mathbf{F}_R\}, \qquad f \circ g = g \circ f.$$

*Proof.* Let $f$ and $g$ be the functions that (3) associates to $R_f$ and $R_g$, respectively, then:

$$\forall [X] \in \mathbf{P}(D), \quad f \circ g([X]) = f(g([X])) = [R_f \cap (R_g \cap X)].$$

In contrast to Proposition 2.5, Proposition 2.6 does not hold for the entire class of the retractions. It says that, from a denotational point of view, the ordering on the composition of functions in $\{\mathbf{F}_R\}$ is unessential. Obviously, that is not true when operational semantics is considered. Operationally, Proposition 2.6 allows to model in a functional paradigm the *declarative* (absence of control) feature of predicative languages.

## 3. A FIRST ORDER FUNCTIONAL LANGUAGE

Our treatment of retractions will be discussed in a first order functional language. The language is essentially Lambda calculus restricted to first order. Lambda calculus is here used as the *abstract functional language* where the concepts of set and of retraction are stated in a clean and simple way.

The language alphabet is a quadruple $A = \langle D, V_D, P, V_p \rangle$ where $D$ is the set of the language data, $P$ is the set of the language primitive operators, $V_D$ and $V_p$ are denumerable set of variables ranging over $D$ and first order functions on $D$, respectively. The language expressions are all the closed forms which can be built starting from $D$ plus $P$, and by $\lambda$-abstraction and application of the fixed point operator, $\mathbf{Y}$ [Milner72]. Each expression has meaning according to $\alpha$, $\beta$, $\mathbf{Y}$ reductions, and to the semantics of the primitive operators. Programs are expressions.

The formal definition of the language syntax and semantics is deferred to Appendix I, while an example of the definition and of the evaluation of a program is reported below. Next Section is devoted to the definition of the structure of the language domain, i. e. $D + P$. In principle, the quadruple $A$ could be arbitrarily set giving origin to several (first order) languages which essentially differ in the language data domain. The structure of the data domain is a relevant point of our construction because it characterizes the structure of the sets which we use to compute with retractions.

*Example* 3.1: Let us consider the following program in the extended syntax

$$\begin{array}{l} f1(x)=x+y \quad \text{where} \quad 1=y; \\ f2(x, y)=f1(x)+y; \\ 7 * x \quad \text{where} \quad f2(2, 3)=x. \end{array}$$

It corresponds to the expression:

$$((\lambda x.7 * x)((\lambda xy.((\lambda x.((\lambda y.x+y)\,1))\,x)+y)\,2\,3))$$

which is a closed form and evaluates to

$$\begin{array}{c} 7 * ((\lambda xy.((\lambda x.((\lambda y.x+y)\,1))\,x)+y)\,2\,3) \\ 7 * ((\lambda xy.((\lambda y.x+y)\,1)+y)\,2\,3) \\ 7 * ((\lambda xy.(x+1)+y)\,2\,3) \\ 7 * ((2+1)+3). \end{array}$$

### 3.1. The values domain: $HU_c^*$.

Because of the complete separation between values domain and functions domain, we can freely enrich the language with the definition of its set of values, $D$, and of the corresponding set of operators $P$. As pointed out in the language definition, these operators will be primitives for the language, and expressions which contain occurrences of these operators will be reduced by $\alpha$, $\beta$, and $\mathbf{Y}$ reductions and, if needs, according to the semantics of the operators.

To model predicates (of Horn clause logic) by retractions, in the choice of $D$ we can limit ourselves to relations on $D$'s which are (cartesian) powers of the Herbrand Universe, $HU_c$.

We briefly recall that $HU_c$, the Herbrand Universe built from $C=\{\underline{Ci_k}\}$ (finite set of constructors $\underline{Ci}$ of arity $k$, such that $C$ includes at least one constructor of arity 0), is the minimum set of terms which satisfies both:

- $\forall\, \underline{Ci_0} \in C, \quad \underline{Ci_0} \in HU_c$
- $\forall\, \underline{Ci_k} \in C, \quad \forall\, t_1, \ldots, t_k \in HU_c, \qquad \underline{Ci_k}(t_1, \ldots, t_k) \in HU_c.$

Given $HU_c$, we define $HU_c^*$ be the (infinite) union of the parts with Scott topology, of the (finite cartesian) powers of $HU_c$. Formally, we have

$HU_c^* = \bigcup_{i \in \mathbf{N}} \mathbf{P}(HU_c^i)$. Definition 3.1 provides a constructive definition for $HU_c^*$.

DEFINITION 3.1 ($HU_c^*$):

$\forall i \in \mathbf{N}^+$, let $HU_{T_i} = \{\langle t_1, \ldots, t_i \rangle \mid t_1, \ldots, t_i \in HU_c \cup \{\varnothing\}\}$ ($i$-tuples),

then $HU_c^*$ is the minimal set of values which satisfies both:

– let $HU_F = \bigcup_{i \in \mathbf{N}} HU_{F_i}$, $HU_F \subset HU_c^*$ (finite sets of $i$-tuples);

– let $HU_\omega = \bigcup_{i \in \mathbf{N}} HU_{\omega_i}$, $HU_\omega \subset HU_c^*$ (infinite sets of $i$-tuples);

where:

+ $HU_{F_i} = \bigcup_{j \in \mathbf{N}} (HU_{T_i})^j_\bullet$

+ $(HU_{T_i})^0_\bullet = \{\varnothing\}$

$(HU_{T_i})^j_\bullet = (HU_{T_i}) \overset{X}{\bullet} (HU_{T_i})^{j-1}_\bullet = \{t_1 \bullet t_2 \mid t_1 \in HU_{T_i},\ t_2 \in (HU_{T_i})^{j-1}_\bullet\}$

+ $HU_{\omega_i} = \{\mathbf{Sup}\{t^j\}\}$, being $\{t^j\}$ a set of members of $HU_{F_i}$, such that:

$$-\forall j \in N, \quad t^j \in (HU_{T_i})^j_\bullet,$$

and

$$-\exists\, t \in HU_{T_i} \text{ such that } t^j = t \bullet t^{j-1}$$

+ $\langle - \rangle$ and $\bullet$ are the tupling and set-constructor operator, respectively.

Though the structure of $HU_c^*$ depends on the properties of $\langle - \rangle$ and $\bullet$, we can see that $HU_c^*$ is a family, $\{HU_i^*\}$, indexed by the classes of tupling, $HU_{T_i}$. Moreover, each $HU_i^*$ contains:

– all the $i$-tuples of elements of $HU_c$, i. e. $HU_{T_i}$;

– all the values obtained by finitely many applications of $\bullet$ to $i$-tuples, i. e. $HU_{F_i}$,

– all the values which are computed by infinitely many applications of $\bullet$ to $i$-tuples and can be obtained as limit of values of $HU_{F_i}$, i. e. $HU_{\omega_i}$.

*Example* 3.2: Let $\underline{0}$ and $\underline{S}$ in $C = \{\underline{0}, \underline{S}\}$ be constructors of arity 0 and 1 respectively, then:

– $\underline{0}$, $\underline{S(0)}$, $\underline{S(S(0))} \in HU_c$;

– $\langle \underline{0}, \underline{0} \rangle$, $\langle \underline{0}, \underline{S(0)} \rangle$, $\langle \underline{0}, \underline{S(S(0))} \rangle \in HU_{T_2}$;

– $\langle \underline{0}, \underline{0} \rangle$, $\langle \underline{0}, \underline{0} \rangle \bullet \langle \underline{0}, \underline{S(0)} \rangle \in HU_{F_2}$;

$(\langle \underline{0}, \underline{0} \rangle \bullet \langle \underline{0}, \underline{S(0)} \rangle) \bullet \langle \underline{0}\ \underline{S(S(0))} \rangle \in HU_{F_2}$;

— the infinite sequence

$$(\ldots((\langle \underline{0}, \underline{0}\rangle \bullet \langle \underline{0}, \underline{S(0)}\rangle) \bullet \langle \underline{0}, \underline{S(S(0))}\rangle) \bullet \ldots \bullet \langle \underline{0}, S^k(0)\rangle) \bullet \ldots$$

computes an element of $HU_{\omega 2}$ [$S^1(0)$ stands for $k\,\underline{S}'s$ followed by $\underline{0}$].

DEFINITION 3.2 ($\langle - \rangle$): Technically, $\langle - \rangle$ is a family of operators, each one indexed by its arity (any positive integer). It is a function $HU_c \times \ldots \times HU_c \to HU_{T_i}$, it computes the minimal congruence on $HU_c^*$ which satisfies the following axiom:

$\forall t_1, \ldots, t_k, \quad t'_1, \ldots, t'_k \in HU,$

$$\langle t_1, \ldots, t_k \rangle = \langle t'_1, \ldots, t'_k \rangle \quad \text{iff } \forall i \in [1, k], \qquad t_i = t'_i$$

or $t_n = \varnothing$, $t'_m = \varnothing$ for some pair $n$, $m$ in $[1, k]$.

Notational remark (bottom element, $\langle \varnothing, \ldots, \varnothing \rangle$, 1-tuple element, $\langle t \rangle$) we will use the notation $\varnothing$ for $\langle t_1, \ldots, \varnothing, \ldots, t_k \rangle$. Though this use is not technically correct (because $\varnothing$ is an element of $HU_{T_1}$, and we should use $\langle \varnothing, \ldots, \varnothing \rangle \in HU_{T_K}$), it equally expresses the theoretical meaning of the above axiom. Another notational freedom is the use of $t$ for the 1-tple $\langle t \rangle$.

DEFINITION 3.3 ($\bullet$, set-constructor): It is a function

$$HU_c^* \times \ldots \times HU_c^* \to HU_c^*,$$

it computes the minimal congruence on $HU_c^*$ which satisfies the following axioms:

(1-idempotent) $t \bullet t = t$;

(2-commutative) $t_1 \bullet t_2 = t_2 \bullet t_1$;

(3-associative) $(t_1 \bullet t_2) \bullet t_3 = t_1 \bullet (t_2 \bullet t_3)$;

(4-zero) $t \bullet \varnothing = t$;

(5-continuous) $t \bullet \mathbf{Sup}\{t^j\} = \mathbf{Sup}\{t \bullet t^j\}$.

The definition of $\bullet$ completes the definition of the structure of the language values domain. The elements of $HU_c^*$ form a model for subsets of the powers of $HU_c$: $\varnothing$ represents the empty set, $\bullet$ represents (possibly infinite but computable) set union, and finally, elements of $HU_{T_i}$, $HU_{F_i}$, $HU_{\omega_i}$ represent singleton, finite and infinite (computable) sets of $i$-tuples of values in $HU_c$, respectively. Note that, $\bullet$ differs from set union because it is only defined on sets of tuples of the same order $i$.

The elements of $HU_c^*$ are partially ordered by the relation $\subseteqq$, defined as follows.

DEFINITION 3.4 ($\subseteqq$): Let $x$ and $y$ be elements of $HU_c^*$, then $x \subseteqq y$ iff $x$, $y \in HU_i^*$ for some $i \in \mathbf{N}$ and,

either: both are elements of $HU_{F_i}$ and $\exists w \in HU_{F_i}$ such that $y = w \bullet x$;
or: $y$ is element of $HU_{\omega_i}$ and, assumed $x = \mathbf{Sup}\{x^i\}$ and $y = \mathbf{Sup}\{y^j\}$,

$$\forall x^i \in \{x^i\}, \qquad \exists y^j \in \{y^j\} \text{ such that } x^i \subseteqq y^j.$$

*Example* 3.3: Let us consider the $C$ of Example 3.2, and let us define: $f(x) = x \bullet f(\underline{S}(x))$. It is easy to note that $f$ is not in $\{\mathbf{F}_R\}$ and is not a retraction. However it is a computable function and can be expressed in our language. In particular, $f(\underline{0})$ computes the sequence

$$\underline{0} \bullet \underline{S(0)} \bullet \underline{S(S(0))} \bullet \ldots \bullet S^k(0) \bullet \ldots$$

which is an element of $HU_{\omega_1}$.

Note that $\forall x \in HU_c^*$, $\varnothing \subseteqq x$. Furthermore, $\subseteqq$ is a *well-founded* ordering on the elements of $HU_F$.

### 3.2. Operators on $HU_c^*$

Although the values domain is completely defined, we need some additional operators. Actually, the elements of $HU_\omega$ can be expressed in the language by expressions which enumerate all the finite approximations [as was the case for $f(\underline{0})$ in the example 3.3]. $HU_\omega$ contains a class of elements which can be expressed without the use of limit operations. This class is sub-class of the recursive sets of tuples of $HU_c$. Moreover, we will see in section 4 that elements of this class are in correspondence with tuples of Herbrand terms. We enrich the set of operators on $HU_c^*$ in order to express the elements of this class in a combinatory form.

#### 3.2.1. *Constructors and inverses*

We associate to each $k$-arity constructor $\underline{Ci_k}$, a 2-arity function (operator) $\mathbf{c}\,i_k$ which, applied to an index $j$ and to a tuple $u$ of $HU_c^*$, behaves like $\underline{Ci_k}$, if $j = 1$ and $u$ is an element of $HU_{T_k}$, i.e. $u$ denotes a singleton of $HU_c^k$, otherwise it computes the element which denotes the set obtained by applying $\underline{Ci_k}$ to the $k$-subtuple at the position $j$ of each member of the set denoted by $u$. Formally, $\mathbf{c}\,i_k$ is defined by the following:

Definition 3.5 ($\mathbf{c}\,i_k$, extension of constructors): Let $\underline{Ci_k}$ be a constructor of arity $k \neq 0$. Then $\mathbf{c}\,i_k$ is a function $\mathbf{N}^+ \times HU_c^* \to HU_c^*$. It computes the minimal congruence which satisfies the following axioms:

- $\mathbf{c}\,i_k(j, \langle t_1, \ldots, t_n \rangle) = \varnothing$ iff $n \geqq k + j - 1$ and $t_h = \varnothing$ for some $h \in [1, n]$;
- $\mathbf{c}\,i_k(j, \langle t_1, \ldots, t_{j-1}, t'_1, \ldots, t'_k, t_{j+1}, \ldots, t_n \rangle)$
$$= \langle t_1, \ldots, t_{j-1}, \underline{Ci_k}(t'_1, \ldots, t'_k), t_{j+1}, \ldots, t_n \rangle;$$

— $\mathbf{c}\, i_k(j, t_1 \bullet t_2) = \mathbf{c}\, i_k(j, t_1) \bullet \mathbf{c}\, i_k(j, t_2)$;

— $\mathbf{c}\, i_k(j, \mathbf{Sup}\{t^i\}) = \mathbf{Sup}\{\mathbf{c}\, i_k(j, t^i)\}$.

Thus, for each $u \in \mathrm{HU}^*_{k+h}$ (i. e. for each set of elements of the power of order $k+h$), $\mathbf{c}\, i_k(j, u)$, such that $h \geqq j-1$, computes the element of $\mathrm{HU}^*_{h+1}$ (i. e. the set of elements of the power of order $h+1$) obtained by applying $\underline{Ci_k}$ to the projection on $\mathrm{HU}^*_k$ of the $j, \ldots, j+k-1$ components of each member in the set denoted by $u$. Though not explicitely given by the above axioms, $\mathbf{c}\, i_k(j, u)$ will be considered *undefined*, if $u$ is such that for no $h \geqq j-1$, $u \in \mathrm{HU}^*_{k+h}$ (i. e. $\# u < k+j-1$).

**Notational remark** (tupling class, $\# t$)

If $t \in \mathrm{HU}^*_k$, we denote by $\# t$ the class of tupling of $t$, i. e. $k$.

*Example* 3.4: Let us consider the $C$ of Example 3.2, and define: $f(x) = x \bullet \mathbf{S}(2, f(x))$. $f(\langle \underline{0}, \underline{0} \rangle)$ computes the sequence in Example 3.2. Note that, the function $f$ is undefined (only) on $\mathrm{HU}^*_1$.

Note also that the constructor operators $\mathbf{c}\, i_k$ induce a further ordering relation on the elements of $\mathrm{HU}^*_k$.

Definition 3.6 ($\ll$): Let $x$ and $y$ be elements of $\mathrm{HU}^*_c$. Then

$$x \ll y \text{ iff } x,\ y \in \mathrm{HU}^*_i \text{ for some } j \in \mathbf{N} \text{ and constructor } \mathbf{c}\, i_k,$$
$$y = \mathbf{c}\, i_k(j, u) \text{ for some } u \text{ such that } x \subseteq u.$$

*Example* 3.5: Let us consider the $C$ and $f$ of Example 3.3, and let $t$ be the element of $\mathrm{HU}^*_2$ computed by $f(\langle \underline{0}, \mathbf{S}(\mathbf{S}(\underline{0})) \rangle)$. Then $t$ has only two less defined elements under $\ll$:

$$f(\langle \underline{0}, \underline{0} \rangle) \ll f(\langle \underline{0}, \mathbf{S}(\underline{0}) \rangle) \ll f(\langle \underline{0}, \mathbf{S}(\mathbf{S}(\underline{0})) \rangle)$$

Noting that $\ll$ is a *well-founded* ordering on all the elements of $\mathrm{HU}^*_c$ and it allows *structural induction* based reasoning on the values computed by **Y**-reductions. We will use this relation in Section 6, to prove the equivalence between a retraction expressed in our language and the relation denoted by a predicate in a Horn clause theory.

Associated to each $\mathbf{c}\, i_k$, we have the *inverse function*, denoted by $\mathbf{c}\, i_k \downarrow$. It is formally defined as follows.

Definition 3.7 ($\mathbf{c}\, i_k \downarrow$, constructor inverses): Let $\underline{Ci_k}$ be a constructor of arity $k \neq 0$. Then $\mathbf{c}\, i_k \downarrow$ is a function $\mathbf{N}^+ \times \mathrm{HU}^*_c \to \mathrm{HU}^*_c$, it computes the minimal congruence which satisfies the following axioms:

– $\mathbf{c}\, i_k \downarrow (j, \langle t_1, \ldots, t_n \rangle) = \varnothing$ iff $n \geqq j$ and, $t_h = \varnothing$ for some $h \in [1, n]$ or $t_j = \mathbf{c}'\, i_k (t'_1, \ldots, t'_k$ and $ci_k \neq c'\, i_k$

– $\mathbf{c}\, i_k \downarrow (j, \langle t_1, \ldots, t_{j-1}, \underline{Ci_k}(t'_1, \ldots, t'_k), t_{j+1}, \ldots, t_n \rangle)$

$$= \langle t_1, \ldots, t_{j-1}, t'_1, \ldots, t'_k, t_{j+1}, \ldots, t_n \rangle$$

– $\mathbf{c}\, i_k \downarrow (j, t_1 \bullet t_2) = \mathbf{c}\, i_k \downarrow (j, t_1) \bullet \mathbf{c}\, i_k \downarrow (j, t_2)$

– $\mathbf{c}\, i_k \downarrow (j, \mathbf{Sup}\{t^i\}) = \mathbf{Sup}\{\mathbf{c}\, i_k \downarrow (j, t^i)\}$.

Noting that $\forall j, u$, $\mathbf{c}\, i_k \downarrow (\mathbf{c}\, i_k (j, u)) = u$, but $\mathbf{c}\, i_k (\mathbf{c}\, i_k \downarrow (j, u)) \sqsubseteq u$, then $\mathbf{c}\, i_k \downarrow$ is a *weak* form of inverse, i. e. the following property holds.

PROPERTY 3.1:

– For each constructor function: $\mathbf{c}\, i_k \downarrow \circ\, \mathbf{c}\, i_k \neq \mathbf{c}\, i_k \circ \mathbf{c}\, i_k \downarrow$.

– Moreover, for each $u$, $\mathbf{c}\, i_k \downarrow (j, u) \ll u$.

The functions $\mathbf{c}\, i_k$ and $\mathbf{c}\, i_k \downarrow$ extend to (sets of) tuples of terms the operations of term construction and subterm selection.

*Notational remark* ($\mathbf{c}\, i_k(u)$): $\mathbf{c}\, i_k (1, u)$ (resp. $\mathbf{c}\, i_k \downarrow (1, u)$) will also be denoted by $\mathbf{c}\, i_k (u)$ (resp. $\mathbf{c}\, i_k \downarrow (u)$) if $\# u = k$.

3.2.2. *Cartesian product* $\otimes - \otimes$

The cartesian product allows to compute the product of elements of $\mathrm{HU}_c^*$. Technically, it is a family of operators, indexed by the arity (any natural). It is a function

$$\mathrm{HU}_{i1}^* \times \ldots \times \mathrm{HU}_{in}^* \to \mathrm{HU}_{i1+\ldots+in}^*,$$

it computes the minimal congruence which satisfies the following axioms:

(1-associative)

$$\otimes\, t_1, \ldots, t_{j-1}, \otimes\, t'_1, \ldots, t'_k \otimes, t_{j+1}, \ldots, t_n \otimes = \otimes\, t_1, \ldots, t_{j-1}, t'_1, \ldots, t'_k, t_{j+1}, \ldots, t_n \otimes;$$

(2-singleton)

$$\otimes\, t_1, \ldots, t_n \otimes = \langle t_1, \ldots, t_n \rangle \quad \text{iff} \quad \forall i \in [1, n],\ t_i \in \mathrm{HU}_{T_1};$$

(3-finite set)

$$\otimes\, t_1, \ldots, t'_i \bullet t''_i, \ldots, t_n \otimes = \otimes\, t_1, \ldots, t'_i, \ldots, t_n \otimes \bullet \otimes\, t_1, \ldots, t''_i, \ldots, t_n \otimes;$$

(4-continuous)

$$\otimes\, t_1, \ldots, \mathbf{Sup}\{t^j\}_i, \ldots, t_n \otimes = \mathbf{Sup}\{\otimes\, t_1, \ldots, t^j, \ldots, t_n \otimes\}.$$

Cartesian product is powerful enough to express the element of $\mathrm{HU}_c^*$ which denotes the set (containing all the terms of) $\mathrm{HU}_c$. As a matter of fact, let $C$

be the following set of constructors

$$C=\{\underline{C1_0}, \ldots, \underline{Cn_0}, \underline{C1_1}, \ldots, \underline{Cn_1}, \ldots, \underline{C1_k}, \ldots, \underline{Cn_k}\}$$

then,

$$\pi: \quad \mathbf{Y}\,\pi.\mathbf{c1}_0 \bullet \ldots \bullet \mathbf{c}n_0 \bullet \mathbf{c1}_1(\pi) \bullet \ldots \bullet \mathbf{c}n_1(\pi) \bullet \ldots$$
$$\bullet\, \mathbf{c}\,1_k(\otimes\,\pi, \ldots, \pi\,\otimes) \bullet \ldots \bullet \mathbf{c}\,n_k(\otimes\,\pi, \ldots, \pi\,\otimes)$$

$\pi$ is a constant function and since $\mathbf{Y}$ is the fixed-point operator, $\pi$ computes the element of $\mathrm{HU}_1^*$ which contains all the terms in $\mathrm{HU}_c$. In expressing predicates and queries through retractions and functional applications, $\pi$ will be used as a constant expression (an additional operator) which models unbound logic variables. $\pi$ is the 1-tuple *top-element* of the elements of $\mathrm{HU}_1^*$, hence the following property holds.

PROPERTY 3.2:

– $\forall\, \underline{Ci_k} \in C$, $\mathbf{c}\,i_k \downarrow (\pi) = \otimes\,\pi, \ldots, \pi\,\otimes$;

– $\forall\, k \in \mathbf{N}^+$, $\otimes\,\pi, \ldots, \pi\,\otimes \in \mathrm{HU}_k^*$;

– $\forall\, n \in \mathbf{N}^+$, $\forall\, u \in \mathrm{HU}_k^*$, $u \subseteq \otimes\,\pi, \ldots, \pi\,\otimes$

(where $\otimes - \otimes$ has arity $k$).

Finally, note that to each function $f_n$ of arity $n$ which maps from $\mathrm{HU}_{k1}^* \times \ldots \times \mathrm{HU}_{kn}^*$ onto $\mathrm{HU}_k^*$, we can associate a function $g_1$ of arity 1 which maps $\mathrm{HU}_{k1+\ldots+kn}^*$ onto $\mathrm{HU}_k^*$ such that:

$$\forall\, x1, \ldots, xn, \quad f_n(x1, \ldots, xn) = g_1(\otimes\, x1, \ldots, xn\,\otimes).$$

### 3.2.3. *Projection* Pr

The projection operator allows us to move from (elements of) cartesian products to (elements of) subproducts. It is a function $\mathbf{N}^+ \times \mathbf{N}^+ \times \mathrm{HU}_c^* \to \mathrm{HU}_c^*$, it computes the minimal congruence which satisfies the following axioms:

(1-singleton)

$$\mathbf{Pr}(j, k, \langle t_1, \ldots, t_j, \ldots, t_{k+j-1}, \ldots, t_n \rangle) = \langle t_j, \ldots, t_{k+j-1} \rangle$$

iff $\forall\, h \in [1, n]$, $t_h \neq \varnothing$;

$$\mathbf{Pr}(j, k, \langle t_1, \ldots, \varnothing, \ldots, t_n \rangle) = \varnothing \qquad \text{iff} \quad n \geqq k+j-1;$$

(2-finite set)

$$\mathbf{Pr}(j, k, t_1 \bullet t_2) = \mathbf{Pr}(j, k, t_1) \bullet \mathbf{Pr}(j, k, t_2);$$

(3-continuous)

$$\mathbf{Pr}(j, k, \mathbf{Sup}\{t^i\}) = \mathbf{Sup}\{\mathbf{Pr}(j, k, t^i)\}.$$

As is the case for functions $\mathbf{c}\,i_k$, in the following $\mathbf{Pr}(j, k, u)$ will be considered undefined if $u$ is such that for no $h \geqq j-1$, $u \in \mathrm{HU}^*_{k+h}$.

*Example* 3.6: Let $\underline{C1}_0$, $\underline{C2}_0$ be two constructors of arity 0 and $\underline{C1}_2$ be a constructor of arity 2:

$$\mathbf{Pr}(2, 1, \mathbf{c}\,1_2 \downarrow (2, \otimes\,\pi, \mathbf{c}\,1_2(\mathbf{c}\,1_0, \mathbf{c}\,2_0)\otimes)) = \langle \underline{C1}_0, \underline{C2}_0 \rangle.$$

**Pr** inherits from set theory the following property.

PROPERTY 3.3: $\forall\, t \in \mathrm{HU}^*_k$, $\forall\, t1, \ldots, tk \in \mathrm{HU}^*_1$, such that $ti = \mathbf{Pr}(i, 1, t)$ then $t \subseteq \mathbf{Pr}(\otimes\, t1, \ldots, tk\,\otimes)$.

3.2.4. *Injection* **In**

The injection operator allows us to lift on the order of cartesian products. It is a function $\mathrm{N}^+ \times \mathrm{HU}^*_1 \to \mathrm{HU}^*_c$, it computes the minimal congruence which satisfies the following axioms:

(1-singleton):

$$\mathbf{In}(i, t) = \langle t_1, \ldots, t_i \rangle \qquad \text{such that} \quad \forall\, h \in [1, i], \quad t_h = t$$

(2-finite set)

$$\mathbf{In}(i, t_1 \bullet t_2) = \mathbf{In}(i, t_1) \bullet \mathbf{In}(i, t_2);$$

(3-continuous)

$$\mathbf{In}(i, \mathbf{Sup}\{t^j\}) = \mathbf{Sup}\{\mathbf{In}(i, t^j)\}.$$

Note that **In**jection is only defined on sets containing 1-tuples. Roughly speaking, this function allows to compute the elements of $\mathrm{HU}^*_c$ which denote sets containing only elements of the form $\langle x, \ldots, x \rangle$, where $x$ is an element of $\mathrm{HU}_c$.

*Example* 3.7: Let $x.x$ be the following constant function:

$$x.x\colon \quad \mathbf{In}(2, \pi).$$

Members of $x.x$ are all the pairs of the form $\langle x, x \rangle$ such that $x$ is a member of $\pi$ (i.e. $x \in \mathrm{HU}_c$).

In expressing predicates and queries through retractions and functional applications, injection allows to model constraints which in logic programming are expressed by the multiple occurrence of variables in the formula. For instance in the atomic formula $P(x, S(x))$, the multiple occurrence of $x$ constrains both the arguments of the predicate $P$, and $\mathbf{S}(2, \mathbf{In}(2, \pi))$ expresses the set of all the terms (of the Herbrand Universe) which satisfy such constraints (*see* Section 4).

3.2.5. *Permutation* **Pe**

It is a function of $\mathbf{N}_k \times \mathrm{HU}_k^* \to \mathrm{HU}_k^*$. Technically, it is an $i$-indexed family of operators (one for each class of tupling). $\mathbf{N}_k$ is the finite space containing all the permutations of the integers in the natural interval $[1, k]$.

**Pe** computes the minimal congruence which satisfies the following axioms:

(1-singleton)

$$\mathbf{Pe}(n_1 \ldots n_k, \langle t_1, \ldots, t_k \rangle) = \langle tn_1, \ldots, tn_k \rangle;$$

(2-finite set)

$$\mathbf{Pe}(n_1 \ldots n_k, t_1 \bullet t_2) = \mathbf{Pe}(n_1 \ldots n_k, t_1) \bullet \mathbf{Pe}(n_1 \ldots n_k, t_2);$$

(3-continuous)

$$\mathbf{Pe}(n_1 \ldots n_k, \mathbf{Sup}\{t^j\}) = \mathbf{Sup}\{\mathbf{Pe}(n_1 \ldots n_k, t^j)\}.$$

*Notational remark* ($n!$, $n!(i)$): From now on we will denote by $n!$ a permutation of the first $n$ integers, and by $n!(i)$ the $i$-th integer of that permutation.

*Example* 3.8: Let $x.y.x$ be the following constant function:

$$x.y.x: \quad \mathbf{Pe}(1\,3\,2, \otimes \mathbf{In}(2, \pi), \pi \otimes).$$

Members of $x.y.x$ are all the 3-tuples of the form $\langle x, y, x \rangle$ such that the first and third components are equal (and, possibly, different from the second one), and $x$, $y$ range over all the elements of $\mathrm{HU}_c$.

**Pe** completes the list of the set operators we need to introduce in the next section, the concept of constant expression and througt it, to extract logical variables from terms. Then definition 3.9 completes the language semantics. Expression evaluations are $\alpha$, $\beta$ and $\mathbf{Y}$ reductions modulo the relation $\approx$. The definition of the operational semantics of $\approx$ is deferred to [Bellia88], where we will introduce the concept of normal form for constant expression and we define a system of reduction rules to compute normal forms.

DEFINITION 3.9 ($\approx$): The *equivalence relation* $\approx$ on expressions is defined as follows. If $E1$ and $E2$ are two constant expressions then $E1 \approx E2$ if and only if $E1$ and $E2$ are in the same congruence class induced on $\mathrm{HU}_c^*$, from the set operators above.

## 4. A COMBINATORY FORMULATION OF THE HERBRAND TERMS

We interpret (ground and non-ground) Herbrand terms as expressions which denote sets. We recall that Herbrand terms are built exactly as the elements of $\mathrm{HU}_c$, starting from a set $\{\underline{Ci_k}\} \cup \{\underline{x}\}$, where $\{\underline{x}\}$ is a denumerable set of variable symbols.

A Herbrand term, $h$, on a Universe $HU_c$, denotes the subset of $HU_c$ containing all the *ground instances* of $h$. All these sets are recursive sets and, being the set of all (computable) subsets of $HU_c$ contained in $HU_c^*$ (i. e. $HU_1^* \subset HU_c^*$), Herbrand terms denote elements of $HU_c^*$. Moreover, being computable the set denoted by a term $h$, methods to enumerate all the ground instances of $h$ are well known. General algorithms which, given $h$, enumerates all its ground instances could easily be defined. However, our main interest is to associate to each $h$ its denotation in $HU_c^*$ which, in case of need, enumerates all the ground instances of $h$.

Definition 4.1 (constant expressions)

1. 0-arity constructors and $\langle - \rangle$ applied to 0-arity cpnstructors, are constant expressions;

2. $\varnothing$ and $\pi$ are constant expressions;

3. If $E_1, \ldots, E_n$ are $n$ constant expressions then $E_1 \bullet \ldots \bullet E_n$ and $\otimes E_1, \ldots, E_n \otimes$ are constant expressions;

4. If $E$ is a constant expression then $\mathbf{In}(k, E)$, $\mathbf{c}\, i_k(j, E)$, $\mathbf{c}\, i_k \downarrow (j, E)$, $\mathbf{Pr}(i, j, E)$, $\mathbf{Pe}(n!, E)$ are constant expressions;

1-5 are the only *constant expressions.*

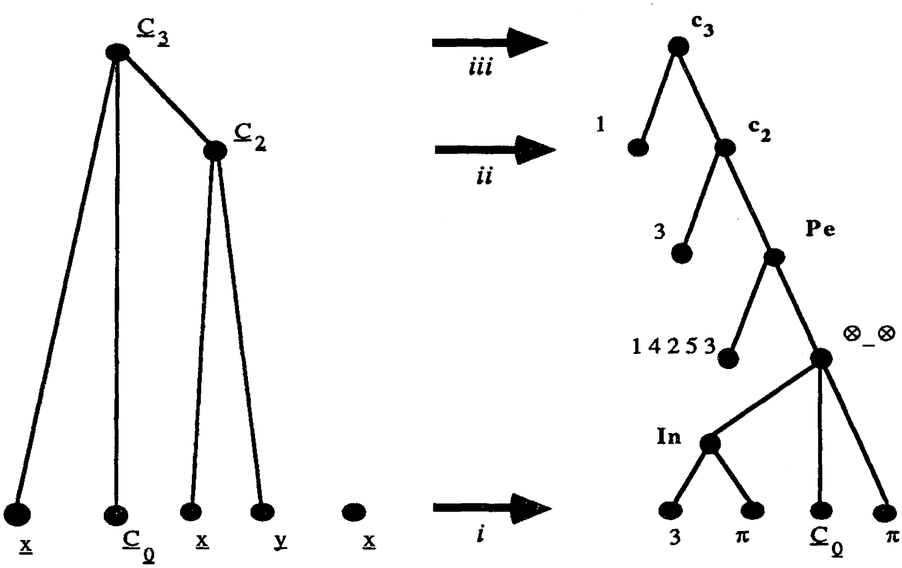
Note that, constant expressions are expressions which do not contain variables and are not infinite applications of $\bullet$, i. e. they are combinatory forms of our set operators.

PROPOSITION 4.1: *There exists a function* $\eta$ *which associates to each tuple* $H = h_1, \ldots, h_n$ *of Herbrand Terms a* constant expression E *on* $HU_n^*$, *containing only occurrences of the function* $\pi$ *and of* 0*-arity constructors, and applications of the operators* $\mathbf{c}\, i_k$, $\otimes - \otimes$, $\mathbf{In}$, *and* $\mathbf{Pe}$. H *and* E *denote the same subset of the cartesian product of order n.*

A constructive proof is reported in Appendix II: We define a function which maps tuples of Herbrand terms into constant expressions, and we show that it satisfies the proposition. Examples of this fact are the constant expression $\pi$ itself which has the same denotation of the single variable term, e. g. $\underline{x}$, the constant expression $\mathbf{In}(2, \pi)$ (*see* Example 3.7) which has the same denotation of the pair of terms $\underline{x, x}$, and the constant expression $\mathbf{Pe}(1\,3\,2, \otimes \mathbf{In}(2, \pi), \pi \otimes)$ (*see* Example 3.8) which has the same denotation of the triple $\underline{x, y, x}$.

*Example* 4.1.: If $C = \{\underline{C_0}, \underline{C_2}, \underline{C_3}\}$ is the constructor set in $HU_c^*$ and $\underline{C_3}(\underline{x, C_0, C_2(x, y)}), \underline{x}$ is a pair of Herbrand term, then, according to the definition of the function $\eta$, as is given in Appendix II, $\eta(\underline{C_3(x, C_0, C_2(x, y)), x})$ is computed as follows.

- by $B$, $\mathbf{c}_3(1, \eta(\underline{x, C_0, C_2(x, y), x}))$;
- by $B$, $\mathbf{c}_3(1, \mathbf{c}_2(3, \eta(\underline{x, C_0, x, y, x})))$;
- by Aii, $\mathbf{c}_3(1, \mathbf{c}_2(3, \mathbf{Pe}(1\,4\,2\,5\,3, \otimes \mathbf{In}(3, \pi), \underline{C_0}, \pi \otimes)))$.




$\underline{C_3(x, C_0, C_2(x, y)), x}$ $\mathbf{c}_3(1, \mathbf{c}_2(3, \mathbf{Pe}(1\,4\,2\,5\,3, \otimes \mathbf{In}(3, \pi), \underline{C_0}, \pi \otimes)))$

**Figure 1. — Transformation of tuples of Herbrand terms into constant expressions.**

Figure 1 shows the tree structure of the tuple $\underline{C_0(x, C_0, C_2(x, y))}$, $x$ and of the corresponding constant expression computed by $\eta$.

In logic languages, unification is used to compare two (or more) Herbrand Terms. Unification computes the Mgu, if any, or fails. Under our interpretation of Herbrand Therms, the following proposition holds.

PROPOSITION 4.2 : *Let $H = h_1, \ldots, h_n$ be a tuple of Herbrand terms, for each* instantiation function $\vartheta$:

$$\eta(H.\vartheta) \subseteq \eta(H) \quad [\textit{or simply}, H.\vartheta \subseteq H].$$

*Proof.* : Let $a_1, \ldots, a_n$ be any ground instance of $H.\vartheta$, that is:

$$\exists \Phi, \quad \text{instantiation function, such that } (H.\vartheta).\Phi = a_1, \ldots, a_n$$

then, by the composition property of the instantiation function, $\vartheta \circ \Phi$ is an instantiation function too and, more important $(H.\vartheta).\Phi = H.(\vartheta \circ \Phi)$ then $a_1, \ldots, a_n$ is an instance of $H$, i.e. $a_1, \ldots, a_n \in \eta(H)$.

*Example* 4.2 : The function $\vartheta$ such that $\vartheta(\underline{x}) = \underline{C_0}$ and $\vartheta(\underline{y}) = \underline{C_2(C_0, C_0)}$ is a ground instantiation function for the tuple in Example 4.1: ( $\underline{C_3(x, C_0, C_2(x, y)), x}).\vartheta = \underline{C_3(C_0, C_0, C_2(C_0, C_2(C_0, C_0))), C_0}$

This tuple, under our interpretation of Herbrand terms, denotes (the singleton set) $U = \langle \mathbf{c}_3(\underline{C_0}, \underline{C_0}, \mathbf{c}_2(\underline{C_0}, \mathbf{c}_2(\underline{C_0}, \underline{C_0}))), \underline{C_0} \rangle$, and is such that:

$$U \subseteq \mathbf{c}_3(1, \mathbf{c}_2(3, \mathbf{Pe}(14253, \otimes \mathbf{In}(3, \pi), \underline{C_0}, \pi \otimes))),$$

because $\underline{C_0} \subseteq \pi$ and $\mathbf{c}_2(\underline{C_0}, \underline{C_0}) \subseteq \pi$.

Proposition 4.2 means that the set of all the terms which are instances of a term $H$ defines a class of subsets of $H$. Note that, this class does not necessarily coincide with the entire class of all the subsets of the term. As an example, consider the single variable Herbrand term $\underline{x}$ in the Universe of Example 4.2, $\underline{C_0} \bullet \mathbf{c}_1(\underline{C_0})$ is subset of $\eta(\underline{x})$, but for no instantiation function $\vartheta$, $\underline{x}.\vartheta = \{\underline{C_0}, \underline{C_1}(\underline{C_0})\}$. The class of all the subsets obtained by instantiation of a term $H$ is included in the class of all the subsets of $H$. However, Proposition 4.3 shows that this class is closed under set intersection.

PROPOSITION 4.3 : *If $H = h_1, \ldots, h_n$ and $H' = h'_1, \ldots, h'_n$ are two tuples of Herbrand terms, then:*

$$\eta(H.\vartheta) = \mathbf{Sup}\{t \in \mathrm{HU}_c^* \mid t \subseteq \eta(H), t \subseteq \eta(H')\},$$

*assuming $\vartheta$ (to exist and) to be the Mgu of $\varphi(h_1, \ldots, h_n)$ and $\varphi(h'_1, \ldots, h'_n)$, where $\varphi$ is a dummy constructor (or predicate).*

*Proof.* : By Proposition 4.2 we have,

$$\eta(H.\vartheta) \subseteq \eta(H) \quad \text{and} \quad \eta(H.\vartheta) \subseteq \eta(H'), \quad \text{then} \quad \eta(H.\vartheta \subseteq \mathbf{Sup}\{t\}.$$

Moreover, by the property of the Mgu $\vartheta$:

$$\forall \Phi, \quad H.\Phi = H'.\Phi \text{ implies } \exists \Phi' \quad \text{such that} \quad \Phi = \vartheta \circ \Phi'$$

and by Proposition 4.2, we have:

$$\forall \Phi, \quad H.\Phi = H'.\Phi \text{ implies } H.\Phi = H.\vartheta \circ \Phi' \subseteq H.\vartheta$$

and,

$$H.\Phi \subseteq H \quad \text{and} \quad H.\Phi \subseteq H'$$

In particular, $H.\Phi \subseteq H.\vartheta$ holds for each ground unification function $\Phi$ (i.e. a unifier mapping $H$ into a tuple of ground terms). If $\{\Phi_i\}$ is the set of all the ground unification functions, then $\{H.\Phi_i\}$ is the set of all the ground terms common to both $H$ and $H'$ and, by definition of $\subseteq$,

– each $t$ in $\{t\}$ is a (possibly infinite) application of $\bullet$ to elements of $\{H . \Phi_i\}$ (which does not necessarily correspond to an instance of $H$ under some unification function);

– each application $u$ of $\bullet$ to elements of $\{H . \Phi_i\}$ is such that $u \subseteqq \eta(H . \vartheta)$ Hence, $\mathbf{Sup}\{t\} \subseteqq \eta(H . \vartheta)$.

Proposition 4.3 tell us that the *most general instance*, Mgi, of two terms is the **Sup**erior of all the subsets which are obtained by instances of the terms under unification functions. Moreover, this set coincides with the **Sup**erior of all the subsets of both terms. Again, Proposition 4.3 allows us to compute the most general instance of two Herbrand terms as the **Sup**erior of an ascending chain of finite applications of $\bullet$ to the elements of $\mathrm{HU}_c^*$ which correspond to instances of the Herbrand terms under the ground unification functions.

Proposition 4.3 is of no use in *resolving* clausal theories, because clauses contain logical variables, and we are mainly interested in the function $\vartheta$ which computes also the variable bindings. In contrast, because of our set interpretation of Herbrand terms, variables occuring in a Herbrand term are considered to stand for (possibly different invocations of) $\pi$ or **Injection** of $\pi$, then only the most general instance is of interest and $\vartheta$ can be ignored.

Finally, note that the right hand side of the formula in Proposition 4.3 is a formulation of set intersection suitable for sets denoted by the elements of $\mathrm{HU}_c^*$. We will use this fact in the following Section 5.

We have seen that to each Herbrand term, $\eta$ associates a constant expression in our language. Moreover, note that several functions $\eta$ exist, due to the fact that infinite congruent constant expressions exist. As a matter of fact, note that $\forall u \in \mathrm{HU}_k^*$, $\forall v \in \mathrm{HU}_c^*$: $\mathbf{Pr}(1, k, \otimes u, v \otimes) = u$.

## 5. *THE OPERATOR Intset*

The previous Section shows how Herbrand terms can be expressed in a combinatory way, and suggests the use of some language operator to compare elements of $\mathrm{HU}_c^*$ and to compute set intersections. With this aim, we introduce the function **Intset.**

It is a function $\mathrm{HU}_i^* \times \mathrm{HU}_i^* \to \mathrm{HU}_i^*$, it computes the minimal congruence which satisfies the following axioms:

(1-idempotent)

$$\mathbf{Intset}(t, t) = t;$$

(2-commutative)

$$\mathbf{Intset}(t_1, t_2) = \mathbf{Intset}(t_2, t_1);$$

(3-associative)

$$\mathbf{Intset}(\mathbf{Intset}(t_1, t_2), t_3) = \mathbf{Intset}(t_1, \mathbf{Intset}(t_2, t_3));$$

(4-zero)

$$\mathbf{Intset}(t, \varnothing) = \varnothing;$$

(5-finite set)

$$\mathbf{Intset}(t_1, t_2) = t \text{ iff } t_1 = t \bullet t'_1,\ t_2 = t \bullet t'_2 \text{ and } \mathbf{Intset}(t'_1, t'_2) = \varnothing;$$

(6-continuous)

$$\mathbf{Intset}(t, \mathbf{Sup}\{t^j\}) = \mathbf{Sup}\{\mathbf{Intset}(t, t^j).$$

**Intset** behaves like set-intersection on the elements of $\mathrm{HU}_c^*$. It satisfies Proposition 4.3.

PROPOSITION 5.1:

$$\forall t_1, t_2 \in \mathrm{HU}_c^*, \quad \mathbf{Intset}(t_1, t_2) = \mathbf{Sup}\{t \in \mathrm{HU}_c^* \mid t \subseteq t_1, t \subseteq t_2\}.$$

Note that Proposition 5.1 means that the Mgi of Herbrand terms corresponds to set intersection defined by **Intset** on the constant expressions that $\eta$ associates to Herbrand terms. As in the case of the previous operators, **Intset** $(u, v)$ will be considered undefined if $u \in \mathrm{HU}_i^*$ and $v \in \mathrm{HU}_j^*$ and $i \neq j$. In all the other cases, **Intset** $(u, v)$ is defined, then the following property holds.

PROPERTY 5.1: Let $H$, $H'$ be any pair of $i$-tuples of Herbrand terms, then

$$\mathbf{Intset}(\eta(H), \eta(H')) = \begin{cases} \eta(H.\vartheta), & \text{if } \vartheta \text{ exists} \\ \varnothing, & \text{otherwise} \end{cases}$$

assuming that $\vartheta$ is the Mgu of $\varphi(h_1, \ldots, h_n)$ and $\varphi(h'_1, \ldots, h'_n)$, where $\varphi$ is a dummy constructor (or predicate).

*Proof:* Since **Intset** computes the minimal congruence, we only need to show that:

(*a*) $\mathbf{Sup}\{t \in \mathrm{HU}_c^* \mid t \subseteq t_1, t \subseteq t_2\}$ satisfies all the axioms (1)-(6). Hence it computes a congruence relation, i.e. $\mathbf{Intset}(t_1, t_2) = T$ implies $\mathbf{Sup}\{t \in \mathrm{HU}_c^* \mid t \subseteq t_1, t \subseteq t_2\} = T$, and

(*b*) $\mathbf{Sup}\{t \in \mathrm{HU}_c^* \mid t \subseteq t_1, t \subseteq t_2\} \subseteq \mathbf{Intset}(t_1, t_2)$, then **Intset** is defined (hence, is the minimal congruence) on the pair $t_1$, $t_2$.

(*a*) is easy to verify.

In case of (*b*), if both $t_1$ and $t_2$ are finite elements (i.e. $t_1, t_2 \in \mathrm{HU}_F$), by definition of $\subseteq$, $T$ is the greatest finite element common to both $t_1$ and $t_2$. Thus by axiom (5), $T \subseteq \mathbf{Intset}(t_1, t_2)$.

Otherwise (i.e. if $t_1$ and $t_2$ are not both finite elements), let $t_1 = \mathbf{Sup}\,\{t_1^i\}$ and $t_2 = \mathbf{Sup}\,\{t_2^j\}$. Then:

– by construction of $\mathrm{HU}_c^*$: $T \in \mathrm{HU}_c^*$ implies that there exists a chain $\{t^i\}$ of elements of $\mathrm{HU}_F$ such that $\mathbf{Sup}\{t^i\} = T$ and for each $i$, $t^i \subseteq t_1$ and $t^i \subseteq t_2$;

– by definition of **Intset:** for each finite $t$ such that $t \subseteq T$, there exist $t_1^i \in \{t_1^i\}$, $t_2^j \in \{t_2^j\}$ such that $t \subseteq t_1^i$ and $t \subseteq t_2^j$.

Therefore $T \subseteq \mathbf{Intset}\,(t_1, t_2)$.

*Example* 5.1: Let $C$ be the set of constructors $\{\underline{0}, \underline{S}\}$, such that $\underline{0}$ has arity 0 and $\underline{S}$ has arity 1, then:

– $\mathbf{Intset}\,(\eta\,(\underline{S\,(0)}), \eta\,(\underline{S\,(0)})) = \mathbf{Intset}\,(\mathbf{S}\,(\underline{0}), \mathbf{S}\,(\underline{0})) = \mathbf{S}\,(\underline{0})$;

– $\mathbf{Intset}\,(\eta\,(\underline{0}), \eta\,(\underline{S\,(0)})) = \mathbf{Intset}\,(\underline{0}, \mathbf{S}\,(\underline{0})) = \varnothing$;

– $\mathbf{Intset}\,(\eta\,(\underline{x, y}), \eta\,(\underline{x, x})) = \mathbf{Intset}\,(\otimes\,\pi, \pi\,\otimes, \mathbf{In}\,(2, \pi))$
$= \eta\,(\underline{x, x}) = \mathbf{In}\,(2, \pi)$.

## 6. LOGIC PROGRAMMING, RELATIONS AND FUNCTIONS

The Horn clause theory (logic program [Kowalski74])

$$\mathrm{ADD}\,(\underline{x, 0, x}) \leftarrow .,\qquad \mathrm{ADD}\,(\underline{x, S\,(y), S\,(z)}) \leftarrow \mathrm{ADD}\,(\underline{x, y, z}). \tag{7}$$

has a minimal model, which is the set of all the triples of the relation:

$$\mathrm{ADD} = \{\langle x, y, z\rangle \mid x \in \mathrm{HU},\ y = 0,\ x = z\} \cup \{\langle x, S\,(y), S\,(z)\rangle \mid \langle x, y, z\rangle \in \mathrm{ADD}\}. \tag{8}$$

Thus, reasoning about the minimal model of the theory (7) is the same as reasoning about the relation (expressed by) (8). However, (8) is an axiomatic theory, even if it differs from (7) because it is not in Horn clause form, contains just one axiom, and uses set operators. Apart from syntax, (8) is a SuperLOGLISP [Robinson82] definition of ADD. In both theories, to "compute an instance" of ADD, we have to handle all the variables as logical variables. Now, consider the following expression in our language:

$$F_{\mathrm{ADD}}\,(w) = u \bullet v$$

where $\mathbf{Intset}\,(w, \mathbf{Pe}\,(1\ 3\ 2, \otimes\,\mathbf{In}\,(2, \pi), \underline{0}\,\otimes)) = u$,
$$\mathbf{S}\,(2, \mathbf{S}\,(3, F_{\mathrm{ADD}}\,(w'))) = v \tag{9}$$

where

$$\mathbf{S}\!\downarrow\!(2, \mathbf{S}\!\downarrow\!(3, z)) = w',\qquad \mathbf{Intset}\,(w, \mathbf{S}\,(2, \mathbf{S}\,(3, \otimes\,\pi, \pi, \pi\,\otimes))) = z$$

It defines a function from $\mathrm{HU}_3^*$ to $\mathrm{HU}_3^*$ which is a retraction. To see that $F_{\mathrm{ADD}}$ is a retraction, note that for each $w$ it computes $u \bullet v$. $u$ is the result of

**Intset** $(w, \mathbf{Pe}(1\,3\,2, \otimes \mathbf{In}(2, \pi), \underline{0} \otimes))$ then $u \subseteq w$. Moreover, $v$ is the result of $\mathbf{S}(2, \mathbf{S}(3, F_{\mathrm{ADD}}(\mathbf{S}\downarrow(2, \mathbf{S}\downarrow(3, z)))))$, where $z$ is such that $z \subseteq w$ and $z$ denotes the subset of the triples in $w$ which have the form $\langle r, S(p), S(q) \rangle$. Examine that expression. Its sub-expression $\mathbf{S}\downarrow(2, \mathbf{S}\downarrow(3, z))$ removes one $S$ from the second and the third component of each triple in (the set denoted by) $z$, then $F_{\mathrm{ADD}}$ is recursively applied to that value (note that, $z \ll w$), and finally one $S$ is put back in the second and third component of each triple.

Thus, by structural induction on $(\mathrm{HU}_c^*, \ll)$, we see that: $\mathbf{S}(2, \mathbf{S}(3, F_{\mathrm{ADD}}(\mathbf{S}\downarrow(2, \mathbf{S}\downarrow(3, z))))) \subseteq w$.

Note also, that $F_{\mathrm{ADD}}$ satisfies Proposition 2.4. In fact, for each $t_1$ and $t_2$ [since $\mathbf{Intset}(t_1 \bullet t_2, t) = \mathbf{Intset}(t_1, t) \bullet \mathbf{Intset}(t_2, t)$],

$$F_{\mathrm{ADD}}(t_1 \bullet t_2) = F_{\mathrm{ADD}}(t_1) \bullet F_{\mathrm{ADD}}(t_2).$$

Moreover, we can show that $F_{\mathrm{ADD}}$ computes the relation ADD, i. e. its union set (*see* Definition 2.2) is ADD. In fact, each triple of ground Herbrand terms $t_1$, $t_2$, $t_3$ such that $\mathrm{ADD}(t_1, t_2, t_3)$ is an instance of $\mathrm{ADD}(\underline{x, 0, x})$, is not an instance of $\mathrm{ADD}(\underline{x, S(y), S(z)})$ and, because of Proposition 5.1 and $\mathbf{Pe}(1\ 3\ 2, \otimes \mathbf{In}(2, \pi), 0 \otimes) = \eta(\underline{x, 0, x})$, is such that:

$$\mathbf{Intset}(\eta(t_1, t_2, t_3), \mathbf{Pe}(1\,3\,2, \otimes \mathbf{In}(2, \pi), 0 \otimes)) = \eta(t_1, t_2, t_3)$$

then,

$$u = \eta(t_1, t_2, t_3) \qquad \text{and} \qquad v = \varnothing.$$

Furthermore, each triple $t_1$, $t_2$, $t_3$ such that $\mathrm{ADD}(t_1, t_2, t_3)$ is an instance of $\mathrm{ADD}(\underline{x, s(y), s(z)})$, is not an instance of $\mathrm{ADD}(\underline{x, 0, x})$ and is such that:

$$\mathbf{Intset}(\eta(t_1, t_2, t_3), \mathbf{S}(2, \mathbf{S}(3, \otimes \pi, \pi, \pi \otimes))) = \eta(t_1, t_2, t_3).$$

Therefore $u = \varnothing$, and $v = \eta(t_1, t_2, t_3)$ iff

$$F_{\mathrm{ADD}}(\mathbf{S}\downarrow(2, \mathbf{S}\downarrow(3, \eta(t_1, t_2, t_3)))) = \mathbf{S}\downarrow(2, \mathbf{S}\downarrow(3, \eta(t_1, t_2, t_3))),$$

i. e. $\mathrm{ADD}(\underline{x, y, z})$ is satisfied for $\underline{x}$ bound to the term $t_1$, and for $\underline{y}$ and $\underline{z}$, respectively bound to $t_2$ and $t_3$, reduced of the first occurrence of $S$.

All the above considerations allows us to conclude that $F_{\mathrm{ADD}}$, when applied to each element $w$ of $\mathrm{HU}_3^*$, computes the element of $\mathrm{HU}_3^*$ which denotes the set of all the triples $\langle x, y, z \rangle$ in $w$ which are also in the relation ADD. Moreover, there is a correspondence between the two members of the set union in (8) and the two clauses in (7) on one side, and the sub-expressions of $F_{\mathrm{ADD}}$:

- $\mathbf{Intset}(w, \mathbf{Pe}(1\ 3\ 2, \otimes \mathbf{In}(2, \pi), \underline{0} \otimes)) = u$;
- $\mathbf{S}(2, \mathbf{S}(3, F_{\mathrm{ADD}}(w'))) = v$ where $\mathbf{S}\downarrow(2, \mathbf{S}\downarrow(3, z)) = w'$,

$$\mathbf{Intset}(w, \mathrm{S}(2, \mathrm{S}(3, \otimes\pi, \pi, \pi\otimes))) = z,$$

respectively, on the other side. According to this correspondence, we can associate to the query:

$$\leftarrow \mathrm{ADD}(h_1, h_2, h_3)$$

where $h_1$, $h_2$, $h_3$ is any triple of (possibly non ground) Herbrand terms, the application:

$$F_{\mathrm{ADD}}(\eta(h_1, h_2, h_3)).$$

It computes the element of $\mathrm{HU}_3^*$ which denotes the set of all the ground instances of $h_1$, $h_2$, $h_3$ which makes valid the query in the theory (7). For instance, consider the query $\leftarrow \mathrm{ADD}(\underline{0, S(0), z})$, which corresponds to the application $F_{\mathrm{ADD}}(\eta(\underline{0, S(0), z}))$, i. e. $F_{\mathrm{ADD}}(\otimes \underline{0}, \mathrm{S}(\underline{0}), \pi\otimes)$. Then the expression $F_{\mathrm{ADD}}(\otimes \underline{0}, \mathrm{S}(\underline{0}), \pi\otimes)$ evaluates to $\langle \underline{0, S(0), S(0)} \rangle$.

As is the case for SuperLOGLISP, our expressions are always deterministic. The *nondeterminism* of PROLOG-like logic programs is embodied in the structure of the elements of $\mathrm{HU}_c^*$ which denote sets of ground Herbrand terms. Moreover, the program *invertibility* feature which in predicative languages is due to logical variables and is supported by the resolution based evaluation rule, is here embodied in the structure of the constant expression and in the properties of our class of retractions. For instance, consider the query $\leftarrow \mathrm{ADD}(\underline{x, y, S(0)})$, which corresponds to the application $F_{\mathrm{ADD}}(\eta(\underline{x, y, S(0)}))$, i. e. $F_{\mathrm{ADD}}(\otimes\pi, \pi, \mathrm{S}(\underline{0})\otimes)$. The expression $F_{\mathrm{ADD}}(\otimes\pi, \pi, \mathrm{S}(\underline{0})\otimes)$ evaluates to $\langle \underline{0, S(0), S(0)} \rangle \bullet \langle \underline{S(0), 0, S(0)} \rangle$, and the expression $\mathbf{Pr}(2, 1, F_{\mathrm{ADD}}(\otimes\pi, \pi, \mathrm{S}(\underline{0})\otimes)))$ evaluates to $\underline{S(0)} \bullet \underline{0}$.

As another example, consider the Horn clause theory:

$$\mathrm{LE}(\underline{0, y}) \leftarrow .,\qquad \mathrm{LE}(\underline{S(x), S(y)}) \leftarrow \mathrm{LE}(\underline{x, y}).$$

We can associate to LE, the retraction $F_{\mathrm{LE}}$ from $\mathrm{HU}_2^*$ to $\mathrm{HU}_2^*$:

$$F_{\mathrm{LE}}(w) = u \bullet v$$

$$\text{where } \mathbf{Intset}(w, \otimes \underline{0}, \pi\otimes) = u, \mathrm{S}(1, \mathrm{S}(2, F_{LE}(w')) = v \tag{10}$$

where

$$\mathrm{S}\downarrow(1, \mathrm{S}\downarrow(2, z)) = \mathrm{w}', \ \mathbf{Intset}(w, \mathrm{S}(1, \mathrm{S}(2, \otimes\pi, \pi\otimes))) = z.$$

Now, we can extend the theory with the following clause which introduces the relation INTERVAL.

$$\mathrm{INTERVAL}(\underline{inf, sup, x}) \leftarrow \mathrm{LE}(\underline{inf, sup}),\qquad \mathrm{LE}(\underline{inf, x}),\qquad \mathrm{LE}(\underline{x, sup}). \tag{11}$$

(11) contains one of the main appealing feature of logic programming, i. e. the *declarative* feature. From a programming point of view, this means that the language sequence control mechanism allows full freedom in the evaluation ordering of the language forms. This is achieved in logic languages by the mechanism used in the (inferential) operational semantics, to select predicative forms in a query (or clause right part). Due to the commutative and associative properties, set intersection in set based functional programming has the same declarative flavour. For example, consider the retraction of $HU_3^* \to HU_3^*$:

$$F_{\mathrm{INTERVAL}}(w) = \mathbf{Intset}(w, u1, u2, u3)$$

where

$$u1 = \otimes F_{\mathrm{LE}}(\mathbf{Pr}(1, 2, w)), \pi \otimes,$$
$$u2 = \mathbf{Pe}(1\,3\,2, \otimes F_{LE}(\mathbf{Pr}(1, 2, \mathbf{Pe}(1\,3\,2, w))), \pi \otimes),$$
$$u3 = \mathbf{Pe}(3\,2\,1, \otimes F_{LE}(\mathbf{Pr}(1, 2, \mathbf{Pe}(3\,2\,1, w))), \pi \otimes).$$

The computation of $F_{\mathrm{INTERVAL}}(w)$ can proceed in different ways in order to reduce $\mathbf{Intset}(w, u\,1, u\,2, u\,3)$. For instance, the computation of $F_{\mathrm{INTERVAL}}(\otimes\, \mathbf{S}(\underline{0}), \underline{0}, \pi \otimes)$ could first reduce both:

$$u2 = \mathbf{Pe}(1\,3\,2, \otimes F_{LE}(\mathbf{Pr}(1, 2, \mathbf{Pe}(1\,3\,2, \otimes \mathbf{S}(\underline{0}), \underline{0}, \pi \otimes))), \pi \otimes),$$

and

$$u\,3 = \mathbf{Pe}(3\,2\,1, \otimes F_{LE}(\mathbf{Pr}(1, 2, \mathbf{Pe}(3\,2\,1, \otimes \mathbf{S}(\underline{0}), \underline{0}, \pi \otimes))), \pi \otimes)$$

before realizing that $u\,1$ is, in any case, reduced to $\varnothing$.

In logic languages, flat composition is the standard composition rule and moreover, suitable and efficient sequence control mechanisms are hard to design [Kowalski79, Byrd80, Gallaire82, Bowen82, Clark82, Pereira82]. That is not true here, since we have functional composition.

The following retraction $F\,1_{\mathrm{INTERVAL}}$ has the same union set of $F_{\mathrm{INTERVAL}}$, but first checks for the correct definition of the interval limits:

$$F\,1_{\mathrm{INTERVAL}}(w) = \mathbf{Intset}(w, u\,2, u\,3)$$

where

$$u\,2 = \mathbf{Pe}(1\,3\,2, \otimes F_{\mathrm{LE}}(\mathbf{Pr}(1, 2, \mathbf{Pe}(1\,3\,2, w\,1))), \pi \otimes),$$
$$u\,3 = \mathbf{Pe}(3\,2\,1, \otimes F_{\mathrm{LE}}(\mathbf{Pr}(1, 2, \mathbf{Pe}(3\,2\,1, w\,1))), \pi \otimes)$$

where

$$w\,1 = \mathbf{Intset}(w, u\,1),$$
$$u\,1 = \otimes F_{\mathrm{LE}}(\mathbf{Pr}(1, 2, w)), \pi \otimes.$$

The effort for controlling logic programs (and providing an orientation to predicative forms, i. e. annotations to transform logic variables in functional variables [Bellia83, Reddy84]) is tightly related to the inability to model functions in logic programming and, has been one of the motivations of the integration of the logic and functional programming paradigms. In our approach, which models predicates with retractions, retractions are a special class of functions. Our language allows to express general functional programming. Functional programming on (data which denote) sets, is not really innovative. Sets are in fact, basic data in SETL [Shwartz75]. However, we admit infinite sets (in SETL only finite sets are allowed) and we do not need any nondeterministic operator to select, for instance, the elements of a set (as is the case for **arb** in SETL).

To express the function SUM on naturals, represented by the ground Herbrand terms of $\mathrm{HU}_c$ with $C=\{\underline{0}, \underline{S}\}$, we can define:

$$
\begin{aligned}
\mathrm{SUM}(x, y) = {} & \mathbf{if}\ \#(x)=1 \text{ and } \#(y)=1 \text{ and } \mathbf{card}(x)=1 \text{ and } \mathbf{card}(y)=1\\
& \quad \mathbf{then\ if}\ x=0\ \mathbf{then}\ y\\
& \quad\quad\quad \mathbf{else}\ \mathrm{S}(\mathrm{SUM}(\mathrm{S}\downarrow(x), y))\\
& \quad \mathbf{else}\ \varnothing
\end{aligned}
$$

SUM is defined on all the values of $\mathrm{HU}_c^*$ and computes $x+y$ for each pair of values in $\mathrm{HU}_{T_1}$ and, $\varnothing$ everywhere else. For instance, the expression $\mathrm{SUM}(\underline{0, \mathrm{S}(0)})$ evaluates to $\underline{S(0)}$, while the expression $\mathrm{SUM}(\underline{0}\bullet\underline{\mathrm{S}}(\underline{0}), \underline{\mathrm{S}}(\underline{0}))$ evaluates to $\varnothing$, because $\underline{0\bullet S(0)}$ is such that $\mathbf{card}(\underline{0\bullet S(0)}) \neq 1$. $\underline{0\bullet S(0)}$ is an element of $\mathrm{HU}_{\mathrm{F}_1}$ and denotes the set of naturals $\{\underline{0, \mathrm{S}(0)}\}$.

A slightly different expression could be given in order to make SUM to compute a partial function:

$$
\begin{aligned}
\mathrm{SUM}(x, y) = {} & \mathbf{if}\ \#(x)=1 \text{ and } \#(y)=1 \text{ and } \mathbf{card}(x)=1 \text{ and } \mathbf{card}(y)=1\\
& \quad \mathbf{then\ if}\ x=0\ \mathbf{then}\ y\\
& \quad\quad\quad \mathbf{else}\ \mathrm{S}(\mathrm{SUM}(\mathrm{S}\downarrow(x), y))\\
& \quad \mathbf{else}\ \mathrm{SUM}(x, y)
\end{aligned}
$$

SUM now computes $x+y$ for each pair of values of $\mathrm{HU}_{T_1}$ and is undefined everywhere else in $\mathrm{HU}_c^*$. SUM could also be considered as a function from $\mathrm{HU}_{T_2}$ in $\mathrm{HU}_{T_1}$, and expressed by:

$$\mathrm{SUM1}(w)=\mathrm{SUM}(x, y) \qquad \mathbf{where} \quad \mathbf{Pr}(1, 1\ w)=x, \qquad \mathbf{Pr}(2, 1\ w)=y.$$

Moreover, we can extend SUM to compute, for instance, the set $\{\underline{S(0), S(S(0))}\}$, when applied to the cartesian product of $\{\underline{0, \mathrm{S}(0)}\}$ and $\{\underline{S(0)}\}$. To express it in our language, we use **Pro**jections of retractions. For example

$$\mathrm{SUM}\,2(w) = \mathbf{Pr}(3,\ 1,\ F_{\mathrm{ADD}}(\otimes\, w,\ \pi\otimes))$$

defines the function SUM extended on sets. It maps a set of pairs $\{\langle x, y\rangle\}$ onto the set $\{x+y \mid \langle x, y\rangle \in w\}$.

We conclude by noting that the language supports relations (predicates) as a special class of functions. However, this class has all the nice features of logic programming. Moreover, relations and functions are combined by the conventional function composition mechanism. For instance, the expression

$$F(x, y) = \mathrm{SUM}\,2(\mathbf{Pr}(3,\ 1,\ \otimes\, F_{\mathrm{INTERVAL}}(\otimes\, \underline{0,\ S(S(S(S(0))))},\ x\otimes)),\ y\otimes)$$

defines the function $F$ which, due to the restrictions on SUM2, is a mapping from $\mathrm{HU}_1^* \times \mathrm{HU}_1^*$ into $\mathrm{HU}_1^*$. For instance, if $k$ is an element of $\mathrm{HU}_{T_1}$, then $F(h, k)$ computes the set resulting from adding $k$ to each natural, $n$, which is in the set denoted by $h$ and such that $n$ satisfies the relation $\mathrm{INTERVAL}(\underline{0,\ S(S(S(S(0))))},\ \underline{n})$.

## 7. CONCLUSIONS

The main contribution of the paper is the identification of a *special class* of set functions, retractions, which perhaps is the most primitive concept which relates logic and functional programming. Retractions are concretely discussed in a first order functional language which has to be considered as a model for a family of functional languages more than another language which integrate logic and functional programming.

There exist two languages [Berkling82, Darlington85] which share with us the use of set functions as the basic logic-functional integration mechanism. However, our proposal contains some remarkable differences. In both the above mentioned languages, a predicate is defined by a function which returns a "set" of tuples of Herbrand terms whose instances are all the terms of the Herbrand Universe of the (minimal) relation which is a valid interpretation of the predicate (in the logic theory). Hence, Herbrand terms are the symbolic data of the language. However, Herbrand terms are not completely symbolic data. In fact they contain logical variables and require some language ability to cope with term unification. This ability is achieved in SuperLOGLISP by a new reduction rule, ε-reduction, which captures unification and in Darlington's language by assuming narrowing as the language basic expression evaluation rule.

This is the first point of difference with the present approach. Here, Herbrand terms are modelled by a special class of symbolic data, constant expressions, which do not involve (logical) variables and do not require unification to compute with term. All the derivations which can be obtained by ε-reduction or by narrowing, on expressions which contain logical variables, are reformulated here as manipulation of symbolic data. Let us recall the main steps of this modelling process.

– Tuples of Herbrand terms are interpreted as sets of tuples of terms of the Herbrand Universe, $HU_c$(*see* Section 4);

– These sets form a subclass of the recursive subsets of the cartesian products of $HU_c$, and have a combinatory formulation in the class of constant expressions (*see* Proposition 4.1);

– Constant expressions are built from a class of set operators which includes constructors, products and injections (*see* Section 3);

– Constructors of $HU_c$ are extended to set operators (*see* Definition 3.5);

– Each constructor has a (weak) inverse operator (see Section 3.2.1). Constructor inverses and set intersection operators allow to model in function application, the bi-directional matchings (i. e. the computation of the most general instance) of tuples of Herbrand terms, and to avoid the use of logical variables and of unification (*see* Sections 5);

– The associative and commutative properties of the intersection operator maintain the declarative flavour of logic programming (*see* Section 6).

A second point of difference is the mechanism used to declare functions and predicates. Both SuperLOGLISP and Darlington's language use Set abstractions, i. e. constructs of the form $\{X \mid C\}$, where $X$ is a set of variables and $C$ contains equations on Herbrand terms or invocations of predicates only. In our approach, predicates are a special class of functions, retractions, and are distinct from ordinary functions only from the semantic viewpoint. There is no syntactic distinction between retractions and functions, and they can be freely combined through the (standard) function composition mechanism. This is fundamental to our approach: Retractions allow to combine logic and functional programming, in a pure functional programming paradigm, and treat both predicates and ordinary functions by the same object: a set function. A similar feature can be found in other languages [Dershowitz84, Dershowitz85, Reddy85, Fribourg85], where predicates are represented as boolean functions expressed in equational theories. However, the main difference with these languages is the use of special evaluation rules which combine inferences and rewritings in order to treat this class of boolean functions as relations, and to interface them with ordinary function evaluations. In contrast, predicates and functions are distinct objects in [Berkling82, Darlington85, Goguen84, Subrahmanyam84, Barbuti85], and can be combined according to some composition rules (through special linguistic constructs). The second approach has clear advantages from the language user viewpoint, since it allows the use of queries to compute with predicates and of (expression) evaluations to compute with functions. However, the main limitation of all these languages is to completely establish complexity and machine realizability of the basic language evaluation rules [Yasuura84]. Our language is oriented to machine architectures and its realization could be directly supported by the reduction machines which are currently developed for functional languages. In this framework, our language can be viewed as an intermediate language (as is the case of combinatory logic for functional languages) able to support all the above mentioned languages, and whose basic mechanisms are well known and easy to realize at the machine level.

In Section 4 we see that constant expressions are enough to represent all the tuples of Herbrand terms. In the same section we see also that tuples of Herbrand terms are less than constant expressions, for instance we see that to each tuple of Herbrand terms we can associate infinite different but equivalent constant expressions. This equivalence is completely but in an abstract way defined by the axiomatization given in Section 3.1 for our operators. In [Bellia88] we show that constant expressions have *normal form* and we give a system of rewrite rules for our operators which reduce constant expressions to normal form. This set of rewrite rules toghether with the rules for $\alpha$, $\beta$, Y reduction, forms also an operational semantic for our calculus with retractions.

The restriction to first order functions is only motivated by our belief that the mechanisms used to unify logic and functional programming are more easy to understand without working about higher order features. Moreover, higher order extensions seem to be rather independent from the present treatment of predicates and functions. They could only require the use of higher order retractions (i. e. retractions as values) and some marginal extensions to our set operators. However, further work is needed to fully capture the nature of the higher order features in logic-functional programming languages [Warren82, Bowen82, Yokomori84, Bowen85].

## REFERENCES

[Abramson84] H. Abramson, *A Prological Definition of HASL, a Purely Functional Language with Unification Based Conditional Binding Expressions*, New Generation Computing, Vol. 2, 1984, pp. 3-35.

[Backus78] J. Backus, *Can Programming be Liberated from the von Neumann Style?* C. ACM, Vol. 21, 1978, pp. 613-641.

[Barendregt84] H. P. Barendregt, *The LAMBDA Calculus. Its Syntax and Semantics.* North-Holland, 1984.

[Barbuti85] R. Barbuti, M. Bellia, G. Levi and M. Martelli, *LEAF: A Language which Integrates Logic, Equations and Functions*, In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom, Eds. Prentice-Hall, 1985.

[Bellia83] M. Bellia, G. Levi and M. Martelli, *On Compiling Prolog Programs on Demand Driven Architectures*, Proc. Logic Programming Workshop'83, 1983, pp. 518-535.

[Bellia84] M. Bellia, E. Dameri, P. Degano, G. Levi and M. Martelli, *A Formal Model for Lazy Implementation of a PROLOG Compatible Functional Language.* In *Implementations of PROLOG*, J. A. Campbell, Ed. Ellis Horwood, 1984, pp. 309-326.

[Bellia86] M. Bellia and G. Levi, *The Relation Between Logic and Functional Languages: A Survey. J. Logic Programming*, 3, 1986, pp. 217-236.

[Bellia87] M. Bellia, *Retractions: A Functional Paradigm for Logic Programming.* Proc. TAPSOFT'87, LNCS, 250, Springer-Verlag, 1987, pp. 260-275.

[Bellia88] M. Bellia, *Logic and Functional Programming by Retractions: Operational Semantics.* To appear in *RAIRO Informatique Théorique et Applications.*

[Berkling75] K. Berkling, *Reduction Languages for Reduction Machines.* Proc. 2nd Int. Symp. on Computer Architectures, IEEE Comp. Society Press, 1975, pp. 133-140.

[Berkling82] K. Berkling, J. A. Robinson and E. E. Sibert, *A Proposal for a Fifth Generation Logic and Functional Programming System, Based on Highly Parallel Reduction Machine Architecture*, Syracuse University, November 1982.

[Bowen82] K. A. Bowen and R. A. Kowalski, *Amalgamating Language and Metalanguage in Logic Programming.* In *Logic Programming*, K. L. Clark and S.-A. Tarnlund, Eds., Academic Press, 1982, pp. 153-172.

[Bowen85] K. A. Bowen and T. Weinberg, *A Meta-level Extension of Prolog*, Proc. 1985 Symp. on Logic Programming IEEE Comp. Society Press, 1985, pp. 48-53.

[Byrd80] L. Byrd, *Understanding the Control Flow of Prolog Programs*, First Workshop on Logic Programming, 1980, pp. 127-138.

[van Caneghem84] M. van Caneghem and D. H. D. Warren (Eds.), *Logic Programming and its Applications*, Ablex Pub. Comp., 1984.

[Clark82] K. L. Clark, F. G. McCabe and S. Gregory, *IC-Prolog: Language Features.* In *Logic Programming*, K. L. Clark and S.-A. Tarnlund, Eds., Academic Press, 1982, pp. 254-266.

[Clark83] K. L. Clark and S. Gregory, *PARLOG: a Parallel Logic Programming Language*, Imperial College Research Report 83/5, May 1983.

[Clarke80] T. J. W. Clarke, P. J. S. Gladstone, C. D. MacLean and A. C. Normal, *SKIM-The S.K.I. reduction machine*, Proc. Lisp 80 Conf., 1980, pp. 128-135.

[Darlington81] J. Darlington and M. Reeve, *ALICE: A Multiprocessor Reduction Machine for the Parallel Evaluation of Applicative Languages.* Proc. Int. Symp. Functional Programming Languages and Computer Architectures, 1981, pp. 32-62.

[Darlington85] J. Darlington, A. J. Field and H. Pull, *The Unification of Functional and Logic Languages.* In *Logic Programming: Functions, Relations and Equations*, D. DeGroot and G. Lindstrom Eds. Prentice-Hall, 1985.

[Dershowitz84] N. DERSHOWITZ and N. A. JOSEPHSON, *Logic Programming by Completion*, Proc. 2nd Int. Logic Programming Conf., 1984, pp. 313-320.

[Dershowitz85] N. DERSHOWITZ and D. A. PLAISTED, *Logic Programming Cum Applicative Programming*, Proc. 1985 Symp. on Logic Programming, IEEE Comp. Society Press, 1985, pp. 54-66.

[Fribourg85] L. FRIBOURG, *SLOG: A Logic Programming Language Interpreter Based on Clausal Superposition and Rewriting*. Proc. 1985 Symp. on Logic Programming, IEEE Comp. Society Press, 1985, pp. 172-184.

[Gallaire82] H. GALLAIRE and C. LASSERRE, *A control Metalanguage for Logic Programming*. In *Logic Programming*, K. L. CLARK and S.-A. TARNLUND Eds., Academic Press, 1982, pp. 173-185.

[Goguen84] J. A. GOGUEN and J. MESEGUER, *Equality, types, modules and (why not?) generics for logic programming*, J. Logic Programming, Vol. 1, 1984, pp. 179-210.

[Hankin85] C. L. HANKIN, P. E. OSMAN and M. J. SHUTE, *COBWEB - A Combinator Reduction Architecture*. Proc. Functional Programming Languages and Computer Architecture, LNCS, Vol. 201, Springer-Verlag, 1985, pp. 89-102.

[Hsiang83] J. HSIANG and N. DERSHOWITZ, *Rewrite Methods for Clausal and Non-clausal Theorem Proving*, Proc 10th ICALP, 1983.

[Khan81] K. M. KAHN, *Uniform: A Language Based Upon Unification which Unifies Much of Lisp, Prolog and Act*1, Proc. 7th IJCAI, 1981.

[Kluge80] W. E. KLUGE and H. SCHLUTTER, *An Architecture for the Direct Execution of Reduction Languages*, Proc. Int. Workshop High Level Computer architecture, 1980.

[Kowalski71] R. A. KOWALSKI and D. KUEHNER, *Linear resolution with selection function*, Artificial Intelligence, Vol. 2, 1971, pp. 227-260.

[Kowalski74] R. A. KOWALSKI, *Predicate Logic as a Programming Language*, Proc. IFIP Congress, 1974, 569-574.

[Kowalski79] R. A. KOWALSKI, *Algorithms = Logic + Control.*, C. ACM, Vol. 22, 1979, pp. 424-436.

[Komorowski82] H. J. KOMOROWSKI, *QLOG - The Programming Environment for Prolog in Lisp*. In *Logic Programming*, K. L. CLARK and S.-A. TARNLUND Eds., Academic Press, 1982, pp. 315-322.

[Kornfeld83] W. A. KORNFELD, *Equality for PROLOG*, Proc. 8th IJCAI, 1983, pp. 514-519.

[Lindstrom85] G. LINDSTROM, *Functional Programming and the Logical Variable*, Proc. 12th ACM Symp. on Principles of Programming Languages, 1985.

[Lloyd84] J. W. LLOYD, *Foundations of Logic Programming*, Springer-Verlag, 1984.

[Mago80] G. A. MAGO, *A Cellular Computer Architecture for Functional Programming*, Proc. IEEE-COMPCON 80, IEEE Comp. Society Press, 1980, pp. 179-187.

[McCarthy60] J. MCCARTHY, *Recursive Functions and Symbolic Expressions and Their Computation by Machine*, C. ACM, Vol. 3, 1960, pp. 184-195.

[Mellish84] C. MELLISH and S. HARDY, *Integrating PROLOG in the POPLOG Environment*. In *Implementations of PROLOG*, J. A. CAMPBELL Ed., Ellis Horwood, 1984, pp. 147-162.

[Milner72] R. MILNER, *Implementation and Application of Scott's Logic for Computable Functions*, Sigplan Notices, Vol. 7, 1972, pp. 1-6.

[Moto-Oka82] T. MOTO-OKA, Ed., *Fifth Generation Computer Systems*, North-Holland, 1982.

[Pereira82] L. M. PEREIRA, *Logic Control with Logic*. In *Implementations of PROLOG*, J. A. CAMPBELL Ed., Ellis Horwood, 1984, pp. 177-193.

[Reddy85] U. S. REDDY, *On the Relationship Between Logic and Functional Languages*. In *Logic Programming: Functions, Relations and Equations*, D. DEGROOT and G. LINDSTROM Eds, Prentice-Hall, 1985.

[Rety85] P. RETY, C. KIRCHNER, H. KIRCHNER and P. LESCANNE, *NARROWER: A New Algorithm for Unification and its Application to Logic Programming*, Proc. First Int. Conf. on Rewriting Techniques and Applications, 1985.

[Robinson65] J. A. ROBINSON, *A Machine-oriented Logic Based on the Resolution Principle*, J. ACM, Vol. 12, 1965, pp. 23-44.

[Robinson82] J. A. ROBINSON and E. E. SIBERT, *LOGLISP: Motivations, Design and Implementation*. In *Logic Programming*, K. L. CLARK and S.-A. TARNLUND Eds., Academic Press, 1982, pp. 299-314.

[Robinson82] J. A. ROBINSON and E. E. SIBERT, *LOGLISP: An Alternative to PROLOG*, Machine Intelligence, Vol. 10, Ellis Horwood, 1982.

[Robinson83] J. A. ROBINSON, *Logic Programming: Past, Present and Future*, New Generation Computing, Vol. 2, 1983, pp. 107-124.

[Sato84] M. SATO and T. SAKURAI, *Qute: a Functional Language Based on Unification*, Proc. FGCS'84, 1984, pp. 157-165.

[Scott76] D. SCOTT, *Data Types as Lattices*, SIAM J. on Computing, Vol. 5, 1976, pp. 522-587.

[Shapiro86] SHAPIRO, E. and L. STERLING, *The Art of Prolog*, MIT Press, 1986.

[Stoy77] J. E. STOY, *Denotational Semantics. The Scott-Strachey Approach to Programming Languages*, MIT Press, Cambridge, 1977.

[Shwartz75] J. T. SHWARTZ, *Automatic Data Structure Choise in a Language of Very High Level*, C. ACM, Vol. 18, 1975, pp. 772-728.

[Srivastava85] A. SRIVASTAVA, D. OXLEY and A. SRIVASTAVA, *An(other) Integration of Logic and Functional Programming*, Proc. 1985 Symp. on Logic Programming, IEEE Comp. Society Press, 1985, pp. 254-260.

[Subrahmanyam84] P. A. SUBRAHMANYAM and J.-H. YOU, *FUNLOG=-Functions+Logic: A Computational Model Integrating Functional and Logic Programming*, Proc. 1984 Int. Symp. on Logic Programming, IEEE Comp. Society Press, 1984, pp. 144-153.

[Turner79] D. A. TURNER, *SASL Language Manual*, Dept. of Computational Science, Univ. of St. Andrews, 1979.

[Warren82] D. H. D. WARREN, *Higher-order Extensions to PROLOG: are they Needed?* Machine Intelligence, Vol. 10, 1982, pp. 441-454.

[Yasuura84] H. YASUURA, *On Parallel Computational Complexity of Unification*, Proc. FGCS' 84, 1984, pp. 235-243.

[Yokomori84] T. YOKOMORI, *A Note on the Set Abstraction in Logic Programming Language*, Proc. FGCS' 84, 1984, pp. 333-340.

# APPENDIX I

## LANGUAGE SYNTAX

(1) The language alphabet is $A=\{D, V_d, P, V_p\}$, where:

– $D$ is a denumerable set of data defined according to the value domain $HU_c^*$, defined in Section 3.1. Moreover, it can include values for Integers, Booleans and other suitable domains (as directly representable data).

– $P$ is a (denumerable) set of primitive operators, which includes the operators $\langle - \rangle$, $\bullet$, $\otimes - \otimes$, $\varnothing$, $\pi$, $\mathbf{c}_k$, $\mathbf{c}_k\downarrow$, **In**, **Pe**, **Pr**, **Intset**, defined in Section 3.1, and, according to $D$, all the arithmetic and boolean operators, the conditional **if-then-else** operator, and the following operators on sets:

– The tupling operator $\#$, which, applied to a data in $HU_i^*$, results $i$, i. e. the class of tupling of the data (Integers and Booleans have tupling 1);

– The cardinality operator **card** which, applied to data in $HU_F$ results the cardinality of the corresponding finite set (Integers and Booleans have cardinality 1). Moreover, **card** is undefined when applied to data in $HU_\omega$;

– The equality operator, $=$, which results true if the arguments are the same data in $HU_F$ (Integer and Boolean). It results false or is undefined if the arguments are different data or are both in $HU_\omega$, respectively.

– $V_d$ is a denumerable set of variables which range over $D$.

– $V_p$ is a family of denumerable sets of variables which range over the first order functions on $D$, and are indexed by the function arity ($V_d$ and $V_p$ are disjoint sets).

(2) The language expressions are all the closed forms: $\{E \mid E\in F, \mathrm{ch}[E]=\{\ \}\}$, where $F$ is the set of the language forms, and $\mathrm{ch}[E]$ is the set of variables which occur free in $E$.

(3) The set $F$ of the forms is:

– $D_F$, set of all the data in $HU_F$ (Integer and Boolean): $\forall E\in D_F$, $\mathrm{ch}[E]=\{\ \}$;

– $V_d$, set of all the variables on $D$: $\forall E\in V_d$, $\mathrm{ch}[E]=\{E\}$;

– $A_F$, set of all the applications of primitive or defined functions:

$$A_F=\{(\mathrm{op}_n\ E1\ldots En) \mid \mathrm{op}_n \text{ has arity } n,\ \mathrm{op}_n\in P\cup V_p,\ Ei\in F\}$$
$$\forall(\mathrm{op}_n E1\ldots En)\in A_F,\quad \mathrm{ch}[(\mathrm{op}_n E1\ldots En)]=\bigcup_{i=1,\,n}\mathrm{ch}[Ei].$$

— $A_\lambda$, set of all the $\lambda$-abstraction applications:

$$A_\lambda=\{((\lambda x1 \ldots xn.E)E1 \ldots En) \mid xi \in V_d,\ xi \neq xj \text{ for } i \neq j,\ E,\ Ei \in F\},$$

$$\forall((\lambda x1 \ldots xn.E)E1 \ldots En) \in A_\lambda,$$

$$\mathrm{ch}[((\lambda x1 \ldots xn.E)E1 \ldots En)]=(\mathrm{ch}[E]-\{x1, \ldots, xn\}) \cup (\bigcup_{i=1,n} \mathrm{ch}[Ei]).$$

— $A_Y$, set of all fixed-point function applications:

$$A_Y=\{((\mathbf{Y}\,\mathrm{op}_n x1 \ldots xn.E)E1 \ldots En) \mid \mathrm{op}_n \text{ has arity } n,$$
$$\mathrm{op}_n \in V_p,\ xi \in V_d,\ xi \neq xj \text{ for } i \neq j,\ E,\ Ei \in F\}$$

$$\forall((\mathbf{Y}\mathrm{op}_n x1 \ldots xn.E)E1 \ldots En) \in A_Y,$$

$$\mathrm{ch}[((\mathbf{Y}\mathrm{op}_n x1 \ldots xn.E)E1 \ldots En)]=(\mathrm{ch}[E]\text{-}\{x1, \ldots, xn\}) \cup (\bigcup_{i=1,n} \mathrm{ch}[Ei])$$

**Language semantics**

Each form has meaning according to the semantics of the primitive operators and to the following reduction rules:

$\alpha$-reduction

$$(\lambda x1 \ldots xn.E)=(\lambda y1 \ldots yn.[y1/x1, \ldots, yn/xn]E),$$

where

$$yi \neq yj \quad \text{for } i \neq j \quad \text{and} \quad yi \notin \mathrm{ch}[E].$$

$\beta$-reduction

$$((\lambda x1 \ldots xn.E)E1 \ldots En)=[E1/x1, \ldots, En/xn]E.$$

**Y**-reduction

$$((\mathbf{Y}\mathrm{op}_n x1 \ldots xn.E)E1 \ldots En)$$
$$=((\lambda x1 \ldots xn.[(\mathbf{Y}\mathrm{op}_n x1 \ldots xn.E)/\mathrm{op}_n]E)E1 \ldots En)$$

where:

$$\forall E \in D_F, \quad [E'/x]E=E;$$

$$\forall E \in V_c \cup V_p, \quad [E'/x]E=\begin{cases} E' & \text{if } x=E \\ E & \text{otherwise} \end{cases}$$

$\forall (\mathrm{op}_n E\,1 \ldots E\,n) \in A_F,$

$[E'/x](\mathrm{op}_n E\,1 \ldots E\,n) = ([E'/x]\,\mathrm{op}_n [E'/x] E\,1 \ldots [E'/x] E\,n);$

$\forall ((\lambda\, x\,1 \ldots xn.\, E)\, E1 \ldots E\,n) \in A_\lambda,$

$[E'/x]((\lambda\, x\,1 \ldots xn.\, E)\, E\,1 \ldots E\,n) = H\,[E'/x] E\,1 \ldots [E'/x] E\,n;$

$\forall ((\mathrm{Yop}_n\, x\,1 \ldots xn.\, E)\, E\,1 \ldots E\,n) \in A_F,$

$[E'/x]((\mathrm{Yop}_n\, x\,1 \ldots xn.\, E)\, E\,1 \ldots E\,n) = H\,[E'/x] E\,1 \ldots [E'/x] E\,n;$

with:

$$H = \begin{cases} (\lambda\, x\,1 \ldots xn.\, E) \text{ resp. } (\mathrm{Yop}_n\, x\,1 \ldots xn.\, E) & \text{if } x \in \{\mathrm{op}_n, x\,1, \ldots, xn\} \\ (\lambda\, x\,1 \ldots xn.\, [E'/x] E) \text{ resp. } (\mathrm{Yop}_n\, x\,1 \ldots xn.\, [E'/x] E) & \\ \text{if } \mathrm{ch}[E'] \cap \{x\,1, \ldots, xn\} = \{\ \} & \end{cases}$$

*Syntactic extensions*

*Sequence of function declarations*

As a syntactic extension, we admit the following two forms

$$f\,1\,(x_1^1, \ldots, x_{n\,1}^1) = E\,1; \ldots; \quad fm\,(x_1^m, \ldots, x_{nm}^m) = E\,m; \quad E$$

$$\mathbf{let}\, f\,1\,(x_1^1, \ldots, x_{n\,1}^1) = E\,1; \ldots; fm\,(x_1^m, \ldots, x_{nm}^m) = E\,m \quad \mathbf{in}\ E$$

which correspond to the expression

$$[F\,1/f\,1](\ldots([F\,m/fm]\,E)\ldots)$$

with:

$$F\,i = \begin{cases} (\lambda\, x_i^i \ldots x_{ni}^i.\, E\,i) & \text{if } E\,i \text{ does not contain occurrences of } fi \\ (Y fi x_1^i \ldots x_{ni}^i.\, E\,i) & \text{otherwise.} \end{cases}$$

Note that $Ei$ can only contains occurrences of $fi$ itself or of functions defined before in the sequence.

– Sequence of function applications and where expression. As a syntactic extension, we admit the following two forms

$$E, E\,1 = x\,1, \ldots, E\,n = xn$$

$$E\,\mathbf{where} \quad E\,1 = x\,1, \ldots, E\,n = xn$$

which have the same meaning and correspond to the expression

$$((\lambda\, x\,1 \ldots xn.\, E)\, E\,1 \ldots E\,n).$$

# APPENDIX II

**Proposition 4.1** (*proof*)

We define a function $\eta$ which satisfies the proposition. To define it we distinguish two cases:

(A) (tuples of constants and variables). Let $H$ be the tuple $v_1, \ldots, v_n$ where each $v_i$ is either a 0-arity constructor or a variable, then:

(i) if each variable in $H$ occurs only once:

$$\eta(v_1, \ldots, v_n) = \otimes\, e_1, \ldots, e_n \otimes$$

where

$$e_i = v_i \quad \text{if} \quad v_i \text{ is a 0-arity constructor}$$
$$e_i = \pi \quad \text{if} \quad v_i \text{ is a variable}$$

(ii) otherwise (i. e. $H$ contains multiple occurrences of some variables):

$$\eta(v_1, \ldots, v_n) = \mathbf{Pe}(n!, \otimes\, e_1, \ldots, e_m \otimes)$$

where

($a$) $\sum_{i=1,m} \# e_i = n$

($b$) $n!$ and $e_1, \ldots, e_m$ are such that:

(1) for each 0-arity constructor $v_i$

$$\exists\, e_j \text{ such that: } \quad e_j = v_i \quad \text{and} \quad \sum_{p=1,\, j-1} \# e_p = n!(i) - 1$$

(2) for each variable $v_i$ which occurs only once

$$\exists\, e_j \text{ such that: } \quad e_j = \pi \quad \text{and} \quad \sum_{p=1,\, j-1} \# e_p = n!(i) - 1$$

(3) for each variable $v_i$ which occurs $k$ times ($k > 1$), let $i1, \ldots, ik$ be its occurrences (i. e. $v_{i1} = \ldots = v_{ik}$ and $i \in \{i1, \ldots, ik\}$)

$$\exists\, e_j \text{ such that: } \quad e_j = \mathbf{In}(k, \pi) \quad \text{and} \quad \sum_{p=1,\, j-1} \# e_p = n!(i1) - 1$$

and for

$$q \in [1, k-1],\ n!(i1+q) = n!(i1) + q.$$

(B) (tuples with constructors of arity greater than 0). Let $H$ be the tuple:

$$v_1, \ldots, v_{m-1}, \underline{C_k}(h'_1, \ldots, h'_k), h_{m+1}, \ldots, h_n$$

where each $v_i$ is either a 0-arity constructor or a variable, then:

$$\eta(v_1, \ldots, v_{m-1}, \underline{C_k}(h'_1, \ldots, h'_k), h_{m+1}, \ldots, h_n)$$
$$= \mathbf{c}_k(m, \eta(v_1, \ldots, v_{m-1}, h'_1, \ldots, h'_k, h_{m+1}, \ldots, h_n)).$$

We prove that $\eta$, as defined above, satisfies Proposition 4.1, that is:

$$\forall a_1, \ldots, a_n \in \mathrm{HU}_c, \quad a_1, \ldots, a_n \in \mathrm{H} \quad \Leftrightarrow \quad \langle a_1, \ldots, a_n \rangle \in \eta(H).$$

(Part $\Rightarrow$). We assume $a_1, \ldots, a_n \in H$, i. e.

$\exists \Phi$, instantiation function of variables to ground terms, such that: $H.\Phi = a_1, \ldots, a_n$.

We prove by induction on the structure of the constructors $\underline{C_k}$ that:

$$\langle a_1, \ldots, a_n \rangle \in \eta(H).$$

A. — [tuples of constants and variables, then $\langle a_1, \ldots, a_n \rangle \in \eta(H) =$

$$\mathbf{Pe}(n!, \otimes e_1, \ldots, e_m \otimes)]:$$

(i) obvious because of the definition of $\pi$ and of the cartesian product $\otimes - \otimes$.

(ii) first note that, since (*a*), $\mathbf{Pe}(n!, \otimes e_1, \ldots, e_m \otimes) \in \mathrm{HU}_n^*$, then denotes sets of $n$-tuples.

Furthermore,

(1) if $h_i$ is a 0-arity constructor then $h_i.\Phi = h_i = a_i$ and by definition of **Pe** the $i$-th component of $\mathbf{Pe}(n!, \otimes e_1, \ldots, e_m \otimes)$ is the $n(\mathrm{i})!$-th component of $\otimes e_1, \ldots, e_m \otimes$, but this is $h_i$.

(2) if $h_i$ is a single occurrence variable then $h_i.\Phi = a_i$ and the $n(i)!$-th component of $\otimes e_1, \ldots, e_m \otimes$ is $\pi$.

(3) if $h_i$ is a multiple occurrence variable then $h_i.\Phi = a_i$ and also $h_{i\,1}.\Phi = a_{i\,1} = \ldots = a_{ik}$, if $i\,1, \ldots, ik$ are all the occurrences of $h_i$ in $H$, then the $n(i)!$-th component of $\otimes e_1, \ldots, e_m \otimes$ is $\mathbf{In}(k, \pi)$ and it has the $n(i\,1)!, \ldots, n(ik)!$ components of $\otimes e_1, \ldots, e_m \otimes$ as its components.

B. — (tuples with constructor of arity greater than 0)

By the proof of A above, and assumed, as inductive step,

$$H' = h_1, \ldots, h_{m-1}, h'_1, \ldots, h'_k, h_{m+1}, \ldots, h_n$$

and $\eta(H')$ to denote the same set, the proof that:

$$\forall \underline{C_k} \quad \text{and} \quad m \in [1, n-k+1],$$
$$H = h_1, \ldots, h_{m-1}, \quad \underline{C_k}(h'_1, \ldots, h'_k), h_{m+1}, \ldots, h_n$$

and $\eta(H)$ denote the same set, immediately follows from part (B) of the definition of $\eta$ and from the definition of the functions $\mathbf{c}_k$.

(Part $\Leftarrow$). We assume $a_1, \ldots, a_n \in \mathbf{Pe}(n!, \otimes e_1, \ldots, e_m \otimes)$, i. e.

$$\forall i \in [1, n], \quad a_i \text{ is the } n(i)!\text{-th component of } \otimes e_1, \ldots, e_m \otimes.$$

We prove the existence of an instantiation function of variables to ground terms, $\Phi$, such that:

$$H . \Phi = a_1, \ldots, a_n.$$

We construct $\Phi$.

(1) if $a_i$ is a 0-arity constructor, then by $(b1)\, h_i = a_i$,

(2) if $a_i$ is $\pi$, then by $(b2)\, h_i$ is a single occurrence variable and we make $\Phi(h_i) = a_i$,

(3) if $a_i$ is component of $\mathbf{In}$ $(k, \pi)$, then by $(b3)\, h_i$ is a multiple occurrence variable and we make $\Phi(h_i) = a_i$.