Imagining CLP (Lambda, alphabeta)

Olivier Ridoux

INRIA

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

Imagining CLP(Λ,$\equiv_{\alpha\beta}$)

Olivier RIDOUX

N° 2388

Octobre 1994

PROGRAMME 2

ISSN 0249-6399

Imagining CLP(Λ,$\equiv_{\alpha\beta}$)

Olivier RIDOUX*

Programme 2 — Calcul symbolique, programmation et génie logiciel
Projet Lande

Rapport de recherche n° 2388 — Octobre 1994 — 19 pages

Abstract: We study under which conditions the domain of λ-terms (Λ) and the equality theory of the λ-calculus ($\equiv_{\alpha\beta}$) form the basis of a usable constraint logic programming language (CLP). The conditions are that the equality theory must contain axiom η, and the formula language must depart from Horn clauses and accept universal quantifications and implications in goals. In short, CLP(Λ, $\equiv_{\alpha\beta}$) must be close to λProlog.

Key-words: CLP, λ-Calculus, λProlog

(Résumé : tsvp)

*ridoux@irisa.fr

Unité de recherche INRIA Rennes
IRISA, Campus universitaire de Beaulieu, 35042 RENNES Cedex (France)
Téléphone : (33) 99 84 71 00 – Télécopie : (33) 99 84 71 71

Imaginons CLP(Λ,$\equiv_{\alpha\beta}$)

Résumé : Nous étudions sous quelles conditions le domaine des λ-termes (Λ) et la théorie de l'égalité du λ-calcul ($\equiv_{\alpha\beta}$) forment une base utilisable pour un langage de programmation logique par contrainte (CLP). Les conditions sont que la théorie de l'égalité doit aussi contenir l'axiome η, et le langage de formule doit étendre celui des clauses de Horn et accepter des quantifications universelles et des implications dans les buts. En fait, CLP(Λ, $\equiv_{\alpha\beta}$) doit ressembler à λProlog.

Mots-clé : CLP, λ-calcul, λProlog

1 Introduction

Logic programming is a programming paradigm in which programs are logical formulas, and executing them amounts to search for a proof. The most famous practical incarnation of logic programming is Prolog, which is based on Horn formulas [30].

The formalism of Horn programs is computationally complete [1, 48], but one has often tried to augment it to gain more flexibility and expressivity. One of these attempts is the paradigm of constraint logic programming [11, 27, 10, 49]. It amounts to replacing unification of first-order terms, considered as a procedure for solving equations on first-order terms, by other procedures for solving a wider range of problems (equations but also inequations and disequations, etc) on a wider range of domains: booleans, finite domains, integers, rationals, reals, etc. An important point is that the constraint handler searchs for satisfiability of the constraints, but not necessarily for solutions. This extension preserves the model-theoretic results of Prolog. It adds flexibility because, computation on some domains must be done in a directional way in Prolog, though it can be done fully relationally when considered a constraint, For instance in Prolog, *13 is X+12*, where variable X is not bound at execution time, is a mistake, though its equational counterpart, *13 = X+12*, makes perfect sense, and is solved by substituting 1 to X. The extension also add efficiency because searching for satisfiability instead of searching for solutions saves non-determinism.

The general scheme for constraint logic programming is noted CLP, instances of which can be formed by passing domains and theories to the scheme. In this work, we study CLP(Λ, $\equiv_{\alpha\beta}$) which is an instance of CLP where the domain is of λ-terms (Λ) and the theory is $\alpha\beta$-equivalence ($\equiv_{\alpha\beta}$).

The study of the general scheme has focused on what properties the domains and theories must satisfy for some model-theoretic results to hold. Much less is said on the *usability* of the solutions of the constraints. By usability we mean the ability to be used by the program itself that generated the constraints in order to analyse the result, take further decisions, build new constraint etc. Usability is in fact a meta-programming capability. Little is said of usability probably for two reasons:

1. it is just natural for numerical or finite domains, for which operations exist in any reasonable instance of Prolog,
2. most examples in the literature, and many reported applications, obey a query-answer scheme in which a CLP program builds a constraints system for solving some problem, pass it to the solver, and reports the solutions to some operator.

Incorporating λ-terms in Prolog stimulates the study of the notion of usability because the condition above are no longer true:

1. it is not just natural since Prolog offers no operation to access λ-terms,
2. and the motivation for incorporating λ-terms is to represent programs or formulas which we want to manipulate in Prolog and not only display to the operator.

We motivate adding λ-terms in Prolog in section 2. Then, we give a proof-theoretic account of Horn clause programming in section 3. We study the consequences of adding λ-terms and $\alpha\beta$-equivalence in Prolog in section 4. Finally, we conclude in section 5. We assume everywhere a basic knowledge of Prolog.

2 Motivations

We give a brief account of the syntax of λ-terms and of the theory of λ-calculus. Then we explain to what purpose λ-terms can be used in Logic Programming.

2.1 The λ-Terms

λ-Terms are generated by the following grammar:

$$\begin{array}{lll} \Lambda & ::= \mathcal{C} \mid \mathcal{V} & \\ \Lambda & ::= \lambda\mathcal{V}\ (\ \Lambda\ ) & \text{(i)} \\ \Lambda & ::= (\ \Lambda \quad \Lambda\ ) & \text{(ii)} \end{array}$$

where the $\mathcal{C}$ and $\mathcal{V}$ are respectively identifiers of constants and identifiers of variables. Rule (i) generates *abstractions*, and rule (ii) generates *applications*. An abstraction can be interpreted as a function, and an application can be interpreted as the result of applying some function to some actual parameter. For instance, the identity function is written $\lambda x(x)$. We assume application associates to the left, which makes some brackets useless: for instance, $(append\ A\ B\ C)$ denotes the same term as $(((append\ A)\ B)\ C)$ does. Similarly we drop nested brackets or brackets in nested abstractions: for instance, $\lambda x \lambda y(x\ y)$ denotes the same term as $\lambda x(\lambda y((x\ y)))$ does.

Abstraction leads to the notions of *heading*, *head*, and *body*. In term $\lambda a \lambda b \lambda c(b\ a\ c)$, the heading is $\lambda a \lambda b \lambda c$, the head is b, and the body is $(b\ a\ c)$. One says a term *binds* the variables of its heading.

One distinguishes between *free* and *bound* occurrences of variables. In the term $(x\ y\ \lambda z \lambda x(x\ y\ z))$, y has only free occurrences, the only occurrence of z is bound, and x has a free occurrence (the first) and a bound one (the second). In the underlined subterm, z and y have only free occurrences, and the only occurrence of x is bound. More generally, an occurrence of some variable is bound in a given term if it is a subterm of an abstraction that binds the variable and is a subterm of the given term. An occurrence of some variable is free if it is not bound. One calls *free variables* of a given term the variables that have a free occurrence in it, *bound variables* those that have a bound occurrence in the term. One writes $\mathcal{FV}(t)$ the *free variables* of t, and $\mathcal{BV}(t)$ the *bound variables*. A term without any free variable is called *closed* or a *combinator*. One writes $[x \leftarrow y]$ the operation of replacing all free occurrences of x by y, and $E[x \leftarrow y]$ the application of this operation to term E.

2.2 A Taste of λ-Calculus

The domain of λ-terms is endowed with an equivalence relation which is defined as the smallest congruence based on the following axioms:

α — $\lambda x(E) \equiv_\alpha \lambda y(E[x \leftarrow y])$, if y has no occurrence in E.

This axiom formalizes the renaming of bound variables. For instance, $\lambda x(f\ x) \equiv_\alpha \lambda y(f\ y)$, but $\lambda x(g\ x\ y) \not\equiv_\alpha \lambda y(g\ y\ y)$.

β — $(\lambda x(E)\ F) \equiv_\beta E[x \leftarrow F]$, if $\mathcal{FV}(F) \cap \mathcal{BV}(E) = \emptyset$.

This axiom formalizes the evaluation of an application by substituting an actual parameter, F, to a formal parameter, x. For instance, $(\lambda x(f\ x)\ 12) \equiv_\beta (f\ 12)$, but $(\lambda x \lambda y(x)\ y) \not\equiv_\beta \lambda y(y)$ because $y \in \mathcal{FV}(y) \cap \mathcal{BV}(\lambda x \lambda y(x))$. However, $(\lambda x \lambda y(x)\ y) \equiv_\alpha (\lambda x \lambda w(x)\ y) \equiv_\beta \lambda w(y)$. In fact, it is always possible to apply axiom β to a term like $(\lambda x(E)\ F)$ if one uses axiom α for renaming the bound variables of E.

η — $\lambda x(E\ x) \equiv_\eta E$, if $x \notin \mathcal{FV}(E)$.

This axiom formalizes the principle of *functional extensionality*: "Different functions yield different results". According to this principle, we have $[\forall x(E\ x) = (F\ x)] \Rightarrow E = F$, which is not provable using axioms α and β alone. For instance, $\lambda x(f\ x) \equiv_\eta f$, but $\lambda x((g\ x)\ x) \not\equiv_\eta (g\ x)$. Note that axiom α can never help applying axiom η.

Axioms α and β are always present in the λ-calculus, but axiom η is optional: it gives nothing as far as the computational properties of the λ-calculus are concerned. However, we will see it is crucial for the equality theory.

Application $(\lambda x(E)\ F)$ is called a *β-redex*. Abstraction $\lambda x(E\ x)$ where variable x is not free in E is called an *η-redex*. A λ-term with no β-redex (resp. η-redex) is called *β-normal* (resp. *η-normal*). The axioms may be oriented to form *rewriting rules*. To apply β-equivalence for suppressing a β-redex is to β-reduce a term. To apply η-equivalence for suppressing a η-redex is to η-reduce a term.

A given term may be applied different reduction rules simply because it contains several redexes. One may wonder whether the reduction order is significant or not. In fact, it is not. This system of reduction rules (with or without η-reduction) enjoys the Church-Rosser property: *"For every two β-equivalent terms A and B, there is a term N to which A and B reduce in some number of steps"*. Hence, every term has at most one normal form. If every term has a normal form, and any arbitrary strategy finds it, we say the calculus enjoys the *strong normalisation property*.

Some terms have no normal form: e.g. term $\Omega = (\lambda x(x\ x)\ \lambda x(x\ x))$ is not in β-normal form, but it only β-reduces to itself. Some terms with a normal form have non-terminating sequences of reductions: e.g. term $(\lambda x \lambda y(x)\ \lambda x(x)\ \Omega)$ β-reduces to $\lambda x(x)$ (a normal form) by its first redex, and it β-reduces to itself by the redex in Ω. Repeating the second reduction produces a non-terminating sequence of reductions.

2.3 λ-Terms and Logic Programming

The motivation to manipulate λ-terms in logic programming is simple: they are the most natural representation for structures that feature either scoping, compositionality, or some form of genericity [32].

Scoping, compositionality and genericity are not exclusive. Among the structures that feature scoping, we see logical and mathematical formulas (logical quantifications: $\forall u$, ...; sums and products: $\sum_{x \in X} x$, $\int_0^1 f(x).dx$, ...; derivations: d/dx, ...), and computer programs (parameterisation: *f(x) int x;* { ... } , ...; blocks: { *int x;* ... } , ...). Among those that feature compositionality, we see expressions of compositional semantics (denotational: $\mathcal{T}_g[\![B_1, B_2]\!] = \lambda\kappa(\mathcal{T}_g[\![B_1]\!] (\mathcal{T}_g[\![B_2]\!] \kappa))$ [8], ...; Montague semantics for natural language [12]). Finally, logical quantifications in automated demonstration feature a form of genericity: one instantiates them using substitutions for building proofs.

The following table pictures some possible representations using λ-terms.

$\forall u.P(u)$	*forall* λ*u(P u)*
$\int_0^1 f(x).dx$	*integral 0 1* λ*x(f x)*
df/dx	*derivative* λ*x(f x)*
f(x) int x; { ... }	*function f int* λ*x(...)*
{ *int x;* ... }	*block int* λ*x(...)*
$\mathcal{T}_g[\![B_1, B_2]\!] = \lambda\kappa(\mathcal{T}_g[\![B_1]\!] (\mathcal{T}_g[\![B_2]\!] \kappa))$	*t_g (B1 and B2)* λ*k(D1 (D2 k)) :- t_g B1 D1, t_g B2 D2*

Scoping introduces the notion of *scoped variable* or *parameter*. The key operation for composing structures is the *substitution* of a term to a parameter; it must be sound with respect to scoping. Axiom β models such a substitution. This is why λ-terms are well suited for representing these structures: abstraction serves as a generic quantification. All these structures can be represented with first-order terms, but the correct handling of substitution with respect to scoping needs a lot of attention. The most frequent solution is to represent the parameters (the *object-level variables*) with Prolog variables (the *metavariables*). There are numerous examples of this solution in text-books. They can be found in chapters dedicated to the manipulation of programs and formulas.

We take as an example the following clause from program 16.2 (page 259) in *The Art of Prolog* by Sterling and Shapiro [47][1].

translate((A,B), (A1,B1), Xs\Ys) :- translate(A, A1, Xs\Xs1) , translate(B, B1, Xs1\Ys) .

This clause is part of a program for translating grammar rules into Prolog. Every clause of predicate *translate* has the same structure. The first parameter is a component of some grammar rule, the second is the corresponding component in the target Prolog clause, and the last one is a pair of variables that must occur in the target Prolog clause. So, these two variables are object-level variables. The problem is that it is no longer possible to give a declarative reading to this clause because the object-level variables are represented directly with metavariables. The declarative reading of Prolog does not spell out the actual rôle of *Xs*, *Xs1* and *Ys*. They are variables of the target clause, and it makes no sense to consider instances of them as the declarative semantics of Prolog does. It works in Prolog because the interpreter always computes the most general solutions. So, it is the operational semantics that gives the meaning of this clause, instead of the declarative semantics.

In the same book, clause *derivative(X,X,s(0))* (program 3.29, page 63) is part of a program that computes the derivative with respect to X of a function. One of its logical consequences is *derivative(12,12,s(0))*, which is absurd. Clause *polynomial(X,X)* (program 3.28, page 62) is a part of the definition of what a polynomial in X is. It has also absurd logical consequences like *polynomial(12,12)*.

In Prolog, substitution of object-level variables is easy at the price of declarativity. Then, it forces the programmer to check the correctness of object-level terms manipulations with respect to the operational semantics.

Instead of the above clauses, we would like to write the following ones[2]:

derivative(λ*x(x), s(0)) .*
polynomial(λ*x(x)) .*
translate((A,B), λ*xs*λ*ys(exists* λ*xs1(A1 xs xs1, B1 xs1 ys)) :- translate(A, A1) , translate(B, B1) .*
exists G :- (G _) .

λ-Terms permits a coherent and declarative handling of scopes and substitutions.

[1] This is an almost verbatim copy of the clause. The only modification is for avoiding a clash with notations used in this article. In the original text, ':-' is noted $\leftarrow$.

[2] It is not only the matter of these clauses, the predicates they belong to must be rewritten entirely along the same scheme.

3 Sequent Proof Theory for Horn Clauses

We present a language of clauses and goals that encompasses the language of Horn clauses. It is a convenient basis for the extensions described in the sequel. Indeed, it stresses a fundamental asymmetry in the language of Horn Clauses that the extensions will eliminate.

Clauses and goals are generated by the following grammar:

$$\mathcal{D} \quad ::= \mathcal{A} \quad | \quad \mathcal{A} \Rightarrow \mathcal{D} \quad | \quad \forall\, \mathcal{V}_t\, \mathcal{D} \quad | \quad \mathcal{D} \wedge \mathcal{D} \qquad \text{(iii)}$$

$$\mathcal{G} \quad ::= \mathcal{A} \qquad \text{(iv)}$$

Rules (iii) and (iv) generate *clauses* and *goals*, respectively.

The semantics of the connectives is given by the following deduction rules of the intuitionistic sequent calculus[3].

$$\frac{P, D[x \leftarrow t] \vdash G}{P, \forall x D \vdash G} \quad \forall_{\mathcal{D}} \qquad t \text{ is an arbitrary term.}$$

$$\frac{P, D_1, D_2 \vdash G}{P, D_1 \wedge D_2 \vdash G} \quad \wedge_{\mathcal{D}}$$

$$\frac{P, D \vdash G \qquad P \vdash A}{P, A \Rightarrow D \vdash G} \quad \Rightarrow_{\mathcal{D}}$$

All these rules are left introduction rules; their connective of interest is in the left part of the conclusion sequent.

To show that this language encompasses Horn formulas, we observe that the right introduction rules of more connectives can be defined by second-order clauses[4]:

$((G_1 \wedge_{\mathcal{G}} G_2) \Leftarrow_{\mathcal{D}} G_1) \Leftarrow_{\mathcal{D}} G_2$

$((G_1 \vee_{\mathcal{G}} G_2) \Leftarrow_{\mathcal{D}} G_1) \wedge_{\mathcal{D}} ((G_1 \vee_{\mathcal{G}} G_2) \Leftarrow_{\mathcal{D}} G_2)$

Thus, we can include conjunctions and disjunctions in goals. Using them allows us to restrict the use of implication for building clauses. Furthermore, we adopt the convention that all conjunctions in clauses are pushed to the top of the $\mathcal{D}$-formulas, and that all universal quantifications are pushed right underneath (clausal form). So, we see that Horn formulas can serve as a concrete syntax for our language.

Because of the clausal form, conjunctions and universal quantifications in clauses need no more be noted explicitly because they can be reconstructed if required. So, the concrete syntax is as follows:

$$\mathcal{P} \quad ::= \mathcal{D} \quad | \quad \mathcal{D} \ .\ \mathcal{P} \qquad \text{(v)}$$

$$\mathcal{D} \quad ::= \mathcal{A} \quad | \quad \mathcal{A} \ \text{:-}\ \mathcal{G} \qquad \text{(vi)}$$

$$\mathcal{G} \quad ::= \mathcal{A} \quad | \quad \mathcal{G} \ ,\ \mathcal{G} \quad | \quad \mathcal{G} \ ;\ \mathcal{G} \qquad \text{(vii)}$$

Rules (v), (vi) and (vii) generates *programs*, *clauses*, and *goals*, respectively. A program is a sequence of input units, all terminated by a full-stop, '.'. Input units are clauses, and every variable that is free in an input unit is considered as universally quantified at the clause level. We call these variables *logical variables* or *unknowns*.

In rule (vi), terminal ':-' is the concrete writings for connective $\Leftarrow_{\mathcal{D}}$. Connective $\forall_{\mathcal{D}}$ has no concrete notation because it is implicit. In rule (vii), terminals ',' and ';' are the concrete writings for connectives $\wedge_{\mathcal{G}}$ and $\vee_{\mathcal{G}}$.

4 Adding λ-Terms and $\alpha\beta$-Equivalence

To add λ-terms to Prolog is still a fuzzy objective; one must decide the circumstances. A first observation is that there is no use for adding λ-terms without $\alpha\beta$-equivalence. Indeed, it is $\alpha\beta$-equivalence that allows us to substitute terms to variables with respect to scoping. So, system CLP(Λ,=) where '=' denotes Prolog's equality, or even CLP(Λ,$\equiv_\alpha$) without β-equivalence, are useless.

[3] A sequent $P \vdash G$ reads "goal G is a consequence of program P". A rule $\frac{Sequent^*}{Sequent}$ reads "conclusion $Sequent$ is true if all premises $Sequent^*$ are true".

[4] Occurrences of connective X in syntactic units of type $\mathcal{Y}$ ($\mathcal{D}$ or $\mathcal{G}$) are noted $X_{\mathcal{Y}}$.

4.1 Typing

In order to add λ-terms and $\alpha\beta$-equivalence to Prolog, one must also add the handling of axioms α and β to unification. One wants also to be sure that there is always a normal form to λ-terms, and that it can be found by any reduction strategy: the strong normalization property.

The domain of simply typed λ-terms has a (almost) well-behaved unification problem, and it is strongly normalizable. The unification problem for simply typed λ-terms modulo axioms α and β is semi-decidable and infinitary[5]. For instance, the unification problem $\lambda z(N\ \lambda x(x)\ z) = \lambda z(z)$ has an infinite number of solutions in N: θ_0=[$N \leftarrow \lambda s \lambda z(z)$], θ_1=[$N \leftarrow \lambda s \lambda z(s\ z)$], θ_2=[$N \leftarrow \lambda s \lambda z(s\ (s\ z))$], θ_i=[$N \leftarrow \lambda s \lambda z(s^i\ z)$], etc. Term $\lambda s \lambda z(s^i\ z)$ is the Church encoding of integer i.

In practice, one uses the semi-algorithm proposed by Huet [25]. The unification problem for simply typed terms is roughly the same whether axiom η is assumed or not. In fact Huet's semi-algorithm searches for pre-unifiers: i.e. substitutions that make the problem trivially solvable, but that are not necessarilly solutions.

Other, more sophisticated, typed domains exist that have also the strong normalization property and a practical unification problem [17, 16, 44, 43, 42].

4.1.1 Simple Types

The new term language is the language of the simply typed λ-terms.

Simple types are generated by the following grammar:

$$\begin{array}{lll} \mathrm{T} & ::= \mathcal{U} \quad | \quad (\ \mathcal{K}_i\ \mathrm{T}^i\ ) & \\ \mathrm{T} & ::= (\ \mathrm{T}\ \text{->}\ \mathrm{T}) & \text{(viii)} \end{array}$$

where the $\mathcal{U}$ and $\mathcal{K}_i$ are identifiers of type variables and type constructors with arity i, respectively. Rule (viii) generates function types. A type *(A->B)* can be interpreted as the type of functions whose domain is A and co-domain is B. We assume $\mathcal{K}_0$ contains at least the constant *'o'* for propositional types. We also assume the arrow, ->, associates to the right. This makes many brackets useless: for instance, *o->o->o* denotes the same type as *(o->(o->o))* does.

We assume type constructors are declared using a directive *kind*: for instance,

```
kind o type .                          % o ∈ K0
kind list type -> type .               % list ∈ K1
```

The declaration of *list* shows it is a type constructor that must be applied to some type to actually produce a type.

4.1.2 Typed Terms

Simply typed λ-terms are generated by the following grammar:

$$\begin{array}{lll} \Lambda_t & ::= \mathcal{C}_t \quad | \quad \mathcal{V}_t & t \in \mathrm{T} \\ \Lambda_{t'\text{->}t} & ::= \lambda \mathcal{V}_{t'}\ (\ \Lambda_t\ ) & t,\ t' \in \mathrm{T} \\ \Lambda_t & ::= (\ \Lambda_{t'\text{->}t}\quad \Lambda_{t'}\ ) & t,\ t' \in \mathrm{T} \end{array}$$

where the $\mathcal{C}_t$ and $\mathcal{V}_t$ are respectively identifiers of constants and identifiers of variables whose type is t. Attributes in terminal and non-terminal symbols are used to constrain types and to ensure the well-typing of generated terms.

Every term of the simply typed λ-calculus has a β-normal form. Because of the Church-Rosser property, it is unique. Moreover, it can be computed using any arbitrary strategy. Hence, the calculus has the strong normalisation property.

We assume the types of constants are declared using a directive *type*: for instance,

```
type [] (list T) .                                  % ∀T ([] ∈ C(list T))
type '.' T -> (list T) -> (list T) .                % ∀T ('.' ∈ CT->(list T)->(list T))
type append (list T) -> (list T) -> (list T) -> o . % ∀T (append ∈ C(list T)->(list T)->(list T)->o)
```

The type of *[]* shows it is a non-functional constant. The type of *'.'* shows it is a function that takes two arguments. These two constants allows us to build *homogeneous* polymorphic lists: polymorphic lists all elements of which have the same type. Finally, the result type of *append*, *'o'*, shows it is a predicate constant.

[5] There can be infinitely many most general unifiers.

4.1.3 Typed Programs

Having chosen a type discipline for the terms does not say everything on the type discipline of programs.

The first point is that we adopt the *prescriptive typing* point of view: the typing discipline defines a *well-typing* predicate, and not well-typed programs (*ill-typed* programs) are simply not considered as programs.

The usual principles of polymorphic typing are transposed to CLP(Λ, $\equiv_{\alpha\beta}$):

- *type* declarations act as a *let* in ML [39], it introduces a type scheme for some constant symbol,
- quantifications (including abstraction) act as an abstraction in ML,
- every occurrence of a constant symbol has a type that is an instance of the type scheme of that constant,
- and every occurrence of a variable has the same type.

The second point is that we adopt the *definitional genericity* principle: *"Every occurrence of a predicate constant in the head of a clause must have a type that is a renaming of its type scheme"*. In other words, the types at these occurrences cannot be just any instance of the type scheme; only renamings are allowed.

This principle is assumed in the work by Mycroft and O'Keefe [40], but not discussed. It appears under the name *head condition* in works by Hanus, and Hill and Lloyd [21, 22]. It also appears under the name *definitional genericity* in Lakshman and Reddy's work [29]. Hanus have shown it is a necessary condition for a *semantic soundness* theorem stating roughly that *"Well-typed programs cannot go wrong"*.

An example illustrates the effect of definitional genericity. Assuming the curried syntax for terms, the CLP(Λ, $\equiv_{\alpha\beta}$) writing for predicate *append* is as follows:

append [] X X .
append [E|X] Y [E|Z] :- append X Y Z .

This program is well-typed with respect to the type schemes given in section 4.1.2. However, the same program plus the clause

append [0,1,2,3,4] [5,6,7,8,9] [0,1,2,3,4,5,6,7,8,9] .

is ill-typed because it violates the head-condition. The occurrence of *append* in the above clause has a type, *(list int)->(list int)->(list int)->o*, that is a strict instance of the type scheme, not just a renaming.

We note CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) the instance of CLP in which the domain is of simply typed terms ($\Lambda_{\rightarrow}$), the equality theory is $\alpha\beta$-equivalence ($\equiv_{\alpha\beta}$), the programs are typed prescriptively, and the definitional genericity principle is adopted.

4.2 Adding $\alpha\beta$-Equivalence

We have seen that some terms of CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) (i.e. those with an arrow type) can be interpreted as functions. We have also seen that proofs in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) are done modulo $\alpha\beta$-equivalence. So, it looks like if one can program in a functional style in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$), and that unification is able to discriminate applications from abstractions and to analyse their components. We will show that these are two misconceptions about the capabilities of λ-terms in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$).

4.2.1 Programming in a functional style in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$)

One must recall that CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) terms are essentially simply typed λ-terms. As such, they have a computing power that is too weak for really serving as a programming language.

ML programs are also essentially made of simply typed λ-terms, but the language also offers a construction (i.e. *letrec*) that is interpreted by a fix-point combinator. That is what gives its computing power to ML.

What gives CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) its computer power (which is the same as for any reasonable programming language) is the structure of its clauses, but not its λ-terms. In other words, β-reduction in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) is not really used for evaluating functions. It is mainly used for instantiating term schemes represented by λ-terms.

However, λ-terms allow us to functionally encode operations that are sufficiently simple. Since variables are allowed in types, but the quantifications of these variables are always prenex, the domain of CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) terms is essentially equivalent to ML$^-$ (ML minus *letrec*). This domain allows us to program extended polynomial functions only [24]. For instance, one may encode concatenation as a λ-term, via a suitable encoding of lists as functional lists [9].

4.2.2 Discriminating applications from abstractions in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$)

A λ-term can be a constant, a variable, an abstraction, or an application. We will show that it is impossible to discriminate abstractions from applications, and to access their components without using a specific programming discipline.

In CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) as in Prolog, the only means for discriminating terms and for accessing their subterms is unification. Hence, a naïve solution to recognize that a term is an application and to access its components is to unify it with the term *(T1 T2)* in order to bind its components to variables T_1 et T_2. In fact, it is much too naïve because term *(T1 T2)* is unifiable modulo $\alpha\beta$-equivalence with *any* term. For instance, terms *12*, *(A B)* and $\lambda x(x)$ are all unifiable with *(T1 T2)*. All three problems have an infinite number of solutions, among which

- problem *12=(T1 T2)* has solution *[T1←λx(12), T2←13]*,
- problem *(A B)=(T1 T2)* has solution *[T1←λx(x B), T2←A]*,
- and problem $\lambda x(x)$*=(T1 T2)* has solution *[T1←λx(x), T2←λx(x)]*.

In the case of term *(A B)*, the solution that allows us to actually access A and B is lost among the infinitely many others without any means for distinguishing it. Hence, the property of being unifiable with an application does not discriminate applications from other terms, and unification does not allow us to access to the components of applications.

Similarly, a too naïve solution to recognize that a term is an abstraction and to access its component (the body) is to unify it with term $\lambda v(T)$ in order to bind the body of the abstraction to variable T. This does not work because term $\lambda v(T)$, which does indeed only unify with abstractions, does not unify with every abstraction. The problem is that the substitution theory that underlies higher-order unification forbids capturing λ-variables outside the abstraction that bind them; binding values of logical variables and parameters of goals must be closed terms.

However, by unifying an abstraction (say $\lambda x(x)$) with $\lambda v(T)$, and expecting that it will bind its body (here x) to T, we are precisely trying to capture a variable. In fact, the body of some abstraction can be substituted to variable T only if the body does not contain any free occurrence of the bound variable. The term $\lambda v(T)$ describes precisely those abstractions that represent constant functions. Hence, to be unifiable with $\lambda v(T)$ does distinguish only very particular abstractions.

One may compare these difficulties with well-known Prolog difficulties. The language of Prolog terms allows for variables in terms, but there is no means in pure Prolog for checking that a term is a variable. In fact, the semantics of Prolog actually uses a saturation of the term domain, the Herbrand universe, in which there is no room for variable terms. Similarly, the semantics of CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) uses $\alpha\beta$-equivalence, which leaves no room for the application/abstraction discrimination.

The problem has two sides: to discriminate applications from abstractions, and to access their components. The solution for the first side is to label with a constructor the applications and abstractions we want to recognize. For instance, the representation of ML terms in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) can use the two following constructors:

```
kind mlt type .
type app mlt -> mlt -> mlt .          % Application: write (app F X) instead of (F X)
type fun (mlt->mlt) -> mlt .          % Abstraction: write (fun λx(x+1)) instead of λx(x+1)
```

Using this technique, one can easily discriminate abstractions from applications. One can also access the components of some discriminated application. The problem of accessing the body of an abstraction is solved in the next section.

For the purpose of further examples, we declare kinds and types of constants for representing formulas of the first-order predicate calculus and terms of the simply typed λ-calculus. We need two types for representing the formulas of the first-order predicate calculus: a type for truth values, and a type for individuals. They are *object-level* truth values and individuals. They are represented by CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) λ-terms, but they must not be mistaken with CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) formulas or terms.

```
kind (formula, individual) type .
```

One also needs connectives for object-level formulas.

```
type (and, or, impl) formula -> formula -> formula .
type not formula -> formula .
```

```
type (forall, exists) (individual->formula) -> formula .
type (p, ...) individual -> individual -> formula .
type (q, ...) formula .
```

So, formula $\forall x(p(x,x) \Rightarrow q)$ is encoded by *(forall λx(impl (p x x) q))*. The only constructors that are original with respect to Prolog are *forall* and *exists*. They have an argument that is an abstraction. Its rôle is to formalize the scope of object-level quantifications and to handle the fact that quantified variables are substitutable.

One also needs two types for representing the terms of the simply typed λ-calculus: a type for λ-terms and another for simple types. As for the representation of object-level formulas of the predicate calculus, they must not be mistaken for the metalevel λ-terms and types.

```
kind (l_term, simple_type) type .
```

One needs constructors for representing object-level applications and abstractions, the arrow of object-level simple types, and constant object-level types and terms.

```
type app l_term -> l_term -> l_term .
type abs (l_term->l_term) -> l_term .
type arrow simple_type -> simple_type -> simple_type .
type (one, two, three, ...) l_term .
type (integer, real, ...) simple_type .
```

The only constructor that is original with respect to Prolog is *abs*. Its argument is an abstraction of CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) whose rôle is to formalize the scope of the object-level abstraction and the fact that the bound variable is substitutable.

It is important to understand that the CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) abstraction in the representation does not model all the semantics of the object-level abstraction. For instance, *(app (abs E) F)* $\not\equiv_{\alpha\beta}$ *(E F)*. If such relation holds at the object level it must be described in CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) by a suitable predicate:

```
type beta_conv l_term -> l_term -> o .
beta_conv (app (abs E) F) (E F) .
```

In this clause, the metalevel β-reduction of *(E F)* performs the actual substitution of term *F* to the variable bound by *(abs E)*.

4.3 Axiom η

We have added simply typed λ-terms and $\alpha\beta$-equivalence to Prolog. Does this result in a usable logic programming language? The answer is no.

We have shown in section 4.2.2 that unification (even higher-order) alone does not allow us do discriminate abstractions from applications or to access their components. Having added simply typed λ-terms and $\alpha\beta$-equivalence to Prolog, we are able to build and compare them, but we cannot analyse them. In other words, we cannot perform an inductive traversal of the terms we are able to build.

The difficulty with abstractions comes from the fact that it is not possible to capture free λ-variables in bindings of logical variables.

The intuition of the solution is that to access the body of an abstraction *A*, the only means is to apply *A* to a term *t* and use term *A'=(A t)*. Knowing *A'* and *t*, one must be able to compute *A* by solving *A'=(X t)* for an unknown *X*. So, the equality theory of the λ-terms and the term *t* must be such that given two distinct terms *A* and *B*, the terms *(A t)* and *(B t)* are also distinct. We call this condition the *conservation condition*. It ensures that accessing the body of an abstraction is a reversible operation.

To apply an abstraction to a term in order to access its body does not allow us to discriminate between η-equivalent terms (e.g. *E* and *λx(E x)*). Indeed, *(λx(E x) t)* $\equiv_{\alpha\beta}$ *(E t)* even if *λx(E x)* $\not\equiv_{\alpha\beta}$ *E*. Hence, the conservation condition implies η-equivalence.

So, CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta}$) is not yet a usable logic programming system. We note CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta\eta}$) the instance of CLP in which the domain is of simply typed terms ($\Lambda_{\rightarrow}$), and the equality theory is $\alpha\beta\eta$-equivalence ($\equiv_{\alpha\beta\eta}$).

4.4 Universal Quantification in Goals : $\forall_{\mathcal{G}}$

4.4.1 Discussion

We have seen that we plan to analyse an abstraction A by applying it to some term t and analysing the result *A'=(A t)*.

If one wants to be able to compute A by solving *A'=(X t)* for an unknown X, it must be that t is recognizable among the subterms of *A'*. As a counter-example, if *A=λx(x+12)* and *t=12*, we get *A'=(12+12)*. The equation *(12+12)=(X 12)* has four solutions among which there is no formal reason to prefer one: *[X←λx(x+x)]*, *[X←λx(x+12)]*, *[X←λx(12+x)]*, and *[X←λx(12+12)]*. The four solutions correspond to four A's that are not $\alpha\beta\eta$-equivalent but still yield the same term *(A t)* for *t=12*. The underlined solution is the one we informally prefer. We need some formal means to select this one.

One could object that it was really gross to choose *t=12* when A already has *12* as a subterm. This objection does not hold for two reasons. First, because a term A can have unknown subterms, it is not always possible to check that a candidate t does not occur in A. Second, even when the term A is fully determined (ground), to choose a t that does not occur in it does not completely solve the problem. For instance, if *A=λx(x+12)* and *t=13*, we get *A'=(13+12)*. Equation *(13+12)=(X 13)* has two solutions among which there is still no formal means for selecting one: *[X←λx(x+12)]* and *[X←λx(13+12)]*. Again, two non-$\alpha\beta$-equivalent A's yield the same *(A t)* for *t=13*. In fact, for any equation *(A t)=(X t)* in X, and if t is an ordinary term, there are always at least solutions *[X←λx(A t)]* and *[X←A]*.

Something must prevent term t from occurring in A and in X. This will bar all the solutions that are not underlined. In other words, it will provides the formal means we want for preferring the underlined solutions. Universal quantification in goals, $\forall_{\mathcal{G}}$, is such a logical means for producing a recognizable term t. Its deduction rule is as follows:

$$\frac{P \vdash G[x \leftarrow c]}{P \vdash \forall x\, G} \quad \forall_{\mathcal{G}} \qquad c \text{ occurs free neither in } P \text{ nor in } G.$$

In order to be coherent with the distinction made between $\mathcal{G}$-formulas and $\mathcal{D}$-formulas one must restrict the universally quantified formulas to be $\mathcal{G}$-formulas. This restriction is compatible with our motivation for adding implication in goal.

The operational semantics of this new connective is as follows. To prove a goal *($\forall_{\mathcal{G}}$ v G)*, prove goal *G[v←c]*, where c is a new constant which has the type of v, taking care that c does not occur in the binding values of older logical variables.

If t is universally quantified in the scope of the quantifications of A and X, then the only solution of equation *(A t)=(X t)* is *[X←A]*. The goal to prove has the following form: $\exists A \exists X \forall_{\mathcal{G}} t(A\ t)=(X\ t)$. Because of η-equivalence, this goal is equivalent to $\exists A \exists X (A=X)$, which has the trivial solution *[X←A]* [6].

This shows a fundamental correspondence between abstraction and universal quantification. They cannot be considered independently. Informally speaking, abstraction is an essentially universal quantification operating at the term level. Abstraction is a storable/manipulable form of universal quantification, and universal quantification is the way for interpreting abstraction.

4.4.2 Application

We apply the technique of using $\forall_{\mathcal{G}}$ to the task of relating a first-order predicate calculus formula to its negative normal form.

A first-order predicate calculus formula is in negative normal form if the negation connective is only applied to atomic formulas. For instance, *(¬A)∨(¬B)* is in negative normal form, but *¬(A∧B)* is not. It is always possible to make a first-order predicate calculus formula into the negative normal form using De Morgan's identities.

We first declare the type of the relation.

```
type nnf formula -> formula -> o .
```

Now we define relation *nnf* by structural induction on type *formula*. The cases for all the connectors are elementary and could be described in Prolog.

```
nnf (and F1 F2) (and G1 G2) :- nnf F1 G1 , nnf F2 G2 .
nnf (or F1 F2) (or G1 G2) :- nnf F1 G1 , nnf F2 G2 .
```

[6] Other most general solutions exist, but they are renaming of this one.

```
nnf (not (and F1 F2)) (or G1 G2) :- nnf (not F1) G1 , nnf (not F2) G2 .
nnf (not (or F1 F2)) (and G1 G2) :- nnf (not F1) G1 , nnf (not F2) G2 .
nnf (p A B) (p A B) .
nnf (not (p A B)) (not (p A B)) .                                   % nnf1
...
```

The cases for quantifications is more interesting. One must continue the induction in the quantified formula. Then a universal quantification in goal is required. It introduces a universal constant of type *individual*, which is not the induction type. Then, augmenting the inductive definition for this new constant is not required.

```
nnf (forall F) (forall G) :- ∀G i( nnf (F i) (G i) ) .
nnf (exists F) (exists G) :- ∀G i( nnf (F i) (G i) ) .              % nnf2
nnf (not (forall F)) (exists G) :- ∀G i( nnf (not (F i)) (G i) ) .
nnf (not (exists F)) (forall G) :- ∀G i( nnf (not (F i)) (G i) ) .  % nnf3
```

A few observations are in order.

- The metalevel universal quantification has nothing to do with the semantics of object-level formulas. It has only to do with their structure. The two object-level quantifications, *exists* and *forall*, presumably respectively existential and universal, are both interpreted by a universal quantification.
- When we use predicate *nnf* for normalizing a formula, the universal quantification works both in analysing and in synthesising abstractions. Abstraction F is analysed and its body passed as a parameter to the recursive call to *nnf*. The second argument is unified with a term *(exists G)* or *(forall G)* where G is an unknown. The application of G to i is also passed as a parameter to the recursive call to *nnf*. The unknown G being quantified out of the scope of the i, it cannot have any occurrence of i in its binding values. Hence, it is bound to an abstraction that becomes more and more precise as long as formula F is traversed.

We present the processing of an actual proof for illustrating the last observation.

1. The question.
   (nnf (exists λx(not (exists λy(p x y)))) X)
2. Resolution with clause nnf_2 of *nnf* (page 12): *[X←(exists G)]*.
   ∀G i(nnf (not (exists λy(p i y))) (G i))
3. Rule $\forall_{\mathcal{G}}$.
   (nnf (not (exists λy(p c y))) (G c))
4. Resolution with clause nnf_3 of *nnf*: *[G←λx(forall (H x)),G'←(H c)]* is the most general solution to *(G c)=(forall G')* [25].
   ∀G i(nnf (not (p c i)) (H c i))
5. Rule $\forall_{\mathcal{G}}$.
   (nnf (not (p c c')) (H c c'))
6. Resolution with clause nnf_1 of *nnf*: *[H←λxλy(not (p x y))]*.
   Success
7. The solution is *[X←(exists λx(forall λy(not (p x y))))]*.

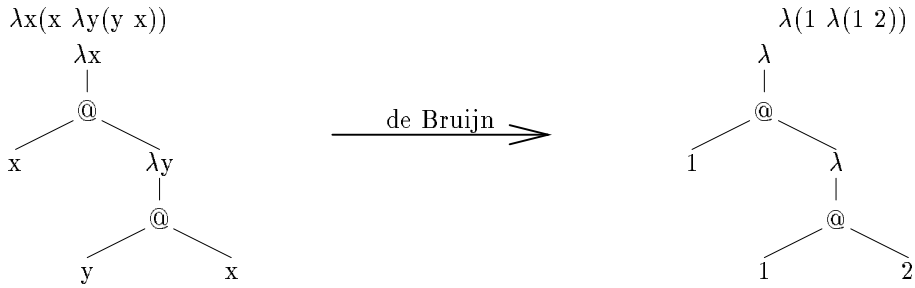
4.5 Implication in Goals : $\Rightarrow_{\mathcal{G}}$

We have seen that the universal quantification allows us to interpret abstractions. We also have seen in section 4.4.1 that the deduction system interprets universal quantifications by substituting a *new* constant to the universal variable. The constant is simply added to the current signature.

4.5.1 Discussion

This new constant is a new problem: how can the programmer take it into account? It is trivial that, because the constant is new, no predicate definition can take it into account. Predicates that are supposed to be defined for every constructor of a data-structure are no longer completely defined when a new constant is introduced.

We illustrate this on an example. Suppose we want to define a predicate for computing de Bruijn's representation of λ-terms [15]. The purpose of de Bruijn's representation is avoid names clash problems, which are usually solved by α-conversion, by simply eliminating names. The idea is to replace every occurrence of names by a number that count how many abstractions separate this occurrence of the name from the abstraction that binds it. The following diagram briefly illustrates de Bruijn's representation using the textual notation for λ-terms and their graph notation.



Object-level λ-terms are supposed to be represented with constants *app* and *abs* (see page 10). De Bruijn's trees can be represented by the following constants:

```
kind db_tree type .
type app_db db_tree -> db_tree -> db_tree .
type abs_db db_tree -> db_tree .
type var_db int -> db_tree .
```

The definition of the relation begins as follows:

```
de_bruijn l_term -> db_tree -> int -> o .

de_bruijn (app T1 T2) (app_db D1 D2) N :- de_bruijn T1 D1 N , de_bruijn T2 D2 N .
de_bruijn (abs T) (abs_db D) N :- ∀G x( de_bruijn (T x) D (succ N) ) .
```

The procedure will eventually reach x at some leaf of the term. At this point, one would like to use the following clause:

```
de_bruijn x (var_db Ix) Nx :- plus N Ix Nx .
```

Variable N is underlined to stress its special status. It is supposed to be the same as the N in the instance of the last clause of the program where x is introduced. This contradict the usual convention that the scope of variables is limited to the clause it occurs in. Variables *Ix* and *Nx* are perfectly normal. They are local to the clause. Variable *Nx* gets its value when the head is unified with a calling goal, and variable *Ix* gets its value when proving the body.

Furthermore, there should be one such clause for every x created during the execution of predicate *de_bruijn*. This is impossible to achieve using standard Prolog tools because one does not know in advance what the x's will be.

We need some means for augmenting a predicate definition during the life of a new constant: i.e. during a subproof. This means must tolerate clause definitions with free variables. Implication in goals, $\Rightarrow_{\mathcal{G}}$, is such a means because it allows us to augment the program for the life-time of a subproof.

The introduction rule for implication in goals is as follows:

$$\frac{P, D \vdash G}{P \vdash D \Rightarrow G} \quad \Rightarrow_{\mathcal{G}}$$

In order to be coherent with the distinction made between $\mathcal{G}$-formulas and $\mathcal{D}$-formulas one must restrict the premises of implications in goal to be $\mathcal{D}$-formulas, and the conclusions to be $\mathcal{G}$-formulas. This restriction is compatible with our motivation for adding implication in goal.

The operational semantics of implication in goal is as follows. To prove a goal $D \Rightarrow_{\mathcal{G}} G$, add clause D to the program and prove G. Clause D is kept in the program during the proof of G. It is suppressed from it as soon as the proof of G is over.

Using implication in goals, the clauses for *abs* and for x are as follows:

```
de_bruijn (abs T) (abs_db D) N :-
      ∀G x  (     ∀D Nx(∀D Ix( de_bruijn x (var_db Ix) Nx :- plus N Ix Nx ))
            ⇒G    de_bruijn (T x) D (succ N) ) .
```

The $\forall_{\mathcal{D}}$'s in the assumed clause make variables *Nx* and *Ix* local to the clause, as it is initially intended. They correspond to the quantifications that are usually implicit. In this case, the nesting of clauses forces us to make them explicit for avoiding confusion of scopes. Variable *N* is intentionally left free in the assumed clause.

4.5.2 Application

We use as an application the problem of relating a λ-term to its type.

The well-typing relation is defined by structural induction on object-level terms. The type of the well-typing relation is as follows:

```
type typing l_term -> simple_type -> o .
```

Then it is defined by induction on the constructors of type *l_term*. The case for applications is elementary and could be written in Prolog.

```
typing (app T1 T2) B :- typing T1 (arrow A B) , typing T2 A .
```

The case for abstractions is more interesting. The induction through object-level abstractions uses a universal quantification because object-level abstractions are represented by CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta\eta}$) abstractions. This universal quantification introduces a universal constant of type *l_term*, which is the type on which the induction operates. So, one must augment the inductive definition for the life-time of the universal constant using an implication.

```
typing (abs E) (arrow A B) :- ∀G x( typing x A ⇒G typing (E x) B ) .
```

The added clause is *(typing x A)*. It contains a free logical variable *A*. This forces all the occurrences of constant x to have the same type. If the added clause had been $\forall_{\mathcal{D}}$ *T(typing x T)*, every occurrence would have had its own type.

The above sequence of reasoning provides only the structure of the CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta\eta}$) program. To work out the full details needs to know the logic of the defined relation. In this case, the logic is given by the deduction rules of the theory of simple types [2, 24]. For the abstraction, the rule is called *arrow introduction*:

$$\frac{\Gamma, x : \alpha \quad \vdash \quad E : \beta}{\Gamma \quad \vdash \quad \lambda x.E : \alpha \rightarrow \beta} \rightarrow_I$$

One may wonder why an implication was required for relations *typing* and *de_bruijn*, but not for relation *nnf*. The answer is in the notion of inductive type. These relations are defined by an induction on the constructors of the λ-terms representation of one of their parameters. If the type of this parameter is *inductive* then it is easy to deduce an induction function on the constructors [5].

One says the type of a data structure is inductive if its constructors admit arguments of that same type only in positive occurrences [46]. We recall that following the *Curry–Howard isomorphism* [13, 23] the arrow of simple types is analogous to implication in propositional intuitionistic calculus. As does implication, arrow introduces a notion of positive and negative occurrences as follows:

```
pos(A->B)  def=  neg(A) ∪ pos(B)
neg(A->B)  def=  pos(A) ∪ neg(B)
pos(T)     def=  {T}          % if T is not an arrow type
neg(T)     def=  ∅            % if T is not an arrow type
```

For instance, *pos((a->b)->(c->d))*={*a,d*} and *neg((a->b)->(c->d))*={*b,c*}.

What causes the differences between the relations is that type *l_term* is not inductive, whereas type *formula* is inductive. When the type is inductive, implication is not required because there cannot be new constants of that type; when it is not inductive, implication is required for handling the new constants of that type.

Let us show that type *l_term* is not inductive. Constant *abs* is a constructor of *l_term*, and its type is *(l_term->l_term)->l_term*. Because *l_term* has a negative occurrence in the type of the only argument of *abs* (*neg(l_term->l_term)*={*l_term*}) it is not inductive.

Let us show that type *formula* is inductive. Its constructors are either like *and*, or like *forall*. Every argument of *and* has type *formula*. The only argument of *forall* has type *individual->formula*. Because *formula* does not occur negatively in them (*neg(formula)*=$\emptyset$ and *neg(individual->formula)*={*individual*}) it is inductive.

If the object-level theory in the negative normal form example had been a higher-order logic in which variables of type *formula* can be quantified, then type *formula* would not have been inductive, and it would have been required to augment the inductive definition using implication.

4.6 Summing-up

This section can be summed-up as follows: for adding λ-terms and $\alpha\beta$-equivalence to Prolog, one needs to type terms in a discipline that makes the unification problem well-defined. For defining relations by structural induction on abstractions, and still satisfy the conservation condition, one must also add η-equivalence and universal quantification in goals. To be able to maintain the completeness of the inductive definitions in presence of universal variables (new constants) of a non-inductive type, one needs to add implication in goals. All this forms the core of λProlog [36, 38, 37]. Formulas generated as presented above are called *hereditary Harrop formulas*.

We have already observed that the right introduction rules of $\wedge$, $\vee$ can be defined by second order Horn clauses. The introduction rule for $\exists$ can also be defined by a second order Horn clause using higher-order terms. Its definition is as follows:

$(\exists_{\mathcal{G}}\ G\ ) \Leftarrow_{\mathcal{D}} (G\ X)$

Thus, all the connectives that belong to $\mathcal{G}$-formulas, but not to $\mathcal{D}$-formulas (i.e. $\exists$ *and* $\vee$), are definable in the language of $\mathcal{D}$-formulas. So, they can be suppressed from the $\mathcal{G}$-formulas, making the two classes of formulas identical. The result is that the logic of λProlog (or of CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta\eta}$)) is the logic of the formulas freely constructed with connectives $\forall$, $\wedge$, and $\Rightarrow$.

The semantics of λProlog is usually given in proof-theoretic terms [38, 37], as opposed to the model-theoretic semantics used for Prolog [30]. The main result is that a class of goal-directed proofs, called *uniform proofs*, is complete with respect to intuitionistic provability for hereditary Harrop formulas. In other words, every hereditary Harrop formula that is a theorem in intuitionistic logic has a uniform proof. In still other words, restricting proofs to be uniform eliminates proofs, but does not eliminate any theorem among hereditary Harrop formulas.

The word "scope" sums-up the new constructs of λProlog:

- abstraction limits the scope of variables in terms,
- quantifications limit the scope of variables in formulas,
- and the deduction rules for universal quantification and implication limit the scope of constants and clauses, respectively, in the proof process.

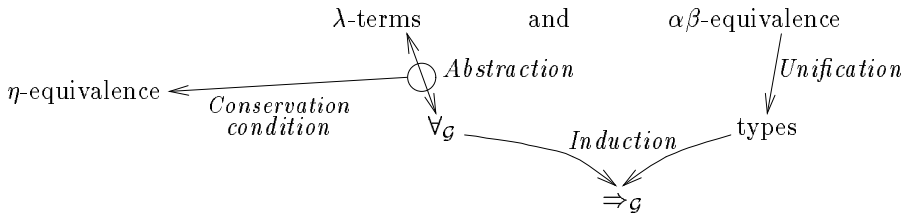
Predicates *nnf* (pp. 11), *de_bruijn* (pp. 13), and *typing* (pp. 14) show that programming in λProlog often amounts to make the three scoping levels interact.

5 Conclusion

5.1 The Structure of λProlog (Alias CLP($\Lambda_{\rightarrow}$, $\equiv_{\alpha\beta\eta}$))

5.1.1 A Summary

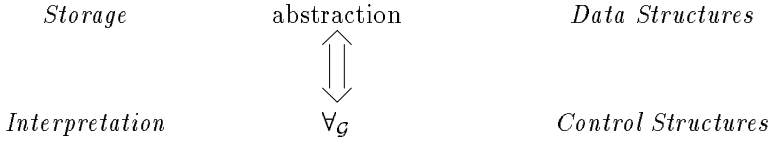
The following picture illustrates the relationships that give a structure to the features of λProlog. Arrows read "requires".



We consider λ-terms and $\alpha\beta$-equivalence as the principal components of λProlog; those that draw every other component. Adding these components to Prolog is motivated by the need for a more declarative handling of notions such as scoping and substitution. Then come types for making unification well-defined and tractable, and universal quantification in goals for handling induction through abstractions. Then η-equivalence in the equality theory is required for making the interpretation of abstraction by universal quantification correct and reversible. Finally, for inductive definitions to remain complete with respect to types when universal quantification introduces new constants, implication is needed for completing the induction.

Among the preliminary designs for CLP($\Lambda_{\rightharpoonup}$, $\equiv_{\alpha\beta\eta}$) we have presented, we see that CLP(Λ, $\equiv_{\alpha\beta}$) and CLP($\Lambda_{\rightharpoonup}$, $\equiv_{\alpha\beta}$) are useless because they feature components that are high in the diagram without featuring the components that are below them (i.e. that come as a consequence).

The symmetry between abstraction and universal quantification in goals makes us consider the first as an essentially universal quantification. This is an important point for programming. This duality is often the cause of a conjoint use of abstraction and universal quantification in goals. It may be pictured as follows:



Abstraction is a *reified* form of universal quantification, the latter being a *reflection* of the former. For analysing/consuming/reflecting an abstraction, one must apply it to a universal variable whose scope is included in the scope of the variable that is bound to the abstraction. For synthesising/producing/reifying an abstraction, it is required to build a term that represents its body in which a unique universal variable represents every occurrence of the variable bound by the abstraction, and compute the function that yields that term when it is applied to the same universal variable (e.g. the proof of *(nnf (exists λx(not (exists λy(p x y)))) X)*, page 12).

Finally, note that the analogy between λProlog and a CLP language still works at the implementation level. Implementing the unification algorithm raises issues on delaying under-specified problems that are familiar to CLP implementers [31].

5.2 The Structure of a Restriction of λProlog

All the steps leading from CLP(Λ, $\equiv_{\alpha\beta}$) to λProlog offer no alternative, except for the way of making the λ-calculus strongly normalizable and the corresponding unification problem tractable.

Miller proposes a fragment of λProlog, L_λ, that restricts more strongly the term domain than simple types do. In this fragment, the unification problem is decidable and unitary [34]. The term domain of L_λ is the greatest subset of λ-terms for which β_0-equivalence (defined below) is equal to β-equivalence.

β_0 — $(\lambda x(E)\ x) \equiv_{\beta_0} E$.

> This axiom formalizes a weak form of β-equivalence for the case the actual parameter is a variable. β_0-Reduction amounts to variable renaming.

The subdomain of the λ-terms for which β_0 is complete with respect to β is described by a restriction of the rule for building applications. The only allowed applications with a variable head are those whose head is applied to distinct essentially universal variables. A similar restriction applies to all systems of Barendregt's cube [3, 2] to make their unification problems unitary and decidable [45].

Despite its algorithmic interest, the L_λ fragment is not widely used because lots of useful definitions do not belong to L_λ. For instance, the definition of $\exists_{\mathcal{G}}$ (p. 15) does not belong to L_λ because of term *(G X)*. Relation *mapfun* is not in L_λ either.

```
type mapfun (A->B) -> (list A) -> (list B) -> o .
mapfun F [] [] .
mapfun F [E|L] [(F E)|FL] :- mapfun F L FL .        % (F E) is not in L_λ
```

However, the L_λ fragment can be used as a heuristic criterion to avoid using the general but costly unification procedure [7, 6].

One may wonder if the L_λ fragment is restricted enough for modifying the overall structure of the language. In fact, L_λ has exactly the same structure as λProlog. Unification in L_λ still does not allow us to analyse every data-structure. The restriction is such that even application $(T_1\ T_2)$ cannot be formed in L_λ. So, one still needs universal quantification, η-equivalence, and implication.

5.3 λProlog's Status

We finish with a few words on the applications and availability of λProlog.

Possible applications are chiefly those that motivated the λ-terms: manipulation of formulas, computation of denotation, etc (see section 2). Among actual applications, we find automatic theorem proving [4, 19], analysis of natural and formal languages [41, 14, 26], and the manipulation of functional programs [20]. Notice also that the structure of λProlog encompasses such constructions as modules [35] and abstract data types [33] without any extra-logical addition.

Two implementations of λProlog can be used: one, called eLP, is an interpreter written in Lisp [18], the other, called Prolog/Mali, is a compiler written in λProlog which generates C programs [6]. A third implementation is in progress [28].

Acknowledgements I wish to thank Catherine Belleannée and Pascal Brisset for the fruitful discussions we add on the matter presented in this paper.

References

[1] H. Andreka and I. Nemeti. *The Generalised Completeness of Horn Predicate-Logic as a Programming Language*. DAI Research Report 21, University of Edinburgh, 1976.

[2] H. Barendregt. Introduction to generalized type systems. *J. Functional Programming*, 1(2):125–154, 1991.

[3] H. Barendregt and K. Hemerik. Types in lambda calculi and programming languages. In N. Jones, editor, *European Symp. on Programming, LNCS 432*, pages 1–35, Springer-Verlag, 1990.

[4] C. Belleannée. *Vers un démonstrateur de théorèmes adaptatif.* Thèse, Université de Rennes I, 1991.

[5] C. Böhm and A. Berarducci. Automatic synthesis of typed λ-programs on term algebras. *Theoretical Computer Science*, 39:135–154, 1985.

[6] P. Brisset and O. Ridoux. The architecture of an implementation of λProlog: Prolog/Mali. In *Workshop on λProlog*, Philadelphia, PA, USA, 1992. ftp: //ftp.irisa.fr/local/lande.

[7] P. Brisset and O. Ridoux. *The Compilation of λProlog and its execution with MALI.* Technical Report 687, IRISA, 1992. ftp: //ftp.irisa.fr/local/lande.

[8] P. Brisset and O. Ridoux. Continuations in λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 27–43, MIT Press, 1993.

[9] P. Brisset and O. Ridoux. Naïve reverse can be linear. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 857–870, MIT Press, 1991.

[10] J. Cohen. Constraint logic programming languages. *CACM*, 33(7):52–68, 1990.

[11] A. Colmerauer. An introduction to Prolog III. *CACM*, 33(7), 1990.

[12] S. Coupet-Grimal. Représentation sémantique dans le traitement des langues naturelles en Prolog. In *Journées Francophones sur la Programmation en Logique*, pages 69–91, Teknea, Nîmes, France, 1993.

[13] H.B. Curry and R. Feys. *Combinatory Logic, Volume I.* North-Holland, Amsterdam, 1968.

[14] M. Dalrymple, S.M. Shieber, and F.C.N. Pereira. Ellipsis and higher-order unification. In *Linguistics and Philosophy*, pages 399–452, 1991.

[15] N.G. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the Church-Rosser theorem. *Indagationes Mathematicae*, 34:381–392, 1972.

[16] C.M. Elliott. *Extensions and Applications of Higher-Order Unification.* Research Report CMU-CS-90-134, School of Computer Science, Carnegie Mellon University, 1990.

[17] C.M. Elliott. Higher-order unification with dependent function types. In N. Derschowitz, editor, *3rd Int. Conf. Rewriting Techniques and Applications, LNCS 355*, pages 121–136, Springer-Verlag, 1989.

[18] C.M. Elliott and F. Pfenning. A semi-functional implementation of a higher-order logic programming language. In P. Lee, editor, *Topics in Advanced Language Implementation*, pages 289–325, MIT Press, 1991.

[19] A. Felty. Implementing tactics and tacticals in a higher-order logic programming language. *J. Automated Reasoning*, 11(1):43–81, 1993.

[20] J. Hannan and D. Miller. From operational semantics to abstract machines. *Mathematical Structures in Computer Science*, 4(2):415–459, 1992.

[21] M. Hanus. Horn clause programs with polymorphic types: semantics and resolution. *Theoretical Computer Science*, 89:63–106, 1991.

[22] P.M. Hill and J.W. LLoyd. *The Gödel Report.* Technical Report TR–91–02, University of Bristol, 1991.

[23] W.A. Howard. The formulae-as-types notion of construction. In J.P. Seldin and J.R. Hindley, editors, *To H.B. Curry: Essays on Combinatory Logic, Lambda Calculus and Formalism*, pages 479–490, Academic Press, London, 1980.

[24] G. Huet. Introduction au λ-calcul typé. In B. Courcelle, editor, *Logique et informatique : une introduction*, pages 137–162, INRIA, 1991.

[25] G. Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[26] S. Le Huitouze, P. Louvet, and O. Ridoux. Logic grammars and λProlog. In D.S. Warren, editor, *10th Int. Conf. Logic Programming*, pages 64–79, MIT Press, 1993.

[27] Joxan Jaffar and Jean-Louis Lassez. Constraint logic programming. In *14th ACM Symp. Principles of Programming Languages*, pages 111–119, ACM, Munich, Germany, 1987.

[28] B. Jayaraman and G. Nadathur. Implementation techniques for scoping constructs in logic programming. In K. Furukawa, editor, *8th Int. Conf. Logic Programming*, pages 871–886, MIT Press, 1991.

[29] T.K. Lakshman and U.S. Reddy. Typed Prolog: a semantic reconstruction of the Mycroft-O'Keefe type system. In *Int. Logic Programming Symp.*, pages 202–217, 1991.

[30] J.W. Lloyd. *Foundations of Logic Programming. Symbolic computation — Artificial Intelligence*, Springer-Verlag, Berlin, FRG, 1987.

[31] S. Michaylov and F. Pfenning. An empirical study of the runtime behavior of higher-order logic programs. In *Workshop on λProlog*, 1992. Preliminary version.

[32] D.A. Miller. Abstract syntax and logic programming. In A. Voronkov, editor, *2nd Russian Conf. Logic Programming, LNCS 592*, Springer-Verlag, 1991.

[33] D.A. Miller. Lexical scoping as universal quantification. In G. Levi and M. Martelli, editors, *6th Int. Conf. Logic Programming*, pages 268–283, MIT Press, 1989.

[34] D.A. Miller. A logic programming language with lambda-abstraction, function variables, and simple unification. *J. Logic and Computation*, 1(4):497–536, 1991.

[35] D.A. Miller. A proposal for modules in λProlog. In R. Dyckhoff, editor, *Int. Workshop Extensions of Logic Programming, LNAI 798*, pages 206–221, Springer-Verlag, 1993.

[36] D.A. Miller and G. Nadathur. Higher-order logic programming. In E. Shapiro, editor, *3rd Int. Conf. Logic Programming, LNCS 225*, pages 448–462, Springer-Verlag, 1986.

[37] D.A. Miller, G. Nadathur, F. Pfenning, and A. Scedrov. Uniform proofs as a foundation for logic programming. *Annals of Pure and Applied Logic*, 51:125–157, 1991.

[38] D.A. Miller, G. Nadathur, and A. Scedrov. Hereditary Harrop formulas and uniform proof systems. In D. Gries, editor, *2nd Symp. Logic in Computer Science*, pages 98–105, Ithaca, New York, USA, 1987.

[39] R. Milner. A theory of type polymorphism in programming. *J. Computer and System Sciences*, 17:348–375, 1978.

[40] A. Mycroft and R.A. O'Keefe. A polymorphic type system for Prolog. *Artificial Intelligence*, 23:295–307, 1984.

[41] R. Pareschi and D.A. Miller. Extending definite clause grammars with scoping constructs. In D.H.D. Warren and P. Szeredi, editors, *7th Int. Conf. Logic Programming*, pages 373–389, MIT Press, 1990.

[42] F. Pfenning. Dependent types in logic programming. In F. Pfenning, editor, *Types in Logic Programming*, pages 285–311, MIT Press, 1992.

[43] F. Pfenning. Logic programming in the LF logical framework. In G. Huet and G. Plotkin, editors, *Logical Frameworks*, pages 149–181, Cambridge University Press, 1991.

[44] F. Pfenning. Partial polymorphic type inference and higher-order unification. In *ACM Conf. LISP and Functional Programming*, pages 153–163, ACM Press, 1988.

[45] F. Pfenning. Unification and anti-unification in the calculus of constructions. In *Symp. Logic in Computer Science*, pages 74–85, 1991.

[46] B. Pierce, S. Dietzen, and S. Michaylov. *Programming in Higher-Order Typed Lambda-Calculi.* Research Report CMU-CS-89-111, School of Computer Science, Carnegie Mellon University, 1989.

[47] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 1986.

[48] S-Å. Tärnlund. Horn clause computability. *BIT*, 17:215–226, 1977.

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

Unité de recherche INRIA Lorraine, Technopôle de Nancy-Brabois, Campus scientifique,
615 rue du Jardin Botanique, BP 101, 54600 VILLERS LÈS NANCY
Unité de recherche INRIA Rennes, Irisa, Campus universitaire de Beaulieu, 35042 RENNES Cedex
Unité de recherche INRIA Rhône-Alpes, 46 avenue Félix Viallet, 38031 GRENOBLE Cedex 1
Unité de recherche INRIA Rocquencourt, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex
Unité de recherche INRIA Sophia-Antipolis, 2004 route des Lucioles, BP 93, 06902 SOPHIA-ANTIPOLIS Cedex

Éditeur
INRIA, Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
ISSN 0249-6399