Let Functions in Logic

Massimo Marchiori
Department of Pure and Applied Mathematics
University of Padova
Via Belzoni 7, 35131 Padova, Italy
max@math.unipd.it

(Extended Abstract)

1 Introduction

Two of the major paradigms that have had a great impetus in these last years are logic programming and functional programming. Both of these paradigms have intrinsic features that make them extremely elegant and expressive. On the one hand, logic programming retains the power of relational programming, where concepts are described by relationship predicates. On the other hand, functional programming retains the power of direct mathematical function definition.

The success of these two paradigms has thus led to the natural expectation to try to combine both of these unique features in a more expressive paradigm

One success has been obtained with narrowing (cf. [3]), that can be seen as the extension of functional programming with unification, a feature typical of logic programming. However, narrowing, properly speaking, is not an extension of these two languages, but a more expressive paradigm.

The original problem, that is incorporating together the two languages, has been and is a matter of extensive research, know an the *amalgamation problem*. A great deal of work has been done on this topic, like usage of residuation principles together with sophisticated forms of incomplete equational unification (for example S-unification in [2], and P-unification in [5]). Practically, also a concrete language, GAPlog, has been implemented (cf. [4, 7]).

All of these sophisticated approaches study new languages/paradigms that extend logic programming with functional programming capabilities, founding however big difficulties in providing a satisfactory solution to the amalgamation problem. A recent study ([9]) has shown what can be the reason for this unsuccess: logic programming as a whole is not functional, in the sense that all logic programs cannot be properly expressed in the functional setting without losing power. In this paper we propose a solution to the amalgamation problem that stems from this recent result. Instead of considering the whole paradigm of logic programming, we take into account a subpart of it which is in some sense more regular: roughly speaking, in place of untyped logic programming we consider typed logic programming. Then, we tackle the amalgamation problem considering the two paradigms of functional programming and of typed logic programming. The solution we found is extremely elegant and presents several advantages over all the previous approaches.

First, we do not define a 'new' operational semantics, like all the previous work do. The idea is that logic programs can just be seen as functional programs, via a natural correspondence. So, we don't have to define every time a new language, with the consequent theoretical problems (semantics, completeness, etc.).

Second, another important problem is the *feasibility* of the solution. All the other approaches define new paradigms, but then the big problem to be able to *really* benefit of the amalgamation is to write a correct (and possibly fast) specific implementation for these languages. With our approach this problem is nicely coped with, since via the transformation from logic to functional programming we can rely on all the existing implementations of functional languages.

Third, once this way is followed we see immediate benefits on the power of the amalgamation: being logic programs just sugaring for functional programs, we can completely naturally interleave logic and functional parts, using interchangeably a relational or functional view.

The nice thing is that, although in principle this approach cannot be strictly more powerful than the other works, since it does not deal with the whole paradigm of logic programming, *in practice* it can cope with all the motivating examples that led to the formulation of the amalgamation problem.

Due to lack of space we will be rather sketchy, re-

ferring to the full version of this paper [10] for all the technical details.

2 Preliminaries

We assume basic knowledge of logic programming (see e.g. [6]) and functional programming (cf. [12]). Logic programs will be considered as executed with leftmost selection rule and depth-first search rule, that is the standard way in which logic programming is implemented (for example, in Prolog). Also, we will consider in full generality conditions that can constrain both the logic program and the goal: so, for notational convenience we will talk by abuse of a *class of logic programs* meaning a collection both of logic programs and of goals.

Sequences of terms will be written in vectorial notation (e.g. $\bar{t}$). Sequences in formulae should be seen just as abbreviations: for instance, $[\bar{t}]$, with $\bar{t} = t_1, \ldots, t_m$, denotes the string $[t_1, \ldots, t_m]$. Accordingly, given two sequences $\bar{s} = s_1, \ldots, s_n$ and $\bar{t} = t_1, \ldots, t_m$, $\bar{s}, \bar{t}$ stands for the sequence $s_1, \ldots, s_n, t_1, \ldots, t_m$.

Given a family S of objects (terms, atoms, etc.), $\mathrm{Var}(S)$ is the set of all the variables contained in it; moreover, S is said to be *linear* if no variable occurs more than once in it.

We deal with strict functional languages, and employ a generic functional notation, without sticking to a particular language syntax. A difference with some widespread functional language is that we represent lists using the notation [X|Y] to denote a list with head X and tail Y, and use [] as the nil constructor: this is the notation used for instance in Concurrent CLEAN, but it differs with e.g. SML or Haskell (where | is used for list comprehension). We chose this notation to make easier the natural correspondence between the logic and functional paradigms, since it is the standard list notation employed in all logic programming languages.

Also we use some other notational convention: we write function definitions with non-linear patterns (in case the language does not support it, one can use the standard modifications, see e.g. [12]), and we write t for the 1-tuple (t) when it enhances readability.

3 The Basic Class

The class of logic programs we will consider makes use of so-called *mode-typings* (also said *directional types*). Roughly speaking, a 'direction' (being an input or an output parameter) and a type are assigned to every argument of a predicate p. For example, in the case of the add predicate to compute the sum of two integers, one could mode/type it as add(*in*:$\mathbb{N}$,*in*:$\mathbb{N}$,*out*:$\mathbb{N}$), indicating that it takes two naturals as input (first two arguments) and returns as output a natural number (third argument).

For notational convenience, we will sometime use a semicolon to separate the input arguments from the output ones, with the convention the inputs are to the left of the semicolon (e.g. in the previous case we would have add(t_1,t_2;t_3)).

As far as the type system is concerned, we will be extremely general, allowing every type system that is closed under substitutions. Possible types are *Any* (all the terms), $\mathbb{N}$ (the natural numbers), *Ground* (all the ground terms), *List* (all the lists) and so on. In the following examples we will assume these basic types are in the type system. Also, we say a type is ground if it is contained in *Ground*.

A term t of type T will be indicated with $t : T$. If $\bar{t} = t_1, \ldots, t_n$ and $\bar{T} = T_1, \ldots, T_n$ are respectively a sequence of terms and types, $\bar{t} : \bar{T}$ is a shorthand for $t_1 : T_1, \ldots, t_n : T_n$.

To reason about types, we employ the standard concept of *type checking*: an expression of the form $\bar{s} : \bar{S} \models \bar{t} : \bar{T}$ indicates that from the fact that $\bar{s}$ has type $\bar{S}$ we can infer that $\bar{t}$ has type $\bar{T}$. More formally, $\bar{s} : \bar{S} \models \bar{t} : \bar{T}$ if for every substitution θ, $\bar{s}\theta \in \bar{S}$ implies $\bar{t}\theta \in \bar{T}$. For instance, $X : Any, Y : List \models [X|Y] : List$.

Another concept we need is that of generic expression A term t is a generic expression of the type T if every $s \in T$ having no common variables with t and unifying with it is an instance of t (i.e. $\exists\theta.\, t\theta = s$). For instance, variables are generic expressions of *Any*, every term is a generic expression of *Ground*, $[], [X], [X|X], [X|Y], [X, Y|Z]$ etc. are generic expressions of *List*.

The idea is to combine usage of types and generic expressions in such a way that during a program execution unification behaves in a more regular way, that is to say it can be performed using repeated applications of pattern matching (see [1, 8]). So, we now introduce the main class studied in the paper:

Definition 3.1 A program is said to be *Regularly Typed* (RT) if for each of its clauses $p(\bar{t}_0 : \bar{T}_0; \bar{s}_{n+1} : \bar{S}_{n+1}) \leftarrow p_1(\bar{s}_1 : \bar{S}_1; \bar{t}_1 : \bar{T}_1), \ldots, p_n(\bar{s}_n : \bar{S}_n; \bar{t}_n : \bar{T}_n)$ we have:

- $\bar{t}_0 : \bar{T}_0, \ldots, \bar{t}_{j-1} : \bar{T}_{j-1} \models \bar{s}_j : \bar{S}_j \quad (j \in [1, n+1])$
- each term in $\bar{t}_i$ is filled in with a generic expression for its corresponding type in $\bar{T}_i$
- if a variable X occurs twice in $\bar{t}_0$, then every term $r \in \bar{t}_0$ has a corresponding ground type.
- $\bar{t}_1, \ldots, \bar{t}_n$ is a linear sequence of variables and $\forall i \in [1, n]$. $\mathrm{Var}(\bar{t}_i) \cap \bigcup_{j=0}^{i} \mathrm{Var}(\bar{s}_j) = \emptyset$. □

For example, the standard program to add two numbers in unary representation

$$add(0, X, X) \leftarrow \qquad add(s(X), Y, s(Z)) \leftarrow add(X, Y, Z)$$

with moding/typing $add(in : Ground, in : Any, out : Any)$ is regularly typed.

The scope of the class RT is quite large: besides the fact that all the standard basic programs *append*, *reverse*, *quicksort*, *member* etc. are all in RT, in general *many* parts of logic programming codes are written, more or less consciously, in the form given by the RT class.

Indeed, this class properly contains the class of simply moded and well typed (SWT) programs introduced in [1], and that class has already been shown to be quite expressive (see for instance the list of programs presented in [1]).

4 Logic as Functional

It has come the moment to briefly describe in what sense regularly typed programs can be naturally seen as functional programs.

Take a clause

$$C = p(\bar{t}_0, \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1, \bar{t}_1), \ldots, p_n(\bar{s}_n, \bar{t}_n)$$

Then define its translation $\mathcal{T}(C)$ as

$$p(\bar{t}_0) = \texttt{let } (\bar{t}_1) = p_1(\bar{s}_1) \texttt{ in}$$
$$\texttt{let } (\bar{t}_2) = p_1(\bar{s}_2) \texttt{ in}$$
$$\ddots$$
$$\texttt{let } (\bar{t}_n) = p_1(\bar{s}_n) \texttt{ in } (\bar{s}_{n+1})$$

Example 4.1
As a simple example, consider the append program with moding/typing append(*in*:*List*,*in*:*List*,*out*:*List*):

append([X | Xs], Ys, [X | Zs]) $\leftarrow$ append(Xs, Ys, Zs)
append([], Ys, Ys) $\leftarrow$

Its translation is

append([X | Xs], Ys) = **let** Zs= append(Xs,Ys) **in** [X | Zs]
append([],Ys) = Ys

□

We now prove that under some reasonable conditions, the above correspondence is a perfect one.

Definition 4.2 Two clauses $p(\bar{t}_0, \bar{s}_{n+1}) \leftarrow \ldots$ and $p(\bar{t}'_0, \bar{s}'_{n+1}) \leftarrow \ldots$ are said to be *input-alternative* if $(\bar{t}_0)$ and $(\bar{t}'_0)$ are not unifiable. A program is said to be input-deterministic if every couple of clauses defining the same predicate is input-alternative. □

Input-determinism corresponds to the very reasonable assumption that predicates have been 'well defined' in the sense that there are no ambiguities in their definition.

Theorem 4.3 *Let $G = \leftarrow p(\bar{s}; \bar{t})$ a goal and P a logic program that are regularly typed and input-deterministic. Then:*

- *$P \cup \{G\}$ gives a computed answer substitution α if and only if $p(\bar{s})$ reduces via $\mathcal{T}(P)$ to $(\bar{t}\alpha)$.*
- *$P \cup \{G\}$ fails if and only if $p(\bar{s})$ raises an unbounding exception when reduced via $\mathcal{T}(P)$.*
- *$P \cup \{G\}$ does not terminate if and only if $p(\bar{s})$ when reduced via $\mathcal{T}(P)$ does not terminate.*

Example 4.4 Reconsider the program seen in the previous Example 4.1: it is trivial to check that it is regularly typed and input-deterministic. So by the above Theorem we are sure its translation is a safe implementation of the append of two lists. □

5 More Functions

Now that we have seen that regularly typed logic programs can be seen as functional programs, it is natural to treat them as functional objects, thus allowing their combination with other functions defined in the functional program to amalgamate.

The only restriction regarding usage of function symbols in a functional program is that no defined symbols can appear inside a definitional pattern. This, using the correspondence seen above, equals to require that given a clause $p(\bar{t}_0; \bar{s}_{n+1}) \leftarrow \ldots$, we cannot use in $\bar{t}_0$ function symbols defined in the functional program to amalgamate (since the clause is seen as a function definition for p, i.e. $p(\bar{t}_0) = \ldots$, and so no other defined function can appear in the definitional pattern $p(\bar{t}_0)$).

This is the only condition one has to impose: function symbols can be freely used in the rest of a regularly typed clause.

Example 5.1 Consider the following GAPlog program to compute the factorial, using the function symbols 0, 1, $*$ and $-$:

factorial(0,1) $\leftarrow$
factorial(X,X*Y) $\leftarrow$ factorial(X-1,Y)

This program is regularly typed taking factorial(*in*:$\mathbb{N}$,*out*:$\mathbb{N}$), and satisfies the above condition. Its functional translation is

factorial(0)=1
factorial(X)= **let** Y=factorial(X-1) **in** X*Y □

As said, the nice thing is that, like the above example, also all the other basic examples motivating the amalgamation problem (cf. [2, 5, 4, 7]) can be dealt with by this approach.

The integration here presented is even deeper, since while all the other amalgamations described in the literature (cf. the introduction) are 'one-way', in the sense that logic programming is extended via external functions but not vice versa, this approach is completely symmetrical: one can use in the functional program functions defined relationally via predicates

in a (possibly extended with functions) logic program, since every atom can be seen via the functional translation as sugaring for a function call.

6 More Power

There are several extremely powerful extensions to the approach we have presented. The first is management of built-in's. To be concretely usable, every logic programming implementation (like Prolog) also provides arithmetical built-in's for natural numbers (e.g. $+$, $*$, $<$, $>$, the "is", etc.). Arithmetical built-in's can be directly translated into the corresponding constructs of the functional program: they can be considered as *tests* (for instance, a clause of the form $p(X,Y) \leftarrow X < 0, \ldots$ can be translated as $p(X,Y) =$ `if` $X < 0$ `then` ... `else` process the next clause). The formal translation is given in [10].

Moreover, we are not limited to just using Prolog built-in's, but we can use all the built-in's present in the functional program, for instance those for the real numbers. Thus we have automatically an extension of logic programming with real arithmetics.

Another extension regards the limitation to input-deterministic programs. In a number of programs this limitation is too strong, since there may be some apparent ambiguities in the definition that are solved in the body of the clause. In [10] it is shown how the translation can be easily refined to deal with such programs, relaxing input-determinism to 'weakly input-determinism'. Roughly speaking, a careful analysis of the body of the clauses is performed in order to determinate when the ambiguity is only apparent. Then, these 'solving relations' hidden inside the body are treated as *guards*, thus enabling to solve ambiguity conflicts right away during the application of the corresponding function definition.

Moreover, the same class of regularly typed logic programs can be extended, using the *conormal* transformation introduced in [11]: this transformation acts on the extremely big class of *safely typed* logic program, simplifying their structure. This simplification is safe for a parallel implementation of logic programming, but may be lossy for a sequential implementation in the sense that termination may be lost. We have developed an effective abstract interpretation technique in order to check whether the transformation is safe, thus enabling to cope with a much bigger class of programs.

Finally, we mention the fact that we can employ all the techniques proper of functional languages in order to get more efficient and comprehensible code. For example, applying the well-known simplifications given by unfolding the `let` constructs we can get much better translations. For example, unfolding the `let` construct in the translation of the GAPlog factorial program seen in Example 4.1, we get

factorial(0)=1
factorial(X)= X*factorial(X-1)

which is exactly the definition of factorial used in functional programming. The same occurs, for instance, with the logic program seen in Example 4.1.

References

[1] K.R. Apt and S. Etalle. On the unification free Prolog programs. *Proc. Int. Conf. on Mathematical Foundations of Computer Science*, vol. 711 of *LNCS*, pp. 1–19. Springer–Verlag, 1993.

[2] S. Bonnier and J. Małuszyński. Towards a clean amalgamation of logic programs with external procedures. *Int. Conf. on Logic Programming*, pp. 311–326. MIT Press, 1988.

[3] M. Hanus. The integration of functions into logic programming: From theory to practice. *Journal of Logic Programming*, 19&20:583–628, 1994.

[4] A. Kågedal and F. Kluźniak. Enriching Prolog with S-unification. *Workshop on Declarative Programming*, Workshops in Computing, pp. 51–65. Springer, 1991.

[5] G. Lindstrom, J. Małuszyński, and T. Ogi. Our LIPS are sealed: Interfacing functional and logic programming systems. *Proceedings Int. Symp. on Programming Language Implementation and Logic Programming*, vol. 631 of *LNCS*, pp. 428–442. Springer, 1992.

[6] J.W. Lloyd. *Foundations of Logic Programming.* Springer–Verlag, second edition, 1987.

[7] J. Małuszyński, S. Bonnier, J. Boye, F. Kluźniak, A. Kågedal, and U. Nilsson. Logic programs with external procedures. *Logic Programming Languages: Constraints, Functions, and Objects*, pp. 21–48. MIT Press, 1993.

[8] M. Marchiori. Localizations of unification freedom through matching directions. *Proc. Int. Symp. on Logic Programming*, pp. 392–406. MIT Press, 1994.

[9] M. Marchiori. The functional side of logic programming. *Proc. ACM Int. Conf. on Functional Programming Languages and Computer Architecture*, pp. 55–65. ACM Press, 1995.

[10] M. Marchiori. On the amalgamation of logic and functions. Technical Report 23, Dept. of Pure and Applied Mathematics, University of Padova, 1996.

[11] M. Marchiori. Proving existential termination of normal logic programs. *Proc. Int. Conf. on Algebraic Methodology and Software Technology*, vol. 1101 of *LNCS*, pp. 375–390. Springer–Verlag, 1996.

[12] S.L. Peyton Jones. *The Implementation of Functional Programming Languages.* International Series in Computer Science. Prentice-Hall, 1987.