The Functional Side of Logic Programming

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

Abstract

In this paper we study the relationships between logic programming and functional programming, trying to answer the following basic question: to what extent is logic programming just functional programming in disguise? We develop a theory to formally express this correspondence, and exhibit a class that can by right be considered as the functional core of logic programming. Moreover, since the functional meaning of each program in this class is provided constructively, via a transformation from logic to functional programs, we show how the obtained theoretical results are useful also in the study of languages integration, termination issues, and practical implementations.

1 Introduction

During the recent years, a huge amount of effort was devoted to integrating the two paradigms of functional programming and logic programming, and to develop languages combining them (see for instance [12]). However, much less attention was devoted to the basic problem of studying what are the intimate relationships between logic programming and functional programming. So far, the research only scratched the surface finding out the *differences* (e.g. usage of unification versus matching, resolution vs. reduction, nondeterminism vs. determinism etc., see for example [7, 12]), but not the *connections*, that is how much they do have in common. To the best of our knowledge, only Reddy (e.g. in [24, 25]) tried to shed some light on this fundamental topic.

In this paper we aim to answer the following basic question: to what extent is logic programming just functional programming in disguise? That is to say, what class of logic programs can be considered the functional core of logic programming?

A first requirement we impose on such a class (if it exists) is to be a neat sub-language of logic programming: one should be able to say of every clause of a logic program whether it is 'functional' or not, the functional flavor should a fundamental property, visible at clause level. This will be referred to as *locality*: every class we consider henceforth will be understood to be local.

Reddy in his work [24] started studying the relationships between functional and logic programming: its outcome was a transformation from a (local) class of logic programs to functional programs that is 'sound', that is informally everything calculated by the obtained functional program is calculated by the original logic program too.

However, Reddy's transformation fails to identify a 'functional' class of logic programming for various reasons.

First, for the transformation to be sound the target functional language must employ an *ad hoc* fair-parallel reduction strategy. This means that the translation of a logic program (its functional meaning) is not a pure functional program, which is quite unsatisfactory. We also remark that, from a practical point of view, not only the obtained code cannot in general run on any implementation not supporting fair parallelism, but even using a parallel functional language an effective use is unfeasible: indeed, the degree of parallelism required is far beyond the so called or-parallelism[1] of logic programming (which is already considered impractical), since even shallow backtracking[2] can instigate an arbitrary number of infinite computations (see also below).

Second, the transformation is *not complete*, that is the obtained functional program may *not* terminate despite the original logic program terminates. Of course, this is unacceptable for a 'functional' class of logic programs: a minimal requirement should be that its transformed functional program behaves *just in the same way*, not losing a fundamental property like termination (in fact, we guess Reddy was aware of this problem since, albeit not mentioning the lack of completeness, he only refers to his transformation as being *sound*).

Thus, Reddy's transformation only started shaping the borders of the problem. In fact, the first thing that should be clarified is what means for a logic program to be 'functional' in nature. As hinted before, this should mean it is equivalent to a functional program behaving in the same way, hence the need for a transformation from logic to functional programs to find out the 'functional meaning'. However, the problem lies in what 'behaving in the same way' should signify. A natural response is requiring the transformation to functional programs to be *complete*, that is to preserve termination and computed answers: this way an external observer should not see any difference on *what* the program computes. Completeness is a minimal requirement but, still, is not the full answer since it does not rule out 'Turing-carpet' low-level transformations: such a transformation would be complete for the whole class of logic programs, hence identifying logic programming with functional

[1] Backtracking due to nondeterminism is implemented via parallel processes.

[2] Failure of a goal to resolve with a clause.

programming, a not very interesting result. This is nothing more than a restatement of the Church-Turing thesis: every other Turing-complete paradigm for calculation would be equivalent having the same computational power.

The answer we found is to slightly stress the notion of equivalence: we said an external observer should not see any difference on *what* the program computes. We also require he should not see any difference on *how* the program computes: it should not increase in time complexity. More formally, logic programs belonging to a certain complexity class should be transformed into functional programs of the same (or lower) complexity class. This turns out to be a good answer: with this requirement, a class can be found that is *maximal* among those complete. A great surprise is that this result holds all the same even if the requirement is considerably weakened: we can allow the transformation to arbitrarily worsen the time complexity of a program, provided only this 'worsening' is computable.

These results will be proven first on definite logic programming, and then also extended to normal logic programming where use of *negation* is allowed.

Also, as a by-product of these correspondence results we will show how such transformational methodology can provide new contributions to a clean integration of logic and functional programming, and to the study of termination properties for logic programs. Finally, we will test this approach from a practical point of view, benchmarking the code produced by the transformation.

2 Preliminaries

2.1 Notation

Sequences of terms will be written in vectorial notation (e.g. $\bar{t}$), and $|\bar{t}|$ denotes their length (viz. if $\bar{t} = t_1, \ldots, t_m$, $m \geq 0$, then $|\bar{t}| = m$). Sequences in formulas 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. To make formulas more readable, we will sometimes omit brackets from the argument of unary functions (e.g. $f(g(X))$ may be written fgX),

For standard logic programming terminology, we will mainly follow [16]. 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). With $mgu(s,t)$ we indicate a most general unifier of two terms s and t.

Finally, $\widetilde{\mathbb{N}}$ stands for the set $\mathbb{N} \cup \{\omega\}$ of natural numbers extended with the ordinal ω.

2.2 Functional Programs

To give relevance to a characterization of a 'functional core' of logic programs, the considered functional language should not employ features particular to a given language, but instead be somehow the common kernel of the various ML, Haskell, Miranda[3] and so on. The functional language considered here is to a certain extent the most 'basic' possible: it just has function definition with patterns.

[3] Miranda is a trademark of Research Software Ltd.

We will consider functional programs of the following kind: a (finite) sequence $\left\{ \begin{array}{rcl} op_1 pat_1 & = & e_1^P \\ op_2 pat_2 & = & e_2^P \\ & \vdots & \\ op_n pat_n & = & e_n^P \end{array} \right\}$ of function definitions, where the op_i are *operators* (not necessarily distinct), the pat_i are *patterns* and the e_i are *expressions*. As usual, in this basic functional language operators are function symbols that cannot occur in patterns, patterns are linear terms and expressions are terms. We remark that patterns in a program can also be overlapping (that is, two or more pat_i's in the definitions of some operator could be unifiable), in which case the different alternatives are tried out one at a time, from top to bottom.

As far as the *reduction strategy* is concerned, in accordance to what said above on the generality of the language, we leave it *unspecified*: that is to say, all the results we are going to prove should hold *independently* on the particular reduction strategy, be it strict or lazy (when this is not the case, we will say it explicitly).

Given a functional program F and an expression e, we will indicate with $(F)(e)$ the result of reducing e w.r.t. F.

2.3 Locality

We study classes of logic programs under a *local* point of view, that is requiring their description can be given *clause-by-clause*:

Definition 2.1 A class of logic programs is said to be *local* when every program belongs to it iff each of its clauses does. □

Such local classes are very useful because, as mentioned in the Introduction, they allow to neatly define a *sub-language* of logic programming simply restricting the syntax of each program clause to those characterizing the class.

Non local classes, instead, are in general not very useful for defining a sub-language: not only do they require the programmer to take into account many program lines all at once, but in case of *program development* two fundamental operations like deleting a clause or adding a clause of the class to the program can be dangerous (indeed, they are both safe operations if and only if the class is local).

Thus, from now on we assume that *every class of logic programs we talk about is understood to be local.*

Also, we will consider in full generality classes that can constrain both the logic program and the goal: for notational convenience we will talk by abuse of a *class* Λ *of logic programs* meaning a collection both of logic programs and of goals.

3 Functionally Moded Programs

The starting idea is quite natural: since logic programming deals with *relations* whereas functional programming deals with *functions*, we could try to give some *directionality* to the predicates, turning them into functions. Indeed, Reddy in [24, 25] followed this path, employing the concept of *moding*:

Definition 3.1 A *mode* for a n-ary predicate p is a map from $\{1, \ldots, n\}$ to $\{\mathrm{in}, \mathrm{out}\}$. A *moding* is a map associating to every predicate p a mode for it. A *moded program* is a program endowed with a moding. An argument position of a moded predicate is called input (resp. output) if it is mapped by the mode into in (resp. out). □

If m is a moding, we sometimes write $p(m(1), \ldots, m(k))$ (or simply $(m(1), \ldots, m(k))$) to show the mode of a k-ary predicate p. Also, $p(\bar{s}; \bar{t})$ denotes a moded atom p having its input positions filled in by the sequence of terms $\bar{s}$, and its output positions filled in by $\bar{t}$.

A moded predicate can be roughly seen as a function from its input arguments to its output ones. For instance, a predicate p with moding $(\mathrm{in}, \mathrm{in}, \mathrm{out})$ should be viewed as a function having two inputs (the first two arguments) and one output (the third one).

Some conditions are needed to ensure this informal interpretation of moded programs is assured.

The idea is to identify the concept of 'being an input argument' with a datum that does not need to be processed further (being an input its value should be fixed and well determined): a safe condition is to require the datum to be *ground*, so that its value cannot be modified by some calculation (if some variable is present, it could be instantiated to some term, hence modifying the datum). This is formalized via the following definition:

Definition 3.2 A derivation is *data driven* if in it every time an atom is selected for unification in a goal, the input arguments of the atom are ground. □

The syntactical criterion used by Reddy to impose directionality on logic programs is the Well Moding:

Definition 3.3 A goal $\leftarrow p_1(\bar{s}_1; \bar{t}_1), \ldots, p_n(\bar{s}_n; \bar{t}_n)$ is *Well Moded* if $\forall i \in [1, n]\ :\ \mathrm{Var}(\bar{s}_i) \subseteq \bigcup_{j=1}^{i-1} \mathrm{Var}(\bar{t}_j)$.
A program is said to be *Well Moded* (WM) if for each of its clauses $p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), \ldots, p_n(\bar{s}_n; \bar{t}_n)$ we have $\forall i \in [1, n+1]\ :\ \mathrm{Var}(\bar{s}_i) \subseteq \bigcup_{j=0}^{i-1} \mathrm{Var}(\bar{t}_j)$. □

This definition roughly says that every variable in an input argument $\bar{s}_i$ of the body (hence, according to our intended interpretation, required as input by some predicate) must appear earlier in the clause as an output argument $\bar{t}_j$ (so it must have been calculated).

Example 3.4 The usual program to add two numbers

```
add(0,X,X) ←
add(s(X),Y,s(Z)) ← add(X,Y,Z)
```

with moding $\mathtt{add}(\mathrm{in}, \mathrm{in}, \mathrm{out})$ is Well-Moded. □

Well Modedness is an appropriate criterion, since:

Theorem 3.5 ([9]) *If P and G are respectively a Well Moded program and goal, then every derivation of $P \cup \{G\}$ is data driven.*

Using Well Modedness, Reddy in [24] introduced a transformation of Well Moded logic programs into (parallel) functional programs. This transformation, as said in the Introduction, was there proved to be *sound*, that is informally everything calculated by the obtained functional program is calculated by the original logic program too.

The above result, however, gives only one side of the coin, since the transformation is not *complete*: the obtained functional programs might not terminate. As said in the Introduction, this is not satisfactory, since completeness is a minimal requirement that should be asked of a logic program which is inherently functional. The problem is that so far only input arguments were considered, imposing on them a 'direction' (groundness) via the Well Moding criterion, but output arguments were neglected.

So, some other condition should be imposed on the output arguments, but what? Intuitively, we will require the output arguments of a predicate to be *different variables*. Indeed, recall the idea that led to constraining the input arguments: analogously, viewing a predicate as a function from its input to its output arguments should imply we cannot predict in advance the result. But if some output argument is not a variable, this means we know in advance something about the output, and analogously if some variable occurs more than once in different arguments. The proper formalization of this idea is:

Definition 3.6 A goal $\leftarrow p_1(\bar{s}_1; \bar{t}_1), \ldots, p_n(\bar{s}_n; \bar{t}_n)$ is *Functionally Moded* if it is Well Moded and the sequence $\bar{t}_1, \ldots, \bar{t}_n$ is linear and composed only of variables.
A program is said to be *Functionally Moded (FM)* if its every clause $p_0(\bar{t}_0; \bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1; \bar{t}_1), \ldots, p_n(\bar{s}_n; \bar{t}_n)$ is Well Moded, the sequence $\bar{t}_0, \ldots, \bar{t}_n$ is linear and $\bar{t}_1, \ldots, \bar{t}_n$ is composed only of variables. □

For instance, it is immediate to check that the program seen in Example 3.4 is Functionally Moded.

It is interesting to notice that *many* parts of logic programming codes are written, more or less consciously, in the 'functional' form given by this class. Also, a great deal of basic logic programs (like e.g. *append*, *reverse*, *quicksort*, *member* etc.) belong to it.

We remark how the above definition concerns *definite* logic programs only (i.e. programs without negation). In Section 9 the full definition of the class for *normal* logic programs (i.e. with negation) will be given.

4 Semantics

Suppose a logic program P is run with goal G. Let us denote with $answer_{P,G}(1)$ the first obtained answer: it is equal to

1. ϑ if the computation terminates successfully giving ϑ as computed answer substitution.
2. Fail if the computation terminates with failure.
3. $\perp$ if the computation does not terminate.

In case 1, the user can activate backtracking to look for the second answer $answer_{P,G}(2)$, and so on till for some $k \geq 1$ $answer_{P,G}(k)$ returns Fail or $\perp$ (in case of infinite answers, we assume $k = \omega$ and $answer_{P,G}(\omega) = \perp$).

Now, the *answer semantics* $\boldsymbol{\alpha}_P(G)$ of a logic program P w.r.t. a goal G is defined as the (possibly infinite) sequence

$$\boldsymbol{\alpha}_P(G) = \ answer_{P,G}(1), \ldots, answer_{P,G}(k)$$

We can now provide a formal definition of termination:

Definition 4.1 A program P is said to *terminate* w.r.t. a goal G if $\boldsymbol{\alpha}_P(G) = \phi_1, \ldots, \phi_k$ with $\phi_k \neq \perp$. □

This kind of termination is usually referred to as *universal* (see e.g. [8]), to distinguish it from *existential* termination (viz. $\phi_1 \neq \perp$): we will come back on other kinds of termination later in Section 8 and Subsection 11.2.

So far, $\perp$ was considered a special symbol to indicate nontermination; however, in the following it may also indicate a nonterminating expression: we will write $e = \perp$ (for some expression e) meaning that e is a nonterminating expression in the given functional program. Also, for notational convenience we will allow usage of every answer ϕ as substitution: in the special cases $\phi = \mathsf{Fail}$ (resp. $\phi = \perp$) we assume $e\phi = \mathsf{Fail}$ (resp. $e\phi = \perp$) for every expression e.

$$\begin{array}{ll}
\textbf{D1} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; [\bar x, y_1, \dots, y_n] \;=\; t \\[6pt]
\textbf{D2} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; \mathsf{Fail} \;=\; \mathsf{Fail} \\[6pt]
\textbf{D3} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; [\bar x|\mathsf{Fail}] \;=\; \mathsf{Fail} \\[6pt]
\textbf{D4} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J([\bar x, y_1, \dots, y_n], v) \;=\; J(t, \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; v) \\[6pt]
\textbf{D5} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J(\mathsf{Fail}, v) \;=\; \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; v \\[6pt]
\textbf{D6} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J([\bar x|\mathsf{Fail}], v) \;=\; \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; v \\[6pt]
\textbf{D7} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J([\bar x|J(u,v)], w) \;=\; \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J([\bar x|u], J([\bar x|v], w)) \\[6pt]
\textbf{D8} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J(J(u,v), w) \;=\; \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J(u, J(v, w)) \\[6pt]
\textbf{D9} & \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; [\bar x|J(u,v)] \;=\; \underset{[\bar x, y_1,\dots,y_n]=t}{\mathcal{D}^{(n)}} \; J([\bar x|u], [\bar x|v])
\end{array}$$

Table 1: The Data-handler equations schema.

5 Completeness

Definition 5.1 A transformation γ for a class Λ of logic programs is a computable map sending goals of Λ into expressions and logic programs of Λ into functional programs. $\Box$

Definition 5.2 A transformation γ is said to be *complete* for a class Λ of logic programs if there is a computable function $\mathbb{A}$ such that for every program $P \in \Lambda$ and goal $G \in \Lambda$, P terminates w.r.t. G implies

$$\mathbb{A}\big(\mathrm{Var}(G), \big(\gamma(P)\big)\big(\gamma(G)\big)\big) \;=\; \boldsymbol{\alpha}_P(G) \qquad \Box$$

The presence of $\mathrm{Var}(G)$ in the above formula is to make explicit that to get a computed answer substitution we have to remember the names of the variables present in G: we could however shorten the definition omitting $\mathrm{Var}(G)$, since the variable names could be encoded directly in the transformation $\gamma(G)$ of the goal.

Observe that for a complete γ if P terminates w.r.t. G then also $\gamma(P)$ terminates w.r.t. $\gamma(G)$ (that is, γ is termination preserving): indeed, if it is not the case then $(\gamma(P))(\gamma(G)) = \bot$, and so also $\mathbb{A}(\mathrm{Var}(G), (\gamma(P))(\gamma(G))) = \bot$, a contradiction since $\boldsymbol{\alpha}_P(G)$ does not contain $\bot$ by assumption.

6 The Transformation

The transformation τ we will now introduce is *compositional*, that is for every two programs P and Q, $\tau(P \cdot Q) = \tau(P) \odot \tau(Q)$ for a suitable map $\odot$. Hence, we can describe it giving its definition only for a goal, a clause and the map $\odot$.

From now on we will assume that the goal is of the form $\leftarrow p(\bar s; \bar t)$: this is not restrictive since, e.g., a Functionally Moded goal of the form $\leftarrow p_1(\bar s_1; \bar t_1), \dots, p_n(\bar s_n; \bar t_n)$ can be split into a goal $\leftarrow p(\bar s_1; \bar t_1, \dots, \bar t_n)$ and a clause $p(\bar s_1; \bar t_1, \dots, \bar t_n) \leftarrow p_1(\bar s_1; \bar t_1), \dots, p_n(\bar s_n; \bar t_n)$ (where p is a new predicate) that are both Functionally Moded, giving an equivalent program. This assumption is not necessary, but it is adopted since it shortens some definitions and results.

Logic programming is intrinsically nondeterministic (multiple choices are processed via backtracking): to utilize this feature into functional programming we use a binary symbols J as 'pairing operator' to Join together the different possibilities arising from the nondeterminism. For instance, if in the logic program a predicate p is nondeterministically defined via two clauses $p \leftarrow q$ and $p \leftarrow r$, in the corresponding functional program we will join these two different choices via the single rule $p \rightarrow J(q, r)$.

A constant Fail will be used to denote failure of a logic program derivation.

We will also employ so-called *Data-handler* operators: A Data-handler of the form $\underset{t_1 = t_2}{\mathcal{D}^{(n)}}$ can be roughly seen as the function $\lambda t_1.t_2$ (i.e. it expects a datum of the form t_1 and gives as output t_2); this is formalized by the rule D1 of Table 1. The difference is that it also has to cope with different structures than t_1, namely when the J and the Fail operators crop up (the superscript n helps to determine what structures are in principle possible). This is illustrated by the remaining rules D2 – D9 of Table 1. Essentially, it produces a failure whenever a Fail result is found (rules D2, D3), otherwise it processes the first result among the ones joined with J: if it is t_1 (resp. a failure) then via rule D4 (resp. D5, D6) the Data-handler $\underset{t_1 = t_2}{\mathcal{D}^{(n)}}$ transforms it into t_2 (resp. skips it), and goes on processing the other results.

Definition 6.1 If $G = \ \leftarrow p(\bar s; \bar t)$ is Functionally Moded, then $\tau(G)$ is $\underset{[X_1,\dots,X_{|\bar t|}] = [X_1,\dots,X_{|\bar t|}]}{\mathcal{D}^{(|\bar t|)}} \; p[\bar s]$.
If $P = p_0(\bar t_0; \bar s_{n+1}) \leftarrow p_1(\bar s_1; \bar t_1), \dots, p_n(\bar s_n; \bar t_n)$ is Functionally Moded, then $\tau(P)$ is:

1. The equation

$$p_0[\bar t_0] \;=\; \underset{[NV(n), \bar t_n] = [\bar s_{n+1}]}{\mathcal{D}^{(|\bar t_n|)}} \; \underset{[NV(n-1), \bar t_{n-1}] = [NV(n)|p_n[\bar s_n]]}{\mathcal{D}^{(|\bar t_{n-1}|)}} \cdots$$

$$\cdots \underset{[NV(0), \bar t_0] = [NV(1)|p_1[\bar s_1]]}{\mathcal{D}^{(|\bar t_0|)}} \; [\bar t_0]$$

where $NV(k) = (\cup_{j=k+1}^{n+1} \mathrm{Var}(\bar{s}_j)) \setminus (\cup_{j=k}^{n} \mathrm{Var}(\bar{t}_j))$.
Incidentally, observe that by the definition of Functionally Moded program it follows $NV(0) = \emptyset$.

2. For every predicate $p(\bar{s}, \bar{t})$ in P, the Data-handler equations for $\underset{[\bar{t}] = [\bar{t}]}{\mathfrak{D}^{(|\bar{t}|)}}$ (see Table 1), and the *otherwise-equation*:

$$p\,x \;=\; \mathsf{Fail}$$

3. For every occurrence of a function $\underset{[\cdots] = [\cdots]}{\mathfrak{D}^{(n)}}$ in the equation of point 2, the corresponding Data-handler equations (see Table 1). □

The idea behind the transformation is that every clause

$$p_0(\bar{t}_0; \bar{s}_{n+1}) \;\leftarrow\; p_1(\bar{s}_1; \bar{t}_1), \ldots, p_n(\bar{s}_n; \bar{t}_n)$$

provides an equation defining $p_0[\bar{t}_0]$ (i.e. p_0 applied to its input arguments). Its output value $[\bar{s}_{n+1}]$ is obtained in the following way.

Informally, $NV(k)$, for $1 \leq k \leq n$, denotes the Variables of $p_1, \ldots, p_{k-1}$ that are Needed for the input arguments of $p_{k+1}, \ldots, p_n$ and for the output argument of the head predicate p_0 (i.e. $\bar{s}_{k+1}, \ldots, \bar{s}_{n+1}$).

We start with the input data $[\bar{t}_0]$. Then, applying the first Data-handler $\underset{[NV(0), \bar{t}_0] = [NV(1) | p_1[\bar{s}_1]]}{\mathfrak{D}^{(|\bar{t}_0|)}}$, we calculate $p_1[\bar{s}_1]$ (that gives its output values for $\bar{t}_1$) together with the values from $\bar{t}_0$ that are needed in the sequel to calculate some other $p_i[\bar{s}_i]$ or the final output $\bar{s}_{n+1}$ (i.e. $NV(1)$). The process goes on till all the $p_1, \ldots, p_n$ have been processed, and the last Data-handler $\underset{[NV(n), \bar{t}_n] = [\bar{s}_{n+1}]}{\mathfrak{D}^{(|\bar{t}_n|)}}$ simply passes to the final output $\bar{s}_{n+1}$ the values previously computed (present in $[NV(n), \bar{t}_n]$).

A graphical representation of the followed approach is illustrated in Figure 1.

In point 2., an *otherwise-equation* (that is, an equation where the pattern is a variable) is introduced: indeed, just like in a logic program a failure occurs when the selected atom of the goal do not unify with every clause head, here when no other equation is applicable to solve a function the otherwise-equation is selected producing Fail as result (having a variable as pattern ensures it behaves like a 'catch-all').

Note that the Data-handler equations produced by points 2. and 3. of the transformation can be deduced from the single equation produced in 1.: hence, for brevity, from now on we will omit them in program listings.

Another useful simplification can be obtained observing that in the equation of point 1. the rightmost Data-handler can be applied right away to the argument $[\bar{t}_0]$. Indeed, since $NV(0) = \emptyset$, the rightmost Data-handler is $\underset{[\bar{t}_0] = [\bar{s}_1]}{\mathfrak{D}^{(|\bar{t}_0|)}}$ if the clause is a fact (i.e. the body is empty), and $\underset{[\bar{t}_0] = [NV(1) | p_1[\bar{s}_1]]}{\mathfrak{D}^{(|\bar{t}_0|)}}$ otherwise.

Hence point 1. can be modified saying that the corresponding equation is

$$p_0[\bar{t}_0] \;=\; [\bar{s}_1]$$

when the clause is a fact, and

$$p_0[\bar{t}_0] \;=\; \underset{[NV(n), \bar{t}_n] = [\bar{s}_{n+1}]}{\mathfrak{D}^{(|\bar{t}_n|)}} \;\; \underset{[NV(n-1), \bar{t}_{n-1}] = [NV(n) | p_n[\bar{s}_n]]}{\mathfrak{D}^{(|\bar{t}_{n-1}|)}} \;\; \cdots$$

$$\cdots \;\; \underset{[NV(1), \bar{t}_1] = [NV(2) | p_2[\bar{s}_2]]}{\mathfrak{D}^{(|\bar{t}_1|)}} \;\; [NV(1) | p_1[\bar{s}_1]]$$

otherwise. We will assume this modification in the following examples.

Example 6.2 Consider the well known append program used to split a list, i.e. with moding append(out,out,in) (we will refer to this usage of append as *split-append*):

```
append([],X,X) ←
append([A|X],Y,[A|Z]) ← append(X,Y,Z)
```

The transformation of the first clause yields

```
append[X] = [[],X]
append X  = Fail
```

and the transformation of the second gives

$$\texttt{append[[A|Z]]} = \underset{[A, X, Y] = [[A|X], Y]}{\mathfrak{D}^{(2)}} \texttt{[A|append[Z]]}$$

```
append X = Fail
```

□

Definition 6.3 ($\odot$ map)
Let a function $\mathtt{M}$ be defined this way:

$$\begin{array}{l}
\mathtt{M}(\{lhs^P = rhs^P\} \cdot P', Q) \;=\; \mathtt{N}(\{lhs^P = rhs^P\}, Q) \cdot \mathtt{M}(P', Q) \\
\mathtt{M}(\{\,\}, Q) \;=\; \{\,\} \\
\mathtt{N}(\{lhs^P = rhs^P\}, \{lhs^Q = rhs^Q\} \cdot Q') \;=\; \\
\quad\;\; \mathtt{if}\ \exists\beta = mgu(lhs^P, lhs^Q)\ \mathtt{then} \\
\quad\quad\;\; \{lhs^p = J(rhs^P\beta, rhs^Q\beta)\} \cdot \mathtt{N}(\{lhs^P = rhs^P\}, Q') \\
\quad\;\; \mathtt{else}\ \mathtt{N}(\{lhs^P = rhs^P\}, Q') \\
\mathtt{N}(\{lhs^p = rhs^P\}, \{\,\}) \;=\; \{\,\}
\end{array}$$

Let $\mathtt{normalize}$ be a function that deletes from its input program S all the otherwise-equations and appends at the end of the program new otherwise-equations corresponding to all the operators in S.

Then, $P \odot Q \;=\; \mathtt{normalize}\big(\mathtt{M}(P, Q) \cdot \tau(P) \cdot \tau(Q)\big)$. □

The function $\mathtt{M}$ can be expressed in this equivalent, more readable way:

Let P be $\left\{ \begin{array}{rcl} op_1^P[\bar{s}_1] & = & e_1^P \\ op_2^P[\bar{s}_2] & = & e_2^P \\ & \vdots & \\ op_n^P[\bar{s}_n] & = & e_n^P \end{array} \right\}$ and Q be $\left\{ \begin{array}{rcl} op_1^Q[\bar{t}_1] & = & e_1^Q \\ op_2^Q[\bar{t}_2] & = & e_2^Q \\ & \vdots & \\ op_m^Q[\bar{t}_n] & = & e_n^Q \end{array} \right\}$

Then $\mathtt{M}(P, Q)$ is simply

$$\{op_i^P[\bar{s}_i\beta] \;=\; J(e_i^P\beta, e_j^Q\beta) \;:\; \exists\beta = mgu(op_i^P[\bar{s}_i], op_j^Q[\bar{t}_j])\}$$

where the equations are ordered lexicographically w.r.t. (i, j) (viz. the indexes i of the first program P have precedence over the indexes j of the second program Q).

The behaviour of the $\odot$ map should be quite clear: First, employing $\mathtt{M}$, whenever two equations overlap in some cases, insert an equation that Joins them for these cases only (this is why a most general unifier is used) via the binary symbol J. Then, the original equations are concatenated (to cope with the other cases). Finally, the result is $\mathtt{normalize}$-d in the sense that the otherwise-equations are placed in the correct position (at the end) to work properly.

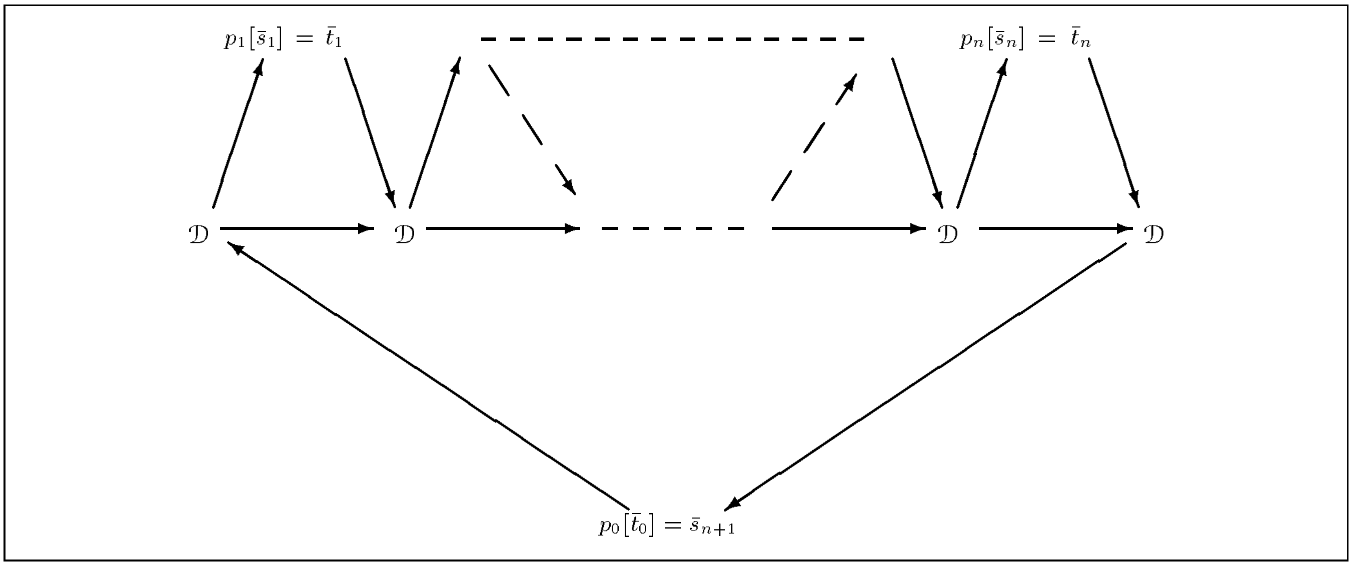

Figure 1: Dataflow representation of a clause $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1;\bar{t}_1),\ldots,p_n(\bar{s}_n;\bar{t}_n)$.

Although $\odot$ as it is works fine, a useful improvement is to get rid of useless equations via the following transition rule:

$$\frac{P\cdot\{op\,pat_1 = e_1\}\cdot Q\cdot\{op\,pat_2 = e_2\}\cdot R}{P\cdot\{op\,pat_1 = e_1\}\cdot Q\cdot R}(\exists\vartheta\; pat_1\vartheta = pat_2)$$

expressing the fact that the equation $op\,pat_2 = e_2$ can be deleted since $op\,pat_1 = e_1$ will be always selected before. We will assume this simplification is added to the definition of $\odot$ in the further examples.

Example 6.4 The transformation of the program split-append (cf. Example 6.2) is:

$$\texttt{append[[A|Z]]} = \texttt{J([[ ],[A|X]],}\ \mathcal{D}^{(2)}_{[A,X,Y]=[[A|X],Y]}\ \texttt{[A|append[Z]])}$$
$$\texttt{append[X]} = \texttt{J([[ ],X],Fail)}$$
$$\texttt{append X} = \texttt{Fail}$$

□

Theorem 6.5 *Let $G = \leftarrow p(\bar{s};\bar{t})$ and P a program, both Functionally Moded. If $\boldsymbol{\alpha}_P(G) = \phi_1,\phi_2,\ldots,\phi_k$ then*

$$\big(\tau(P)\big)\big(\tau(G)\big) = J([\bar{t}\phi_1], J([\bar{t}\phi_2],\ldots J([\bar{t}\phi_{k-1}],[\bar{t}\phi_k])\ldots))$$

Example 6.6 We will calculate all the possible splittings of the list `[a]` via the translation by τ of the split-append program.

The goal is $\leftarrow$ `append(X,Y,[a])`, that is transformed via τ into $\mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}$ `append[[a]]`.

Reducing this expression yields (when a Data-handler equation is applied, it is indicated under the equality symbol):

$$\mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{append[[a]]}$$
$$= \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{J([[],[a]],}\ \mathcal{D}^{(2)}_{[A,X,Y]=[[A|X],Y]}\ \text{[a|append[[]]])}$$
$$= \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{J([[],[a]],}\ \mathcal{D}^{(2)}_{[A,X,Y]=[[A|X],Y]}\ \text{[a|J([[],[]],Fail)])}$$
$$\underset{D9}{=} \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{J([[],[a]],}\ \mathcal{D}^{(2)}_{[A,X,Y]=[[A|X],Y]}\ \text{J([a,[],[]],[a|Fail]))}$$
$$\underset{D4}{=} \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{J([[],[a]], J([[a],[]],}\ \mathcal{D}^{(2)}_{[A,X,Y]=[[A|X],Y]}\ \text{[a|Fail]))}$$
$$\underset{D3}{=} \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{J([[],[a]], J([[a],[]],Fail))}$$
$$\underset{D4}{=} \text{J([[],[a]],}\ \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{J([[a],[]],Fail))}$$
$$\underset{D4}{=} \text{J([[],[a]], J([[a],[]],}\ \mathcal{D}^{(2)}_{[X_1,X_2]=[X_1,X_2]}\ \text{Fail))}$$
$$\underset{D2}{=} \text{J([[],[a]], J([[a],[]],Fail))}$$

□

From Theorem 6.5 it easily follows:

Corollary 6.7 *The transformation τ is complete.*

6.1 Modular Programming

An important feature of the transformation τ is that it integrates well with modular programming. Indeed, let us denote with $\lhd$ the module relation (that is, $P \lhd Q$ if P does not define predicates/functions that are defined in Q). The transformation is *module-preserving* in the sense that:

Lemma 6.8 $P \lhd Q \Rightarrow \tau(P) \lhd \tau(Q)$

(to be fussy, note that actually if $P \lhd Q$ some Data-handler equation defined in $\tau(P)$ could be repeated in $\tau(Q)$: besides this is not a problem in practice, it can be fixed via a suitable renaming of the Data-handler equations).

Also, the compositional behaviour of τ gets simpler:

Lemma 6.9 $P \lhd Q \Rightarrow\ \tau(P\cdot Q) = \tau(P)\cdot\tau(Q)$

Hence when dealing with modules $\odot$ becomes program composition.

Thanks to Lemmata 6.8 and 6.9, all the benefits of modular programming can be retained when passing from logic to functional programming: modules/libraries in logic programming can be translated into corresponding modules of the target functional language. Also, in project development this avoids global re-transformation in case a single module is modified.

7 Complexity

As said in the Introduction, we are going to use complexity arguments to provide a suitable notion of functionality.

We take as (time) complexity of a logic program (resp. of a functional program) P which is run with a goal (resp. an expression) G, the number of resolution steps (resp. of rewrite steps) it performs, and denote this measure by $\mathcal{C}\big(P,G\big)$. Having defined a suitable measure of the execution time of a program, it makes sense to talk about the corresponding complexity classes with respect to this measure. Now we have at our disposal a mean to say when a class Λ of logic programs is inherently functional in nature: it must have a transformation that translates every logic program in Λ into a functional program having the same (or lower) complexity. This is expressed by the following definition:

Definition 7.1 A transformation γ for a class Λ of logic programs is said to be *complexity preserving* if there is a $k \in \mathbb{N}$ such that for every program $P \in \Lambda$ and goal $G \in \Lambda$

$$\mathcal{C}\big(\gamma(P), \gamma(G)\big) \leq k \cdot \mathcal{C}\big(P, G\big) \qquad \square$$

Observe that the linear factor k is necessary to ensure the independence of the particular unity measure employed.

With the aid of this new requirement, we can formalize the vague concept of 'being functional':

Definition 7.2 A class of logic programs Λ is *functional* if it has a transformation which is *complete* and *complexity preserving*. $\square$

Functionally Moded programs, as expected, satisfies the above condition:

Theorem 7.3 *The class of Functionally Moded programs is functional.*

7.1 Optimality

We will now show that the class of Functionally Moded programs can be by right considered *the* functional core of logic programs, since it cannot be enlarged without losing functionality. In fact, we will prove even more. Let us weaken the notion of functional class this way:

Definition 7.4 A class of logic programs Λ is *hardly functional* if it has a transformation γ which is *complete* and there is a computable function $\mathcal{F}$ such that for every program $P \in \Lambda$ and goal $G \in \Lambda$

$$\mathcal{C}\big(\gamma(P), \gamma(G)\big) \leq \mathcal{F}\big(\mathcal{C}\big(P, G\big)\big) \qquad \square$$

Hence, we allow the transformation to arbitrarily worsen the run-time behaviour of the logic program, only provided the worsening can be expressed by a computable function $\mathcal{F}$ (for instance, with $\mathcal{F}(x) = 2^{(2^x)}$ a class of logic programs in P could be translated into one in 2-EXPTIME).

We can now state the following surprising result:

Theorem 7.5 (Optimality) *Every class of logic programs that properly contains the Functionally Moded programs is not hardly functional.*

Since by Theorem 7.3 *FM* is also functional, and functional implies hardly functional, we obtain straight away the following corollary:

Corollary 7.6 *The class of Functionally Moded programs is maximal among the functional ones.*

That is to say, we cannot improve on the *FM* class without loosing functionality (even hardly functionality!).

8 Beyond Completeness

As we have seen, completeness suffices to prove the Optimality Theorem. However, from a practical point of view, we could be interested in a stronger correspondence. This happens when we are not interested in *all* the answers of a logic program, but only in part of them. For instance, if $\boldsymbol{\alpha}_P(G) = \phi_1, \perp$ the logic program yields one answer, and a complete transformation could not grasp this information because the program is nonterminating. A natural way to fix this problem is to employ a lazy reduction strategy for the functional program, and defining a corresponding notion of completeness. However, we stick to the generality requirements expressed in Subsection 2.2, and define an extension of completeness that also preserves the answers contained in a nonterminating logic program *disregarding* the specific reduction strategy:

Definition 8.1 A transformation γ is said to be *hypercomplete* for a class Λ of logic programs if there are two families $(\mathbb{A}_n)_{n\in\widetilde{\mathbb{N}}}$ and $(\mathbb{B}_n)_{n\in\widetilde{\mathbb{N}}}$ of computable functions such that for every program $P \in \Lambda$ and goal $G \in \Lambda$, if $\boldsymbol{\alpha}_P(G) = \phi_1, \ldots, \phi_k$ then

$$\forall i \in \widetilde{\mathbb{N}}.\ \mathbb{A}_i\big(\mathrm{Var}(G), \big(\gamma(P)\big)\big(\mathbb{B}_i(\gamma(G))\big)\big) = \phi_1, \ldots, \phi_{\min(i,k)}$$

$\square$

If for a transformation hypercompleteness holds when the reduction strategy is fixed to be lazy, then we call such a transformation *lazy-complete*.

Thus, an hypercomplete transformation allows to calculate also finite approximations of the answer semantics, allowing to fully reconstruct the answer semantics not only of terminating, but also of nonterminating logic programs.

The transformation τ here presented is not hypercomplete. It can however be modified to be such, essentially making all the functions parametric to store how many answers have been so far calculated, and modifying the equations using continuations techniques (cf. [2]).

Being completeness implied by hypercompleteness, the existence of such a transformation for the class of Functionally Moded programs makes possible to restate the analogous of the Optimality Theorem, where functionality is replaced by the stronger notion of 'hyperfunctionality' (replacing completeness by hypercompleteness in Definition 7.2): *FM* is maximal among the 'hyperfunctional' classes of logic programs[4].

Using lazy reduction, however, makes things simpler. Indeed, Theorem 6.5 holds for nonterminating logic programs as well. Hence if a logic program P is run with a goal G (both in *FM* of course) yielding $\boldsymbol{\alpha}_P(G) = \phi_1, \ldots, \phi_k$ with $\phi_k = \perp$, then $\big(\tau(P)\big)\big(\tau(G)\big) = J([\bar{t}\phi_1], J([\bar{t}\phi_2], \ldots J([\bar{t}\phi_{k-1}], \perp)\ldots))$. Thus via lazy reduction one can extract one by one the answers ϕ_i $(i < k)$ avoiding to calculate the inner nonterminating part corresponding to ϕ_k.

This can be formally expressed by saying that τ *is lazy-complete*: it suffices to take $\mathbb{B}_\omega(x) \equiv x$, and for $i < \omega$

$$\mathbb{B}_i(x) \equiv \mathcal{E}(\underbrace{s(\ldots s(0)\ldots)}_{i\ times}, x)$$

where $\mathcal{E}$ is so defined:

$$\begin{aligned} \mathcal{E}(s(s(x)), J(u, J(v,w))) &= J(u, \mathcal{E}(s(x), J(v,w))) \\ \mathcal{E}(s(x), \mathsf{Fail}) &= \mathsf{Fail} \\ \mathcal{E}(s(0), J(u, J(v,w))) &= u \end{aligned}$$

[4] Of course this follows from a stronger result (cf. Theorem 7.5) of maximality among the 'hardly hyperfunctional' classes...

8.1 Infinite Lists

As said, usage of logic programming directly in functional programming can be extremely useful when lazy reduction is used. In this case, besides the usual advantages of using logic programming when expressing relations, it is added the further power of expressing infinite lists via logic programs with infinite answers.

As a simple example, the infinite list of naturals $[0, s(0), s(s(0)), \ldots]$ can be expressed in a functional program just using the definition of natural number ([26]):

```
natural(0) ←
natural(s(X)) ← natural(X)
```

where the moding is `natural`(out).

9 Normal Logic Programs

After having analyzed definite logic programming, we extend the results previously obtained to normal logic programming, that is allowing usage of *negation*. As usual in Prolog, negated atoms are solved using the negation as finite failure procedure, i.e. they succeed if and only if they finitely fail (see e.g. [3]).

Since we have already given the definition of functionally moded for a definite goal/clause, we can give the definition of Functionally Moded for a normal logic program only giving it for goal or clauses with at least one occurrence of the *not* operator:

Definition 9.1 A goal *not* $(p(\bar{s};\bar{t}))$ is *functionally moded* if $p(\bar{s};\bar{t})$ is such. A clause $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1;\bar{t}_1), \ldots, \mathit{not}\ (p_k(\bar{s}_k;\bar{t}_k)), \ldots, p_n(\bar{s}_n;\bar{t}_n)$ of a normal logic program is *functionally moded* if both $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1;\bar{t}_1), \ldots, p_k(\bar{s}_k;\bar{t}_k)$ and $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1;\bar{t}_1), \ldots, p_{k-1}(\bar{s}_{k-1};\bar{t}_{k-1}), p_{k+1}(\bar{s}_{k+1};\bar{t}_{k+1}), \ldots, p_n(\bar{s}_n;\bar{t}_n)$ are such. A program is *Functionally Moded* if each its clause is such. □

Example 9.2 Suppose p and q are both moded (in, out). Then $p(X, f(Z)) \leftarrow q(X,Y), \mathit{not}\ (p(Y,Z)), q(Y,Z)$ is Functionally Moded since both $p(X, f(Z)) \leftarrow q(X,Y), p(Y,Z)$ and $p(X, f(Z)) \leftarrow q(X,Y), q(Y,Z)$ are. □

The transformation τ can be extended to cope with Functionally Moded (normal) logic programs with little modifications.

First, new equations defining the *not* are added (see Table 2).

Second, every literal of the form $\mathit{not}\ p(\bar{s};\bar{t})$ that appears in a goal or a clause is viewed by the transformation like $(\mathit{not}\ p)(\bar{s};)$ (that is, the compound function *not* p has as inputs the old inputs of p but produces no output, correspondingly to the fact that a successful negative literal produce no bindings).

For instance, $p(x;y) \leftarrow \mathit{not}\ q(x;z), r(y;z)$ is seen as $p(x;y) \leftarrow (\mathit{not}\ q)(x;), r(y;z)$.

With these simple extensions, all the results previously proved only for definite programs generalize verbatim to normal logic programs.

Example 9.3 In order to briefly illustrate the behaviour of the *not*, suppose the usual split-append program is called with the goal $\leftarrow \mathit{not}\ (\mathit{append}(X,Y,[a]))$: τ translates this goal into the expression $\underset{[\,]=[\,]}{\mathcal{D}^{(0)}}\ \mathit{not}\ (\mathit{append}[[a]])$. Analogously to the reduction seen in Example 6.6, this expression reduces to $\underset{[\,]=[\,]}{\mathcal{D}^{(0)}}\ \mathit{not}\ J([[\,],[a]], J([[a],[\,]], \mathsf{Fail}))$ that via the *not* equation N5 reduces to $\underset{[\,]=[\,]}{\mathcal{D}^{(0)}}\ \mathsf{Fail}$, and via the Data-handler equation D2 finally gives Fail. □

N1	$\mathit{not}\,[\,] = \mathsf{Fail}$
N2	$\mathit{not}\,[u\vert v] = \mathsf{Fail}$
N3	$\mathit{not}\,\mathsf{Fail} = [\,]$
N4	$\mathit{not}\ J([\,],v) = \mathsf{Fail}$
N5	$\mathit{not}\ J([u\vert v],w) = \mathsf{Fail}$
N6	$\mathit{not}\ J(\mathsf{Fail},v) = \mathit{not}\ v$
N7	$\mathit{not}\ J(J(u,v),w) = \mathit{not}\ J(u,J(v,w))$

Table 2: The *not* equations.

10 Integration of Languages

The transformation τ presented here is of obvious importance also in the integration of logic and functional programming (cf. [12] for a survey), since it allows to combine in a direct and clean way logic programming with functional programming.

The first and simpler form of integration offered by the transformation is to mix every functional language with the functionally moded logic programs: functional code can be interleaved directly with such logic programming code, and hence the programmer can decide to write a certain part of the program in functional or logic programming style according to what paradigm suites better.

Another kind of integration is to allow external functions directly in the logic program (this is also known as *amalgamation*). Much work has been done on this topic, like usage of residuation principles together with sophisticated forms of incomplete equational unification (e.g. S-unification in [4], and P-unification in [15]); along with these studies also a concrete language, GAPlog, has been implemented (cf. [14, 17]).

All these approaches have to face many difficulties both of theoretical and of practical (implementative) nature, essentially due to the fact that logic programming as a whole is *not* functional, as shown in this paper.

Instead, focusing only on the functional core of logic programming here found, we can give a much simpler and cleaner solution to the amalgamation problem.

Call P the Functionally Moded logic program, and F the functional program to be amalgamated. We divide the function symbols that can be used in a logic program into *external* and *internal* ones: external symbols can appear outside the logic program (i.e. in F), while internal symbols cannot.

The *amalgamation condition* for P and F is simply: in every clause $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1;\bar{t}_1), \ldots, p_n(\bar{s}_n;\bar{t}_n)$ of P, external symbols that are operators in F cannot occur in $\bar{t}_0$.

Example 10.1 As a simple example, consider the following GAPlog program to compute the factorial, using the external symbols 0, 1, $*$ and $-$ (interpreted as usual in the functional program to amalgamate):

```
factorial(0,1) ←
factorial(X,X*Y) ← factorial(X-1,Y)
```

This program is Functionally Moded taking `factorial` moded (in, out), and satisfies the amalgamation condition. □

The amalgamation condition is completely natural in view of the functional nature of the *FM* class: indeed, a clause $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow \ldots$ is seen as a function definition for p_0 (i.e. $p_0[\bar{t}_0] = \ldots$), and so the unique restriction it should satisfy is that no operator (i.e. no other defined function) can appear in the pattern $[\bar{t}_0]$. Hence, the amalgamation condition for a logic program P and a functional program F becomes simply the module relation $\tau(P) \lhd F$ (cf. Subsection 6.1).

It should be noted that amalgamated programs as defined in [4] can also cope with external *predicate* symbols that can be used as boolean tests. However, we can consider such apparent extra power just as syntactic sugar: every occurrence of 'external predicates' $p(\bar{s})$ can be seen more properly as an occurrence of $test(p(\bar{s}))$, where $test$ is defined in the logic program by $test(\text{true}) \leftarrow$, and both p and $true$ are external symbols ($true$ is intended to be interpreted, as usual in every reasonable functional language, as the boolean truth function).

Example 10.2 Consider the following program after [4]:

```
leq(0,x) ←
leq(y+1,x) ← 1≤x, leq(y,x-1)
```

The `leq` predicate is designed to be the 'less than or equal to' relation on naturals, to be called with its first argument uninstantiated (e.g. ← `leq(x,2)` would give as answers for `x` the naturals `0`, `1` and `2`).

In [4] $\leq$ is an external *predicate* symbol. But, as said, the above program can be seen as syntactic sugar for the following one:

```
leq(0,x) ←
leq(y+1,x) ← test(1≤x), leq(y,x-1)
test(true) ←
```

where now `0`, `1`, `+`, `-`, $\leq$ and `true` are all external symbols (interpreted as usual in the functional program to amalgamate).

And, in fact, with moding `leq`(out, in) and `test`(in) the above program is Functionally Moded and satisfies the amalgamation condition. □

Let us summarize what are the differences of our approach to the integration of logic and functional programs with the others present in the literature.

Being logic and functional programming different paradigms, all their integrations have been so far obtained by defining a new language that *extends* logic (or functional) programming to cope with these differences.

Here, we follow *just the opposite way*: we define a new (sub-)language that *diminishes* logic programming to drop these differences.

On the one hand, all the other approaches try to seize the whole expressive power of logic programming. On the other hand, they need to define every time a new language, not only with theoretical problems (semantics, completeness, efficiency etc.) but also with the *practical* problem of building a correct (and possibly fast!) specific implementation for the language. Instead, our approach does not try to retain the full power of logic programming, but does not have the aforementioned problems since via the transformation we can rely on the existing implementation of the chosen target functional language.

Also, the amalgamation proposed in all the papers cited in this section is only 'one-way', that is the logic program is extended via external functions but not vice versa. Our proposal, instead, is completely symmetrical, since also the functional program can be extended using (possibly negated) atoms (since, via τ, they are just syntactic sugaring for function calls).

11 Further Applications

11.1 Using τ Beyond Functionally Moded

In this subsection we show how in some cases τ can be useful also outside the scope of Functionally Moded programs.

First, when in the definition of Functionally Moded programs we drop the requirement that $\bar{t}_0$ must be linear in a clause $p_0(\bar{t}_0;\bar{s}_{n+1}) \leftarrow p_1(\bar{s}_1;\bar{t}_1),\ldots,p_n(\bar{s}_n;\bar{t}_n)$, the transformation τ produces a functional program with possibly non-linear patterns. This program is directly utilizable if the target language supports them (like, for instance, Miranda). Even if the language does not support non-linear patterns but support guards and the equality test (as it is the case for a great deal of functional languages), then it can be proved that the usual implementation of non-linear patterns via guards (see e.g. [21, Ch. 4]) is safe. In most of cases, both of these two alternatives do not significantly affect the execution time.

Second, *FM* could be extended by making use of *program transformation*: Suppose to have a transformation $\mathfrak{T}$ from a class Λ of logic programs into *FM*. This transformation cannot both be complexity preserving and retain the answer semantics by the Optimality Theorem 7.5. Anyway, one of these two properties could be relaxed, and tested singularly on each program of Λ using a sufficient criterion. If the test is passed, the program is functional (and can be transformed into an equivalent functional program using the compound transformation $\tau \circ \mathfrak{T}$).

An example of this method has been given in [19], where it is defined a transformation $\mathcal{C}$ (called *conormal*) that translates into *FM* a big class of normal logic programs (e.g. all Well Moded programs (Def. 3.3) belong to this class). This program transformation is complexity preserving, and preservation of the answer semantics is relaxed into preservation of the least Herbrand Model (cf. [16]). Also, $\mathcal{C}$ is compositional (more precisely, $\mathcal{C}(P \cdot Q) = \mathcal{C}(P) \cdot \mathcal{C}(Q)$) and module-preserving. To check when $\mathcal{C}$ preserves the answer semantics, some global criteria are employed.

Finally, we remark that the program transformation technique may be also useful to study termination properties of logic programs (see the next subsection).

11.2 Termination

An interesting application of hypercomplete transformations is the study of termination properties of logic programs. In logic programming, indeed, the study of termination is a very hard problem already for universal termination, and existential termination is still an open problem (cf. [8]).

A hypercomplete transformation γ can provide a mean for studying a whole spectrum of termination properties: the only requirement is that the functions $\mathbb{B}_i$ can be defined in the functional language used with γ (indeed, this is the case for the modified τ we hinted at in Section 8). Let's call $def(\mathbb{B}_i)$ this definition. Then:

- P (universally) terminates w.r.t. G iff $(def(\mathbb{B}_\omega) \cup \gamma(P))(\mathbb{B}_\omega(\gamma(G)))$ terminates
- P existentially terminates w.r.t. G iff $(def(\mathbb{B}_1) \cup \gamma(P))(\mathbb{B}_1(\gamma(G)))$ terminates

- P k-terminates w.r.t. G iff $(def(\mathbb{B}_k) \cup \gamma(P))(\mathbb{B}_k(\gamma(G)))$ terminates (a program k-terminates iff its first k answers are different from $\perp$)

Although we did not explicitly include in this paper the modifications to make τ hypercomplete, τ itself can be very useful in view of the following result:

Theorem 11.1 *Let P and G be respectively a Functionally Moded program and goal: then P terminates w.r.t. G if and only if $\tau(P)$ terminates w.r.t. $\tau(G)$.*

Hence the techniques to prove termination of functional programs (e.g. [5, 10, 11]) can be used, via τ, also for Functionally Moded logic programs.

The above theorem can also be combined with program transformation: if we have a transformation $\mathcal{T}$ from a class Λ of logic programs into *FM* that preserves nontermination (i.e. if $P \cup \{G\}$ does not terminate then $\mathcal{T}(P) \cup \{\mathcal{T}(G)\}$ does not terminate as well), then we can look for sufficient termination criteria of the programs in the whole class Λ simply analyzing their corresponding functional programs (obtained via the composition of τ with $\mathcal{T}$). For instance, the conormal transformation mentioned in Subsection 11.1 preserves nontermination, and hence it can be used to study a much bigger class than *FM*.

It should be noted that there are also other transformational approaches to study termination of logic programs (e.g. the works of [23] and [18]), where a logic program is translated into a term rewriting system. Anyway, those approaches are in principle less general than ours, since they can cope only with universal termination, and cannot treat negation.

12 Efficiency

We have seen that τ, besides its use in the proof of the Optimality Theorem, can have a variety of applications. In particular, it can be used to integrate (Functionally Moded) logic programming directly in functional programming (or to amalgamate them as seen in Section 10). In this section we try to give some indications on how this approach is good not only in theory, but also in practice.

We will test the possible use of the transformation τ in conjunction with an existing implementation of a functional language like program compiler of Functionally Moded programs, comparing the efficiency of the obtained code with the original Prolog one.

It should be remarked that the transformation we presented in this paper has not been specifically designed for an implementation but only for presentation purposes. Among the practicable improvements we mention:

- Simple and time-gaining optimizations of τ are possible (e.g. via a simple analysis many Data-handlers can be removed).
- When it is known what reduction strategy is utilized, further optimizations are possible (for instance, the Data-handler equations schema can be written in a more efficient way).
- The runtime behaviour can be greatly improved making use of features typical of the target language, when this is known in advance (e.g. guards).

Program	Sicstus Prolog	τ + Clean	Saving
Ackermann	0.56	0.17	69.6%
Engineering	3.09	0.05	98.4%
Fourcolors	2.27	0.56	75.3%
Quicksort	0.68	0.57	16.2%
Reverse	3.18	1.02	67.9%
Split-append	13.30	0.55	95.9%

Table 3: Benchmark Comparison with Sicstus Prolog.

Despite of this, in the benchmarks we have used the pure code described by the transformation τ as presented in this version of the paper.

The languages employed are the commercial compiler Sicstus Prolog v2.1 (cf. [1]), and Concurrent Clean v0.8.4 (cf. [20, 22]), a public domain non-strict functional language (that can also be used as a strict language on request).

We chose six different programs: Ackermann (from [26, pag. 41]) is the usual Ackermann function, Reverse is the list reversal ([26, pp. 48–49]), Quicksort is the classical sort program (after [26, pag. 56]), Split-append is the example program seen in this paper (Example 6.2), Fourcolors is a program by L. Pereira and A. Porto to solve the four colors map problem ([6, pp. 200–201]), and Engineering is a planning problem for civil engineering after [6, pp. 189–190].

All the tests were executed using a Sparc Classic running SunOS 4.1.3C with 48Mbytes of RAM and a cache memory of 128Kbytes. Like in [13], timings are calculated taking the best value obtained out of ten runs.

Also, the Sicstus Prolog compiler was not used with its default byte code, but with the fastcode option specific for Sun architectures (roughly three times faster than the default), and on the other hand no 'heap tuning' (cf. [13]) was made to improve Clean execution.

The results are presented in Table 3 (run times are expressed in seconds). Although, of course, the transformation should be tested with heavier applications, we think the results give the flavor that employing transformational techniques like that used here can be productive not only from a theoretical but also from a practical point of view.

Acknowledgements

I wish to thank Krzysztof R. Apt and Jan Willem Klop. Thanks also to the referees for some useful style hints.

References

[1] J. Andersson, S. Andersson, K. Boortz, M. Carlsson, H. Nilsson, J. Widén, and T. Sjöland. Sicstus prolog user's manual. SICS Technical Report T93:01, Swedish Institute of Computer Science, 1993.

[2] A.W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

[3] K.R. Apt and R. Bol. Logic programming and negation: A survey. *Journal of Logic Programming*, 19,20:9–72, 1994.

[4] S. Bonnier and J. Małuszyński. Towards a clean amalgamation of logic programs with external procedures. In *International Conference on Logic Programming*, pages 311–326. MIT Press, 1988. Also in Proc. Int'l Conf. on Algebraic and Logic Programming, pages 20–34, LNCS, Springer, 1988.

[5] R. Cartwright. Recursive programs as definitions in first order logic. *SIAM Journal on Computing*, 12(2):374–408, 1984.

[6] H. Coelho and J.C. Cotta. *Prolog by Example*. Symbolic Computation Series. Springer–Verlag, 1988.

[7] J. Darlington, A.J. Field, and H. Pull. The unification of functional and logic languages. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 37–70, Englewood Cliffs, New Jersey, 1986. Prentice–Hall.

[8] D. de Schreye and S. Decorte. Termination of logic programs: The never-ending story. *Journal of Logic Programming*, 19,20:199–260, 1994.

[9] P. Dembinski and J. Małuszyński. AND-parallelism with intelligent backtracking for annotated logic programs. In *Proceedings of the International Symposium on Logic Programming*, pages 29–38, Boston, 1985.

[10] N. Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[11] S. Feferman. Logics for termination and correctness of functional programs. In Y.N. Moschovakis, editor, *Logic from Computer Science: Proceedings of a Workshop held November 13-17, 1989*, pages 95–127. Springer-Verlag, 1992.

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

[13] P.H. Hartel and K.G. Langedoen. Benchmarking implementations of lazy functional languages. In *Proceedings FPCA*, pages 341–349, 1993.

[14] A. Kågedal and F. Kluźniak. Enriching Prolog with S-unification. In J. Darlington and R. Dietrich, editors, *Workshop on Declarative Programming*, Workshops in Computing, pages 51–65. Springer, 1991.

[15] G. Lindstrom, J. Małuszyński, and T. Ogi. Our LIPS are sealed: Interfacing functional and logic programming systems. In *Proceedings of the 4th International Symposium on Programming Language Implementation and Logic Programming*, volume 631 of *LNCS*, pages 428–442. Springer, 1992.

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

[17] J. Małuszyński, S. Bonnier, J. Boye, F. Kluźniak, A. Kågedal, and U. Nilsson. Logic programs with external procedures. In K. Apt, J. de Bakker, and J. Rutten, editors, *Logic Programming Languages: Constraints, Functions, and Objects*, pages 21–48. MIT Press, 1993.

[18] M. Marchiori. Logic programs as term rewriting systems. In G. Levi and M. Rodríguez-Artalejo, editors, *Proceedings of the Third International Conference on Algebraic and Logic Programming*, volume 850 of *LNCS*, pages 223–241. Springer–Verlag, 1994.

[19] M. Marchiori. On safe transformation of logic programs. Draft. Forthcoming as technical report, 1995.

[20] E.G.J.M.H. Nöcker, J.E.W. Smetsers, M.C.J.D. van Eekelen, and M.J. Plasmeijer. Concurrent Clean. In E.H.L. Aarts, J. van Leeuwen, and M. Rem, editors, *3rd Parallel architectures and languages Europe (PARLE)*, volume 505/506 of *LNCS*, pages 202–220. Springer–Verlag, 1991.

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

[22] R. Plasmeijer and M. van Eekelen. *Functional Programming and Parallel Graph Rewriting*. International Computer Science Series. Addison-Wesley, 1994.

[23] K. Rao, D. Kapur, and R.K. Shyamasundar. A transformational methodology for proving termination of logic programs. In *Proceedings of the Fifth Conference on Computer Science Logic*, volume 626 of *LNCS*, pages 213–226, Berlin, 1992. Springer–Verlag.

[24] U.S. Reddy. Transformation of logic programs into functional programs. In *Proceedings of the International Symposium of Logic Programming*, pages 187–197. IEEE, February 1984.

[25] U.S. Reddy. On the relationships between logic and functional programming. In D. DeGroot and G. Lindstrom, editors, *Logic Programming: Functions, Relations and Equations*, pages 3–36. Prentice–Hall, Englewood Cliffs, New Jersey, 1986.

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