Functional Logic Programming in GCLA

Olof Torgersson*
Department of Computing Science, Chalmers University of Technology
S-412 96 Göteborg,Sweden
oloft@cs.chalmers.se

Abstract

We describe a definitional approach to functional logic programming, based on the theory of Partial Inductive Definitions and the Programming Language GCLA. It is shown how functional and logic programming are easily integrated in GCLA using the features of the language, that is combining functions and predicates in programs becomes a matter of programming methodology. We also give a brief description of a way to automatically generate efficient procedural parts to the described definitions.

1 Introduction

Through the years there have been numerous attempts to combine the two main declarative programming paradigms functional and logic programming into one framework providing the benefits of both. The proposed methods varies from different kinds of translations, embedding one of the methods into the other, to more integrated approaches such as Horn Clause Logic with equality [5] and Constraint Logic Programming as in the language Life [2].

A notion shared between functional and logic programming is that of a definition, we say that we *define* functions and predicates. The programming language can then be seen as a formalism especially designed to provide the programmer with an as clean and elegant way as possible to define functions and predicates respectively. Of course these formalisms are not created out of thin air but are explained by an appropriate theory.

In GCLA [1, 6] we take a somewhat different approach, we do talk about definitions but these definitions are not given meaning by mapping them on some theory about something else, but are instead understood through a theory of *definitions* and their properties, the theory of *Partial Inductive Definitions* (PID) [4]. This theory is designed to express and study properties of definitions, so we look at the problem from a different angle and try to answer the questions; what are the specific properties of function and predicate definitions and how

*This work was carried out as part of the work in the ESPRIT working group GENTZEN and was funded by The Swedish National Board for Industrial and Technical Development (NUTEK).

can they be combined and interpreted to give an integrated functional logic computational framework based on PID.

A GCLA program consists of two communicating partial, inductive definitions which we call the (*object level*) *definition* and the *rule definition* respectively. The rule definition is used to give a meaning to the conditions in the definition and it is also through this the user queries the definition. We present a rule definition to a class of functional logic program definitions. This rule definiton implicitly determines the structure of function, predicate and integrated functional logic program definitions. We also try to give a brief description of how to write functional logic GCLA programs. Furthermore we show how the knowledge that a definition defines a functional logic program can be used to automatically generate better proof-search strategies, enhancing efficiency and enabling us to write more concise definitions. The approach taken to functional logic programming in GCLA in this paper is inspired by some earlier work found in [1, 6].

2 The GCLA Programming Language

The programming system GCLA [1, 6, 3](to be pronounced "gisela") is a logical programming language that is based on a generalization of Prolog. This generalization is unusual in that it takes quite a different view of the meaning of a logic program - a definitional view rather than the traditional logic view.

Compared to Prolog, what has been added to GCLA is the possibility of assuming conditions. For example, the clause

```
a <= (b -> c).
```

should be read as: "`a` holds if `c` can be proved while assuming `b`".

There is also a richer set of queries in GCLA than in Prolog. In GCLA, a query corresponding to an ordinary Prolog query is written

```
\- a.
```

and should be read as "Does `a` hold (in the definition D)?" We can also assume things in the query, for example

```
c \- a.
```

which should be read as "Assuming `c`, does `a` hold (in the definition D)?", or "Is `a` derivable from `c`?"

To execute a program, a query G is posed to the system asking wether there is a substitution σ such that $G\sigma$ holds according to the logic defined by the program. The goal G has the form $\Gamma \vdash c$, where Γ is a list of assumptions, and c is the conclusion from the assumptions Γ. The system tries to construct a deduction showing that $G\sigma$ holds in the given logic.

2.1 GCLA Programs

A GCLA program consists of two parts; One part is used to express the declarative content of the program, called the *definition* or the *object level*, and the other part is used to express rules and strategies acting on the declarative part, called the *rule definition* or the *meta level*.

2.1.1 The Definition.

The definition constitutes the formalization of a specific problem domain and in general contains a minimum of control information. The intention is that the definition by itself gives a purely declarative description of the problem domain while a procedural interpretation of the definition is obtained only by putting it in the context of the rule definition.

2.1.2 The Rule Definition.

The rule definition contains the procedural knowledge of the domain, that is the knowledge used for drawing conclusions based on the declarative knowledge in the definition. This procedural knowledge defines the possible inferences made from the declarative knowledge.

The rule definition contains *inference rule* definitions which define how different inference rules should act, and *search strategies* which control the search among the inference rules.

The general form of an inference rule is

$Rule(A_1, \ldots, A_m, PT_1, \ldots, PT_n)$ <=
  $Proviso$,(PT_1 -> Seq_1,...,(PT_n -> Seq_n)
  -> Seq.

and the general forms of a strategy are

$Strat(A_1, \ldots, A_m)$ <= $PT_1, \ldots, PT_n$.

$Strat(A_1, \ldots, A_m)$ <=
  ($Proviso_1$ -> Seq_1),...,($Proviso_k$-> Seq_k).
$Strat(A_1, \ldots, A_m)$ <= PT1, ..., PTn.

where

- A_i are arbitrary arguments.
- *Proviso* is a conjunction of provisos, that is calls to Horn clauses defined elsewhere. The *Proviso* could be empty.
- Seq and Seq_i are sequents which are on the form ($Antecedent \vdash Consequent$), where *Antecedent* is a list of terms and *Consequent* is an ordinary GCLA term.
- PT_i are proofterms, that is terms representing the proofs of the premises, Seq_i.

2.2 Example: Default Reasoning

Assume we know that an object can fly if it is a bird and if it is not a penguin. We also know that Tweety and Polly are birds as well as all penguins, and that Pengo is a penguin. This knowledge is expressed in the following definition:

```
flies(X) <= bird(X),(penguin(X) ->false).

bird(tweety).   bird(polly).   bird(X) <= penguin(X).

penguin(pengo).
```

One possible rule definition enabling us to use this definition the way we want, is:

```
fs <=
   right(fs),           % First try standard right rules.
   left_if_false(fs).   % else if consequent is false...

left_if_false(PT) <=    % Is the consequent false?
   (_ \- false).
left_if_false(PT) <=    %  If so perform left rules.
   no_false_assump(PT),false_left(_).

no_false_assump(PT) <= not(member(false,A)) -> (A \- _).
no_false_assump(PT) <= left(PT).

member(X,[X|_]).        % Proviso Definition
member(X,[_|Ys]) :- member(X,Ys).
```

If we want to know which birds can fly, we pose the query

```
fs \\- (\- flies(X)).
```

and the system will respond with `X = tweety` and `X = polly`. If we want to know which birds cannot fly, we can pose the query

```
fs \\- (flies(X) \- false).
```

and the system will respond with `X = pengo`.

3 Functional Logic Programming: A First Example

The key to using GCLA as a (kind of) first-order functional programming language is the inference rule *D-left*, *Definition Left* also called the rule of *Definitional Reflection* [4, 6], which gives us the opportunity to reason on the basis of assumptions with respect to a given definition. The rule of Definitional Reflection tells us that if we have an atom a as an assumption and C follows from everything that defines a then C follows from a:

$$\frac{\Gamma, A \vdash C}{\Gamma, a \vdash C}\ D\vdash \text{ for each } a \in D(a)$$

where $D(a) = \{A \mid (a \Leftarrow A) \in D\}$. With this rule at hand functional evaluation becomes a special case of hypothetical reasoning. Asking for the value of the function `f` in `x` is done with the query:

```
f(x) \- C.
```

3.1 Defining addition

Functional programming languages can be regarded as syntactical sugar for typed lambda-calculus, or as languages especially constructed to define functions. In ML we can write a definition of addition using successor arithmetic:

```
datatype nat = zero | s of nat

fun plus(zero,N) = N
  | plus(s(M),N) = s(plus(M,N))
```

Functions in GCLA are also defined by a number of equations so the naive way to define addition as a function in GCLA would be to mimic the ML program and write the definition:

```
plus(zero,N)<= N.
plus(s(M),N)<= s(plus(M,N)).
```

Unfortunately this will not give us a function that computes anything useful since we have not defined what the value of `s(plus(M,N))` should be. What we forgot was that in the (strict) functional language ML the type definition and the computation rule together give a way to compute a value from `s(plus(M,N))`. The type definition says that `s` is a constructor function and since the language is strict the argument of `s` is evaluated before a dataobject is constructed. One way to achieve the same thing in GCLA is to introduce an object constructing function `succ`:

```
succ(X) <= (X ->Y) -> s(Y).
```

We read this as "the value of `succ(X)` is defined to be `s(Y)` if the value of `X` is `Y`" . The function `succ` plays much the same role as the constructor function `s` in the functional program, it is used to build dataobjects. We will sometimes call such functions that are used to build dataobjects *objectfunctions*.

How the function `succ` works can be seen in the following derivation:

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathtt{Y} = \mathtt{0}\}}{\mathtt{0} \vdash \mathtt{Y}}\ \mathit{Axiom}}{\vdash \mathtt{0} \rightarrow \mathtt{Y}}\ \mathit{A\text{-}right} \qquad \dfrac{\{\mathtt{s(Y)} = \mathtt{X}\}}{\mathtt{s(Y)} \vdash \mathtt{X}}\ \mathit{Axiom}}{(\mathtt{0} \rightarrow \mathtt{Y}) \rightarrow \mathtt{s(Y)} \vdash \mathtt{X}}\ \mathit{A\text{-}left}}{\mathtt{succ(0)} \vdash \mathtt{X}}\ \mathit{D\text{-}left}$$

The functional GCLA definition we have given so far only allow nested function calls in its second argument, an expression like `plus(plus(zero,zero),zero)` is undefined. Since we have no external computation rule telling us to evaluate arguments to functions before they are called we must add the following extra clause to handle this case:

```
plus(Exp,N)#{Exp\=zero, Exp\=s(_)}<=(Exp -> M) -> plus(M,N).
```

The guard is needed to make the different clauses defining addition mutually exclusive.

So at last we have a useful definition of addition. If we use the standard rules we are unfortunately still able to construct some undesired derivations like

$$\cfrac{\cfrac{\cfrac{}{\mathtt{false} \vdash \mathtt{X}}\;\textit{False-left}}{\mathtt{s(zero)} \vdash \mathtt{X}}\;\textit{D-left}}{\mathtt{plus(zero,s(zero))} \vdash \mathtt{X}}\;\textit{D-left}$$

which succeeds without binding `X`. The general problem that needs to be solved is to choose correctly between the rules *D-left* and *Axiom*. To do this we need to distinguish canonical values from non-canonical values. We achieve this by giving the canonical values circular definitions:

```
zero <= zero.
s(X) <= s(X).
```

Now the canonical values are defined so they can not be absurd, but on the other hand the definition is empty in content so there is nothing to be gained from using *D-left*. We also restrict our inference rules so that the *axiom* rule only is applicable to circularly defined atoms, and symmetrically restrict the rule *D-left* to be applicable only to atoms not circularly defined. We call atoms with circular definitions *canonical objects*.

In the final definition below the first three clauses corresponds to the *type definition* in the ML program, while the last three really constitutes the *definition* of the addition function.

```
0 <= 0.
s(X) <= s(X).

succ(X) <= (X <- Y) -> s(Y).

plus(0,N)<= N.
plus(s(M),N) <= succ(plus(M,N)).
plus(Exp,N)#{Exp \= 0, Exp \= s(_)}<= (Exp -> M) -> plus(M,N).
```

4 A Calculus for Functional Logic Programming

In this section we describe a basic rule definition to functional logic program definitions. We present the inference rules as a sequent calculus to enhance readability.

We call the rule definition consisting of the rules below *FL* (for functional logic). This rule definition shows implicitly the choices we have made as for what counts as a valid functional logic program. We have chosen to work with the common condition constructors ',' ,' ;' and ' ->' , which all have more or less their usual meaning. However, the inference rules are restricted to specialize them to functional logic programs. We have also added a special condition constructor ,'not' ,for negation.

One of the goals of *FL* is to give a useful rule definition producing as few redundant answers as possible, to achieve this we have made the calculus deterministic in the sense that at most one inference-rule can be applied to each object-level sequent.

FL is very similar to the calculus *DOLD* [6] used to interpret the meta-level of a GCLA program (that is, *DOLD* is used to interpret the GCLA code of the rules here described) This should not be surprising since the meta-level of a GCLA program is nothing but an indeterministic functional logic program run backwards.

4.1 Rules of Inference

The inference rules of *FL* can be naturally divided into two groups, rules relating atoms to a definition and rules for constructed conditions.

4.1.1 Rules Relating Atoms to a Definition.

$$\frac{\vdash C\sigma}{\vdash c \;\; \sigma}\ \textit{D-right}\quad (b \Leftarrow C) \in D, \sigma = mgu(b,c), C\sigma \neq c\sigma$$

$$\frac{D(a\sigma) \vdash C\sigma}{a \vdash C \;\; \sigma}\ \textit{D-left}\ \ \sigma \text{ is an } a\text{-sufficient substitution }, a\sigma \neq D(a\sigma)$$

$$\frac{}{a \vdash c \;\; \sigma\tau}\ \textit{D-Axiom}\ \ \sigma \text{ is an } a\text{-sufficient substitution }, a\sigma = D(a\sigma), \tau = mgu(a\sigma, c\sigma)$$

The restrictions we put on these definitional rules are such that they are mutually exclusive,a very important feature to minimize the number of possible answers.

4.1.2 Rules for Constructed Conditions.

The rules for constructed conditions are essentially Gentzen's sequent calculus rules restricted to allow at most one element in the antecedent. We only show a few of them as examples:

$$\frac{}{false \vdash false}\ \textit{False-left}$$

$$\frac{A \vdash B}{\vdash A \rightarrow B}\ \textit{A-right}$$

$$\frac{C_1 \vdash C \quad C_2 \vdash C}{(C_1; C_2) \vdash C}\ \textit{O-left}$$

4.2 Queries

Both the left and right hand side of sequents in queries may be arbitrarily complex as long as we remember that whenever there is something in the left hand side the right hand side *must* be a *variable* or a (partially instantiated) *canonical object*.

Assuming that `f`,`g` and `h` are functions, `p` and `q` predicates and `a`, `b` and `c` canonical objects some possible examples are

```
\- p(X).
```

"Does `p` hold for some `X`"

```
p(b) -> g(a,b) \- C
```

"What is the value of `g` in `(a,b)` provided that `p(b)` holds"

```
(p(X);q(X)) -> (f(X);h(X)) \- a.
```

"Find a value of `X` such that `p(X)` *or* `q(X)` holds and both `f`*and* `h` has the value `a` in X"

5 Writing Functional Logic Program Definitions

Now that we have given a calculus for functional logic program definitions, it is in order that we also describe the structure of the definitions it can be used to interpret and how to write programs. Note that all readings we give of definitions, conditions and queries are with respect to the given rule definition (calculus), *FL*, and that there is nothing intrinsical in the definitions *themselves* that forces this particular interpretation. For example, if we allow contraction, and several elements in the antecedent, function definitions as we describe them cease to make sense since it is then possible to prove things like

```
plus(s(0),s(0)) \- s(s(s(0))).
```

using the definition of addition given in Sect. 3.

Conditions are built up using the condition constructors '`,`' , '`;`' , '`->`' , '`true`' , '`false`' and '`not`'. Both functions and predicates are defined using the same condition constructors and there is no easily recognizable syntactic difference between functions and predicates. The difference in interpretation of functions and predicates instead comes from whether they are intended to be used to the left or to the right in queries.

When we read and write functional logic program definitions the condition constructors have different interpretations depending on wether they occur to the left or to the right,so when we write a function or a predicate we must always keep in mind wether the condition we are currently writing will be used to the left or to the right to understand its meaning. The basic principle is that predicates are used to the right (negation excepted), while functions are used to the left.

In order to show that it is not always so obvious to see what constitutes function and predicate definitions respectively, we look at a simple example:

```
q(X) <= p(X) -> r(X).
```

If we read this as defining a predicate, we read it as "`q(X)` holds if the value of `p(X)` is `r(X)`" , read as a (conditional) function it becomes "the value of `q(X)` is defined to be `r(X)` provided that `p(X)` holds" .

5.1 Defining Predicates

Since pure Prolog is a subset of GCLA, defining predicates is very much the same thing in this context as in Prolog even though the theoretical foundation is different. The predicate definitions allowed by *FL* also provides two extensions of pure Prolog in predicates ; use of functions in conditions defining predicates and constructive negation.

A predicate definition defining the predicate P consists of a number of definitional clauses

$P(t_1,\ldots,t_n) <= C_1.$
...
$P(t_1,\ldots,t_n) <= C_m\ .\ n \geq 0, m > 0$

where each C_i is a *Predicate Condition* as described below, and all variables occurring in C_i not occurring in the head of the clause are understood as existentially quantified.

We say that a condition C is a Predicate Condition if

- C is an atom
- $C = true$ or $C = false$
- $C = (C_1,C_2)$, where both C_1 and C_2 are predicate conditions. We read this as "C holds if C_1 and C_2 holds"
- $C = (C_1;C_2)$, where both C_1 and C_2 are predicate conditions. We read this as "C holds if C_1 or C_2 holds"
- $C = (C_1$ -> $C_2)$, where C_1 is a *Functional Condition* as described below and C_2 is a variable or a (partially instantiated) canonical object. We read this as "C holds if the value of C_1 is C_2"
- C = not(C_1) , where C_1 is a predicate condition. We read this as "C holds if C_1 can be proven false".

5.2 Defining Canonical Objects

Each atom that should be regarded as a canonical object in a definition, in the sense that it could possibly be the result of some function, is given a circular definition. From an operational point of view this is essential to prevent further application of the rule *D-left* and allow application of the rule *D-axiom*. These circular definitions also set canonical objects apart as belonging to a special class of terms since they can not be proven true or false in *FL*.

Generally the definition of the canonical objects S of arity n is

$$S(X_1, \ldots, X_n) <= S(X_1, \ldots, X_n).$$

where each X_i is a variable.

Note that we make a distinction between a *canonical object* and a *canonical value*. Any atom which has a circular definition is a canonical object, while a canonical value is a canonical object where each subpart is a canonical value (a canonical object of arity zero is also a canonical value).

5.2.1 Object-Functions and Implicit Type Definitions

In the definition of addition in Sect. 3 we used the objectfunction `succ` to ensure that all numbers we constructed were fully evaluated canonical values built up from `s` and `0` only. Generally to each canonical object S of arity n we create an objectfunction F to be used whenever we want to build an object of type S in a definition. F has the definition:

$$F(X_1, \ldots, X_n) <= (X_1 \text{ -> } Y_1), \ldots, (X_n \text{ -> } Y_n) \text{ -> } S(X_1, \ldots, X_n) \; .$$

The canonical objects and objectfunctions together form an *Implicit Type Definition*. The implicit type definition for lists is:

```
[] <= [].
[X|Xs] <= [X|Xs] .

cons(X,Xs) <= (X -> Y),(Xs ->Ys) -> [Y|Ys].
```

5.2.2 Laziness and Strictness.

When we use *FL* to evaluate programs it is the definition and not the computation rule that determines if a function is strict or lazy. To define a strict function we always use an objectfunction when we construct a dataobject in a program, this will ensure that evaluation continues until we reach a fully evaluated value. The definition of `plus` given in Sect. 3 follows this convention, as does the following definition of `append` which uses the implicit type definition of lists:

```
append([],Ys) <= Ys.
append([X|Xs],Ys) <= cons(X,append(Xs,Ys)).
append(E,Ys)#{E\=[],E\=[_|_]}<= (E -> Xs) -> append(Xs,Ys).
```

In lazy functional languages expressions are evaluated to WHNF, in our context this means that evaluation stops whenever we reach a canonical object, whether or not its subparts are fully evaluated. When we write function definitions this simply means that we omit the object-functions whose purpose is to evaluate the subparts of canonical objects. The lazy version of `append` then becomes:

```
append([],Ys) <= Ys.
append([X|Xs],Ys) <= [X|append(Xs,Ys)].
append(E,Ys)#{E\=[],E\=[_|_]}<= (E -> Xs) -> append(Xs,Ys).
```

If we use lazy evaluation function definitions must be uniform and have only shallow patterns [7].

5.3 Function Definitions

A definition of a function F really consists of two parts, an implicit type definition T made up of definitions of canonical objects and objectfunctions used in F, and the definition of F itself.

A function definition defining F then consists of a number of definitional clauses

$F(t_1,\ldots,t_n) <= C_1.$
...
$F(t_1,\ldots,t_n) <= C_m\ .\ n \geq 0, m > 0$

where each C_i is a *Functional Condition* as described below, and all variables occurring in C_i not occurring in the head are understood as universally quantified.

We say that a condition C is a Functional Condition if

- C is an atom
- $C = (C_1,C_2)$, where both C_1 and C_2 are functional conditions. We read this as "the value of C is the value of C_1 or C_2"
- $C = (C_1 \text{->} C_2)$, where C_1 is a predicate condition and C_2 is functional condition. We read this as "the value of C is C_2 provided that C_1 holds"
- $C = (C_1;C_2)$, where both C_1 and C_2 are functional conditions. We read this as "the value of C is B if B is the value of both C_1 and C_2".

If the heads of two or more clauses defining a function are overlapping all the corresponding bodies must evaluate to the same value, since the definiens operation used in *D-left* collects all clauses defining an atom. For example consider the function definition:

```
f(0) <= 0.
f(N) <= s(0)
```

Even though `f(0)` is defined to be `0` the derivation of the query

```
f(0) \- C.
```

will fail since `f(0)` is also defined to be `s(0)`, the definition of `f` is *ambiguous*.

5.4 Examples

If we use the lazy version of append and ask the query

```
append([0],[s(0)]) \- C.
```

we will get the single answer `C = [0|append([],[s(0)])]`, since this is a canonical value and cannot be evaluated further. To get a fully evaluated answer we have to force evaluation somehow, one way is to use the following function `show` which takes canonical values apart and evaluate the subparts:

```
show(0) <= 0.
show(s(X)) <= (show(X) -> Y) -> s(Y).
show([]) <= [].
show([X|Xs]) <= (show(X) ->Y),(show(Xs)->Ys) -> [Y|Ys].
show(E)#{E\=0,E\=s(_),E\=[],E\=[_|_]} <= (E -> V) -> show(V).
```

As another example we define a function `odd_double` defined only for odd numbers, this function uses the definition of plus from Sect.3. Note how the predicate `odd` is used to restrict the applicability of `odd_double` to odd numbers only.

```
odd_double(X) <= (X ->X1),
  odd(X1) -> plus(X1,X1).

odd(s(X)) <= even(X).

even(0).
even(s(X)) <= odd(X).
```

6 Generating Specialized Rule Definitions

In Sect. 3 we noticed that *FL* is deterministic, thus making search strategies more or less superfluous, the answers will be the same no matter in what order the inference rules are tried.

However, there are several reasons to use search strategies anyway. The most important concerns integration of functional logic programs into large systems using other programming paradigms as well; without a proper search strategy all the rules and strategies concerning the rest of the system will also be tested. Another reason is to enhance efficiency, the strategies we present in this section will almost always try the correct rule. Yet another reason is that the specialized rules we create allows us to write definitions without "evaluation clauses" for arguments thus making definitions shorter and more elegant.

In [3] we suggested the method *Local Strategies* as a way to write efficient rule definitions to a given definition. We also suggested that it should be possible to more or less automatically generate rule definitions according to this method to some programs, the rules we generate are applications of this method.

6.1 Specialized D-Rules

It is beyond the scope of this paper to describe the algorithms and heuristics used to generate rules, in short we try to decide which atoms are functions and which are predicates and then create specialized rules which as far as possible precompile the rule-sequences needed to use each particular function and predicate. Sometimes it is impossible to distinguish functions from predicates, in such cases an oracle (the user) is queried. It is also beyond the scope of this paper to describe the resulting rules and strategies in any greater detail, we simply give schematic descriptions of the D-Rules created for each function and predicate

6.1.1 Predicate Definitions and their D-right Rules

When we create specialized D-right rules to predicates we let them evaluate all arguments before we try to find a unifiable clause. This means that we may have functional expressions as arguments to predicates. It also means that the only allowed patterns in the heads of predicate definitions (and function definitions) are variables and canonical values. This corresponds closely to the approach taken in functional logic languages based on a constructor discipline [5]. Generally if P is a predicate of arity n its corresponding D-right rule becomes:

$$\frac{X_1 \vdash Y_1 \ldots X_n \vdash Y_n \quad \vdash B}{\vdash \; P(X_1, \ldots, X_n)} \quad B \in D(P(Y_1, \ldots, Y_n))$$

If we use strict functions in the arguments of predicates this approach works well enough, but if we combine lazy functions and predicates the patterns allowed in the heads of predicate definitions must be further restricted to shallow patterns with no repeated variables.

6.1.2 Function Definitions and their D-left Rules

The usual meaning of a strict function definition is that its arguments are evaluated before the function is called. It is therefore perfectly reasonable to associate with each strict function a rule which evaluates each of the arguments and then tests to see if the resulting atom is defined. If F is a function of arity n the rule becomes:

$$\frac{X_1 \vdash Y_1 \ldots X_n \vdash Y_n \quad Dp \vdash C}{F(X_1, \ldots, X_n) \; \vdash \; C} \quad Dp = D(F(Y_1, \ldots, Y_n))$$

In lazy functions we only evaluate arguments which have a non-variable pattern, if we do not want to risk evaluating to many arguments it is important that the function definitions are uniform, the patterns must also be linear, that is no variable may occur more than once in the head of a clause.

If we use lazy functions the rule-generator also generates a strategy which makes it possible to evaluate a functional condition to a fully evaluated canonical value which makes show-functions like the one given in Sect. 5 superfluous. It is up to the user to specify if a function is strict or lazy.

6.2 Examples

We now give some examples to show how functional logic programs may be written if we create specialized rules to each program.

6.2.1 Addition Revisited

The strict definition of addition given in Sect. 3 may now be reformulated as:

```
0 <= 0.
s(X) <= s(X).
succ(X) <= s(X).

plus(0,N) <= N.
plus(s(M),N) <= succ(plus(M,N)).
```

The definition of succ may look a bit strange but remember that succ has a corresponding specialized *D-left* rule which evaluates the argument.

6.2.2 Mixing Functions and Predicates

Let append and mem have the definitions:

```
[] <= [].
[X|Xs] <= [X|Xs].
cons(X,Xs) <= [X|Xs].

append([],Ys) <= Ys.
append([X|Xs],Ys) <= cons(X,append(Xs,Ys)).

mem(X,[X|_]).
mem(X,[_|Ys]) <= mem(X,Ys).
```

and let plus be defined as above, we may then ask queries like:

```
\- mem(plus(s(0),X),append(cons(0,[s(0)]),[s(s(0))])).
X = 0 ? ;
X = s(0) ? ;
```

6.2.3 Computing Prime Numbers

As our last example we show a definition that can be used to compute prime numbers using the well-known sieve of Erastothenes. To implement this algorithm we use a combination of lazy functions and predicates, and also the possibility to regard numbers as canonical objects and use the ordinary arithmetic operations. We also use a special condition constructor if (that has a corresponding inference rule) which chooses one of two functional conditions depending on whether a predicate is provable or not.

```
[] <= [].
[X|Xs] <= [X|Xs].

primes <= sieve(from(2).

sieve([P|Ps]) <= [P|sieve(filter(P,Ps))].
```

```
filter(_,[]) <= [].
filter(N,[X|Xs]) <= if(p(N,X),
                       filter(N,Xs),
                       [X|filter(N,Xs)]).

p(A,B) <= (B mod A) =:= 0.

from(M) <= (M -> N) -> [N|from(N+1)].
```

We also need something to force evaluation. We can not use a show strategy since the result is infinite. The solution is to define a predicate to print the elements of the infinite list of primes

```
print_list(E) <= (E -> [X|Xs]),put(X),print_list(Xs).
```

where `put` is defined elsewhere. Now the query

```
\- print_list(primes).
```

will print `2 3 5 7 11` and so on until we run out of memory.

References

[1] M. Aronsson, *GCLA, The Design Use and Implementation of a Program Development System*, Ph D thesis, Department of Computer Sciences, University of Stockholm, 1993.

[2] H. Ait-Kaci, A. Podelski, Towards a Meaning of Life, *Journal of Logic Programming*, vol.16 pp 195-234, 1993.

[3] G. Falkman, O. Torgersson, Programming Methodologies in GCLA, in, *Extensions of Logic Programming*, Springer LNCS 798, 1994.

[4] L. Hallnäs, Partial Inductive Definitions, *Theoretical Computer Science*, vol 87,pp 115-142,1991.

[5] M. Hanus, The Integration of Functions into Logic Programming; From Theory to Practice, *Journal of Logic Programming*, vol 19/20, pp 583-628,1994.

[6] P. Kreuger, GCLA II, A Definitional Approach to Control, in, *Extensions of Logic Programming*, Springer LNCS 596, 1992.

[7] H. Kuchen, J,J Moreno-Navarro, M. Rodriguez-Artalejo. Implementing a Lazy Functional Logic Language with Disequality Constraints, in *Proc. Proc. of the 1992 Joint International Conference and Symposium on Logic Programming.* MIT Press,1992.