A Definitional Approach to Functional Logic Programming

Olof Torgersson*

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

Abstract. We describe a definitional approach to the combination of functional and logic programming based on the theory of Partial Inductive Definitions. The described method produces programs directly executable in the definitional programming language GCLA. We show both a basic calculus for functional logic program definitions and discuss a refined version where the rules definitional resolution, definitional reflection, and definitional axiom are altered to be better suited for functional evaluation and equation solving.

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, [21, 26], to more integrated approaches such as narrowing languages [9, 13, 20, 24] based on Horn clause logic with equality [23], some kind of higher order logic as in Escher [18], and constraint logic programming as in Life [1].

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.

An alternative approach is to study the definitions *themselves*. In *definitional programming*, programs are understood as definitions, more precisely as *Partial Inductive Definitions* (PID) [10]. The theory of partial inductive definitions is general enough to naturally express both (Horn clause) logic programs and (first order) functional programs. Accordingly it provides a natural theoretical basis for combing functional and logic programming within the same computational

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

framework. In this paper we will describe how this integration of functions and predicates can be carried out. The work presented is inspired by some earlier work [4, 5, 25] on programming in the definitional programming language GCLA and has a double purpose: (i) to demonstrate how functional logic programming comes naturally in GCLA, (ii) to give ideas for how a specialized functional logic language on a definitional basis could be designed. All example programs are given in GCLA but we would like to stress the general nature of the material. Before we proceed we give a small example program.

Example 1 (Mixing Functions and Predicates). Consider the following definition defining addition and comparision of natural numbers in successor arithmetics:

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

0 + N <= N.
s(M) + N <= s(M + N).

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

Here we have defined `0` and `s` as *canonical objects* by giving them circular definitions. Addition is defined as a function and '`=<`' as a predicate. To use this definition we also need four inference rules, of which two are specializations of the rules ($\vdash D$) and ($D \vdash$), also known as the rule of *definitional resolution* and *definitional reflection* [10, 12, 16], and the third is the *definitional axiom* introduced in [15]:

$$\frac{\mathtt{M} \vdash \mathtt{M}_1 \quad D(\mathtt{M}_1 + \mathtt{N}) \vdash \mathtt{C}}{\mathtt{M} + \mathtt{N} \vdash \mathtt{C}}\ (D_+ \vdash) \qquad\qquad \frac{}{\mathtt{a} \vdash \mathtt{a}}\ D\text{-}ax \quad D(a) = a \ .$$

$$\frac{\mathtt{X} \vdash \mathtt{X}_1 \quad \mathtt{Y} \vdash \mathtt{Y}_1 \quad \vdash \mathtt{C}}{\vdash \mathtt{X} \leq \mathtt{Y}}\ (\vdash D_{\leq}) \quad C \in D(\mathtt{X}_1 \leq \mathtt{Y}_1) \ . \qquad \frac{}{\vdash \mathit{true}}\ \mathit{truth}$$

Here, $D(a)$, the *defininens* of the atom a, gives all conditions defining a. A sequent $F \vdash C$ is best understood as evaluate F to C. A more complete formulation of definitional rules involving variables and substitions is given in Sect. 3. We are now able to find `Z` and `W` such that `Z + W =< 0` as follows:

$$\cfrac{\cfrac{\cfrac{\{\mathtt{W} = \mathtt{X}_1\}}{\mathtt{W} \vdash \mathtt{X}_1}\ D\text{-}ax}{\mathtt{Z} + \mathtt{W} \vdash \mathtt{X}_1}\ D_+ \vdash \ \{\mathtt{Z} = \mathtt{0}\} \quad \cfrac{\{\mathtt{Y}_1 = \mathtt{0}\}}{\mathtt{0} \vdash \mathtt{Y}_1}\ D\text{-}ax \quad \cfrac{}{\mathtt{true}}\ \mathit{truth}}{\vdash \mathtt{Z} + \mathtt{W} \leq \mathtt{0}}\ (\vdash D_{\leq})\ \{\mathtt{X}_1 = \mathtt{0}\}$$

As an optimization we have omitted the leftmost premise in the rule ($D_+ \vdash$) since it is not needed in this example.

The rest of this paper is organized as follows. Section 2 gives some background on definitional programming. A basic calculus is presented in Sect. 3. This calculus shows the basic ideas of definitional functional logic programming. Section 4 shows the structure of programs. In Sect. 5, a refined calculus is given and, finally, Sect. 6 contains some conclusions and directions for future research.

2 Preliminairies

We recall some basic notions of PID and the programming language GCLA[2]. More details of GCLA, its underlying theory and general programming methodology, can be found in [5, 6, 8, 10, 11, 12, 14, 16].

A program in GCLA consists of two (finitary) partial inductive definitions which we refer to as the *definition* or the *object-level definition* and the *rule definition* respectively. We will sometimes refer to the definition as **D** and to the rule definition as **R**. Since both **D** and **R** are understood as definitions we speak of *programming with definitions*, or *definitional programming*.

GCLA shares several features with most logic programming languages, like logical variables. Computations are performed by posing queries to the system and the variable bindings produced are regarded as the answer to the query. Search for proofs are performed depth-first with backtracking and clauses of programs are tried by their textual order. Some familiarity with basic logic programming concepts [17] and rudimentary knowledge of sequent calculus is assumed.

2.1 Basic Notions

Atoms, Terms, Constants, and Variables. We start with an infinite signature, Σ, of *term constructors*, and a denumerable set, $\mathcal{V}$, of *variables*. We write variables starting with a capital letter. Each term constructor is distinguished by its *name* and *arity*. The term constructor t of arity n is written t/n. We omit the arity when there is no risk of ambiguity. A *constant* is a term constructor of arity 0. *Terms* are built up according to the following:

1. all variables and constants are terms
2. if f is a term constructor of arity n and $t_1, \ldots, t_n$ are terms then $f(t_1, \ldots, t_n)$ is a term.

An *atom* is a term which is not a variable.

Conditions. *Conditions* are built from terms and *condition constructors*. The set $\mathcal{CC}$ of condition constructors always include *true* and *false*. We then have:

1. *true* and *false* are conditions
2. all terms are conditions

[2] To be pronounced "Gisela"

3. if $p \in \mathcal{CC}$ is a condition constructor of arity n, and $C_1, \ldots, C_n$ are conditions then $p(C_1, \ldots, C_n)$ is a condition. Condition constructors can be declared to appear in infix position like in $C_1 \rightarrow C_2$.

In **R** the set of condition constructors is predefined, while in **D** any symbol can be declared to become a condition constructor.

Clauses. If a is an atom and C a condition then

$$a \Leftarrow C.$$

is a *definitional clause*, or a clause for short. We refer to a as the *head* and to C as the *body* of the clause. The clause a. is short for $a \Leftarrow true$. The clause $a \Leftarrow false$. is equivalent to not defining a at all. A *guarded definitional clause* has the form

$$a\#\{G_1, \ldots, G_n\} \Leftarrow C.$$

where a is an atom, C a condition, and each G_i is a *guard*. If t_1 and t_2 are terms then $t_1 \neq t_2$ and $t_1 = t_2$ are guards. Guards are used to restrict variables occurring in the heads of guarded definitional clauses.

Definitions. A definition is a finite sequence of (guarded) definitional clauses:

$$\begin{array}{l} a_1 \Leftarrow C_1. \\ \vdots \\ a_n \Leftarrow C_n. \end{array}$$

Note that both **D** and **R** are definitions in the sense described here.

Operations on Definitions. The *domain*, $Dom(D)$, of a definition D, is the set of all atoms defined in D, that is, $Dom(D) = \{a\sigma \mid \exists A(a \Leftarrow A \in D)\}$. The *definiens*, $D(a)$, of an atom a is the set of all bodies of clauses in D whose heads matches a, that is $\{A\sigma \mid (b \Leftarrow A) \in D, b\sigma = a\}$. If there are several bodies defining a then they are separated by ';'. A closely related notion is that of *a-sufficiency.* Given an atom a a substitution σ is called *a-sufficient* if $D(a\sigma)$ is closed under further substitution, that is, for all substitutions τ $D(a\sigma\tau) = (D(a\sigma))\tau$. If $a \notin Dom(D)$ then $D(a) = false$. For more details see [7, 12, 14, 16].

Sequents and Queries. A *sequent* is as usual $\Gamma \vdash C$, where, in GCLA, Γ is a (possibly empty) list of *assumptions* and C is the *conclusion* of the sequent. A *query* has the form

$$S \Vdash (\Gamma \vdash C). \qquad (1)$$

where S is a *proofterm*, that is some more or less instantiated condition in **R**. The intended interpretation of (1) is that we ask for an object-level substitution σ such that $S\phi \Vdash (\Gamma\sigma \vdash C\sigma)$. holds for some meta-level substitution ϕ.

2.2 The Definition

In **D** the programmer should state the declarative content of a problem. A definition **D** has no procedural interpretation without its associated procedural part **R**. **R** supplies the necessary information to get a program fulfilling the intent behind **D**. The programmer can use the predefined set of condition constructors, or replace or mix them with new ones.

The default set of condition constructors include '**,**', '**;**', and '$\rightarrow$', with an interpretation given by the standard rule definition. This rule definition implements the calculus $\mathcal{OLD}$ [14], which in turn is a variant of $\mathcal{LD}$ given in [12].

2.3 The Rule Definition

The rule definition consists of *inference rules*, *search strategies*, and *provisos*, which together form a procedural interpretation of the definition. The rule definition implements a sequent calculus giving meaning to the condition constructors in **D**. The set of condition constructors available in **R** is fixed to ',', ';', $\rightarrow$, *true*, and *false*. The interpretation of the condition constructors in **R** is given by a fixed calculus, $\mathcal{DOLD}$ [14].

Also available in the rule definition are a number of primitives to handle the communication between **R** and **D**. Some of these are described in below.

Inference Rules. The interpretation of conditions in **D** is given by inference rules in **R**. Inference rules (or rules for short) are coded as functions from the proofs of the premises of a rule to its conclusion. Generally, the form of an inference rule is

$$\begin{aligned} r(PT_1, \ldots, PT_n) \Leftarrow\ & P_1, \ldots, P_k, \\ & (PT_1 \rightarrow Seq_1), \\ & \ldots, \\ & (PT_n \rightarrow Seq_n) \\ & \rightarrow Seq. \end{aligned}$$

where

- $PT_1, \ldots, PT_n$ are proof terms, that is, more or less instantiated functional expressions representing the *proofs* of the premises, Seq_i
- $P_1, \ldots, P_k$ for $k \geq 0$ are provisos, that is, side conditions on the applicability of the rule
- Seq and Seq_i are sequents, $\Gamma \vdash C$, where Γ is a list of (object level) conditions and C is a condition.

We read this as "If P_1 to P_k hold and each PT_i proves Seq_i then $r(PT_1, \ldots, PT_n)$ proves Seq." In actual proof search derivations are constructed bottom-up, so the functions representing rules are evaluated backwards [5, 14].

Search Strategies. Search strategies are used to combine rules together guiding search. The basic building blocks of strategies are rules and provisos. Combining rules, stragies, and provisos together we can build more and more complex structures. The general form of a strategy is

$$\begin{aligned} Strat &\Leftarrow P_1 \to Seq_1, \\ &\quad \ldots, \qquad\qquad n \geq 0 \\ &\quad P_n \to Seq_n. \\ Strat &\Leftarrow PT_1, \ldots, PT_m. \end{aligned}$$

where P_i are provisos, PT_i proof terms, and Seq_i sequents. We read this as "If P_i holds, $i \leq n$, and some PT_j, $1 \leq j \leq m$, proves Seq_i then $Strat$ proves Seq_i."

Provisos. A *proviso* is a side condition on the applicability of a rule or strategy. Provisos can be predefined or user defined. User defined provisos are described in [5]. Among the predefined provisos there are really three provisos handling the communication between **R** and **D** and various provisos implementing different kinds of simple tests like `var`, `atom`, `number` etc.

The provisos handling the communication between **R** and **D** are:

- $definiens(a, Dp, n)$ which holds if $D(a\sigma) = Dp$, where σ is an a-sufficient substitution and n the number of clauses defining a. If $n > 1$ then the different clauses defining a are separated by ';'.
- $clause(b, B)$ which holds if $c \Leftarrow C \in D$, $\sigma = mgu(b, c)$, and $B = C\sigma$.
- $unify(t, c)$ which unifies the two object level terms t and c.

Example 2 (Example 1 continued). The inference rules $(D_+ \vdash)$, D-ax, and $truth$ used in Ex. 1 can be coded in GCLA as

```
'+_d_left'(PT) <= (PT -> ([X] \- X1)),
                  definiens(X1+Y,Dp,N),
                  (PT -> ([Dp] \- C))
                  -> ([X+Y] \- C).


d_ax <= circular(T),        '=<_d_right'(PT) <= (PT -> ([X] \- X1)),
        unify(T,C)                              (PT -> ([Y] \- Y1)),
        -> ([T] \- C).                          clause(X1=<Y1,B),
                                                (PT -> ([] \- B))
truth <= ([] \- true).                          -> ([] \- X=<Y).
```

where `circular` is a special proviso that ensures that `T` can only be bound to canonical objects (atoms with circular definitions). One suitable strategy for the definition in Ex. 1 is

```
s <= '+_d_left'(s), '=<_d_right'(s), d_ax, truth.
```

that simply tells us to try the rules in the given order. The query shown in Ex. 1 becomes

```
s \\- \- Z + W =< 0.
```

giving the only answer `Z = 0, W = 0`.

3 A Calculus for Functional Logic Programming

We describe a basic calculus for definitional functional logic programs. In GCLA this calculus is implemented as a rule definition. We call the calculus *FL* for functional logic. *FL* illustrates the basic ideas of the definitional approach to functional logic programming. An important property of *FL* is that it is *deterministic* in the sense that there is at most one inference rule that apply to each given sequent. The only source of non-determinism are the definitional rules where it is possible that several clauses in **D** can be used. *FL* is very similar to $\mathcal{DOLD}$[14] used to interpret **R** which is not very surprising since the rule definition really is a kind of functional logic program.

3.1 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.

Rules Relating Atoms to a Definition.

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

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

$$\frac{}{a \vdash c \quad \sigma\tau}\ D\text{-}ax \quad \sigma \text{ is an } a\text{-suff. substitution}, a\sigma = D(a\sigma), \tau = mgu(a\sigma, c\sigma)\ .$$

Note that *D-ax* may only be applied to atoms with circular definitions. For a in-depth description and motivation of these rules, in particular *D-ax*, see [15, 16].

Rules for Constructed Conditions. The rules for constructed conditions are essentially the standard GCLA and PID rules [10, 14] restricted to allow at most one element in the antecedent. Note also that *falsity* can only be applied if both the antecedent and the consequent are false.

$$\frac{}{\vdash true}\ truth \qquad\qquad \frac{}{false \vdash false}\ falsity$$

$$\frac{A \vdash B}{\vdash A \to B}\ (\vdash\to) \qquad\qquad \frac{\vdash A \quad B \vdash C}{A \to B \vdash C}\ (\to\vdash)$$

$$\frac{\vdash C_1 \quad \vdash C_2}{\vdash (C_1, C_2)}\ (\vdash,) \qquad\qquad \frac{C_i \vdash C}{(C_1, C_2) \vdash C}\ (,\vdash) \quad i \in \{1,2\}$$

$$\frac{\vdash C_i}{\vdash (C_1; C_2)}\ (\vdash;) \quad i \in \{1,2\} \qquad\qquad \frac{C_1 \vdash C \quad C_2 \vdash C}{(C_1; C_2) \vdash C}\ (;\vdash)$$

$$\frac{C \vdash false}{\vdash not(C)}\ (\vdash not) \qquad\qquad \frac{\vdash A}{not(A) \vdash false}\ (not \vdash)$$

$$\frac{A \vdash C}{(pi\ X \backslash A) \vdash C}\ (pi \vdash) \qquad\qquad \frac{\vdash C}{\vdash si\ X \backslash C}\ (\vdash si)$$

3.2 A Sample Program

The basic principle when we combine functions and predicates is that we place functions to be evaluated in the antecedent and get the result in the consequent, while a predicates are proved to the right using rules operating on the consequent only. Functions and predicates are glued together by the rules ($\vdash\rightarrow$) and ($\rightarrow\vdash$).

Example 3 (Computing the size of a list). Let the size of a list be the number of distinct elements in the list. We express this in **D** in the following way:

```
size([]) <= 0.
size([X|Xs]) <= pi Y \ if(mem(X,Xs),
                           size(Xs),
                          ((size(Xs) -> Y) -> s(Y))).
```

with the *intended* reading "the size of the empty list is 0, and the size of `[X|Xs]` is `size(Xs)` *if* `X` is a member of `Xs`, else evaluate `size(X)` to `Y` and take as result of the computation the successor of `Y`." The definition of `if` becomes

```
if(Pred,Then,Else) <= (Pred -> Then),(not(Pred) -> Else).
```

Note that `Pred` represents a predicate, while `Then` and `Else` are functional expressions. The arrow works as a "switch" between functions and predicates. To complete the program we need the circular definitions of `0` and `s` given in Ex. 1 plus the definition of the predicate `mem`:

```
mem(X,[X|_]).
mem(X,[Y|Xs])#{X \= Y} <= mem(X,Xs).
```

Now, using *FL* we can handle queries like

```
fl \\-  (size([0,X,s(0)]) \- L).
```

which first gives `L = s(s(0))`, `X = 0`, then `L = s(s(0))`, `X = s(0)`, and finally `L = s(s(s(0)))`, `0 \= X`, `X \= s(0)`.

4 Definitional Functional Logic Programs

We give a brief description of how **D** is interpreted by *FL*. Keep in mind that a predicate is proved by placing it to the right of the turnstyle, $\vdash P$, while a function is evaluated to the left with the result given in the consequent, $F \vdash C$. All irreducable terms, the *canonical objects*, are given circular definitions. As can be seen from *FL* this is what controls application of the rules ($D \vdash$) and *D-ax*. We cannot simply regard undefined atoms as canonical objects since if $a \notin Dom(D)$ then $D(a) = false$. Generally the definition of the canonical object s/n is

$$s(X_1, \ldots, X_n) \Leftarrow s(X_1, \ldots, X_n).$$

where each X_i is a variable. Note the 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).

4.1 Defining Predicates

Predicate definitions are very similar to pure Prolog with two extensions: use of functions in conditions defining predicates and constructive negation.

A predicate definition defining the predicate p consists of a number of definitional clauses:

$$\begin{array}{ll} p(t_1,\ldots,t_n) \Leftarrow C_1. & \\ \vdots & n \geq 0, m > 0 \\ p(t_1,\ldots,t_n) \Leftarrow C_m. & \end{array}$$

where each C_i is a *predicate condition*. We say that a condition C is a predicate condition if:

- C is an atom.
- $C = \mathit{true}$ or $C = \mathit{false}$.
- $C = (C_1, C_2)$, 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".
- $C = C_1 \rightarrow C_2$, 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".
- $C = \mathit{not}(C_1)$, where C_1 is a predicate condition. We read this as "C holds if C_1 can be proven false".
- $C = \mathit{si}\ X\backslash C$, that is, existensial quantification of X in C.

We may omit *si* understanding all variables not occurring in the head as existensially quantified.

4.2 Defining Functions

A definition defining a function f consists of a number of definitional clauses:

$$\begin{array}{ll} f(t_1,\ldots,t_n) \Leftarrow C_1. & \\ \vdots & n \geq 0, m > 0 \\ f(t_1,\ldots,t_n) \Leftarrow C_m. & \end{array}$$

where each C_i is a *functional condition*. 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”.
- $C = (C_1 \to C_2)$, 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”.
- $C = pi\ X \backslash C$, that is, universal quantification of the variable X in C.

We may omit *pi* understanding all variables not occurring in the head as universally quantified. 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 $(D\vdash)$ collects all clauses defining an atom.

Example 4 (Overlapping Clauses). Overlapping clauses may be used as usual in predicates but only in special cases in functions. Consider the following definition of the function **max** (we assume appropriate definitions of >, =<, and canonical objects):

```
max(X,Y) <= X > Y -> X.
max(X,Y) <= X =< Y -> Y.
```

An attempt to compute `max(s(0),0)` will fail as can be seen below:

$$\dfrac{\dfrac{\dfrac{\dfrac{\vdots}{\vdash \mathtt{s(0)} > \mathtt{0}}\,(\vdash D) \quad \dfrac{\{\mathtt{C} = \mathtt{s(0)}\}}{\mathtt{s(0)} \vdash \mathtt{C}}\,D\text{-}ax}{\mathtt{s(0)} > \mathtt{0} \to \mathtt{s(0)} \vdash \mathtt{C}}\,(\to\vdash) \quad \dfrac{\dfrac{fails}{\vdash \mathtt{s(0)} \leq \mathtt{0} \to \mathtt{0}}\,(\vdash D) \dfrac{fails}{\mathtt{0} \vdash \mathtt{s(0)}}}{\mathtt{s(0)} \leq \mathtt{0} \to \mathtt{0} \vdash \mathtt{C}}\,(\to\vdash)}{(\mathtt{s(0)} > \mathtt{0} \to \mathtt{s(0)}); (\mathtt{s(0)} \leq \mathtt{0} \to \mathtt{0}) \vdash \mathtt{C}}\,(;\vdash)}{\mathtt{max(s(0),0)} \vdash \mathtt{C}}\,(D\vdash)$$

To compute **max** we instead write the definition

```
max(X,Y) <= (X > Y -> X), (X =< Y -> Y).
```

where we have a *choice* between two conditions in the body. We leave it to the reader to work out the details of how **max** can be computed using this definition. The definition of **max** reflects a conceptual difference between our approach and narrowing languages. It can be argued that while we define the *function* **max** a narrowing language allowing the version with overlapping clauses really define a conditional term rewriting system or the equality predicate. Alternatively, **max** can of course be defined and executed as a predicate using overlapping clauses.

Example 5 (Lazy Evaluation). *FL* is well-suited for lazy evaluation since evaluation always stops when a canonical object is reached. A simple definition illustrates this:

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

```
from(X) <= [X|from(s(X))].

drop(0,Xs) <= Xs.
drop(s(N),L) <= (L -> [_|Xs]) -> drop(N,Xs).
```

The function `drop` removes the first n elements from its second argument. For instance

```
drop(s(s(0)),from(0)) \- L.
```

binds `L` to `[s(s(0))|from(s(s(s(0))))]`.

4.3 Queries

There are two general forms of queries: $\vdash P$ proves the predicate condition P and $F \vdash C$ evaluates the functional condition F. Note that F may be much more complex than a simple function call.

Example 6 (Complex Queries). Assume that the definition of `mem` in Ex. 3 is added to Ex. 5. We may then ask things like "let `L` be the list `[0,s(0),s(s(0))]`, if `X` and `Y` are members of `L`, what is then the value of `drop(X,from(Y))`":

```
([0,s(0),s(s(0))] -> L),mem(X,L),mem(Y,L) -> drop(X,from(Y)) \- C.
```

5 Specialized Definitional Rules

The definitional rules presented in Sect. 3 are sufficient to handle flat programs. However, we could not use the definition of `size` given in Ex. 3 to evaluate the atom `size(append([0],[s(0)]))` since there is no clause defining it. One alternative could of course be to flatten both **D** and all queries, but we will not go into that here. Instead, we will show how to refine **R**. We think that there are at least two good motivations why nested functional applications should be allowed in **D** and the information of how to handle them be kept in **R**, namely:

1. Nesting functional expressions is an essential feature of a functional programming style. Consequently it is desirable to describe a computational mechanism that can handle them.
2. In a realization in GCLA it is well in line with the basic programming methodology to have such procedural details as when to evaluate arguments as part of **R** in favor of transforming **D**.

Arguments to functions and predicates can be evaluated by creating specialized definitional rules – one to each function and predicate. These definitional rules are refinements of the rules $(D \vdash)$ and $(\vdash D)$ given in Sect. 3.

In the sequel we only consider lazy evaluation and assume that all patterns in heads of clauses are *shallow*, that is, if $f(t_1, \ldots, t_n)$ is the head of a clause then each t_i is either a variable, or a canonical object $c(X_1, \ldots, X_n)$, where each X_i is a variable. We also assume that no variable occurs more than once in any head.

5.1 Specialized Definitional Rules for Functions

As default, the arguments that have a non-variable pattern in some clause defining a function f are evaluated before the definiens operation is applied. If $t_1, \ldots, t_n$ denotes the arguments having a non-variable pattern and $s_1, \ldots, s_n$ denotes the variable patterns we get the rule

$$\frac{t_1 \vdash u_1 \ldots t_n \vdash u_n \quad D(f(u_1, \ldots, u_n, s_1, \ldots, s_m)) \vdash C}{f(t_1, \ldots, t_n, s_1, \ldots, s_m) \vdash C} \; (D_f \vdash) \; .$$

If the function definition is *uniform* then this approach will only evaluate *needed* arguments. For an example of this kind of rule see $(D_+ \vdash)$ in Ex. 1.

5.2 Specialized Definitional Rules for Predicates

We allow functional expressions as arguments to predicates and use the same default rules for what arguments to evaluate as for functions. It is possible for the programmer to override the default (both for functions and predicates) and decide exactly what arguments to evaluate. Generally, if p is a predicate and $t_1, \ldots, t_n$ denotes the arguments having a non-variable pattern and $s_1, \ldots, s_n$ denotes the variable patterns we get the rule

$$\frac{t_1 \vdash u_1 \ldots t_n \vdash u_n \quad \vdash B}{\vdash p(t_1, \ldots, t_n, s_1, \ldots, s_m)} \; (\vdash D_p) \quad B \in D(p(u_1, \ldots, u_n, s_1, \ldots, s_m)) \; .$$

5.3 Refined Axioms

If we had used strict evaluation (strict evaluation is discussed in [25]) then the definitional axiom *D-ax* would have been sufficient. Also, if we discard equation solving completely there is no need for anything more sophisticated. However the combination of lazy evaluation and partially instantiated terms occurring to the right in sequents gives a need for a new even more restricted axiom and a special *decomposition* rule. A very simple example will do to illustrate this:

Example 7 (Simple Equation Solving). Consider the definition of addition given in Ex. 1 and the sequent

```
X + s(0) \- s(s(0)).
```

The first (and only) rule to try on this sequent is $(D_+ \vdash)$:

$$\frac{\dfrac{\{\mathtt{X} = \mathtt{X}_1\}}{\mathtt{X} \vdash \mathtt{X}_1} \, D\text{-}ax \qquad \dfrac{fails}{\mathtt{s}(\mathtt{Y} + \mathtt{s}(0)) \vdash \mathtt{s}(\mathtt{s}(0))}}{\mathtt{X} + \mathtt{s}(0) \vdash \mathtt{s}(\mathtt{s}(0))} \; (D_+ \vdash) \quad \{\mathtt{X}_1 = \mathtt{s}(\mathtt{Y})\}$$

We can get no further since we have no mechanism for looking inside canonical objects, comparing subparts.

Generally, there are five possible cases to handle a sequent $A \vdash C$ that each should have a separete treatment instead of using the rule *D-ax*, namely:

1. Both A and C are variables. In this case the variables should be unified but restricted so that they may only be bound to canonical objects.
2. A is a canonical object and C a variable – unify the two sides.
3. A is a variable and C a canonical object – unify the two sides.
4. Both A and C are canonical objects having the same term constructor s/n. In this case the subparts of A and C must be checked against each other.
5. A is a variable or a canonical object and C a nonvariable functional condition but not a canonical object. In this case we have to evaluate C to C_1 and then continue to prove $A \vdash C_1$.

Expressed as inference rules we get:

$$\frac{}{X \vdash Y \quad \{X = Y\}}\ V\text{-}ax \quad X \text{ may only be bound to canonical objects} \ .$$

$$\frac{}{a \vdash X \ \ \sigma}\ D_V\text{-}ax \qquad \frac{}{X \vdash a \ \ \sigma}\ D_V\text{-}ax \quad D(a) = a, \sigma = mgu(a, X) \ .$$

$$\frac{C \vdash C_1 \quad A \vdash C_1}{A \vdash C}\ (\vdash eval) \quad A \in \mathcal{V} \text{ or } D(A) = A,\ C \notin \mathcal{V},\ D(C) \neq C \ .$$

$$\frac{t_1 \vdash s_1 \ldots t_n \vdash s_n}{c(t_1, \ldots, t_n) \vdash c(s_1, \ldots, s_n)}\ decompose \quad c/n \text{ is a canonical object} \ .$$

The simplified test for circularity (without a-sufficient substitutions) in these rules compared to the one in the rule *D-ax* in Sect. 3.1 is motivated by the simple structure of definitions of canonical objects. We could use the same condition as in Sect. 3.1 but it is not needed. Using these rules conditions in equations $A \vdash C$ will be evaluated just as much as is needed. The formulation of these rules have some similarities with the rules of the narrowing calculus `LNC` [22].

Example 8 (The Last Element of a List). A simple equation solving program is the following, using append to compute the last element of a list:

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

last(Xs) <= (append(_,[E]) -> Xs) -> E.
```

To this program we create a specialized rule ($D_{\text{append}} \vdash$) that evaluates the first argument of `append` as described above. A sample query is

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

giving the expected answer `E = s(0)` once.

Example 9 (Demanded Evaluation). Our refined axiom rules only forces evaluation as much as is needed to give an answer. To compute the first and third natural number we gan use the definition of `from` in Ex. 5 and ask the query

```
from(0) \- [X,_,Y|_].
```

which computes the answer `X = 0, Y = s(s(0))`.

6 Concluding Remarks

We have implemented a rule generator that makes it easy to write programs in line with the ideas described in Sect. 5. This rule generator takes a definition and perform some analysis to tell function definitions and predicate definitions apart. It then creates specialized definitional rules to each function and predicate and also precompile the sequence of rules for constructed conditions as far as possible, into efficient search strategies. It is possible for the programmer to override the default evaluation schemes described in Sects. 5.1 and 5.2 and decide exactly what arguments to evaluate.

Some problems remain though, most notably the restriction to shallow patterns due to lazy evaluation. Another is that, at times, too many arguments to functions and predicates are evaluated. As far as the evaluation of arguments is concerned, it is well in line with programming methodology in GCLA to let the programmer decide this explicitly. The restriction to shallow patterns is more difficult. We either need to apply some transformation to **D** or change the definiens operation. Of course, both these problems would have to be adressed if one decided to create a definitional functional logic programming language.

A lot of work has been done in the narrowing setting to transform and analyse certain classes of rewrite systems (function definitions). Several of these, for instance [3, 19] use the notion of a *definitional tree* [2] as a basis for program transformation and finding narrowing strategies. One possibility could be to use something similiar to transform **D**. However, the choice is not obvious since it is possible that some sophisticated rule definition would do.

References

1. H. Aït-Kaci and A. Podelski. Towards a meaning of life. *Journal of Logic Programming*, 16:195–234, 1993.
2. S. Antoy. Definitional trees. In *Int. Conf. on Algebraic and Logic Programming ALP'92*, Springer LNCS 632, pages 143–157. Springer-Verlag, 1992.
3. S. Antoy, R. Echahed, and M. Hanus. A needed narrowing strategy. In *Proc. 21st ACM Symposium on Principles of Programming Languages*, pages 268–279, 1994.
4. M. Aronsson. A definitional approach to the combination of functional and relational programming. Research Report SICS T91:10, Swedish Institute of Computer Science, 1991.
5. M. Aronsson. Methodology and programming techniques in GCLAII. In *Proc. of Extensions of logic programming, ELP'91*, Springer LNAI 596, 1992.
6. M. Aronsson. *GCLA, The Design, Use, and Implementation of a Program Development System.* PhD thesis, Stockholm University, Stockholm, Sweden, 1993.
7. M. Aronsson. Implementational issues in GCLA: A-sufficiency and the definiens operation. In *Proc. of Extensions of logic programming, ELP'92*, Springer LNAI 660, 1993.
8. G. Falkman and O. Torgersson. Programming methodologies in GCLA. In *Extensions of logic programming, ELP'93*, pages 120–151, Springer LNAI 798, 1994.
9. E. Giovannetti, G. Levi, C. Moiso, and C. Palamidessi. Kernel-LEAF: A logic plus functional language. *Journal of Computer and System Sciences*, 42:139–185, 1991.

10. L. Hallnäs. Partial inductive definitions. *Theoretical Computer Science*, 87(1):115–142, 1991.
11. L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(2):261–283, 1990. Part 1: Clauses as Rules.
12. L. Hallnäs and P. Schroeder-Heister. A proof-theoretic approach to logic programming. *Journal of Logic and Computation*, 1(5):635–660, 1991. Part 2: Programs as Definitions.
13. M. Hanus. The integration of functions into logic programming; from theory to practice. *Journal of Logic Programming*, 19/20:593–628, 1994.
14. P. Kreuger. GCLA II: a definitional approach to control. In *Extensions of logic programming, ELP91*, Springer LNAI 596, 1992.
15. P. Kreuger. Axioms in definitional calculi. In *Extensions of logic programming, ELP93*, Springer LNAI 798, 1994.
16. P. Kreuger. *Computational Issues in Calculi of Partial Inductive Definitions*. PhD thesis, Department of Computing Science, University of Göteborg, Göteborg, Sweden, 1995.
17. J. Lloyd. *Foundations of Logic Programming*. Springer Verlag, second extended edition, 1987.
18. J. Lloyd. Combining functional and logic programming languages. In *Proceedings of the 1994 International Logic Programming Symposium, ILPS'94*, 1994.
19. R. Loogen, F. López-Fraguas, and M. Rodríguez-Artalejo. A demand driven computation strategy for lazy narrowing. In *Proc. of the 5th Int. Symposium on Programming Language Implementation and Logic Programming,PLIP'93*, pages 184–200, Springer LNCS 714, 1993.
20. J. J. Moreno-Navarro and M. Rodríguez-Artalejo. Logic programming with functions and predicates: The language BABEL. *Journal of Logic Programming*, 12:191–223, 1992.
21. L. Naish. Adding equations to NU-Prolog. In *Proc. of the 3rd Int. Symposium on Programming Language Implementation and Logic Programming*, pages 15–26, Springer LNCS 528, 1991.
22. S. Okui, A. Middeldorp, and T. Ida. Lazy narrowing: Strong completeness and eager variable elimination (extended abstract). In *TAPSOFT'95: Theory and Practice of Software Development*, pages 394–408. Springer LNCS 915, 1995.
23. P. Padawitz. *Computing in Horn Clause Theories*, volume 16 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, 1988.
24. U. S. Reddy. Narrowing as the operational semantics of functional languages. In *Proceedings of the IEEE International Symposium on Logic Programming*, pages 138–151. IEEE Computer Soc. Press, 1985.
25. O. Torgersson. Functional logic programming in GCLA. In *Proceedings of the 6th Nordic Workshop on Programming Theory*. Aarhus, 1994.
26. D. H. D. Warren. Higher-order extensions to prolog—are they needed? In D. Mitchie, editor, *Machine Intelligence 10*, pages 441–454. Edinburgh University Press, 1982.