Translating functional programs to GCLA *

Olof Torgersson

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

Abstract This paper presents an attempt to do lazy evaluation in GCLA by translating functional programs into GCLA definitions and evaluating these using a proper procedural part. The resulting GCLA programs are described with some detail to illuminate different aspects of GCLA programming.

1 Introduction

This paper presents a translation from functional programs into GCLA programs. This translation has its origins in a project where we planned to apply KBS -technology to build a programming environment for a functional language. The translation into GCLA programs was then intended to be the representation of the functional programs in a system to do debugging etc. The presentation of the resulting definitions, and the procedural part used to evaluate these is carried out with some detail to give some light upon GCLA programming.

The rest of this paper is organized as follows. In Sect 2 we give a brief introduction to GCLA. In Sect 3 we describe the process of translating functional programs into GCLA definitions. In Sect. 4 we give the inference-rules and search-strategies needed to give a procedural interpretation of the GCLA definitions produced. In Sect 5 finally, we discuss possible future work and connections to other attempts at doing lazy evaluation in logic programming.

2 Introduction to GCLA

The programming system *G*eneralized Horn *C*lause *La*nguage (GCLA) [3,4,5,6,16]is a logical programming language (specification tool) that is based on a generalization of Prolog. This generalization is unusual in that it takes a quite 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 to assume 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*)?” In GCLA one can also assume things in the

* 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).

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?”

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.

2.2 Example: Pure Prolog

The division of GCLA programs into a declarative part and a procedural part makes it possible to write programs in a number of styles. For example pure prolog programming is achieved by substituting ‘<=’ for ‘:-’ in Prolog programs. A proper procedural behavior is then given by the following rule definition

```
:- include_rules(lib('rules.rul')).         % use standard rules, see [4].

pureprolog <= pp(_,pureprolog).

pp(C,_) <= (_ \- C).                        % these two clauses are read
pp(C,PT) <= correct_rule(C,PT).             % conjunctively.

correct_rule(true,_) <= true_right.         % choose the proper predefined
correct_rule((_,_),PT) <= v_right(_,PT,PT).  % inference rule.
correct_rule((_;_),PT) <= o_right(_,_,PT).
correct_rule(C,PT)#{C \= true,C \= false,C \= (_,_),C \= (_;_)} <= d_right(C,PT).
```

As a simple example of a definition we take the definition of the append relation.

```
append([],L,L).
append([X|Xs],L1,[X|L2]) <= append(Xs,L1,L2).

| ?-  pureprolog \\- (\-   append([2],L1,L2),append(L3,L1,[8,2,34])).

L1 = [8,2,34], L2 = [2,8,2,34], L3 = [] ? ;

L1 = [2,34], L2 = [2,2,34], L3 = [8] ? ;

L1 = [34], L2 = [2,34], L3 = [8,2] ? ;

L1 = [], L2 = [2], L3 = [8,2,34] ? ;

no
```

3 Translating functional programs into GCLA

Although GCLA is perhaps best described as logical programming language it also incorporates a kind of functional programming. To evaluate a function we put the expression to be evaluated to the left of the turnstile '`\-` ' and get the result as a binding of some variable(s) occurring to the right. For example:

```
| ?- somestrat \\- plus(s(0),s(s(0))) \- X.
X = s(s(0)))
```

So, the problem to be solved is to translate functions written in a functional language into GCLA-functions and to construct a procedural part that evaluates these functions correctly.

3.1 The basic Idea

The basic idea behind the translation described in this paper is the similarity between a functional program containing only supercombinators and a functional GCLA-definition. According to [18] a supercombinator is defined as

> **Definition** A supercombinator, $S of arity n is a lambda expression of the form
> $\lambda x_1.\lambda x_2...\lambda x_n.E$
> where E is not a lambda abstraction, such that
> i) $S has no free variables.
> ii) any lambda abstraction in E is a supercombinator.
> iii) n >= 0, that is there need be no lambdas at all.

A supercombinator redex then consists of the application of a supercombinator of arity n to n arguments, and a supercombinator reduction replaces a supercombinator by an instance of its body. For example take the following supercombinator program adopted from [18]

```
$Y w y = + y w
$X x = $Y x x
$main = $X 4
```

The corresponding GCLA definition is

```
y(W,Y) <= +(Y,W).
x(X) <= y(X,X).
main <= x(4).
```

The supercombinator - program is then evaluated by performing supercombinator reductions, where the combinators are replaced by instances of their bodies until we reach the built-in function `+` and evaluate `4+4` to `8`. Similarly in GCLA if we have the query

```
somefunstrat \\- (main \- C)
```

we replace each head by its body by successive applications of the `d_left/3` [4,15] rule until we get the expression `+(4,4)` which is then evaluated to `8`, and we get a binding of `C` to `8`. We see that reductions correspond to applications of the `d_left/3` rule. All that we have to do then is to take a functional program and perform some of the usual transformations made by compilers until we get a supercombinator program which can then be turned into a GCLA definition that can be evaluated with some suitable set of inference rules and search strategies.

Unfortunately it is not quite as simple as that since we want lazy evaluation and supercombinator programs may contain both functions as arguments, and combinators waiting for more arguments. It is not obvious how to express those things in the first order language GCLA, but we try hard.....................

3.2 A Tiny Functional Language

Since our original aim was to apply KBS technology to build an environment for program development, we choose to work with a subset of an existing programming language. The language chosen was a very small subset of LML [9], called TML (Tiny ML) in this paper. TML is a subset of LML in the sense that any correct TML program also is a correct LML program.

The major limitation made in TML is that there are no type declarations in the language, that is, is it not possible to introduce any new types. The only possible types of objects in TML are those already known to the system, namely integers, booleans, lists and tuples. Although this is a serious limitation, we do not believe that the translation into GCLA presented in this paper would be seriously affected if we introduced the possibility to declare new types.

Some examples of TML programs are given below. As can be seen from the examples ordinary features such as pattern matching, anonymous patterns, lambda abstractions and higher order functions are included in TML.

```
let rec append nil ys = ys
   || append (x.xs) ys = x.(append xs ys)
   and foldr f u nil = u
   || foldr f u (x.xs) = f x (foldr f u xs)
   and fromto n m = if n = m+1 then nil
   else n.fromto (n+1) m
   and flat xss = foldr append nil xss
in flat [fromto 1 5;fromto 2 6 ;fromto 3 7]
```

```
let rec filter p nil = nil
   || filter p (x.xs) = if (p x) then x.(filter p xs)
   else filter p xs
   and from n = n.from (n+1)
   and take 0 _ = nil
   || take n (x.xs) = x.take (n-1) xs
   and sieve (p.ps) = p.sieve (filter (\n.n%p ~=0) ps)
in take 20 (sieve (from 2))
```

3.3 The Translation Process

One of the motivations of writing the translator presented here was to further evaluate GCLA as a programming tool, and to initiate work on a programming methodology. Some first results on programming methodology extracted from this and other projects have been presented in [12]. The translator, which is written entirely in GCLA is one of the largest GCLA- applications developed so far, and shows that it is possible to use GCLA to do other things than develop Knowledge Based systems.

The translator works in a number of passes where the most important are

- Lexical analysis and parsing
- Some rewriting to remove any remaining syntactical sugar
- Pattern matching compiler
- Lambda lifting
- A pass to create a number of GCLA-functions - one for each supercombinator produced by the lambda lifting pass.

The lexical analyzer and parser were written by Göran Falkman and the rest of the translator by the author of this paper. The methods used are from [7,8,14,18,19] and are not described here since they are standard material in compilers for functional languages. We will simply make some remarks on how they help us achieve our purposes.

3.3.1 Removing syntactic sugar

Some constructs in TML like if expressions, boolean operators and pattern matching in function definitions are simply syntactic sugar for case expressions. These constructs are removed as described in [7,8]. For us this is advantageous since it gives us a much more limited language to work with. All constructed values are represented uniformly with a constructor number and a number of fields (the parts of the constructed value) as in [8,19]. This representation supports our claim that the translation presented here would not be seriously affected if we allowed arbitrary data types.

3.3.2 Pattern Matching Compiler

All patterns are transformed into case expressions. The pattern matching compiler [8,18]is then used to enhance the efficiency of pattern matching. In Sect. 4 we show how these are evaluated by a special inference rule in GCLA.

3.3.3 Lambda Lifting

In GCLA all function definitions are at the same level, that is there are no local functions or lambda abstractions. By lambda lifting the programs [14,18,19] we get a program where all function definitions (the supercombinators) are at the same outermost level which is exactly what we want.

3.3.4 Creating a GCLA Definition

When we have performed the above transformations we get a list of supercombinator definitions and an expression to be evaluated. We then create a GCLA definition according to the following, where each identifier beginning with an uppercase letter is a GCLA variable

- The expression to be evaluated becomes the definition of main
  `main <= expression_to_evaluate`.

- For each super combinator definition we create the definitions
  ```
  name(X1,....,Xn) <= combinatorbody.
  fn(name,[X1,...,Xn]) <= name(X1,...,Xn).
  ```
  Where the X_i's are GCLA variables, and the second clause can be seen as a kind of declaration which says that there is a function of arity n called `name`.

- Constructed values, that is all objects but functions, are written as
  `c(`*`constructornumber`*`:`*`constructorname`*`, `*`list_of_parts`*`)`
  for example a list containing the element 5 becomes
  `c(2:cons,[c(5:int,[]),c(1:nil,[])])`
  For each type constructor there is also a declaration like the ones for functions, for instance the constructor `cons` has the declaration
  `fn(cons,[Hd,Tl]) <= c(2:cons,[Hd,Tl])`

- As an example of a case expression we take the following which evaluates to true if `List` is an empty list and to false otherwise
  ```
  case(List,[
   (1:[] -> c(1.true,[])),
   (2:[Hd,Tl] -> c(2:false,[]))])
  ```

-
  ```
  let([
   X1 = exp1,
  ......
  ```

```
 Xn = expn],
inexpression)
```

Is the translation of let expressions. Since the X_i's are GCLA variables we cannot handle programs with bindings like in,`let rec ones = 1.ones in ones`, which would lead to circular unification. To handle declarations of this kind we would need to lift them out to become definitions like `ones <= c(cons/2,[c(1:int,[]),ones])`, but this is not done at the moment.

- All function applications `f` *exp_1*`...` *exp_n* where the function is applied to enough arguments, that is a supercombinator redex, are replaced by `f(`*exp_1*`,...,`*exp_n*`)`

3.3.5 Examples

In the examples below we have edited the variable and function names since the translator renames identifiers to make them unique.

```
let rec from n = n.from (n+1)
    and take n (x.xs) = if n=0
                        then nil
                        else x.take (n-1) xs
in take 5 (from 3)

% Corresponding GCLA definition.
from(X)<=c(2:cons,[X,from(+(X,c(1:int,[])))]).

take(N,Xs)<=case(Xs,[
             (1:[] -> error),
             (2:[Hd,Tl] -> case(=(N,c(0:int,[])),[
                            (1:[] -> c(1:nil,[])),
                            (2:[] -> c(2:cons,[Hd,take(-(N,c(1:int,[])),Tl)]))]))]).

main <=take(c(5:int,[]),from(c(3:int,[]))).

let rec odd 0 = false
    ||  odd n = even (n-1)
    and even 0 = true
    ||  even n = odd (n-1)
in even 5

% Corresponding GCLA definition.
odd(N)<=case(N,[
         (0:[] -> c(2:false,[])),
         (V:[] -> even(-(N,c(1:int,[]))))]).

even(N)<=case(N,[
          (0:[] -> c(1:true,[])),
          (V:[] ->odd(-(N,c(1:int,[]))))]).

main <=even(c(int/5,[])).
```

We could of course do some sugaring of the syntax, the definition of take could for example be written as below with very small changes to the procedural part described in the next section.

```
take(N,Xs)<=case(Xs,[
             (nil:[] -> error),
             (cons:[Hd,Tl] -> case(N=n(0),[
                            (true:[] -> []),
                            (false:[] ->[Hd|take(N-n(1),Tl])])]).
```

3.4 Remaining Problems

All program examples above were first order programs, and all functions were applied to enough arguments to form a super combinator redex. Not many functional programs have these properties though, so we need some way of handling higher order programs.

As yet we do not know of any natural way to do this in GCLA, but it is not very difficult to do some syntactical rewriting to handle higher order functions. In the translation presented here functions waiting for more arguments are simply represented with the construction `fn(`*`name`*`,`*`ListOfArgs`*`)`, and function application denoted with the operator `¤`. If we have a function like the ordinary map function which takes two arguments, an expression like `fn(map,[])¤`*`funexp`* is evaluated to `fn(map,[`*`funexp`*`])`, and an expression `fn(map,[`*`funexp`*`])¤`*`listexp`* is evaluated to `fn(map,[`*`funexp`*`,`*`listexp`*`])` . We then use the definition `fn(map,[F,L]) <= map(F,L)`. to replace it by the definition of the function map.
As an example, if we translate the program in Sect 3.2, the functions `foldr` and `flat` come out as

```
foldr(F,U,L)<=case(L),[
              (1:[] -> U),
               (2:[Hd,Tl] -> F¤ Hd¤ foldr(F,U,Tl))])

flat(Xss)<=foldr(fn(append,[]),c(1:nil,[]),Xss)
```

4 The Procedural Part : Evaluating the Programs

All we have produced so far is a definition, to give the definition a procedural interpretation we also need a procedural part, defining how the definition is to be used. Sect 4.1 gives some background on how to read the rules and strategies presented in Sect 4.2, and can be skipped by the experienced GCLA programmer (if there are any).

4.1 The Procedural Part Revisited

The general form of a GCLA query is $S \Vdash Q$ where S is a proofterm, that is some more or less instantiated inference rule or strategy, and Q is an object level sequent. One way of reading this query is “S includes a proof of $Q\sigma$ for some substitution σ”. When the GCLA system is started the user is provided with a basic set of inference rules and some standard strategies implementing common search behaviors among these rules. The standard rules and strategies are very general, that is they are potentially useful for a large number of definitions, and provide the possibility to pose a wide variety of queries. We show some of the standard inference rules and strategies here, in order to make the rest of the paper easier to understand.The rest of the standard rules can be found in [4,15].

One simple inference-rule is `axiom/3` which states that anything holds if it is assumed, the standard axiom rule is applicable to any terms and is defined by

```
   axiom(T,C,I)<=
      term(T),                                   % provisio
      term(C),                                   % provisio
      unify(T,C)                                 % provisio
      ->(I@[T|R] \- C).                          % conclusion
```

The rule tells us that if we have an assumption `T` among the list of assumptions `(@[T|R])`,(`@` is an infix append operator) and if both `T` and the conclusion `C` are terms, and if `T` and `C` are unifiable then `C` holds. Another standard rule is definition right, `d_right/2`, the conclusions that can be made from this rule depend on the particular definition at hand. The d_right rule applies to all atoms

```
   d_right(C,PT)<=
      atom(C)                                    % C must be an atom
```

```
    clause(C,B),                              % proviso
    (PT -> (A \- B))                          % premise, use PT to prove it
    -> (A \- C).                              % conclusion
```

This rule is read as,if we have a sequent `A \- C`, and if there is a clause `C <= B` in the definition, and if we can show that the sequent `A \- B` holds using some of the proofs represented by the proofterm `PT` then `C` holds. There is also an inference rule, definition left, which uses the definition to the left, just as `d_right`, this rule called d_left/3 is applicable to all atoms

```
  d_left(T,I,PT) <=
   atom(T),                                   % T must be an atom
   definiens(T,Dp,N),                         % Dp is the definiens of T
   (PT -> (I@[Dp|Y] \- C))                    % premise, use PT to prove it
   -> (I@[T|Y] \- C).                         % conclusion.
```

The definiens operation is described in [15], if `T` is not defined `Dp` is bound to `false`.

One very general search strategy among the predefined inference rules is `arl`, which in each step of the derivation first tries the axiom rule, then all standard rules operating on the consequent of a sequent and after that all standard rules operating on the antecedent, it is defined by:

```
  arl <=
    axiom(_,_,_)                              % try axiom first,
    right(arl)                                % then try the strategy right,
    left(arl)                                 % then try the left strategy.
```

In the definition below of the strategy right, `user_add_right/2` can be defined by the user to contain any new inference rules or strategies desired.

```
  right(PT) <=
    user_add_right(_,PT),                     % try users specific rules first
    v_right(_,PT,PT),                         % then standard right rules
    a_right(_,PT),
    o_right(_,_,PT),
    true_right,
    d_right(_,PT).
```

4.2 A procedural part.

An alternative to building the procedural part from the predefined inference rules and search strategies is to implement new, specific rules and strategies to give exactly the desired procedural interpretation of the definition at hand. In this section we use this method to develop a procedural part to evaluate the function definitions described in previous sections. This procedural part gives an evaluation order corresponding to normal order evaluation in lambda calculus.

4.2.1 The Rules

There are five general rules dealing with normal order evaluation and a number of special rules to handle specific constructs like case expressions and predefined functions. We show the five general rules here and some of the others. The comments are possible readings of the rules.

```
  evaluated(T,C,I) <=                         % succeeds when T is a value
    (functor(T,c,2);                          % values are c(_,_), or
    (functor(T,fn,2),                         % fn(N,A), where fn(N,A) is
    clause(T,B),B = false)),                  % not defined.
    unify(C,T) -> (I@[T|_] \- C).             % result is C.


  rewrite(T,I,PT)#{T \= let(_,_),T \= case(_,_),T \= c(_,_),T \= (_ -> _),T \= (_
  ¤_)} <=                                     % guard
    definiens(T,Dp,N),                        % calculate definiens of T,if Dp \=
    Dp \= false,                              % false, then T is defined and is
    (PT -> (I@[Dp|R] \- C))                   % replaced by its definiens.
    -> (I@[T|R] \- C).                        % value of T is C.
```

```
apply_fn((fn(_,_)¤_),I,PT) <=              % this rule is needed to avoid
   definiens(fn(Name,Args),Dp,N),          % collecting to many arguments
   Dp \= false,                            % If fn(Name,Args) is defined
   (PT -> (I@[Dp¤Arg|R] \- C))             % it is replaced by its definition
   -> (I@[fn(Name,Args)¤Arg|R] \- C).      % value is C.


collect_arg((fn(_,_)¤_),I,PT) <=           % rule to collect arguments to
   append(Args,[Arg],Args1),               % functions. Arg is appended to
   (PT -> (I@[fn(N,Args1)|R] \- C))        % Args and evaluation continues
   -> (I@[fn(N,Args)¤Arg|R] \- C).         % with fn(N,Args1).


normalorder((M¤N),I,PT1,PT2)#{M \= fn(_,_)} <=
   (PT1 -> (I@[M|R] \- M1)),               % first evaluate M to M1
   (PT2 -> (I@[M1¤N|R] \- C))              % then evaluate M1 applied to N
   -> (I@[M¤N|R] \- C).                    % value is C.
```

The built in function + and case expressions are handled by the following rules, the provisio `ev/4` is a call to the underlying prolog system to perform addition.

```
plus_left(+(X,Y),I,PT1,PT2)<=              % rule to handle +:
   (PT1 -> (I@[X|R] \- N1)),               % use PT1 to compute X to N1
   (PT2 -> (I@[Y|R] \- N2)),               % use PT2 to compute Y to N2
   ev('+',N1,N2,N),                        % add N1 and N2
   unify(C,N)                              % unify N with conclusion C
   -> (I@[+(X,Y)|R] \- C).                 % value of X+Y is C.


case(I,PT) <=                              % case expression rule:
   (PT -> (I@[E|R] \- E1)),                % evaluate E to E1
   E1 = c(T:N,F),                          % then instantiate F and Exp
   (det_axiom  -> (CL \- T:F -> Exp)),     % using a deterministic axiom
   (PT -> (I@[Exp|R] \- C))                % evaluate Exp to C
   -> (I@[case(E,CL)|R] \- C).             % value of case is C.
```

4.2.2 The strategy

```
tml<=
   (predef(_,_,tml) <- true),              % first try strategy predef/3
   (rewrite(_,_,tml) <- true),             % then try rule rewrite/3
   (apply_fn(_,_,tml) <- true),            % then try rule apply_fn/3
   (collect_arg(_,_,tml) <- true),         % then try rule collect_arg/3
   (normalorder(_,_,tml,tml) <- true),     % then try rule normalorder/4
   evaluated(_,_,_).                       % else try rule evaluated.


predef(T,I,PT)#{T \= fn(_,_),T \= c(_,_),T \= (_¤_)} <= (I@[T|_] \- C).
predef(T,I,PT)#{T \= fn(_,_),T \= c(_,_),T \= (_¤_)} <= predef_i(T,I,PT).


% continue with the appropriate choice
predef_i(case(_,_),I,PT) <= case(I,PT).
predef_i(let(_,_),I,PT) <= let(I,PT).
predef_i(+(_,_),I,PT) <= plus_left(_,_,PT,PT).
predef_i(-(_,_),I,PT) <= minus_left(_,_,PT,PT).
predef_i(*(_,_),I,PT) <= mult_left(_,_,PT,PT).
predef_i(/(_,_),I,PT) <= div_left(_,_,PT,PT).
predef_i(mod(_,_),I,PT) <= mod_left(_,_,PT,PT).
predef_i(<(_,_),I,PT) <= lt_left(_,_,PT,PT).
predef_i(>(_,_),I,PT) <= gt_left(_,_,PT,PT).
predef_i(=<(_,_),I,PT) <= lte_left(_,_,PT,PT).
predef_i(>=(_,_),I,PT) <= gte_left(_,_,PT,PT).
predef_i(=(_,_),I,PT) <= eq_left(_,_,PT,PT).
```

The search strategy `tml/0` determines the order in which the different inference rules are tried. The order is somewhat arbitrary, the only important thing is that the rule `apply_fn/3` is tried before the rule `collect_arg/3`. The backward arrow '<-' is described in [5]. Used the way it is here it means that if the rule/strategy occurring to the left of the arrow succeeds, then the following rules/strategies are not tried on backtracking. The strategy `tml/0` is deterministic, that is we only get one answer, which we feel is the correct behavior of functional programs.

4.3 Lazier Evaluation

The programs described so far are not lazy, that is expression such as arguments to functions risk being evaluated over and over again. In order to achieve lazier evaluation we attach an extra logical variable to each variable occurring more than once in an expression. We write this as `(X -> XV)` meaning that `X` is evaluated to `XV`. With this technique the function from becomes

```
  from(X) <= c(2:cons,[X->XV,from((X->XV)+c(1:int,[]))]).
```

We also add an extra rule lazy/3 to the strategy `predef`

```
  lazy(1,I,PT) <=                              % first clause of lazy
     Q = (X->Y),                               % Q = (X -> Y)
     var(Y),                                   % if Y is a variable then
     (PT -> (I@[X|R] \- C)),                   % evaluate X to C
     unify(Y,C),                               % unify Y and C
     -> (I@[Q|R] \- C).
  lazy(2,I,PT) <=                              % second clause of lazy
     Q = (X->Y),                               % Q = (X -> Y)
     nonvar(Y),                                % if Y is not a variable
     unify(Y,C)                                % unify Y and C
     -> (I@[Q|R] \- C).
```

4.4 Usage

As a last example we take a program that computes an infinite list of all primes

```
let rec filter p nil = nil
     || filter p (x.xs) = if (p x)
                          then x.(filter p xs)
                          else filter p xs
    and from n = n.from (n+1)
    and take 0 _ = nil
     || take n (x.xs) = x.take (n-1) xs
    and sieve (p.ps) = p.sieve (filter (\n.n%p ~=0) ps)
  in take 10 (sieve (from 2))
```

If we run the translation of this program with the strategy `tml` the evaluation will of course stop almost immediately, but with a printing mechanism to show the results we get the infinite list

```
  | ?- inf \\- sieve(from(c(2:int,[]))) \- L.
  cons(2 cons(3 cons(5 ..........................
```

Since the programs described in this paper are ordinary GCLA programs, they can of course be used in other GCLA programs. A definition to enumerate all subsets of a list on backtracking is given below, used as in the query below we get all subsets of the three first primes.

```
  subset(c(1:nil,[]),[]).
  subset(c(2:cons,[Hd,Tl]),[n(Num)|Tl1]) <=
     (Hd -> c(Num:int,[])),subset(Tl,Tl1).
  subset(c(2:cons,[Hd,Tl]),Tl1) <= subset(Tl,Tl1).
  subset(L,L2)#{L \= c(_,_)} <= (L -> L1),subset(L1,L2).
  n(N) <= c(N:int,[]).
```

```
| ?-  subset \\- \- subset(take(n(3),sieve(from(n(2)))),S).
S = [n(2),n(3),n(5)] ? ;
S = [n(2),n(3)] ? ;
S = [n(2),n(5)] ? ;
S = [n(2)] ? ;
S = [n(3),n(5)] ? ;
S = [n(3)] ? ;
S = [n(5)] ? ;
S = [] ? ;
no
```

5 Concluding Remarks

Our original aim with this work was to find a representation of TML to facilitate reasoning about TML programs in GCLA. So far we have only created this representation. Interesting future work would be to see if it is possible to use it to do other things than evaluate the programs. For example is it possible to write other procedural parts and some auxiliary definitions to do things like debugging or some kind of abstract interpretation? In short develop the programming environment originally planned.

5.1 Alternative Translations

The translation presented here is of course by no means the only possible solution. We have also tried translating the TML programs into a fixed set of combinators (S,K, I and some more) as is done in [18], and to extended lambda calculus using the Y combinator. The procedural part then contained one inference rule for each combinator. Another interesting alternative is to use GCLA pattern matching instead of doing all pattern matching with case expressions. One possible such translation of the program in Sect 4.4 is given below. The main disadvantages of this method is that the different clauses defining a function need to be mutually exclusive, and that we also need a number of extra clauses which evaluates unevaluated arguments. The definition of the function take illustrates this problem.

```
fn(modzero,[X,Y]) <= modzero(X,Y).

modzero(X1,X2) <=
   if((X2 mod X1) = n(0)),                    % if
   b(false),                                  % then
   b(true)).                                  % else

filter(_,[]) <= [].
filter(F,[X|Xs]) <=
   if(F¤ (X=>XV),                             % if
   [(X=>XV)|filter(F,Xs)],                    % then
   filter((X=>XV),Xs)).                       % else
filter(F,Xs)#{Xs \= [],Xs \= [_|_]} <= (Xs -> Xs1) -> filter(F,Xs1).

take(n(0),L) <= [].
take(n(N),[X|Xs])#{N \= 0} <= [X|take(n(N)-n(1),Xs)].
take(N,L)#{N \= n(_)} <= (N -> N1) -> take(N1,L).
take(N,L)#else <= (L -> L1) -> take(N,L1).% if no other clause match

from(X) <= [(X=>XV)|from((X=>XV)+n(1))].

sieve([X|Xs]) <= [(X=>XV)|sieve(filter(fn(modzero,[(X=>XV)]),Xs))].
sieve(Xs)#{Xs \= [],Xs \= [_|_]} <= (Xs -> Xs1) -> sieve(Xs1).

main <= take(n(10),sieve(from(n(2)))).

% definition of if.
if(b(true),Then,_) <= Then.
if(b(false),_,Else)<= Else.
if(B,Then,Else)#{B \= b(_)} <= (B -> B1) -> if(B1,Then,Else).
```

5.2 Related Work

The work in this paper is perhaps closest related to different attempts at doing functional (lazy) evaluating in logic programming, and to work on combining functional/relational programming into one framework. An survey of this research area is given in [11]. GCLA itself [5,15] is of course one example of a programming system which combines the two styles. [1] classify this kind of research into four main trends: an *embedding* approach; a *syntactic* approach; an *algebraic* approach and a *higher-order logic* approach.

Our method should then probably be classified as an example of the syntactic approach, that is we take a program written in some syntax and transform it statically into an alternate syntax which behaves according to the intended semantics. An early example of this approach is [20], later examples which include normal order (lazy) evaluation are [2,16,17]. [16] use an approach very much like ours to avoid computing expressions several times. As compared to [2,16,17] we get much cleaner programs since in GCLA we can evaluate functions, and we also separate the declarative and procedural parts of a program, thus freeing the definitions from control information.

As an example of a logic programming system including equations (functional programming) we mention the language ALF [13], and as an example of a functional language with a logic component the language LML [10].

References

[1] H. Ait-Kaci, R. Nasr, Integrating Logic and Functional Programming, *Lisp and Symbolic Computation*, 2 pp 51-89, 1989.

[2] S. Antoy, Lazy Evaluation in Logic LNCS 528

[3] M. Aronsson, Methodology and Programming Techniques in GCLA II, SICS Research Report R92:05, also published in: L-H. Eriksson, L. Hallnäs, P. Shroeder-Heister (eds.), *Extensions of Logic Programming, Proceedings of the 2nd International Workshop held at SICS, Sweden, 1991, Springer Lecture Notes in Artificial Intelligence*, vol. 596, pp 1-44, Springer-Verlag, 1992.

[4] M. Aronsson, GCLA User's Manual, SICS Research Report T91:21A.

[5] M. Aronsson, GCLA - The Design, Use, and Implementation of a Program Development System, Ph D thesis, Department of Computer and Systems Sciences, The Royal Institute of Technology and Stockholm University, Sweden 1993.

[6] M. Aronsson, P. Kreuger, L. Hallnäs, L-H. Eriksson, A Survey of GCLA — A Definitional Approach to Logic Programming, in: P. Shroeder-Heister (ed.), *Extensions of Logic Programming, Proceedings of the 1st International Workshop held at the SNS, Universität Tübingen, 1989, Springer Lectures Notes in Artificial Intelligence*, vol. 475, Springer-Verlag, 1990.

[7] L. Augustsson, A Compiler for Lazy ML, in *Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming*, pp 218-227, Austin Texas 1984.

[8] L- Augustsson, *Compiling Functional Languages, part II, Ph D thesis*, Chalmers University of Technology, Sweden, 1987.

[9] L. Augustsson, T. Johnsson, *Lazy ML, user's manual*

[10] A. Brogi, P. Mancarella, D. Pedreschi, F. Turini, Logic Programming within a Functional Framework, LNCS, 456. pp. 372 - 386.

[11] D. DeGroot, D Lindstrom, (eds), *Logic Programming. Functions, relations and equations*, Prentice Hall, New Jersey, 1986.

[12] G. Falkman, O. Torgersson, Programming Methodologies in GCLA, to be published in LNCS ?

[13] M. Hanus, Compiling Logic Programs with Equality, LNCS 456.

[14] T. Johnsson, *Compiling lazy functional languages, Ph.D* thesis, Chalmers University of technology, Sweden, 1987.

[15] P. Kreuger, GCLA II, A Definitional Approach to Control, Ph L thesis, Department of Computer Science, University of Göteborg, Sweden, 1991, also published in: L-H. Eriksson, L. Hallnäs, P. Shroeder-Heister (eds.), *Extensions of Logic Programming, Proceedings of the 2nd International Workshop held at SICS, Sweden, 1991, Springer Lecture Notes in Artificial Intelligence*, vol. 596, pp 239-297, Springer-Verlag, 1992.

[16] S. Narain, A Technique For Doing Lazy Evaluation in Logic, *The Journal of Logic Program*ming, vol 3 ,1986, pp 259-276.

[17] S. Narain, Lazy Evaluation in Logic Programming, Proceedings 1990.

[18] S. L. Peyton Jones, *The Implementation of Functional Programming Languages*. Prentice Hall, 1987.

[19] S. L. Peyton Jones and D. Lester, *Implementing Functional Languages: A Tutorial.* Prentice Hall, 1992.

[20] D. H. D. Warren, Higher-order extensions to Prolog -are they needed?, in : D. Mitchie (ed) *Machine Intelligence 10*, pp. 441-454 Edinburgh University Press, Edinburgh, UK (1982).