Relational Programs in the RelView System*

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40
D-24098 Kiel, Germany

1 Introduction

RelView is a relation-based computer system for visualization, analysis and manipulation of discrete structures. It allows the user to manipulate and analyse relations by combining them with the basic operators of relational algebra. The elementary operators can be accessed through simple mouse-click. But they may also be combined into functions and even imperative programs which then can be stored and applied to many sets of input data. A RelView program, also called relational program in the following, essentially is a while-program based on the datatype of binary relations only. In the following, we give a short description of the programming language of the RelView system.

2 Structure of a RelView Program

First, we give a short and informal overview of the structure of a relational program. Such a program has many similarities with a function procedure in the programming languages PASCAL or MODULA-2.

A relational program starts with a head line containing the program's name and the list of formal parameters. After the keyword `DECL` follows the declaration part. It consists of the declarations of relational domains, followed by the declarations of the local functions, and, finally, of the local variables for relations. The declarations of domains and functions are separated by semicolons whereas the declared variables are separated by commas. The third part of a relational program is its body, the keyword `BEG` followed by a sequence of statements which are separated by semicolons. Relational programs compute values. Hence, the last part of such a program is the `RETURN`-clause, which consists of the keyword `RETURN` followed by a relational term. The keyword `END` and a dot behind `END` indicate the end of a relational program. Summing up, we have the following syntactic form:

```
≫ident≪( ≫ident≪, ≫ident≪, ..., ≫ident≪)
 DECL  ≫declaration of local domains≪;
       ≫declaration of local functions≪;
       ≫declaration of local variables≪;
 BEG   ≫statement≪;
           ...
       ≫statement≪
       RETURN  ≫term≪
 END.
```

*WWW: http://www.informatik.uni-kiel.de/~progsys/relview.shtml

In the actual version of the RelView system *an identifier can consist of up to sixteen characters* only.

Domain declarations introduce binary relational products and binary relational sums together with some natural relations like projections and injections. A declaration of a binary relational direct product has got the form

$$\gg \text{ident} \ll = \texttt{PROD}(\gg \text{term} \ll, \gg \text{term} \ll)$$

and a declaration of a binary relational sum has got the form

$$\gg \text{ident} \ll = \texttt{SUM}(\gg \text{term} \ll, \gg \text{term} \ll)\,,$$

where in the arguments of the domain formers `PROD` and `SUM` as constants only the global relations of the working memory and parameters of the superior relational program may occur.

A declaration of a local (non-recursive) relational function with several variables is done as usual in mathematics. Hence, it has the form

$$\gg \text{ident} \ll (\gg \text{ident} \ll, \ldots, \gg \text{ident} \ll) = \gg \text{term} \ll$$

with a function name, a list of formal parameters, and a relational expression again containing globally defined relations from the working memory and parameters from the superior relational program as the only constants.

Note that it is allowed to put more than one relational program into one file. In addition, a file can contain definitions of global functions. Such a definition looks like a definition of a local function followed by a "`.`". See the section on files, loading, and invokation below. Furthermore, comments can be added everywhere in a relational program. Like in PASCAL and Modula-2, a comment must be enclosed by the brackets "{" and "}".

3 Program Calls and Return Values

A call of a relational program looks like an ordinary function application in mathematics and computer science. I.e., it has the form

$$\gg \text{ident} \ll (\gg \text{term} \ll, \ldots, \gg \text{term} \ll)\,,$$

where the arguments (also called actual parameters) of a program call can be any relational terms.

The result of a program execution then is a value, i.e., a relation. Based on the local relational functions and the values of the global relations in the working memory resp. the local variables after the execution of the statements of the relational program, the expression of the `RETURN`-clause is evaluated. The calculated value of that expression is the result of the program call. So in this setting a program call can be seen as a relational term. See again the section on files, loading, and invokation below.

In RelView, program names (and function names as well) are internally distinguished from relation names by the trailing left parenthesis. As a consequence, the left parenthesis belongs to the program name and in a call *between the identifier and the left parenthesis there must not be whitespace.*

4 Datatypes

In a relational program there is only one datatype: All local variables are from the type "relation". Especially there is no type "Boolean" for truth values like in PASCAL or MODULA-2. Instead the truth values are represented by homogeneous relations of type $\mathbf{1} \leftrightarrow \mathbf{1}$ (where $\mathbf{1}$ is a singleton set), i.e., by quadratic 1×1 matrices. The value `FALSE` is modelled by the empty relation on $\mathbf{1}$, a 1×1 matrix with a single zero entry, and the other value `TRUE` is represented by the universal relation on $\mathbf{1}$. The corresponding 1×1 matrix has a single 1 entry.

This specific representation of the truth values is due to the fact that the most important test on relations, viz. inclusion $R \subseteq S$, can be described by a relational expression in R and S which evaluates to `TRUE` if inclusion holds and to `FALSE`, otherwise. In RelView this expression is implemented as a base function `incl`. Thus, further relational properties consisting of inclusions, such as equality, functionality, totality, surjectivity, injectivity, transitivity, reflexivity, antisymmetry, and many others, can then easily be reformulated in terms of `incl` since, clearly, the propositional connectives directly correspond to the Boolean operations on relations.

5 Syntax of Statements

Essentially, the body of a relational program is a sequence of statements. This section gives a short and informal overview of the statements of the RelView programming language:

1. The simple statements of the language are the *assignments*

   $$\gg \text{ident} \ll = \gg \text{term} \ll$$

   with a local variable on the left-hand side and a relational expression on the right-hand side. This expression may contain global relations of the working memory, local variables, relational operations, and calls of relational functions as well as relational programs. In particular, we allow recursive calls of relational programs. Since on the left-hand side of an assignment only local variables are allowed, the execution of a program produces no side-effects with respect to globally defined relations.

2. *Sequential composition* is the first kind of compound statements. It is denoted by a semicolon:

   $$\gg \text{statement} \ll ; \gg \text{statement} \ll$$

3. Like in PASCAL and MODULA-2, there are two different kinds of *conditional statements* in the programming language of RelView. Without an else part a conditional looks like this:

   $$\texttt{IF} \gg \text{term} \ll \texttt{THEN} \gg \text{statement} \ll \texttt{FI}$$

   An else part in a conditional is also allowed. This two-sided conditional then looks as follows:

   $$\begin{array}{l} \texttt{IF} \gg \text{term} \ll \texttt{THEN} \gg \text{statement} \ll \\ \qquad\qquad\qquad\quad \texttt{ELSE} \gg \text{statement} \ll \texttt{FI} \end{array}$$

   At execution time the value of the expression in a conditional statement has to be a relation of type $\mathbf{1} \leftrightarrow \mathbf{1}$, in RelView represented by a 1×1-matrix. See the section on datatypes above.

4. The *while-loop* of the programming language of the RelView system has the following form:

   $$\begin{array}{l} \texttt{WHILE} \gg \text{term} \ll \texttt{DO} \\ \quad \gg \text{statement} \ll \texttt{OD} \end{array}$$

   As in the case of conditional expressions at execution time the value of the condition of the loop has to be a relation of type $\mathbf{1} \leftrightarrow \mathbf{1}$.

Note that `RETURN` $\gg$ term $\ll$ is not a statement. Furthermore, note that a skip statement is not part of the RelView programming language. As a consequence, *before the key words* `ELSE`, `FI`, `OD` *and the* `RETURN`*-clause there must not be a semicolon.*

6 Examples

To get an impression of the programming language of the RelView system, in this section we give some examples of relational programs concerning algorithms on directed and undirected graphs.

a) Our first example is *testing cycle-freeness of a directed graph via removal of sources.* Input is a homogeneous relation R, output is the 1×1 universal relation, if the input is cyclefree, and the 1×1 empty relation, otherwise.

```
cycfree(R)
  DECL s, v
  BEG  v = Ln1(R);
       s = v & -(R^ * v);
       WHILE -empty(s) DO
         v = v & -s;
         s = v & -(R^ * v)
         OD
       RETURN empty(v)
  END.
```

Here, in the Boolean matrix model, `Ln1(R)` is the universal vector with the same row number as `R` and `empty(R)` tests, whether `R` is empty. Both, `Ln1` and `empty` are predefined base functions of RelView.

b) Now, we show how to compute the *reduced graph of a directed graph* using RelView. Again, the input is a homogeneous relation `R` and output is the relation of the cyclefree reduced graph. The auxiliary program `scc` enumerates the strongly connected components of its input as the columns of a relation.

```
scc(R)
  DECL B, S, p, x
  BEG  B = refl(trans(R)) & refl(trans(R))^;
       S = B * point(Ln1(R));
       x = -S;
       WHILE -empty(x) DO
         p = atom(x);
         S = (S^ +  (B * p)^)^;
         x = x & -(B * p)
         OD
       RETURN S
  END.

redgraph(R)
  DECL S
  BEG  S = scc(R)
       RETURN ipa(S^ * R * S)
  END.
```

Here the base function `refl` computes the reflexive closure of a relation, the base function `trans` computes the transitive closure of a relation, and the base function `point` yields for a non-empty vector a point contained in it. Furthermore, `ipa(X) = X & -I(X)` is a function from the start-up file and computes the irreflexive part of a relation. In this definition, the base function `I` computes for a homogeneous relation the identity relation with the same dimension.

c) Our third example deals with *computing a spanning tree of a simple graph.* Here the input is a homogeneous, non-empty, symmetric, and connected relation `E` and the result is a relation, describing a directed spanning tree of `E`.

```
spantree(E)
  DECL T, s, v, w
  BEG  T = atom(E);
       s = dom(T);
       v = s | ran(T);
       WHILE -empty(-v) DO
         T = T | atom(v * -v^ & E);
         v = s | ran(T)
         OD
       RETURN T
  END.
```

In this relational program, the base function `dom` computes the domain of a relation and the base function `atom` yields for a non-empty relation an atom, i.e. a single-pair relation (i.e., an arc) contained in it. Furthermore, `ran(X) = dom(X^)` is a function from the start-up file and computes for a relation its range.

d) As already mentioned, the programming language of RelView allows even recursion. An example is the following program `reach`. Its input is a homogeneous relation `R` and a vector `s`, denoting a set of vertices. The result of a call of `reach(R,s)` is the vector of the vertices which are reachable from `s`. The relational program `reach` uses an auxiliary program `dfs` which is a nested-recursive depth first search algorithm.

```
dfs(R,s,t)
  DECL p, v
  BEG  IF empty(t)
         THEN v = s
         ELSE p = point(t);
              IF incl(p,s)
                THEN v = dfs(R,s,t&-p)
                ELSE v = dfs(R,dfs(R,s|p,R^*p),t&-p)
              FI
       FI
       RETURN v
  END.


reach(R,s)
  DECL v
  BEG  v = dfs(R,On1(s),s)
       RETURN v
  END.
```

In this program, the call `incl(R,S)` of the already mentioned base function `incl` tests whether `R` is contained in `S` and `On1(R)` is the empty vector with the same row number as `R`.

e) Our last example makes use of a global function definition. The following program `CutPoints` computes the vector of all cutpoints of an undirected graph by checking vertex after vertex. Its input is a homogeneous relation `E` describing the undirected graph.

```
del(R,x) = inj(-x) * R * inj(-x)^.

CutPoints(E)
  DECL iscut(E,p) = -eq(trans(del(E,p)),del(trans(E),p));
       c, p
  BEG  p = init(Ln1(E));
       c = iscut(E,p);
       WHILE -empty(next(p)) DO
         p = next(p);
         c = c + iscut(E,p)
         OD
       RETURN c
  END.
```

The call `del(E,p)` of the globally defined function `del` removes the vertex `p` together with all arcs connected to `p` from the undirected graph `E`. The application `iscut(E,p)` tests whether `p` is a cutpoint of the undirected graph `E`. It yields a relation of type $\mathbf{1} \leftrightarrow \mathbf{1}$ representing a Boolean value. The basic function `init` computes an initial point of a given vector. The successor of the point `p` is obtained by the call `next(p)`. The operation "+" computes the sum of two relations.

7 Files, Loading, Invokation

The extension of file names of files containing programs and global function definitions is ".prog". Each file can hold an arbitrary number of relational programs and global function definitions in any order. Please note that each relational program and each global function definition has to be terminated by a dot "`.`".

Relational programs contained in a file can be loaded into the RelView system using the filechooser which can be opened by pressing the push button `FILES` in the first button row of the main window. The list of loaded programs is shown in the second part of the directory window.

In the RelView system, a program execution can be invoked exactly in the same way as an expression evaluation using, for example, the evaluation window, popped up by pressing the push button `EVAL` in the third button row of the menu window.