Simple Rectangle-based Functional Programs for Computing Reflexive-transitive Closures

Rudolf Berghammer

Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Joint work with Sebastian Fischer

Reachability Closures of Relation

Given a relation R on a set X,

- its *reflexive-transitive closure* is $R^* = \bigcup_{n\geq 0} R^n$,
- its *transitive closure* is $R^+ = \bigcup_{n\geq 1} R^n$.

R^* specifies reachability in the graph $g = (X, R)$ and R^+ specifies reachability via non-empty paths.

Fundamental properties:

- $\mathsf{O}^* = \mathsf{I}$
- $R^* = \mathsf{I} \cup R^+$
- $(R \cup S)^* = R^*(SR^*)^*$ (star-decomposition rule)
- R is transitive iff $R = R^+$

Warshall's Algorithm

- Traditional imperative computation of R^* using a representation of relations by 2-dimensional Boolean arrays.

```
$Q := R;$
for $i \in X$ do
    for $j \in X$ do
        for $k \in X$ do
            $Q[j,k] := Q[j,k] \vee (Q[j,i] \wedge Q[i,k])$
for $i \in X$ do
    $Q[i,i] := \mathrm{True}$
```

Warshall

- Arrays are unfit for representing relations if they are of "medium density" or even sparse.

  Computational linguistics, XML-query processing, order theory, . . .

  Here successor lists are much more economic; but such a representation sacrifices the simplicity and efficiency of the algorithm.

- Arrays with side-effects are also problematic if the functional paradigm is used.

Aim of the Talk

To show how systematically to derive simple functional programs for computing reflexive-transitive closures that

- base on a common schematic algorithm,
- use a representation of relations via successor lists,
- have for specific instantiations the same cubic running time as the imperative algorithm.

The used tools and techniques are as in the case of the RAMiCS 12 talk:

- Relation algebra for problem specification, the derivation of the generic algorithm and its specializations.
- Data refinement to obtain list representations and for the translation into Haskell.

For visualization purposes we depict relations as Boolean matrices, drawn by the computer system RelView.

Relation Algebra

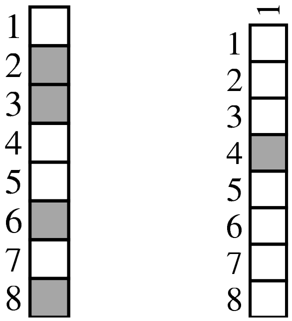
Relations:

- If R is a relation with source X and target Y, we write $R : X \leftrightarrow Y$.
- $[X \leftrightarrow Y]$ is the *type/set* of all relations with source X and target Y.

Signature of relation algebra:

- Constants: $\mathsf{O}, \mathsf{L}, \mathsf{I}$.
- Operations: $R \cup S, R \cap S, RS, \overline{R}, R^{\mathsf{T}}$.
- Tests: $R \subseteq S, R = S$.

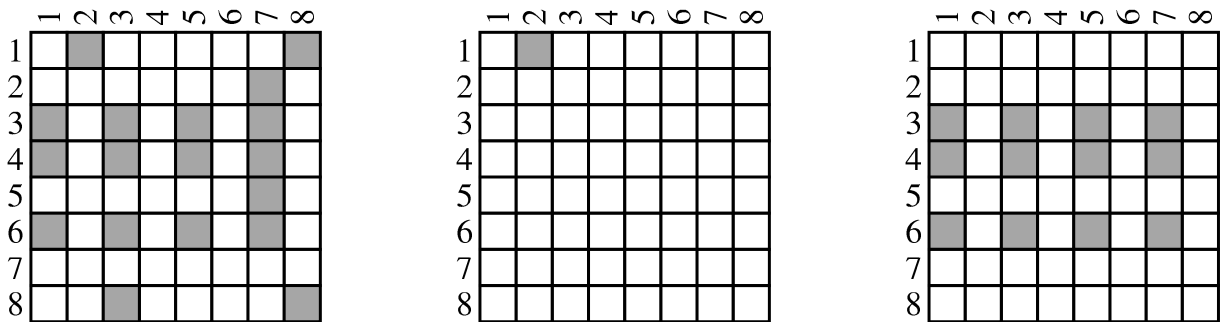
Properties;



- *Reflexivity*: $\mathsf{I} \subseteq R$.
- *Transitivity*: $RR \subseteq R$.
- *Vector*: $v\mathsf{L} = v$.
- *Point*: $p\mathsf{L} = p$, $\mathsf{L}p = \mathsf{L}$ (*surjectivity*) and $pp^{\mathsf{T}} \subseteq \mathsf{I}$ (*injectivity*).

Rectangles and Reachability Closures

- A relation $S : X \leftrightarrow X$ is a *rectangle* if there exist $A, B \subseteq X$ such that
$$S = A \times B.$$
- Examples, where $X = \{1, \ldots, 8\}$:



R S_1 S_2

R:

	1	2	3	4	5	6	7	8
1		▨						▨
2							▨	
3	▨		▨		▨		▨	
4	▨		▨		▨		▨	
5							▨	
6	▨		▨		▨		▨	
7								
8			▨					▨

S_1:

	1	2	3	4	5	6	7	8
1		▨						
2								
3								
4								
5								
6								
7								
8								

S_2:

	1	2	3	4	5	6	7	8
1								
2								
3	▨		▨		▨		▨	
4	▨		▨		▨		▨	
5								
6	▨		▨		▨		▨	
7								
8								

  R is not a rectangle. S_1 is a rectangle since $S_1 = \{1\} \times \{2\}$, and S_2 is a rectangle since $S_2 = \{3, 4, 6\} \times \{1, 3, 5, 7\}$.

- Relation-algebraic specification: $S : X \leftrightarrow X$ is a rectangle iff
$$S \mathsf{L} S \subseteq S.$$

Theorem. Assume $R : X \leftrightarrow X$ and let $S : X \leftrightarrow X$ be a rectangle. Then

$$(R \cup S)^* = R^* \cup R^*SR^*.$$

Proof. Transitivity $SR^*SR^* \subseteq S\mathsf{L}SR^* \subseteq SR^*$. yields $(SR^*)^+ = SR^*$. By means of this equation, the statement follows from

$$\begin{aligned}
(R \cup S)^* &= R^*(SR^*)^* && \text{star-decomposition} \\
&= R^*(\mathsf{I} \cup (SR^*)^+) && \\
&= R^*(\mathsf{I} \cup SR^*) && SR^* \text{ transitive} \\
&= R^* \cup R^*SR^* && \text{distributivity.}
\end{aligned}$$

Generic Algorithm. The following functional algorithm computes R^*:

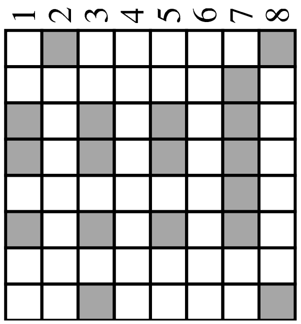
```
rtc : $[X \leftrightarrow X] \to [X \leftrightarrow X]$
rtc($R$) = if $R = \mathsf{O}$ then $\mathsf{I}$
           else let $S$ = rectangle($R$)
                    $C$ = rtc($R \cap \overline{S}$)
                in  $C \cup CSC$
```

Here *rectangle*(R) yields a non-empty rectangle S with $S \subseteq R$.

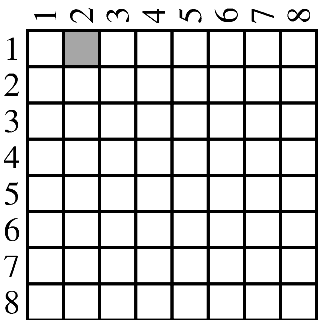
Possibilities for *rectangle*(R):

- Selection of an atomic relation contained in R.
- Selection of a relation that singles out a row of R.
- Selection of a relation that singles out a column of R.
- Selection of a maximal (a non-enlargable) rectangle contained in R
  - started vertically
  - started horizontally.

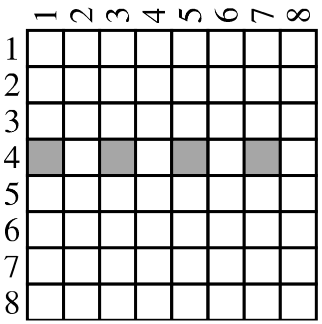
Examples, where $X = \{1, \ldots, 8\}$:



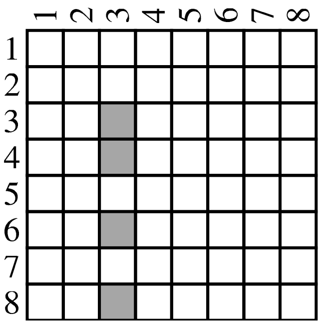
R



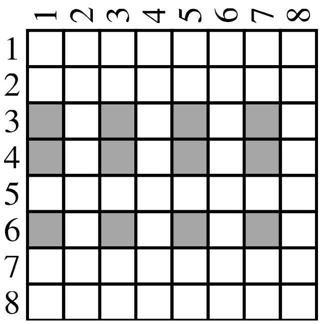
S_1



S_2



S_3



S_4

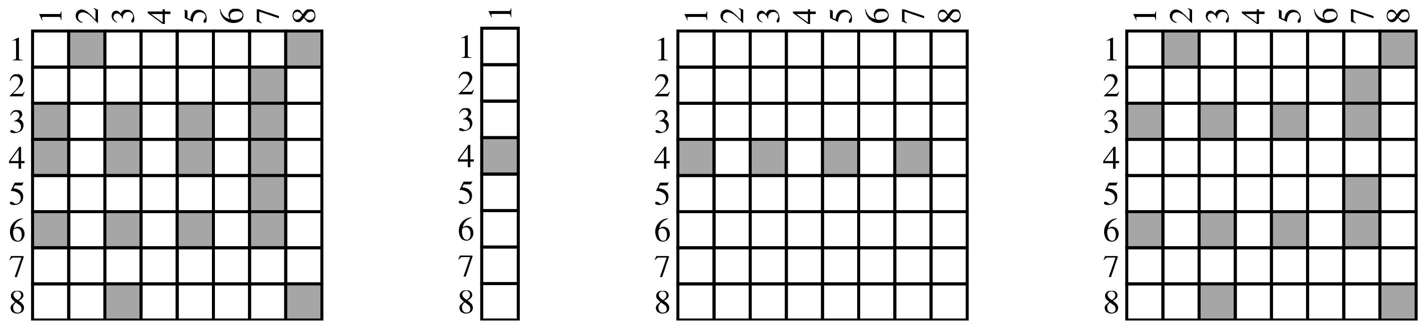
Singling Out Rows

For $R : X \leftrightarrow X$ and a *point* $p : X \leftrightarrow \mathbf{1}$ with $p \subseteq R\mathsf{L}$ ("it has successors"), we define:

$$S := pp^{\mathsf{T}}R$$

The rectangle $S = pp^{\mathsf{T}}R$ corresponds to the row of R designated by the point p and $R \cap \overline{S}$ "zeroes out" out this row.

Example, where $X = \{1, \dots, 8\}$ and p describes the element 4 of X:



	1	2	3	4	5	6	7	8
1		▨						▨
2							▨	
3	▨		▨		▨		▨	
4	▨		▨		▨		▨	
5							▨	
6	▨		▨		▨		▨	
7								
8			▨					▨

R

	1
1	
2	
3	
4	▨
5	
6	
7	
8	

p

	1	2	3	4	5	6	7	8
1								
2								
3								
4	▨		▨		▨		▨	
5								
6								
7								
8								

$S = pp^{\mathsf{T}}R$

	1	2	3	4	5	6	7	8
1		▨						▨
2							▨	
3	▨		▨		▨		▨	
4								
5							▨	
6	▨		▨		▨		▨	
7								
8			▨					▨

$R \cap \overline{S}$

Proof of the rectangle property of S (using the *vector property* of p):

$$S\mathsf{L}S = pp^{\mathsf{T}}R\mathsf{L}pp^{\mathsf{T}}R \subseteq p\mathsf{L}p^{\mathsf{T}}R = pp^{\mathsf{T}}R = S$$

Proof of the inclusion $S \subseteq R$ (using the *injectivity* of p):

$$S = pp^{\mathsf{T}}R \subseteq \mathsf{I}R = R$$

Proof of non-emptiness of S by contradiction (using the *surjectivity* of p):

$$S = \mathsf{O} \iff pp^{\mathsf{T}}R \subseteq \mathsf{O} \iff \ldots \iff p \subseteq \overline{R\mathsf{L}}$$

Refined Algorithm. Singling out a row leads to:

```
$rtc : [X \leftrightarrow X] \to [X \leftrightarrow X]$
$rtc(R) =$ if $R = \mathsf{O}$ then $\mathsf{I}$
          else let $p = point(R\mathsf{L})$
                   $S = pp^{\mathsf{T}}R$
                   $C = rtc(R \cap \overline{S})$
               in  $C \cup CSC$
```

Here $point(R\mathsf{L})$ yields a point p with $p \subseteq R\mathsf{L}$.

From Relation Algebra to Functions

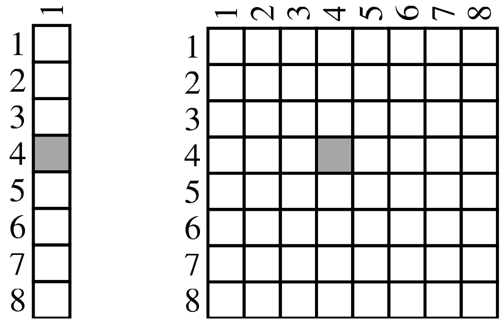
Represent $F : X \leftrightarrow X$ by $f : X \rightarrow 2^X$ such that $(x,y)\in F$ iff $y\in f(x)$.

- The empty relation O is represented by
$$\lambda x.\emptyset.$$
- The identity relation I is represented by
$$\lambda x.\{x\}.$$
- If $R \neq \mathsf{O}$ is represented by r, then the choice of a point p with $p \subseteq R\mathsf{L}$ corresponds to the choice of an element $n\in X$ with $r(n) \neq \emptyset$.
- With r and n from above, $R \cap \overline{pp^{\mathsf{T}}R}$ is represented by
$$r[n \leftarrow \emptyset].$$
- If C is represented by c, then $C \cup Cpp^{\mathsf{T}}RC$ is represented by
$$\lambda x.\textbf{if } n\in c(x) \textbf{ then } c(x) \cup \bigcup\{c(k) \mid k\in r(n)\} \textbf{ else } c(x).$$

Let $ks := \bigcup\{c(k) \mid k \in r(n)\}$. Then the last representation follows from

$$\begin{array}{rl}
& (x,y) \in C \cup Cpp^{\mathsf{T}}RC \\
\Leftrightarrow & (x,y) \in C \vee (x,y) \in Cpp^{\mathsf{T}}RC \\
\Leftrightarrow & (x,y) \in C \vee \exists i : (x,i) \in C \wedge \exists j : (i,j) \in pp^{\mathsf{T}} \wedge (j,y) \in RC \\
\Leftrightarrow & (x,y) \in C \vee \exists i : (x,i) \in C \wedge \exists j : i = n \wedge j = n \wedge (j,y) \in RC \\
\Leftrightarrow & (x,y) \in C \vee ((x,n) \in C \wedge (n,y) \in RC) \\
\Leftrightarrow & (x,y) \in C \vee ((x,n) \in C \wedge \exists k : (n.k) \in R \wedge (k,y) \in C) \\
\Leftrightarrow & y \in c(x) \ \vee \textbf{if } n \in c(x) \textbf{ then } \exists k : k \in r(n) \wedge y \in c(k) \textbf{ else } \textit{false} \\
\Leftrightarrow & y \in c(x) \vee \textbf{if } n \in c(x) \textbf{ then } y \in \bigcup\{c(k) \mid k \in r(n)\} \textbf{ else } y \in \emptyset \\
\Leftrightarrow & y \in c(x) \vee y \in \textbf{if } n \in c(x) \textbf{ then } y \in ks \textbf{ else } \emptyset \\
\Leftrightarrow & y \in \textbf{if } n \in c(x) \textbf{ then } c(x) \cup ks \textbf{ else } c(x) \\
\Leftrightarrow & y \in (\lambda x.\textbf{if } n \in c(x) \textbf{ then } c(x) \cup ks \textbf{ else } c(x))(x)
\end{array}$$

using that $(i,j) \in pp^{\mathsf{T}}$ iff $i = n$ and $j = n$.



Refined Algorithm on Functions.

We replace

- all relations by their representing functions,
- the choice of p by that of n.

Doing so, we arrive at the following **refined algorithm on functions**:

$$
\begin{array}{l}
\mathit{rtc} : (X \to 2^X) \to (X \to 2^X) \\
\mathit{rtc}(r) = \textbf{if } r = \lambda x.\emptyset \textbf{ then } \lambda x.\{x\} \\
\qquad\qquad\qquad\qquad\quad \textbf{else } \begin{array}[t]{ll}
\textbf{let} & n = \mathit{elem}(\{x \in X \mid r(x) \neq \emptyset\}) \\
& c = \mathit{rtc}(r[n \leftarrow \emptyset]) \\
& ks = \bigcup\{c(k) \mid k \in r(n)\} \\
\textbf{in} & \lambda x.\textbf{if } n \in c(x) \textbf{ then } c(x) \cup ks \textbf{ else } c(x)
\end{array}
\end{array}
$$

Here $\mathit{elem}(\{x \in X \mid r(x) \neq \emptyset\})$ yields a $n \in X$ with $r(n) \neq \emptyset$.

List Representation of Output Functions

Assuming $X = \{0, \ldots, m\}$ we represent $f : X \to 2^X$ by $[f(0), \ldots, f(m)]$.

- For $\lambda x.\{x\}$ we get the representation

$$[\{x\} \mid x \in [0..m]].$$

- For $\lambda x.\textbf{if } n \in c(x) \textbf{ then } c(x) \cup ks \textbf{ else } c(x)$ we get the representation

$$[\textbf{if } n \in ms \textbf{ then } ms \cup ks \textbf{ else } ms \mid ms \in cs],$$

where $cs \in (2^X)^*$ represents c. Note that then $c(k)$ becomes $cs!!k$.

Moving from the output functions to the list representations, we get:

$$\begin{array}{l}
\textit{rtc} : (X \to 2^X) \to (2^X)^* \\
\textit{rtc}(r) = \textbf{if } r = \lambda x.\emptyset \textbf{ then } [\{x\} \mid x \in [0..m]] \\
\qquad\qquad\qquad\qquad \textbf{else } \textbf{let } n = \textit{elem}(\{x \in X \mid r(x) \neq \emptyset\}) \\
\qquad\qquad\qquad\qquad\qquad\quad cs = \textit{rtc}(r[n \leftarrow \emptyset]) \\
\qquad\qquad\qquad\qquad\qquad\quad ks = \bigcup\{cs!!k \mid k \in r(n)\} \\
\qquad\qquad\qquad\qquad\quad \textbf{in } [\textbf{if } n \in ms \textbf{ then } ms \cup ks \textbf{ else } ms \mid ms \in cs]
\end{array}$$

List Representation of Input Functions

We represent $f : X \to 2^X$ as list of the pairs $(x, f(x))$ for which $f(x) \neq \emptyset$.

- If $rs \in (X \times 2^X)^*$ represents r, testing $r = \lambda x.\emptyset$ reduces to

$$rs = [\,].$$

- An n with $r(n) \neq \emptyset$ is then given by the first component of *head*(rs).
- The list representation of $r[n \leftarrow \emptyset]$ is then

$$\textit{tail}(rs).$$

If we use pattern matching in the **let**-clause, we get:

```
$\textit{rtc} : (X \times 2^X)^* \to (2^X)^*$
$\textit{rtc}(rs) =$ if $rs = [\,]$ then $[\{x\} \mid x \in [0..m]]$
                    else let $(n, ns) = \textit{head}(rs)$
                             $cs = \textit{rtc}(\textit{tail}(rs))$
                             $ks = \bigcup\{cs!!k \mid k \in ns\}$
                         in [if $n \in ms$ then $ms \cup ks$ else $ms \mid ms \in cs$]
```

Refined Algorithm on Lists of Sets

The last version of *rtc* also can be written in the following form.

- The auxiliary function *step* performs the essential computations of the recursion.

$$\begin{aligned}
&\mathit{step} : (X \times 2^X) \times (2^X)^* \to (2^X)^* \\
&\mathit{step}((n, ns), cs) = \textbf{let } ks = \bigcup\{cs!!k \mid k \in ns\} \\
&\qquad\qquad\qquad\qquad \textbf{in } [\textbf{if } n \in ms \textbf{ then } ms \cup ks \textbf{ else } ms \mid ms \in cs]
\end{aligned}$$

- The main program:

$$\begin{aligned}
&\mathit{rtc} : (X \times 2^X)^* \to (2^X)^* \\
&\mathit{rtc}(rs) = \textbf{if } rs = [\,] \textbf{ then } [\{x\} \mid x \in [0..m]\,] \\
&\qquad\qquad\qquad\qquad\quad \textbf{else } \mathit{step}(\mathit{head}(rs), \mathit{rtc}(\mathit{tail}(rs)))
\end{aligned}$$

Translation into Haskell

Let a Haskell constant `m` of type `Int` for m be at hand.

An obvious implementation of subsets of X in Haskell is given by *sorted lists* over X *without multiple occurrences of elements*.

- $\emptyset$ is implemented by `[]`.
- Set-membership is implemented by `elem`.
- An implementation of set union using linear running time is

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
  case compare x y of EQ -> x : merge xs ys
                      LT -> x : merge xs (y:ys)
                      GT -> y : merge (x:xs) ys
```

(merging and removal of multiple occurrences of elements).

The Haskell list for $\bigcup\{cs!!k \mid k\in ns\}$ is obtained by merging all sorted lists `cs!!k`, where `k` ranges over the elements of `ns`. Haskell code for *step*:

```
step :: (Int,[Int]) -> [[Int]]  -> [[Int]]
step (n,ns) cs =
  let ks = foldr merge [] [cs!!k | k <- ns]
  in  [if elem n ms then merge ms ks else ms | ms <- cs]
```

The running time is is $O(m^2)$ since during the whole merging process each argument and result of `merge` has at most length $m+1$.

Also the main function *rtc* can be seen as a right-fold over the Haskell counterpart `rs` of rs.

```
rtc :: [(Int,[Int])] -> [[Int]]
rtc rs =
  foldr step [[x] | x <- [0..m]] rs
```

Slightly dissatisfying is that the Haskell function `rtc`

- depends on the constant `m`
- works with two different list representations of relations.

Because of

```
step (n,[]) cs = cs
```

we can change it in such a way that it is independent of `m` and also the input `rs` is a list of type `[[Int]]` that contains as x-th component the (possible empty) successor list of x.

```
rtc :: [[Int]] -> [[Int]]
rtc rs =
  let xs = [0..length rs - 1]
  in  foldr step [[n] | n <- xs] (zip xs rs)
```

The complexity of the entire program is $O(m^3)$.

The Complete Haskell Program

```
merge :: [Int] -> [Int] -> [Int]
merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys) =
  case compare x y of EQ -> x : merge xs ys
                      LT -> x : merge xs (y:ys)
                      GT -> y : merge (x:xs) ys

step :: (Int,[Int]) -> [[Int]]  -> [[Int]]
step (n,ns) cs =
  let ks = foldr merge [] [cs!!k | k <- ns]
  in  [if elem n ms then merge ms ks else ms | ms <- cs]

rtc :: [[Int]] -> [[Int]]
rtc rs =
  let xs = [0..length rs - 1]
  in  foldr step [[n] | n <- xs] (zip xs rs)
```

Concluding Remarks

Some improvements with regard to practical running times are possible.

- A user-defined `elem`-test that takes advantage of the fact that the successor lists are sorted.
- A linear time computation of `[cs!!k | k <- ns]` that also uses that all successor lists are sorted.
- Transformation into tail-recursive version.

Further improvement:

- Replace `zip xs rs` by

  ```
  zip xs (map (nub.sort) rs).
  ```

  This is an implementation that does not rely on the successor lists to be strictly increasing without sacrificing time complexity.

Present and future work within the Kiel group:

Combination of
relation algebra and techniques of functional programming
for the formal develompent of efficient functional programs
on relation-based discrete structures.