Relations, Equivalences, Partitions

Jan van Eijck

March 11, 2014

Abstract

This reports documents the implementation of equivalence relations as partitions, and of reflexive and symmetric relations (similarity relations) as covers. The fusion of a cover is the finest coarsening of the cover that turns it into a partition. This corresponds to the transitive closure of the similarity defined by the cover. These are important ingredients of a concise and efficient representation of multi-agent epistemic models, where knowledge of agents is captured by equivalences. The final chapter computes generated subdomains and bisimulation-minimal partition tables.

Chapter 1

Introduction

The implementation is in literate programming style [5], using Haskell [3].

```
module EREL where
import Data.List
```

It is useful to have an operator for material implication:

```
infix 1 ==>
(==>) :: Bool -> Bool -> Bool
p ==> q = (not p) || q
```

And the following quantifier is also handy:

```
forall = flip all
```

Construct a function from a table; lookup for arguments not in the table generate an error:

```
apply :: Eq a => [(a,b)] -> a -> b
apply table x = let
  Just y = lookup x table
 in
  y
```

Restriction of a table:

```
restrTable :: Eq a => [a] -> [(a,b)] -> [(a,b)]
restrTable xs t = [ (x,y) | (x,y) <- t, elem x xs ]
```

Chapter 2

Relations

Relations represented as lists of pairs.

```
type Rel a = [(a,a)]
```

We will assume that domains are ordered, and sets are represented as lists without duplicates. Set union is implemented as merge:

```
merge :: Ord a => [a] -> [a] -> [a]
merge xs [] = xs
merge [] ys = ys
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
```

Extend this to lists of lists:

```
mergeL :: Ord a => [[a]] -> [a]
mergeL = foldl' merge []
```

The domain of a relation:

```
domR :: Ord a => Rel a -> [a]
domR [] = []
domR ((x,y):pairs) = mergeL [[x],[y],domR pairs]
```

Cartesian product of two lists:

```
cprod :: [a] -> [b] -> [(a,b)]
cprod xs ys = [(x,y) | x <- xs, y <- ys ]
```

In terms of this, total relation:

```
totalR :: [a] -> Rel a
totalR xs = cprod xs xs
```

Least fixpoint operator:

```
lfp :: Eq a => (a -> a) -> a -> a
lfp f x | x == f x  = x
        | otherwise = lfp f (f x)
```

Use this to compute the transitive closure of a relation.

```
tc :: Ord a => [(a,a)] -> [(a,a)]
tc r = lfp (\ s -> (s `merge` (r@@s))) r
```

This uses the composition of two relations:

```
infixr 5 @@

(@@) :: Eq a => Rel a -> Rel a -> Rel a
r @@ s =
  nub [ (x,z) | (x,y) <- r, (w,z) <- s, y == w ]
```

Chapter 3

Similarities and Covers, Equivalences and Partitions

A partition β of a set X is a family of subsets of X with the following properties:

1. $\bigcup \beta = X$,
2. $Y \in \beta$ implies $Y \neq \emptyset$,
3. $Y, Z \in \beta \wedge Y \neq Z$ implies $Y \cap Z = \emptyset$.

Call a family of subsets of X satisfying just (1) and (2) a *cover* of X. Call a relation that is reflexive and symmetric a *similarity*.

It is not hard to see that covers correspond to similarities, in the following way:

If β is a cover of X then $R \subseteq X^2$ given by

$$\{(x, y) \in X^2 \mid \exists Y \subseteq \beta : \ \{x, y\} \subseteq Y\}$$

is a reflexive and symmetric relation, and conversely, if R is a reflexive and symmetric binary relation on X, the family β given by

$$\{\{xRy \mid y \in X\} \mid x \in X\}$$

is a cover of X.

If α, β are covers of X, then $\alpha \cup \beta$ is also a cover of X. Similarities are closed under union.

If α, β are partitions of X, then $\alpha \cup \beta$ is not necessarily a partition of X. Equivalences are not closed under union.

To see that the union of two equivalence relations need not be an equivalence, take the example of R and S given by $R = \{(1,1),(1,2),(2,1),(2,2),(3,3)\}$ and $S = \{(1,1),(2,2),(2,3),(3,2),(3,3)\}$. Then

$$R \cup S = \{(1,1),(1,2),(2,1),(2,2),(2,3),(3,2),(3,3)\}.$$

Since $(3,2)$ and $(2,1)$ are in $R \cup S$, but $(3,1) \notin R \cup S$, the relation $R \cup S$ is not transitive.

Proposition 1 *If R, S are similarities on X, then $R \cup S$ is a similarity on X.*

Proof. Let R, S be similarities on X. Then $I \subseteq R$, so $I \subseteq R \cup S$, i.e., $R \cup S$ is reflexive. Consider $x, y \in X$ with $(x,y) \in R \cup S$. Then $(x,y) \in R$ or $(x,y) \in S$, and therefore, by symmetry of R and S, $(y,x) \in R$ or $(y,x) \in S$. It follows that $(y,x) \in R \cup S$. □

It follows immediately from Proposition 1 that the union of two equivalences is a similarity, and that the union of an equivalence and a similarity is a similarity.

Similarity relations represented as covers, equivalence relations represented as partitions:

```
type Erel a = [[a]]
```

Overlap of lists, on the assumption that the lists are ordered:

```
overlap :: Ord a => [a] -> [a] -> Bool
overlap [] ys = False
overlap xs [] = False
overlap (x:xs) (y:ys) = case compare x y of
  EQ -> True
  LT -> overlap xs (y:ys)
  GT -> overlap (x:xs) ys
```

Check that a cover is a partition. Note: this assumes each block in the cover is ordered.

```
isPart :: Ord a => Erel a -> Bool
isPart []     = True
isPart (b:bs) = all (not.overlap b) bs &&
                isPart bs
```

The domain of a cover or partition:

```
domE :: Ord a => Erel a -> [a]
domE = mergeL
```

The rank of a partition α of X is given by $|X| - |\alpha|$:

```
rank :: Ord a => Erel a -> Int
rank r = let
   dom = domE r
 in
   length dom - length r
```

The block of an element in a partition:

```
bl :: Eq a => Erel a -> a -> [a]
bl r x = head (filter (elem x) r)
```

Notes that blocks need not be unique if the cover is not a partition. If the cover is a partition, the following can be used to check whether two items are equivalent.

```
related :: Eq a => Erel a -> a -> a -> Bool
related p x y = elem y (bl p x)
```

Functions f generate equivalences with the recipe $\lambda xy \mapsto fx = fy$. From a function to an equivalence:

```
fct2equiv :: Eq a => (b -> a) -> b -> b -> Bool
fct2equiv f x y = f x == f y
```

From a function to a partition, given a domain:

```
fct2erel :: (Eq a,Eq b) => (b -> a) -> [b] -> Erel b
fct2erel f [] = []
fct2erel f (x:xs) = let
   xblock = x: filter (\ y -> f x == f y) xs
   rest   = xs \\ xblock
 in
   xblock: fct2erel f rest
```

Examples:

```
*EREL> fct2erel (\x -> mod x 3 == 1) [1..10]
[[1,4,7,10],[2,3,5,6,8,9]]
```

```
*EREL> fct2erel (\x -> mod x 3) [1..10]
[[1,4,7,10],[2,5,8],[3,6,9]]
*EREL> fct2erel (mod 3) [1..10]
[[1,3],[2],[4,5,6,7,8,9,10]]
```

The partition equivalent of a total relation:

```
totalE :: [a] -> Erel a
totalE xs = [xs]
```

From a characteristic function to a relation:

```
cfct2rel :: Eq a => [a] -> (a -> a -> Bool) -> Rel a
cfct2rel domain f =
  [(x,y) | (x,y) <- totalR domain, f x y ]
```

From a partition to a relation:

```
erel2rel :: Ord a => Erel a -> Rel a
erel2rel r =
  [(x,y) | (x,y) <- totalR (domE r), elem y (bl r x) ]
```

From a characteristic function of a relation to the cover corresponding to the reflexive and symmetric closure of the relation:

```
cfct2erel :: Eq a =>
             [a] -> (a -> a -> Bool) -> Erel a
cfct2erel [] r = []
cfct2erel (x:xs) r = xblock : cfct2erel rest r
  where
  (xblock,rest) = (x:filter (r x) xs,
                   filter (not . (r x)) xs)
```

Example:

```
*EREL> cfct2erel [1..10] (\ x y -> x+y < 5)
[[1,2,3],[4],[5],[6],[7],[8],[9],[10]]
```

Coarsening, Refinement Let α, β be partitions on the same set X. Then $\alpha \leq \beta$ (α is finer than β, α is a refinement of β, β is coarser than α) if every element Y of α is a subset of some element Z of β.

The following implementation of `sublist` assumes that the input lists are ordered and without duplicates.

```
sublist :: Ord a => [a] -> [a] -> Bool
sublist [] ys = True
sublist xs [] = False
sublist (x:xs) (y:ys) = case compare x y of
  LT -> False
  EQ -> sublist xs ys
  GT -> sublist (x:xs) ys
```

```
finer :: Ord a => Erel a -> Erel a -> Bool
finer r s = all (\xs -> any (sublist xs) s) r
```

```
coarser :: Ord a => Erel a -> Erel a -> Bool
coarser r s = finer s r
```

The finer-than relation $\leq$ is a partial order on the set of all partitions of a set X. If X is finite, the set of partitions of X forms a geometric lattice [2], i.e., a lattice that is finite, atomistic and semimodular:

1. Any two elements α, β have a least upper bound $\alpha \vee \beta$ and a greatest lower bound $\alpha \wedge \beta$.

2. The lattice has a bottom element $\bot = \{\{x\} \mid x \in X\}$ and a top element $\top = \{\{X\}\}$.

3. The atoms of the lattice are the α with rank 1; these are the partitions with one element of size 2, and all other elements of size 1.

4. Every element α is the least upper bound of a set of atoms.

5. The rank function r satisfies the semi-modularity law:

$$r(\alpha) + r(\beta) \geq r(\alpha \wedge \beta) + r(\alpha \vee \beta).$$

The finer-than relation $\leq$ also applies to covers. In particular, for any cover α we can talk about the finest partition β with $\alpha \leq \beta$.

Proposition 2 *The transitive closure of a similarity is an equivalence.*

Proof. Let S be a similarity (reflexive and transitive relation) on X, and let S^* be the transitive closure of S (the smallest transitive relation that contains S).

We have to show that S^* is reflexive and symmetric. Reflexivity follows from the fact that $I \subseteq S \subseteq S^*$. For symmetry, assume xS^*y. Then there is a path $x_1 \ldots x_n$ with $x_i S x_{i+1}$ and $x = x_1$, $x_n = y$. By symmetry of S $x_n \ldots x_1$ is an S-path from y to x. Thus, yS^*x. □

Call the finest coarsening of the cover that turns it into a partition the *fusion* of the cover. This corresponds to the transitive closure of the similarity given by the cover. The following function computes this partition, by fusing all blocks in the cover that have elements in common, until a partition results:

```
fusion :: Ord a => [[a]] -> Erel a
fusion [] = []
fusion (b:bs) = let
   cs = filter (overlap b) bs
   xs = mergeL (b:cs)
   ds = filter (overlap xs) bs
 in
   if cs == ds
     then xs : fusion (bs \\ cs)
     else fusion (xs : bs)
```

Compute the partition of the reflexive transitive symmetric closure of the input relation:

```
rel2erel :: Ord a => Rel a -> Erel a
rel2erel []  = []
rel2erel ((x,y):pairs) = let
   xypairs = filter (\ (u,v) ->
             elem u [x,y] || elem v [x,y]) pairs
   rest    = pairs \\ xypairs
   xyblock = domR ((x,y):xypairs)
 in
   fusion (xyblock : rel2erel rest)
```

If R and S are equivalences on X, it does not follow that $R \circ S$ is an equivalence on X. Consider $R = \{(1,1),(1,2),(2,1),(2,2),(3,3)\}$ and $S = \{(1,1),(2,2),(2,3),(3,2),(3,3)\}$. Then

$$R \circ S = \{(1,1),(1,2),(1,3),(2,1),(2,2),(2,3),(3,2),(3,3)\}.$$

Since $(3,1) \notin R \circ S$, the result is not an equivalence.

Compute the partition corresponding to the reflexive transitive symmetric closure of the composition of two (equivalence) relations:

```
concatE :: Ord a => Erel a -> Erel a -> Erel a
concatE r s = fusion [ merge b (bl s x) | b <- r, x <- b ]
```

Example:

```
r = [[1,2],[3]]
s = [[1],[2,3]]
```

```
*EREL> concatE r s
[[1,2,3]]
```

The following operation computes the partition corresponding to the finest equivalence that includes the union of two equivalences. This corresponds to the transitive closure of the union of the two equivalences. By Proposition 1 this is an equivalence.

```
unionRel ::  Ord a => Erel a -> Erel a -> Erel a
unionRel r s = fusion (r++s)

unionRels :: Ord a => [Erel a] -> Erel a
unionRels = foldl1' unionRel
```

The restriction of a partition to a domain:

```
restrict :: Ord a => [a] -> Erel a -> Erel a
restrict domain =  nub . filter (/= [])
                   . map (filter (flip elem domain))
```

Split a partition α with a subset Y, to get:

$$\{Z \cap Y \mid Z \in \alpha, Z \cap Y \neq \emptyset\} \cup \{Z - Y \mid Z \in \alpha, Z - Y \neq \emptyset\}.$$

```
split :: Ord a => Erel a -> [a] -> (Erel a, Erel a)
split r xs = let
   domain = domE r
   ys     = domain \\ xs
 in
   (restrict xs r, restrict ys r)
```

Proposition 3 *The intersection of two equivalences is an equivalence. The intersection of any set of equivalences is an equivalence.*

Proof. Let R, S be equivalence relations on X. Then reflexivity and symmetry of $R \cap S$ are immediate. For transitivity, let $xR \cap Sy$ and $yR \cap Sz$. Then xRy, xSy, yRz, ySz. By transitivity of R and S: xRz and xSz. It follows that $xR \cap Sz$.

This is easily generalized to the intersection of a set of equivalences. □

The following function computes the partition that corresponds to the intersection of two equivalences, given as partitions:

```
intersectRel ::  Ord a => Erel a -> Erel a -> Erel a
intersectRel r [] = []
intersectRel r (b:bs) = let
   (xs,ys) = split r b
 in
   xs ++ intersectRel ys bs
```

Intersection of a list of relations, given as partitions.

```
intersectRels :: Ord a => [Erel a] -> Erel a
intersectRels = foldl1' intersectRel
```

Call the coarsest refinement of a cover that is a partition the *fission* of the cover. The following function computes this partition, by splitting all blocks in the cover that have elements in common.

```
fission :: Ord a => Erel a -> Erel a
fission [] = []
fission (b:bs) = let
   xs = filter (overlap b) bs
   ys = bs \\ xs
   zs = [ [b\\c, c\\b, b `intersect` c] | c <- xs ]
   us = filter (/= []) $ concat zs
 in
   if xs == [] then b: fission bs
   else fission (us++ys)
```

Proposition 4 *Any similarity is a union of equivalences.*

Proof. Let S be a similarity (reflexive and symmetric relation) on X. For any $x \in X$, define the relation S_x as

$$S_x = \{(y, z) \in X^2 \mid y = z \vee (xSy \wedge xSz \wedge ySz)\}.$$

Then each S_x is an equivalence relation on X, and $S = \bigcup_{x \in X} S_x$. □

It is immediately clear from the representation of similarities as covers that the fission of a cover corresponds to the intersection of the set of all maximal equivalences contained in a relation. More precisely, $\check{R}$, the fission of R, is given by

$$\check{R} = \bigcap \{S \mid S \subseteq U \subseteq R \mid S \text{ equiv and if } S \neq U \text{ then } U \text{ not equiv }\}.$$

If a cover represents R then the fission of the cover is a partition that represents $\check{R}$.

Question: can $\check{R}$ be defined from R in terms of first order operations plus transitive closure?

Chapter 4

Product of Equivalences

For product update of an S5 model with an S5 action model, we need a definition of the product of two equivalence relations (partitions). The product of two equivalences is an equivalence. Here is the corresponding partition:

```
prod :: Erel a -> Erel b -> Erel (a,b)
prod r s = [ [ (x,y) | x <- b, y <- c ] | b <- r, c <- s ]
```

This gives:

```
*EREL> prod [[1,2],[3,4]] [[1,3],[2,4]]
[[(1,1),(1,3),(2,1),(2,3)],[(1,2),(1,4),(2,2),(2,4)],
 [(3,1),(3,3),(4,1),(4,3)],[(3,2),(3,4),(4,2),(4,4)]]
```

For product update in the sense of [1], this has to be combined with a check whether the pairs satisfy a restriction:

```
restrictedProd :: (Eq a,Eq b) =>
                  Erel a -> Erel b -> [(a,b)] -> Erel (a,b)
restrictedProd r s domain =
  [ [ (x,y) | x <- b, y <- c, elem (x,y) domain ] |
              b <- r, c <- s ]
```

Applying this to two tables of partitions:

```
restrictedProdT :: (Eq a, Ord b, Ord c) =>
         [(a,Erel b)] -> [(a,Erel c)]
          -> [(b,c)] -> [(a,Erel (b,c))]
restrictedProdT table1 table2 domain =
   [ (i, restrictedProd r s domain) |
      (i,r) <- table1, (j,s) <- table2, i == j ]
```

It is useful to be able to convert tables of type `[(a,Erel (b,c))]` again into tables of type `[(a,Erel d)]`. This can be done with the following universal conversion function for partition tables.

```
convert :: Ord a => [b] -> [(t, Erel a)] -> [(t, Erel b)]
convert newstates table = let
   sts = states table
   f = apply (zip sts newstates)
 in
   [ (a,map (map f) erel) | (a,erel) <- table ]
```

Chapter 5

Constraints on Equivalence Relations

From a list of equality constraints to a partition:

```
eqs2erel :: Ord a => Rel a -> Erel a
eqs2erel = rel2erel
```

From a list of inequalities to a check on a partition:

```
ineqsProp :: Ord a => Rel a -> Erel a -> Bool
ineqsProp [] _ = True
ineqsProp ((x,y):pairs) p = let
    domain = domE p
    check (u,v) =
      elem u domain && elem v domain
        ==> bl p u /= bl p v
  in
    check (x,y) && ineqsProp pairs p
```

Use this to check that a list of equality constraints and a list of inequality constraints are consistent with each other.

```
consistentCs :: Ord a => Rel a -> Rel a -> Bool
consistentCs eqs ineqs =
  ineqsProp ineqs (eqs2erel eqs)
```

This gives, e.g.:

```
*EREL> consistentCs [(1,2),(2,3),(4,5)] [(1,4)]
True
*EREL> consistentCs [(1,2),(2,3),(4,5)] [(1,3)]
False
```

Chapter 6

Generated Subdomain, Bisimulation Closure

The `fusion` function can be used to find the domain of a submodel generated from an element x and a set of equivalences: the set (or, in the implementation, list) of all elements that are reachable from x via the equivalences.

```
gendomain :: Ord a => a -> [Erel a] -> [a]
gendomain x rs = head (fusion $ [x]: concat rs)
```

Applying this to a table of partitions:

```
genD :: Ord a => a -> [(b,Erel a)] -> [a]
genD x = gendomain x . map snd
```

To compute bisimulation minimal partition tables, we need to start out from a valuation. A valuation can be any function, represented as a table. Here is a check whether two worlds have the same valuation:

```
sameVal :: (Eq a,Eq b) => [(a,b)] -> a -> a -> Bool
sameVal = fct2equiv.apply
```

The problem of finding the smallest Kripke model modulo bisimulation is similar to the problem of minimizing the number of states in a finite automaton [4]. In the particular case where all the relations are equivalences, all we need as input are a partition table and a valuation table. The algorithm for finding a bisimulation minimal version of the partition table uses partition refinement, in the spirit of [6]. Note: in our particular application, we will construct a 'table of partitions of partitions', i.e., a table of type `[(a,Erel [b])]`.

Initializing a partition by putting worlds that have the same valuation in the same block. The valuation is given by a table.

```
initPartition :: (Eq a,Eq b) => [(a,b)] -> [a] -> [[a]]
initPartition = fct2erel.apply
```

If `a` is the type of agents, then `[(a,Erel b)]` is the type of a table with equivalence partitions, one for each agent. Agents of a table, on the assumption that the table is non-empty:

```
agents :: [(a,Erel b)] -> [a]
agents = map fst
```

The states in a table, again on the assumption that the table is non-empty:

```
states :: Ord b => [(a,Erel b)] -> [b]
states = domE.snd.head
```

Accessible blocks for an agent, given a relation table, and an actual state:

```
accBlocks :: (Eq a,Eq b) =>
           [(a,Erel b)] -> [[b]] -> a -> b -> [[b]]
accBlocks table part ag s = let
    rel = apply table ag
    xs  = bl rel s
  in
    nub [ bl part x | x <- xs ]
```

The relation of having the same accessible blocks (for a list of agents given by a table), given a partition.

```
sameAB :: (Eq a,Eq b) =>
          [(a,Erel b)] -> [[b]] -> b -> b -> Bool
sameAB table part s t = let
   ags = agents table
   f   = accBlocks table part
 in
   and [ f ag s == f ag t | ag <- ags ]
```

If two worlds in a block b have the same accessible blocks, then there is no need to split the block b; otherwise there is. This block splitting refines the partition.

So a refine step in the partition refinement algorithm works like this: split the blocks in the current partition, using the `sameAB` relation to do the splitting.

```
refineStep :: (Eq a,Eq b) =>
              [(a,Erel b)] -> [[b]] -> [[b]]
refineStep table p = let
   f bl =
    (cfct2erel bl (sameAB table p) ++)
 in
   foldr f [] p
```

Carry out refinement steps until the process stabilizes, using least fixpoint:

```
refine :: (Eq a,Eq b) =>
          [(a,Erel b)] -> [[b]] -> [[b]]
refine table = lfp (refineStep table)
```

Miminize a table of partitions, given a valuation.

```
minimize :: (Eq a, Ord b, Eq c) =>
            [(a,Erel b)] -> [(b,c)] -> [(a,Erel [b])]
minimize table val = let
   sts   = states table
   initP = initPartition val sts
   sts'  = refine table initP
   f     = bl sts'
 in
   [ (a, map (nub.(map f)) erel) | (a, erel) <- table ]
```

Example:

```
table = [(1,[[1,2,3],[4,5,6]]), (2,[[1,2,3,4,5,6]])]
val   = map (\ n -> (n,even n)) [1..6]

mini = minimize table val
```

This gives:

```
*EREL> mini
[(1,[[[1,3],[2]],[[4,6],[5]]]),(2,[[[1,3],[2],[4,6],[5]]])]
```

Note that in the output we have blocks of worlds rather than worlds as ingredients of epistemic partitions.

Computing the bisimulation-minimal version of a table of partitions, given a valuation. Definition in point-free style, but see the line that is commented out:

```
bisim :: (Eq a, Enum a, Ord b, Eq c, Num d, Enum d) =>
         [(a, Erel b)] -> [(b,c)] -> [(a, Erel d)]
--bisim table f = convert [0..] $ minimize table f
bisim = (convert [0..] .)  .  minimize
```

Example again:

```
example = bisim table val
```

The result:

```
*EREL> example
[(1,[[0,1],[2,3]]),(2,[[0,1,2,3]])]
```

Bibliography

[1] A. Baltag, L.S. Moss, and S. Solecki. The logic of public announcements, common knowledge, and private suspicions. In I. Bilboa, editor, *Proceedings of TARK'98*, pages 43–56, 1998.

[2] Garrett Birkhoff. *Lattice Theory*. American Mathematical Society, 1995.

[3] The Haskell Team. The Haskell homepage. `http://www.haskell.org`.

[4] J.E.Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Zvi Kohavi and Azaria Paz, editors, *Theory of Machines and Computations*. Academic Press, 1971.

[5] D.E. Knuth. *Literate Programming*. CSLI Lecture Notes, no. 27. CSLI, Stanford, 1992.

[6] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6):973–989, 1987.