Module Rellens_types

attributes as strings

```
type attr = string

module Attr = struct
  type t = attr
(* let compare = Pervasives.compare *) (* 4.07 and earlier *)
  let compare = Stdlib.compare
end
```

set of attributes, ranged over by A, B, C, as OCaml set of attr

```
module SetofAttr = Set.Make(Attr)
```

homogeneous set of values, ranged over by a, b, c

```
type value =
    Int of int
  | Flt of float
  | Str of string
  | Bol of bool
```

comparator of value

```
let compare_value : value → value → int = (* Pervasives.compare *) Stdlib.compare

module MapofAttr = Map.Make(Attr)
```

Records, ranged over by m, n, l, partial functions from attributes to values, as OCaml map from attr to value

```
type record = value MapofAttr.t
```

Relation, ranged over by M, N, L, as sets of records

```
module Record = struct
  type t = record
  let compare = MapofAttr.compare compare_value
end

module SetofRecord = Set.Make(Record)

type relation = SetofRecord.t
```

expressions for predicates

```
type phrase =
    PCns of value (* constant *)
  | PAnd of phrase × phrase (* conjunction *)
  | POr of phrase × phrase (* disjunction *)
  | PNot of phrase (* negation *)
  | PVar of attr (* attribute reference *)
  | PLt of phrase × phrase (* < *)
  | PGt of phrase × phrase (* > *)
  | PLte of phrase × phrase (* ≤ *)
  | PGte of phrase × phrase (* ≥ *)
  | PEq of phrase × phrase (* = *)
  | PCase of
      ((phrase × phrase) list) (* when ... then *)
    × phrase (* else *)
```

phrase_map: apply function f to every node in phrase p

```
let rec phrase_map (f : phrase → phrase) p : phrase = match p with
    PCns _ | PVar _ → f p
  | PAnd (p1, p2) → f (PAnd (phrase_map f p1, phrase_map f p2))
  | POr (p1, p2) → f (POr (phrase_map f p1, phrase_map f p2))
  | PLt (p1, p2) → f (PLt (phrase_map f p1, phrase_map f p2))
  | PGt (p1, p2) → f (PGt (phrase_map f p1, phrase_map f p2))
  | PLte (p1, p2) → f (PLte (phrase_map f p1, phrase_map f p2))
  | PGte (p1, p2) → f (PGte (phrase_map f p1, phrase_map f p2))
  | PEq (p1, p2) → f (PEq (phrase_map f p1, phrase_map f p2))
  | PNot p1 → f (PNot (phrase_map f p1))
  | PCase (when_clause_list, else_clause) →
      f (PCase
        (List.map (fun (p1, p2) → (phrase_map f p1, phrase_map f p2)) when_clause_list,
         phrase_map f else_clause))
```

phrase type

```
type ptype =
    TInt (* int *)
  | TFlt (* float *)
  | TStr (* string *)
  | TBol (* bool *)
```

type environment : maps attr to ptype

```
type tenv = ptype MapofAttr.t
```

simple FD representation : pair of sets of attributes

```
type fd = SetofAttr.t × SetofAttr.t

let compare_fd (fd1 : fd) (fd2 : fd) : int =
  let ((x, y), (x', y')) = (fd1, fd2) in
  let c = SetofAttr.compare x x' in
  if c = 0
  then SetofAttr.compare y y'
  else c
```

set of fd

```
module FD = struct
  type t = fd
  let compare = compare_fd
end

module SetofFD = Set.Make(FD)
```

set of set of attributes

```
module PSetofAttr = Set.Make(SetofAttr)
```

set map

```
let setmap_PSetofAttr (f : SetofAttr.t → PSetofAttr.t) (ss : PSetofAttr.t) =
    PSetofAttr.fold (fun s → PSetofAttr.union (f s)) ss PSetofAttr.empty
```

map of nodes

```
module MapofSetofAttr = Map.Make(SetofAttr)
```

convert function f to a finite map on given domain D
$\{v \mapsto f(v) \mid v \leftarrow D\}$

```
let f2map_PSetofAttr (f : SetofAttr.t → α) (ss : PSetofAttr.t) : α MapofSetofAttr.t =
  PSetofAttr.fold (fun (s : SetofAttr.t) →
          MapofSetofAttr.add (s : SetofAttr.t) (f s)) ss MapofSetofAttr.empty
```

relation name

```
type rname = string
```

database instances, ranged over by I, J, are finite maps from relation names to relations

```
module Rname = struct
  type t = rname
  let compare = (* Pervasives *)Stdlib.compare
end

module MapofRname = Map.Make(Rname)
```

sort : quadruple of domain U, (attribute type environment), predicate P and functional dependencies F

```
type sort = SetofAttr.t × tenv × phrase × SetofFD.t
```

database: map from rname to pair of sort and relation

```
type database = (sort × relation) MapofRname.t
```

change set
multiplicity mult: delete, keep, insert

```
type mult =
    Delete (* -1 *)
  | Keep (* 0 *)
  | Insert (* +1 *)
```

change entry as a pair of record (Horn's row) and multiplicity

```
type change_entry = record × mult

let compare_change_entry (ce1 : change_entry) (ce2 : change_entry) : int =
    let ((rec1, mul1), (rec2, mul2)) = (ce1, ce2) in
    let c = MapofAttr.compare compare_value rec1 rec2 in
    if c = 0
    then (* Pervasives *)Stdlib.compare mul1 mul2
    else c

module ChangeEntry = struct
  type t = change_entry
  let compare = compare_change_entry
end

module SetofChange = Set.Make(ChangeEntry)

module PRecord = struct
   type t = record × record
  let compare (rec1, rec2) (rec1', rec2') =
    let c = MapofAttr.compare compare_value rec1 rec1' in
    if c = 0
    then MapofAttr.compare compare_value rec2 rec2'
    else c
end

module SetofPRecord = Set.Make(PRecord)
```

pair of FD and set of pair of records

```
module FDSPRecord = struct
  type t = fd × SetofPRecord.t
  let compare (fd, s) (fd′, s′) =
    let c = compare_fd fd fd′ in
    if c = 0
    then SetofPRecord.compare s s′
    else c
end
```

(fd * ((record * record) set)) set used for update set

```
module SetofFDSPRecord = Set.Make(FDSPRecord)
```

tentative type definition for lens in the thesis

```
type ilens = sort × relation
```

$v \in \Sigma \leftrightarrow \Delta$

```
type lens =
    Select of rname × phrase × rname (* select from R where P as S *)
  | JoinDL of (rname × rname) × rname (* join_dl R, S as T *)
  | Drop of attr × ((attr list) × value) × rname × rname
                (* drop A determined by (X, a) from R as S *)
  | Compose of lens × lens
```

Module Rellens

Relational Lenses by Bohannon et al. and its incremental version by Horn

```
open Rellens_types
open Print
```

domain of a record m
$dom(m) = \{A \mid (A \mapsto a) \in m\}$

```
let dom (m : record) : SetofAttr.t =
  SetofAttr.of_list (List.map fst (MapofAttr.bindings m))
```

attribute access $m(A)$

```
let attribute (a : attr) (m : record) : value =
  MapofAttr.find a m
```

record extension $m[A \to a]$
$m[A \to a] = m \cup \{A \mapsto a\}$

```
let extend (a : attr) (v : value) (m : record) : record =
  MapofAttr.add a v m
```

record restriction $m[X]$
`restrict` $X\ m = \{A \mapsto a \mid (A \mapsto a) \in m, A \in X\}$

```
let restrict (x : SetofAttr.t) (m : record) : record =
   MapofAttr.filter (fun key a → SetofAttr.mem key x) m
```

$M : U$ – relation M has domain U
dynamic check

```
let of_domain (u : SetofAttr.t) (m : relation) : bool =
   SetofRecord.for_all (fun r → (SetofAttr.equal u (dom r))) m
```

relational projection $M[X]$ as a restriction lifted to relation
$\{m[X] \mid m \in M\}$

```
let restrict_relation (x : SetofAttr.t) (m : relation) : relation =
   SetofRecord.fold (fun r → SetofRecord.add (restrict x r)) m SetofRecord.empty
```

record equivalence

```
let record_equal (m : record) (m′ : record) : bool =
  MapofAttr.equal (=) m m′
```

natural join $M \bowtie N$ by simple nested loop join

```
let nat_join (m : relation) (n : relation) : relation =
  if (SetofRecord.is_empty m) then SetofRecord.empty
  else
    if (SetofRecord.is_empty n) then SetofRecord.empty
    else
      let uS = dom (SetofRecord.choose m) in
      let vS = dom (SetofRecord.choose n) in
      if ((of_domain uS m) ∧ (of_domain vS n))
      then
        let iS = SetofAttr.inter uS vS in
        SetofRecord.fold (fun rm rms →
          SetofRecord.union
          (let rm′ = restrict iS rm in
          (SetofRecord.fold (fun rn rns →
            let rn′ = restrict iS rn in
            if record_equal rm′ rn′
            then SetofRecord.add (MapofAttr.merge
                                        (fun k a b → match (a, b) with
```

```
                                          (Some a, Some b) → if a = b then Some a else failwith "mism
                                        | (Some a, None ) → Some a
                                        | (None, Some b) → Some b
                                        | (None, None) → None) rm rn) rns
          else rns) n rms)) rms) m SetofRecord.empty
    else SetofRecord.empty

let lookup_type (a : attr) (env : tenv) : ptype =
   try (MapofAttr.find a env) with
     Not_found →
       Format.fprintf Format.err_formatter "lookup_type␣%a␣not␣found␣in␣type␣env␣%a@."
         pp_attr a pp_tenv env;
       raise Not_found

let merge_tenv (tenv1 : tenv) (tenv2 : tenv) : tenv =
  MapofAttr.merge
     (fun k a b → match (a, b) with
       (Some a, Some b) → if a = b then Some a else failwith "mismatch"
     | (Some a, None ) → Some a
     | (None, Some b) → Some b
     | (None, None) → None) tenv1 tenv2
```

trivial type inference

```
let rec qtype (env : tenv) (p : phrase) : ptype = match p with
  PCns (Int _ ) → TInt
| PCns (Flt _ ) → TFlt
| PCns (Str _ ) → TStr
| PCns (Bol _ ) → TBol
| PVar a → lookup_type a env
| PCase (wt_phrase_list, else_phrase) →
    (match wt_phrase_list with
      [] → qtype env else_phrase
    | (w, t) :: wts →
        let check_when p =
          if qtype env p ≠ TBol then failwith "qtype:␣non-boolean␣WHEN␣in␣CASE" in
        check_when w;
        let tt = qtype env t in
        let rec qt accum_type l =
          (match l with
            [] →
              if accum_type = qtype env else_phrase then accum_type else
              failwith "qtype:␣invalid␣else␣type␣in␣CASE"
```

```
        | (w, t) :: wts  →
            check_when w;
            if accum_type  =  qtype env t then qt accum_type wts
            else failwith "qtype: invalid then clause in CASE") in
      qt tt wts)
| PAnd (p1, p2) | POr (p1, p2)  →
    let t1, t2  =  qtype env p1, qtype env p2 in
    (match t1, t2 with
      TBol, TBol  →  TBol
    | _  →  failwith "qtype: invalid operand type for boolean")
| PNot p1  →
    let t1  =  qtype env p1 in
    (match t1 with
      TBol  →  TBol
    | _  →  failwith "qtype: invalid operand type for boolean")
| PLt (p1, p2) | PGt(p1, p2) | PLte(p1, p2) | PGte(p1, p2)  →
    let t1, t2  =  qtype env p1, qtype env p2 in
    (match t1, t2 with
      TInt, TInt | TFlt, TFlt | TStr, TStr  →  TBol
    | _  →  failwith "qtype: invalid operand type for comparison")
| PEq (p1, p2)  →
    let t1, t2  =  qtype env p1, qtype env p2 in
    if t1  =  t2 then TBol else failwith "qtype: invalid operand type for comparison"
```

$[\![p]\!]_r$: interpret phrase p on record r

```
let rec eval (r : record) (p : phrase)  :  value  = match p with
  PCns v  →  v
| PAnd (p1, p2)  →
    let v1  =  eval r p1 in
    let v2  =  eval r p2 in
    (match (v1, v2) with
      (Bol b1, Bol b2)  →  Bol (b1  ∧  b2)
    | _  →  failwith "PAnd: non-boolean operand"
    )
| POr (p1, p2)  →
    let v1  =  eval r p1 in
    let v2  =  eval r p2 in
    (match (v1, v2) with
      (Bol b1, Bol b2)  →  Bol (b1  ∨  b2)
    | _  →  failwith "POr: non-boolean operand")
```

```
| PNot p1 →
    let v = eval r p1 in
    (match v with
       (Bol b) → Bol (¬ b)
    | _ → failwith "PNot: non-boolean operand"
    )
| PVar attr → MapofAttr.find attr r
| PLt (p1, p2) | PGt (p1, p2) | PLte (p1, p2) | PGte (p1, p2) →
    (let (v1, v2) = (eval r p1, eval r p2) in
     let value_comparison_op : value → value → bool =
        match p with
        | PLt _ → (<)
        | PGt _ → (>)
        | PLte _ → (≤ )
        | PGte _ → (≥ )
        | _ → failwith "eval: unexpected operator" in
     match (v1, v2) with
        (Int _, Int _)
     | (Flt _, Flt _)
     | (Str _, Str _) → (* use value type to rely on Stdlib.compare *)
                           Bol (value_comparison_op v1 v2)
     | (Bol _, Bol _) → failwith "comparison: boolean operands"
     | _, _ → failwith "comparison: mismatch operand")
| PEq (p1, p2) →
     (let (v1, v2) = (eval r p1, eval r p2) in Bol (v1 = v2) )
| PCase (wlist, else_phrase) → eval_case r wlist else_phrase
and eval_case (r : record) (wlist : (phrase × phrase) list) (else_phrase : phrase) : value =
  match wlist with
    (p1, p2) :: ps →
      let vp1 = eval r p1 in
     (match vp1 with
         Bol b →
           if b then eval r p2
           else eval_case r ps else_phrase
       | _ → failwith "non-boolean value in when condition")
  | [] → eval r else_phrase
```

evaluator specific to boolean predicates

```
let eval_bool (r : record) (p : phrase) : bool =
  let v = eval r p in
```

```
match v with Bol b → b
| _ → failwith (Format.sprintf "eval_bool: non-boolean result %s" (toStr pp_value v))
```

predicates, ranged over by P and Q

$\top_U$ (top) : set of all records over the domain U

negation of predicate : set complement

$\neg_U M : \top_U \setminus M$

$P \cap M$: relational selection

P *ignores* X : predicate P does not refer to attributes in X

```
let rec ignores (p : phrase) (x : SetofAttr.t) : bool = match p with
  PCns v → true
| PAnd (p1, p2) | POr (p1, p2) | PLt (p1, p2) | PGt (p1, p2) | PLte (p1, p2) |
 PGte (p1, p2)
| PEq (p1, p2) → ignores p1 x ∧ ignores p2 x
| PNot p1 → ignores p1 x
| PVar attr → ¬ (SetofAttr.mem attr x)
| PCase (wlist, else_phrase) →
    List.for_all (fun (w, t) → ignores w x ∧ ignores t x) wlist
      ∧ ignores else_phrase x
```

$M \models X \to Y$: M models functional dependency $X \to Y$

list of all pairs of a list

```
let list_pairs (l : α list) : (α × α) list =
   List.fold_right (fun e →
    (@) (List.map (fun e' → (e, e')) l)) l []
```

naive test of $M \models X \to Y$

by testing if $m[X] = m'[X] \Rightarrow m[Y] = m'[Y]$ for all $m, m' \in M$

```
let models (m : relation) ((x, y) : fd) : bool =
  let lp = list_pairs (SetofRecord.elements m) in
  List.for_all (fun (m, m') →
    if record_equal (restrict x m) (restrict x m')
    then record_equal (restrict y m) (restrict y m')
    else true) lp
```

$\models$ lifted to set of fd

$M \models F = \bigwedge_{fd \in F} M \models fd$

```
let models_fds (m : relation) (fds : SetofFD.t) : bool =
  SetofFD.for_all (models m) fds
```

functions on functional dependencies

$\mathit{left}(F) = \cup\{X \mid X \to Y \in F\}$

```
let left (fds : SetofFD.t) : SetofAttr.t =
    SetofFD.fold (fun (x, y) →
      SetofAttr.union x) fds SetofAttr.empty
```

$right(F) = \cup\{Y \mid X \rightarrow Y \in F\}$

```
let right (fds : SetofFD.t) : SetofAttr.t =
    SetofFD.fold (fun (x, y) →
      SetofAttr.union y) fds SetofAttr.empty
```

$names(F) = left(F) \cup right(F)$

```
let names (fds : SetofFD.t) : SetofAttr.t =
   SetofAttr.union (left fds) (right fds)
```

$outputs(F) = \{A \in U \mid \exists X \subseteq U.A \notin X \text{ and } F \models X \rightarrow A\}$

```
let outputs (fds : SetofFD.t) : SetofAttr.t =
  SetofAttr.filter (fun attr →
    SetofFD.exists (fun (x, y) →
      (SetofAttr.mem attr y) ∧ (¬ (SetofAttr.mem attr x))) fds) (right fds)
```

(literal) tree form
distinctness of sources and destinations of FD

```
exception Overlap_Between_Attributes (* overlap between sets of attributes *)
```

```
let merge_attrs (attrs : SetofAttr.t) (attrss : PSetofAttr.t) : PSetofAttr.t =
  if SetofAttr.is_empty attrs then failwith "empty set of attributes"
  else
    if (PSetofAttr.mem attrs attrss)
    then attrss
    else
      if PSetofAttr.for_all
          (fun s → SetofAttr.is_empty (SetofAttr.inter s attrs)) attrss
      then PSetofAttr.add attrs attrss
      else raise Overlap_Between_Attributes
```

```
exception Is_Cyclic
exception Multiple_Indegree
```

set of nodes (distinctness test)

```
let nodes (fds : SetofFD.t) : PSetofAttr.t =
  SetofFD.fold (fun (x, y) ss →
      (merge_attrs y (merge_attrs x ss))) fds PSetofAttr.empty
```

```
let fwd (v : SetofAttr.t) (fds : SetofFD.t) : PSetofAttr.t =
   let edges = fds in
   SetofFD.fold (fun (x, y) ss →
    if SetofAttr.equal v x then PSetofAttr.add y ss
    else ss) edges PSetofAttr.empty
let bwd (v : SetofAttr.t) (fds : SetofFD.t) : PSetofAttr.t =
   let edges = fds in
   SetofFD.fold (fun (x, y) ss →
    if SetofAttr.equal v y then PSetofAttr.add x ss
    else ss) edges PSetofAttr.empty
```

tree form test: check if a set of FDs is in tree form

```
let is_tree (fds : SetofFD.t) : bool =
  let it () : bool =
    let nS : PSetofAttr.t = nodes fds in
    let _ =
      PSetofAttr.iter (fun (v : SetofAttr.t) →
          if PSetofAttr.cardinal (bwd v fds) > 1 then raise Multiple_Indegree) nS in
    let is_cyclic (fds : SetofFD.t) (nS : PSetofAttr.t) : bool =
      let fwd v = fwd v fds in
      (* finite map from left hand side of FD to the set of right hand sides of FD *)
```

(* $\{v_1 \mapsto fwd(v_1), v_2 \mapsto fwd(v_2), \ldots, v_n \mapsto fwd(v_n)\}$ *)

```
      let initial_closure_map : PSetofAttr.t MapofSetofAttr.t = f2map_PSetofAttr fwd nS in
```

(* extend the closure by the sets of attributes obtained by one time application of fwd *)

```
      let extend_by_fwd (closure : PSetofAttr.t) : PSetofAttr.t =
        setmap_PSetofAttr (fun v → PSetofAttr.add v (fwd v)) closure
      in
```

(* complete the map from the node to its irreflexive transitive closure of fwd by repeating one step extension of the closure $|nS|$ times (number of nodes) to the initial closure map *)

```
      let closure_map =
        PSetofAttr.fold (fun _ → MapofSetofAttr.map extend_by_fwd) nS initial_closure_map in
```

(* $\vee\{v \in closure(v) \mid (v \mapsto closure(v)) \in closure\_map\}$ *)

(* $= \bigvee_{(v \mapsto closure(v)) \in closure\_map} v \in closure(v)$ *)

```
      MapofSetofAttr.exists PSetofAttr.mem closure_map in
    if is_cyclic fds nS then raise Is_Cyclic else
    true in
    try (it ()) with
```

```
    Is_Cyclic →
      Format.fprintf Format.err_formatter "%s@." "cyclic functional dependency";
      false
  | Overlap_Between_Attributes →
      Format.fprintf Format.err_formatter "%s@." "overlap between sets of attributes";
      false
  | Multiple_Indegree →
      Format.fprintf Format.err_formatter "%s@." "more than one in-degree";
      false
```

exception *Not_in_Tree_Form*

$leavs(F)$: nodes that have no overlap with left parts of F
$\{Y \mid \exists X.X \to Y \in F \text{ and } Y \cap \mathit{left}(F) = \emptyset\}$

```
let leaves (fds : SetofFD.t) : PSetofAttr.t =
  if ¬ (is_tree fds) then raise Not_in_Tree_Form else
  let leftS = left fds in
  SetofFD.fold (fun (x, y) ss →
    if SetofAttr.is_empty (SetofAttr.inter leftS y)
    then PSetofAttr.add y ss
        else ss) fds PSetofAttr.empty
```

$roots(F)$: nodes that have no overlap with the right parts of F
$\{Y \mid \exists X.X \to Y \in F \text{ and } Y \cap \mathit{right}(F) = \emptyset\}$

```
let roots (fds : SetofFD.t) : PSetofAttr.t =
  if ¬ (is_tree fds) then raise Not_in_Tree_Form else
  let rightS = right fds in
  SetofFD.fold (fun (x, y) ss →
    if SetofAttr.is_empty (SetofAttr.inter rightS x)
    then PSetofAttr.add x ss
        else ss) fds PSetofAttr.empty
```

right-biased combination of records m and n
$m \leftarrow\!\!+ n$
$m = \{A \to a1, B \to b1, C \to c1\}$
$n = \{A \to a2, B \to b2, D \to d1\}$
$m \leftarrow\!\!+ n = \{A \to a2, B \to b2, C \to c1, D \to d1\}$

```
let rbcr (m : record) (n : record) : record =
  MapofAttr.merge (fun k a b → match (a, b) with
  (Some a, Some b) → Some b (* dom(m) ∩ dom(n) : right bias *)
| (None, Some b) → Some b (* dom(n) : agree with n *)
| (Some a, None ) → Some a (* dom(m) \ dom(n) : agree with m *)
```

| $(None, None)$ → $None$) m n

single-dependency record revision

$m \xrightarrow[N]{X \to Y} m'$

```
let sdrr (m : record) ((x, y) : fd) (n : relation) : record =
    if ¬ (models n (x, y))
    then failwith (Format.sprintf "relation %s does not satisfy functional dependency %s"
                        (toStr pp_relation n) (toStr pp_fd (x, y)))
    else
      let mX = restrict x m in
      let n′ = SetofRecord.filter
          (fun rn → record_equal mX (restrict x rn)) n in
      let n′ = restrict_relation (SetofAttr.union x y) n′ in
      match (SetofRecord.cardinal n′) with
        1 → rbcr m (restrict y (SetofRecord.choose n′)) (* (C-Match) *)
      | 0 → m (* C-NoMatch *)
      | _ → failwith (Format.sprintf
            "multiple match %s under functional dependency %s@."
                        (toStr pp_relation n′) (toStr pp_fd (x, y)))
```

record revision of m to n under set of functional dependencies F

$m \xRightarrow[L]{F} n$

```
let rec record_revision (m : record) (fds : SetofFD.t) (l : relation) : record =
  if SetofFD.is_empty fds
  then m (* (FC-Empty) *)
  else (* (FC-Step) *)
    if ¬ (models_fds l fds) then failwith
      (Format.sprintf
          "relation %s does not satisfy functional dependencies %s@."
        (toStr pp_relation l) (toStr pp_fds fds))
    else (* L ⊨ F, X → Y *)
      if ¬ (is_tree fds) then raise Not_in_Tree_Form
      else (* F, X → Y is in tree form *)
        (* choose an FD with root on its left *)
        let fd =
          let rS = roots fds in
          SetofFD.choose
            (SetofFD.filter (fun (x′, y′) →
              PSetofAttr.mem x′ rS) fds) in
        let _ = Format.fprintf Format.err_formatter "x->y = %a@." pp_fd fd in
```

```
let f = SetofFD.remove fd fds in
let _ = Format.fprintf Format.err_formatter "f = %a@." pp_fds f in
let m' = sdrr m fd l in
    record_revision m' f l
```

relation revision
$M \leftarrow_F L = \{m' \mid m \xRightarrow[L]{F} m' \text{ for some } m \in M\}$

```
let relation_revision (m : relation) (fds : SetofFD.t) (l : relation) : relation =
  SetofRecord.fold (fun m ms →
    SetofRecord.add (record_revision m fds l) ms) m SetofRecord.empty
```

relational merge
$M \overset{\cup}{\leftarrow}_F L = M \leftarrow_F L \cup L$

```
let relational_merge (m : relation) (fds : SetofFD.t) (l : relation) : relation =
  SetofRecord.union (relation_revision m fds l) l
```

sort function $sort(R)$

```
let sort (r : rname) (db : database) : sort =
  fst (MapofRname.find r db)
```

instance $I(R)$

```
let instance (r : rname) (db : database) : relation =
  snd (MapofRname.find r db)
```

functional dependencies of relation $R : fd(R)$

```
let fd (r : rname) (db : database) : SetofFD.t =
  let ((_, _, _, fds), _) = MapofRname.find r db in
  fds
```

select lens

$$\frac{\begin{array}{c} sort(R) = (U, Q, F) \\ sort(S) = (U, P \cap Q, F) \\ F \text{ is in tree form} \quad Q \text{ ignores } outputs(F) \end{array}}{\begin{array}{c} \texttt{select from } R \texttt{ where } P \texttt{ as } S \in \\ \Sigma \uplus \{R\} \Leftrightarrow \Sigma \uplus \{S\} \end{array}} \text{ (T-SELECT)}$$

type check and inference for select

```
let typeinf_select (p : phrase) (srt : sort) : sort =
  let (u, tenv, q, fds) = srt in
  if qtype tenv p ≠ TBol then failwith
        (Format.sprintf
          "typeinf_select: non-boolean predicate %s@." (toStr pp_phrase p)) else
   if ¬ (is_tree fds) (* check tree form *) then raise Not_in_Tree_Form else
   let o_fds = outputs fds in
   if ¬ (ignores q o_fds) (* check if Q ignores outputs(F) *) then failwith
       (Format.sprintf "predicate %s does not ignore outputs(%s)=%s@."
          (toStr pp_phrase q) (toStr pp_fds fds) (toStr pp_SetofAttr o_fds)) else
     (u, tenv, PAnd (p, q), fds)
```

get part of select lens

$v \nearrow (I) = I \setminus_R [S \mapsto P \cap I(R)]$

```
let get_select (r : rname) (p : phrase) (s : rname) (db : database) : database =
   let (sortR, relation) = MapofRname.find r db in
   let sortS = typeinf_select p sortR in
   let db' = MapofRname.remove r db in
   MapofRname.add s (sortS, SetofRecord.filter
                              (fun r → eval_bool r p) relation) db'
```

put part of select lens

$$\begin{array}{rrcl} & v \searrow (J, I) & = & J \setminus_S [R \mapsto M_0 \setminus N_\#] \\ \text{where} & M_0 & = & (\neg P \cap I(R)) \overset{\cup}{\leftarrow}_F J(S) \\ & N_\# & = & (P \cap M_0) \setminus J(S) \\ & F & = & fd(R) \end{array}$$

```
let put_select (r : rname) (p : phrase) (s : rname)
     ((dbJ, dbI) : (database × database)) : database =
  let (sortR, iR) = MapofRname.find r dbI in
  let (_, _, _, fds) = sortR in
  let iR' = SetofRecord.filter (fun r → eval_bool r (PNot p)) iR in
  let (_ , jS) = MapofRname.find s dbJ in
  let m0 = relational_merge iR' fds jS in
  let n_sharp = SetofRecord.diff (SetofRecord.filter (fun r → (eval_bool r p)) m0) jS in
  let dbI' = MapofRname.remove s dbJ in
   MapofRname.add r (sortR, SetofRecord.diff m0 n_sharp) dbI'
```

join of two predicates $P \bowtie Q$

$p, q : record \rightarrow bool$

$p \bowtie q$ is true for a record r in $\{rp \mid r \in U, p\ rp\} \bowtie \{rq \mid rq \in V, q\ rq\}$

$[\![P \bowtie Q]\!]_r$ is true for a record r in $\{r \mid r \in U, [\![P]\!]_r\} \bowtie \{r \mid r \in V, [\![Q]\!]_r\}$

Example
$p: \quad a \times b \times c \qquad\qquad \to bool$
$q: \qquad\qquad\quad c \quad \times d \times e \to bool$
for overlapped atributes, they are equal.
so $p \bowtie q = \lambda r \to p\ r[U] \land q\ r[V]$
$[\![P \bowtie Q]\!]_r = [\![P]\!]_r \land [\![Q]\!]_r$

$$\frac{\begin{array}{c} sort(R) = (U, P, F) \quad sort(S) = (V, Q, G) \\ sort(T) = (UV, P \bowtie Q, F \cup G) \\ G \models U \cap V \to V \\ F \text{ is in tree form} \quad G \text{ is in tree form} \\ P \text{ ignores } outputs(F) \quad Q \text{ ignores } outputs(G) \end{array}}{\texttt{join\_dl}\ R, S\ \texttt{as}\ T\ \in\ \Sigma \uplus \{R, S\} \Leftrightarrow \Sigma \uplus \{T\}} \text{ (T-JOIN)}$$

type check and inference for join

```
let typeinf_join_dl (sortR : sort) (sortS : sort) : sort =
  let ((uR, tenvR, predR, fdsR)) = sortR in
  if ¬ (is_tree fdsR) (* check tree form *) then raise Not_in_Tree_Form else
   let o_fdsR = outputs fdsR in
   if ¬ (ignores predR o_fdsR) (* check if P ignores outputs(F) *) then failwith
      (Format.sprintf "predicate %s does not ignore outputs(%s)=%s@."
         (toStr pp_phrase predR) (toStr pp_fds fdsR) (toStr pp_SetofAttr o_fdsR)) else
  let ((uS, tenvS, predS, fdsS)) = sortS in
  if ¬ (is_tree fdsS) (* check tree form *) then raise Not_in_Tree_Form else
   let o_fdsS = outputs fdsS in
   if ¬ (ignores predS o_fdsS) (* check if Q ignores outputs(G) *) then failwith
      (Format.sprintf "predicate %s does not ignore outputs(%s)=%s@."
         (toStr pp_phrase predS) (toStr pp_fds fdsS) (toStr pp_SetofAttr o_fdsS)) else
  let sortT = (SetofAttr.union uR uS,
                   merge_tenv tenvR tenvS,
                                      PAnd (predR, predS)
                         ,
                         SetofFD.union fdsR fdsS) in
  sortT
```

get part of join lens
$v \nearrow (I) = I\backslash_{R,S}[T \mapsto I(R) \bowtie I(S)]$
TODO: check if $G \models U \cap V \to V$

```
let get_join_dl (r : rname) (s : rname) (t : rname) (db : database) : database =
  let (sortR, iR) = MapofRname.find r db in
  let (sortS, iS) = MapofRname.find s db in
```

$\textsf{let } \mathit{sortT} = \mathit{typeinf\_join\_dl}\ \mathit{sortR}\ \mathit{sortS} \textsf{ in}$
$\textsf{let } \mathit{instanceT} = \mathit{nat\_join}\ \mathit{iR}\ \mathit{iS} \textsf{ in}$
$\textsf{let } \mathit{db} = \mathit{MapofRname.remove}\ r\ \mathit{db} \textsf{ in}$
$\textsf{let } \mathit{db} = \mathit{MapofRname.remove}\ s\ \mathit{db} \textsf{ in}$
$\mathit{MapofRname.add}\ t\ (\mathit{sortT}, \mathit{instanceT})\ \mathit{db}$

put part of join lens

$$\begin{array}{rrcl} & v \searrow (J, I) & = & J\backslash_T[R \mapsto M][S \mapsto N] \\ \text{where} & (U, P, F) & = & \mathit{sort}(R) \\ & (V, Q, G) & = & \mathit{sort}(S) \\ & M_0 & = & I(R) \overset{\cup}{\leftarrow}_F J(T)[U] \\ & N & = & I(S) \overset{\cup}{\leftarrow}_G J(T)[V] \\ & L & = & (M_0 \bowtie N) \setminus J(T) \\ & M & = & M_0 \setminus L[U] \end{array}$$

$\textsf{let } \mathit{put\_join\_dl}\ (r : \mathit{rname})\ (s : \mathit{rname})\ (t : \mathit{rname})$
$\quad ((\mathit{dbJ}, \mathit{dbI}) : (\mathit{database} \times \mathit{database})) : \mathit{database} =$
$\textsf{let } (\mathit{sortR}, \mathit{iR}) = \mathit{MapofRname.find}\ r\ \mathit{dbI} \textsf{ in}$
$\textsf{let } (u, \_, \_, f) = \mathit{sortR} \textsf{ in}$
$\textsf{let } (\mathit{sortS}, \mathit{iS}) = \mathit{MapofRname.find}\ s\ \mathit{dbI} \textsf{ in}$
$\textsf{let } (v, \_, \_, g) = \mathit{sortS} \textsf{ in}$
$\textsf{let } (\mathit{sortT}, \mathit{jT}) = \mathit{MapofRname.find}\ t\ \mathit{dbJ} \textsf{ in}$
$\textsf{let } \mathit{m0} = \mathit{relational\_merge}\ \mathit{iR}\ f\ (\mathit{restrict\_relation}\ u\ \mathit{jT}) \textsf{ in}$
$\textsf{let } n = \mathit{relational\_merge}\ \mathit{iS}\ g\ (\mathit{restrict\_relation}\ v\ \mathit{jT}) \textsf{ in}$
$\textsf{let } l = \mathit{SetofRecord.diff}\ (\mathit{nat\_join}\ \mathit{m0}\ n)\ \mathit{jT} \textsf{ in}$
$\textsf{let } m = \mathit{SetofRecord.diff}\ \mathit{m0}\ (\mathit{restrict\_relation}\ u\ l) \textsf{ in}$
$\textsf{let } \mathit{dbI}' = \mathit{MapofRname.remove}\ t\ \mathit{dbJ} \textsf{ in}$
$\textsf{let } \mathit{dbI}' = \mathit{MapofRname.add}\ r\ (\mathit{sortR}, m)\ \mathit{dbI}' \textsf{ in}$
$\textsf{let } \mathit{dbI}' = \mathit{MapofRname.add}\ s\ (\mathit{sortS}, n)\ \mathit{dbI}' \textsf{ in}$
dbI'

Union lens (not part of Bohannon's relational lenses, but necessary to encode our integration scenario)

$$\frac{\begin{array}{c} \mathit{sort}(R) = (U, P, F) \quad \mathit{sort}(S) = (V, Q, G) \\ \mathit{sort}(T) = (U, P \cup Q, F) \\ U = V \quad F \equiv G \\ F \text{ is in tree form} \quad G \text{ is in tree form} \quad \textit{(tentative)} \\ P \text{ ignores } \mathit{outputs}(F) \quad Q \text{ ignores } \mathit{outputs}(G) \quad \textit{(tentative)} \end{array}}{\texttt{union } R, S \texttt{ as } T \in \Sigma \uplus \{R, S\} \Leftrightarrow \Sigma \uplus \{T\}} \ (\text{T-UNION})$$

type check and inference for union

```
let typeinf_union (sortR : sort) (sortS : sort) : sort =
  let ((uR, tenvR, predR, fdsR)) = sortR in
  if ¬ (is_tree fdsR) (* check tree form *) then raise Not_in_Tree_Form else
   let o_fdsR = outputs fdsR in
   if ¬ (ignores predR o_fdsR) (* check if P ignores outputs(F) *) then failwith
      (Format.sprintf "predicate␣%s␣does␣not␣ignore␣outputs(%s)=%s@."
         (toStr pp_phrase predR) (toStr pp_fds fdsR) (toStr pp_SetofAttr o_fdsR)) else
  let ((uS, tenvS, predS, fdsS)) = sortS in
  if ¬ (is_tree fdsS) (* check tree form *) then raise Not_in_Tree_Form else
  let o_fdsS = outputs fdsS in
  if ¬ (ignores predS o_fdsS) (* check if Q ignores outputs(G) *) then failwith
      (Format.sprintf "predicate␣%s␣does␣not␣ignore␣outputs(%s)=%s@."
         (toStr pp_phrase predS) (toStr pp_fds fdsS) (toStr pp_SetofAttr o_fdsS)) else
  if ¬ (MapofAttr.equal (=) tenvR tenvS) then failwith
      (Format.sprintf "incompatible␣type␣environments:␣%s␣%s@." (toStr pp_tenv tenvR)
            (toStr pp_tenv tenvS)) else
  (* TODO: current check is syntactical equivalence, which is too strong *)
  if ¬ (SetofFD.equal fdsR fdsS) then failwith
      (Format.sprintf "incompatible␣FDs:␣%s␣%s@." (toStr pp_SetofFD fdsR)
            (toStr pp_SetofFD fdsS)) else
  let sortT = (uR,
               tenvR,
                    POr (predR, predS),
                    fdsR) in
  sortT
```

get part of union lens
$v \nearrow (I) = I \backslash_{R,S}[T \mapsto I(R) \overset{\cup}{\leftarrow}_F I(S)]$
TODO: determine necessary condition and check in particular, simple record union may violate FD. it should be checked at run time.

```
let get_union (r : rname) (s : rname) (t : rname) (db : database) : database =
  let (sortR, iR) = MapofRname.find r db in
  let (sortS, iS) = MapofRname.find s db in
  let sortT = typeinf_union sortR sortS in
  let ((uT, tenvT, predT, fdsT)) = sortT in
  let instanceT = relational_merge iR fdsT iS in (* TODO check conflict in merge.
currently, S wins. (by rbcr) *)
  let db = MapofRname.remove r db in
  let db = MapofRname.remove s db in
  MapofRname.add t (sortT, instanceT) db
```

```
let put_union (r : rname) (s : rname) (t : rname)
    ((dbJ, dbI) : (database × database)) : database =
  let (sortR, iR) = MapofRname.find r dbI in
  let (u, _, predR, f) = sortR in
  let (sortS, iS) = MapofRname.find s dbI in
  let (v, _, predS, g) = sortS in
  let (sortT, jT) = MapofRname.find t dbJ in
  let (v, _, _, fdsT) = sortT in
  let jT_org = relational_merge iR fdsT iS in (* TODO check conflict in merge *)
  let delT = SetofRecord.diff jT_org jT in (* deleted records *)
  let insT = SetofRecord.diff jT jT_org in (* inserted records *)
  let iR' = SetofRecord.diff iR delT in (* delete if match *)
  let iS' = SetofRecord.diff iS delT in (* delete if match *)
  let iR' = SetofRecord.union iR' (SetofRecord.filter (fun r → eval_bool r predR) insT) in
  let iS' = SetofRecord.union iS' (SetofRecord.filter (fun r → eval_bool r predS) insT) in
  let dbI' = MapofRname.remove t dbJ in
  let dbI' = MapofRname.add r (sortR, iR') dbI' in
  let dbI' = MapofRname.add s (sortS, iS') dbI' in
  dbI'
```

`change_set` $T\ T' = \{(t, -1) \mid t \in T \setminus T'\} \cup \{(t, 0) \mid t \in T \cap T'\} \cup \{(t, +1) \mid t \in T' \setminus T\}$

```
let change_set (table : relation) (table' : relation) : SetofChange.t =
 let sDelete =
  SetofRecord.fold (fun t →
    SetofChange.add (t, Delete)) (SetofRecord.diff table table') SetofChange.empty in
 let sKeep =
  SetofRecord.fold (fun t →
    SetofChange.add (t, Keep)) (SetofRecord.inter table table') SetofChange.empty in
 let sInsert =
  SetofRecord.fold (fun t →
    SetofChange.add (t, Insert)) (SetofRecord.diff table' table) SetofChange.empty in
  SetofChange.union (SetofChange.union sDelete sKeep) sInsert
```

relational delta merge

$M \xleftarrow{\Delta\cup}_F \Delta N = (M - \{t \mid (t, m) \in \Delta N \wedge m = -1\}) \xleftarrow{\cup}_F \{t \mid (t, m) \in \Delta N \wedge m = +1\}$

```
let relational_delta_merge (m : relation) (fds : SetofFD.t) (n : SetofChange.t) : relation =
  relational_merge
    (SetofRecord.diff m
      (SetofChange.fold (fun (t, m) rs →
            if m = Delete then SetofRecord.add t rs else rs) n SetofRecord.empty))
    fds
```

$(SetofChange.fold\ (\mathsf{fun}\ (t, m)\ rs\ \rightarrow$
$\quad \mathsf{if}\ m\ =\ Insert\ \mathsf{then}\ SetofRecord.add\ t\ rs\ \mathsf{else}\ rs)\ n\ SetofRecord.empty)$

match on
`match on` $t\ t'\ X = (t[X] = t'[X]) = \bigwedge\{t[A] = t'[A] \mid A \in X\}$

$\mathsf{let}\ match\_on\ (t : record)\ (t' : record)\ (cols : SetofAttr.t)\ : bool =$
$\quad record\_equal\ (restrict\ cols\ t)\ (restrict\ cols\ t')$

unflatten list of phrases as conjunction

$\mathsf{let\ rec}\ uf\_and\ (l : phrase\ list)\ :\ phrase\ =$
$\quad \mathsf{match}\ l\ \mathsf{with}$
$\quad\ \ [\,]\ \rightarrow\ PCns\ (Bol\ \mathsf{true})$
$\quad \mid [p]\ \rightarrow\ p$
$\quad \mid p :: ps\ \rightarrow\ PAnd\ (p, uf\_and\ ps)$

unflatten list of phrases as disjunction

$\mathsf{let\ rec}\ uf\_or\ (l : phrase\ list)\ :\ phrase\ =$
$\quad \mathsf{match}\ l\ \mathsf{with}$
$\quad\ \ [\,]\ \rightarrow\ PCns\ (Bol\ \mathsf{false})$
$\quad \mid [p]\ \rightarrow\ p$
$\quad \mid p :: ps\ \rightarrow\ POr\ (p, uf\_or\ ps)$

production of expression to compare all columns in *cols* to a record t
$\langle\!\langle \bigwedge [\![\{ \langle\!\langle [\![c]\!] = [\![t[c]]\!] \rangle\!\rangle \mid c \in cols \}]\!] \rangle\!\rangle$

$\mathsf{let}\ match\_on\_expr\ (t : record)\ (cols : SetofAttr.t)\ :\ phrase\ =$
$\quad \mathsf{let}\ col\_list\ =\ SetofAttr.elements\ cols\ \mathsf{in}$
$\quad \mathsf{let}\ phrase\_list\ =\ List.map\ (\mathsf{fun}\ c\ \rightarrow\ PEq\ (PVar\ c, PCns\ (attribute\ c\ t)))\ col\_list\ \mathsf{in}$
$\quad uf\_and\ phrase\_list$

multiplicity is complement with each other : $m = -m'$

$\mathsf{let}\ is\_compl\ (m : mult)\ (m' : mult)\ : bool =$
$\quad (m, m') = (Insert, Delete)\ \lor\ (m, m') = (Delete, Insert)$

complement rows
not sure if attributes key should always be indeed keys
$\{(t', m') \mid (t', m') \in \Delta R, t[key] = t'[key], m' = -m\}$

$\mathsf{let}\ compl\_rows\ ((t, m) : change\_entry)\ (dR : SetofChange.t)\ (key : SetofAttr.t)\ :\ SetofChange.t$
$= SetofChange.filter\ (\mathsf{fun}\ (t', m')\ \rightarrow$
$\quad match\_on\ t\ t'\ key\ \land\ is\_compl\ m\ m')\ dR$

`get_fd_updates` $\Delta R\ (X \rightarrow Y) = (X \rightarrow Y, \{(t[X], t[Y]) \mid (t, +1) \in \Delta R\})$

```
let get_fd_updates (dR : SetofChange.t) ((x, y) : fd) : fd × SetofPRecord.t =
    ((x, y), SetofChange.fold
                  (fun (t, m) ss →
                    if m = Insert then SetofPRecord.add (restrict x t, restrict y t) ss
                    else ss) dR SetofPRecord.empty)
```

{get_fd_updates ΔR f | $f \in F$}

```
let get_updateset (dR : SetofChange.t) (f : SetofFD.t) : SetofFDSPRecord.t =
   SetofFD.fold (fun fd →
      SetofFDSPRecord.add (get_fd_updates dR fd)) f SetofFDSPRecord.empty
```

var_case_expr map c or $(X \to Y) :=$
⟪**CASE**
    ⟦{⟪**WHEN** ⟦match_on_expr t X⟧ **THEN** ⟦$t'[c]$⟧⟫ | $(t, t') \in map$}⟧
ELSE⟦or⟧ **END**⟫

```
let var_case_expr (map : SetofPRecord.t) (attr : attr) (or_phrase : phrase) (fd : fd) : phrase =
   let where_clause_list = List.map
      (fun (t, v) → (match_on_expr t (fst fd), PCns (attribute attr v)))
         (SetofPRecord.elements map) in
      PCase (where_clause_list, or_phrase)

let rec calc_updated_var_expr (chl : SetofFDSPRecord.t)
      (key : SetofAttr.t) (col : SetofAttr.t) (attr : attr) (or_phrase : phrase)
      (map : SetofPRecord.t option) : phrase =
   let (f, changes) =
      let s = SetofFDSPRecord.filter
                (fun ((x, y), changes) → SetofAttr.equal col y) chl in
      let _ = Format.printf "s=%a@." pp_SetofFDSPRecord s in
      match SetofFDSPRecord.cardinal s with
         1 → SetofFDSPRecord.choose s
      | _ → failwith
              (Format.sprintf "invalid␣cardinality␣%i␣of␣s␣%s@." (SetofFDSPRecord.cardinal s)
                 (toStr pp_SetofFDSPRecord s)) in
   let map' = match map with
      None → changes
   | Some map →
        let m =
                SetofPRecord.fold (fun (k, k') ss →
             (SetofPRecord.fold (fun (k'', v) ss' →
                if record_equal k' k'' then SetofPRecord.add (k, v) ss'
                else ss') map ss)) changes SetofPRecord.empty in
```

```
      Format.printf "m=%a@." pp_SetofPRecord m; m in
  if (* SetofAttr.equal key (fst f) *)
     SetofAttr.subset (fst f) key
     then
    var_case_expr map' attr or_phrase f
  else
    calc_updated_var_expr chl key (fst f)
      attr (var_case_expr map' attr or_phrase f) (Some map')
let updated_var_expr (chl : SetofFDSPRecord.t) (key : SetofAttr.t) (col : attr)
 : phrase =
  calc_updated_var_expr chl key (SetofAttr.singleton col) col (PVar col) None
```

sort: target sort of the input

```
let updated_pred (dR : SetofChange.t) (sort : sort) (p : phrase) : phrase =
  let (u, _, pred, fds) = sort in
  let chl = get_updateset dR fds in
  let key =
    let s = roots fds in
    match PSetofAttr.cardinal s with
      1 → PSetofAttr.choose s
    | _ → let error_message =
            (Format.sprintf
               "updated_pred: > 1 cardinality of keys of functional dependency %s. Merging
               (toStr pp_PSetofAttr s)) in
           Format.fprintf Format.err_formatter "%s@." error_message;
           PSetofAttr.fold (fun s → SetofAttr.union s) s SetofAttr.empty in
  phrase_map (fun node →
    match node with
      PVar n →
        if ¬ (SetofAttr.mem n (right fds)) then node else updated_var_expr chl key n
    | _ → node) p
```

$\sigma_{\langle\langle [\![\neg P]\!] \wedge [\![\texttt{updated\_pred}\ \Delta R\ l\ P]\!] \rangle\rangle}(l)$

```
let query_deleted_rows (dR : SetofChange.t) (r : rname) (p : phrase) (dbI : database) : relation =
  let (sortR, iR) = MapofRname.find r dbI in
  let (_, tenv, _, _) = sortR in
  if qtype tenv p ≠ TBol then failwith
      (Format.sprintf "query_deleted_rows: ill-typed predicate %s@."
         (toStr pp_phrase p)) else
  let pred = (PAnd (PNot p, updated_pred dR sortR p)) in
  SetofRecord.filter (fun r → eval_bool r pred) iR
```

$\Delta R \cup \{(t, -1) \mid t \in \texttt{query\_deleted\_rows}\ \Delta R\ l\ P\}$

```
let delta_put_select (r : rname) (dbI : database) (p : phrase) (dR : SetofChange.t)
      : SetofChange.t =
   SetofChange.union dR
    (SetofRecord.fold (fun t ->
       SetofChange.add (t, Delete))
          (query_deleted_rows dR r p dbI) SetofChange.empty)
```

Thesis ignores multiplicity of change set, but deletion should be ignored
$\langle\!\langle \bigvee [\![\{\texttt{match\_on\_expr}\ t\ x \mid (t, m) \in \Delta R\}]\!] \rangle\!\rangle$

```
let any_match_expr (dR : SetofChange.t) (x : SetofAttr.t) : phrase =
   let change_list = SetofChange.elements dR in
   let phrase_list = List.map (fun (t, m) -> match_on_expr t x)
     change_list in uf_or phrase_list
```

$\sigma_{\texttt{any\_match\_expr}\ \Delta R\ X}(I(R)[X \cup \{A\}])$

```
let query_lookup_table (dR : SetofChange.t) (r : rname) (db : database)
        (x : SetofAttr.t) (a : attr) : relation =
   let (_, tenv, _, _) = sort r db in
   let pred = any_match_expr dR x in
   if qtype tenv pred ≠ TBol then failwith
         (Format.sprintf "query_lookup_table:␣ill-typed␣predicate␣%s@."
             (toStr pp_phrase pred)) else
        SetofRecord.filter (fun r -> eval_bool r pred)
         (restrict_relation (SetofAttr.add a x) (instance r db))
```

```
let lookup_col (t : record) (a : attr) (x : SetofAttr.t)
    (v : value) (l : SetofRecord.t) =
     let c = SetofRecord.filter (fun t' -> match_on t t' x ) l in
     match SetofRecord.cardinal c with
     | 0 -> extend a v t
     | _ ->
          let t' = SetofRecord.choose c in
          extend a (attribute a t') t
```

delta translation for drop
let $L = \texttt{query\_lookup\_table}\ \Delta R\ l\ X\ A$
$\{(\texttt{lookup\_col}\ t\ A\ X\ a\ L, m) \mid (t, m) \in \Delta R\}$

```
let delta_put_drop (r : rname) (db : database) (a : attr) (x : SetofAttr.t) (v : value)
    (dR : SetofChange.t) =
  let l = query_lookup_table dR r db x a in
  SetofChange.fold (fun (t, m) →
    SetofChange.add
      (lookup_col t a x v l, m)) dR SetofChange.empty
```

decomposition of functional dependency
$F = F' \cup \{X \to A\}$
arbitrary decomposition

```
let decompose_fd (f : SetofFD.t) (a : attr) : SetofFD.t × fd =
  let rec df acc fds = match fds with
    [] → failwith (Format.sprintf
                                "decompose_fd: decomposition of FDs %s w.r.t. %s failed."
                                (toStr pp_SetofFD f) (toStr pp_attr a))
  | ((x, y) :: ss) →
      if SetofAttr.mem a y then
        let f′ =
          let y′ = SetofAttr.remove a y in
          SetofFD.union (SetofFD.of_list ss)
          (if SetofAttr.is_empty y′ then acc else
           SetofFD.add (x, y′) acc) in
        (f′, (x, SetofAttr.singleton a))
      else
        df (SetofFD.add (x, y) acc) ss in
  df SetofFD.empty (SetofFD.elements f)
```

FD split for projection lens, using attributes defining attribute a

```
let decompose_fd (f : SetofFD.t) (x′ : SetofAttr.t) (a : attr) : SetofFD.t × fd =
  let rec df acc fds = match fds with
    [] → failwith (Format.sprintf
                  "decompose_fd: decomposition of FDs %s w.r.t. (%s,%s) failed."
                  (toStr pp_SetofFD f) (toStr pp_SetofAttr x′) (toStr pp_attr a))
  | ((x, y) :: ss) →
      if SetofAttr.mem a y ∧ SetofAttr.equal x x′ then
        let f′ =
          let y′ = SetofAttr.remove a y in
          SetofFD.union (SetofFD.of_list ss)
          (if SetofAttr.is_empty y′ then acc else
           SetofFD.add (x, y′) acc) in
        (f′, (x, SetofAttr.singleton a))
```

```
      else
        df (SetofFD.add (x, y) acc) ss in
  df SetofFD.empty (SetofFD.elements f)
```

tentative conversion to conjunctive normal form (CNF)
$\neg\neg P \Rightarrow P$
$\neg(P \wedge Q) \Rightarrow \neg P \vee \neg Q$
$\neg(P \vee Q) \Rightarrow \neg P \wedge \neg Q$
$(P \wedge Q) \vee R \Rightarrow (P \vee R) \wedge (Q \vee R)$

```
let rec cnf (p : phrase) : phrase = match p with
    PCns _ → p
  | PAnd (p1, p2) →
      let p1' = cnf p1 in
      let p2' = cnf p2 in
      PAnd (p1', p2')
  | POr (p1, p2) →
      let p1' = cnf p1 in
      let pR = cnf p2 in
      (match p1' with
        PAnd (pP, pQ) → (* distribution *) PAnd (cnf (POr (pP, pR)), cnf (POr (pQ, pR)))
      | _ →
          (match pR with
            PAnd (pP, pQ) →
              PAnd (cnf (POr (p1', pP)), cnf (POr (p1', pQ)))
          | _ → POr (p1', pR)))
  | PNot p1 →
       let p1' = cnf p1 in
       (match p1' with
          PNot p1'' → p1'' (* double negation elimination *) | PAnd (p1', p2') → (* De
Morgan *) cnf (POr (PNot p1', PNot p2'))
       | POr (p1', p2') → (* De Morgan *) PAnd (cnf (PNot p1'), cnf (PNot p2'))
       | _ → PNot p1'
       )
  | PVar _ → p
  | PLt _ → p
  | PGt _ → p
  | PLte _ → p
  | PGte _ → p
  | PEq _ → p
  | PCase _ → p
```

projection of predicates
after turning into conjunctive normal form, find a disjunction that only contains reference to A and detach from the rest.

```
(A = 1) and ((B = 2) or (C = 3)) -> B=2 or  C=3, A=1
 B = 2  and   A = 3  and C = 2   -> B=2 and C=2, A=3
```

set of free variables appear in the given phrase p

```
let rec freevars (p : phrase) : SetofAttr.t = match p with
   PCns v → SetofAttr.empty
 | PNot p1 → freevars p1
 | PVar attr → SetofAttr.singleton attr
 | PAnd (p1, p2) | POr (p1, p2)
 | PLt (p1, p2) | PGt (p1, p2) | PLte (p1, p2) | PGte (p1, p2) | PEq (p1, p2) →
     SetofAttr.union (freevars p1) (freevars p2)
 | PCase (wlist, else_phrase) →
      List.fold_right (fun (w, t) ss →
           SetofAttr.union (freevars w)
             (SetofAttr.union (freevars t) ss)) wlist (freevars else_phrase)
```

split predicate $P : U$ into a pair $(P[U - A], P[A])$
used for decomposition $P = P[U - A] \bowtie P[A]$ in drop lens

```
let split_phrase (p : phrase) (a : attr) : phrase × phrase =
 let p = cnf p (* convert p into CNF *) in
 (* flatten as list of disjunctions *)
 let rec fl (p : phrase) : phrase list = match p with
    PAnd (p1, p2) → fl p1 @ fl p2
 | _ → [p] in
 let l = fl p in
 let (aps, ps) = List.partition (fun p → SetofAttr.mem a (freevars p)) l in
  (* unflatten *)
 let (faps, fps) = (uf_and aps, uf_and ps) in
 match (SetofAttr.cardinal (freevars faps)) with
    0 | 1 → (fps, faps)
 | _ → (* faps contains variables other than A *)
      failwith (Format.sprintf
        "split_phrase: conjunctive extraction of %s part from predicate %s failed"
                  (toStr pp_attr a) (toStr pp_phrase p))
```

drop lens
`drop` A `determined by` (X, a) `from` R `as` S

$$\frac{\begin{array}{c} sort(R) = (U, P, F) \\ A \in U \quad F \equiv F' \cup \{X \to A\} \\ sort(S) = (U - A, P[U - A], F') \\ P = P[U - A] \bowtie P[A] \quad \{A = a\} \in P[A] \end{array}}{\begin{array}{c} \texttt{drop}\ A\ \texttt{determined by}\ (X, a)\ \texttt{from}\ R\ as\ S \in \\ \Sigma \uplus \{R\} \Leftrightarrow \Sigma \uplus \{S\} \end{array}} \text{ (T-DROP)}$$

type check and inference for drop

let $typeinf\_drop\ (a : attr)\ ((x, v) : SetofAttr.t\ \times\ value)\ (sortR : sort)\ :\ sort\ =$
  let $(u, tenv, p, f)\ =\ sortR$ in
  if $\neg\ (SetofAttr.mem\ a\ u)$ then $failwith\ (Format.sprintf$
   "get_drop:␣dropped␣attribute␣%s␣is␣not␣in␣the␣domain␣%s"
   $(toStr\ pp\_attr\ a)\ (toStr\ pp\_SetofAttr\ u))$ else
  let $(f', (xS, aS))\ =\ decompose\_fd\ f\ x\ a$ in
  let $(pRest, pA)\ =\ split\_phrase\ p\ a$ in
  if $(Bol$ true$)\ \neq\ eval\ (MapofAttr.singleton\ a\ v)\ pA$ then
    $failwith\ (Format.sprintf$ "value␣v=%s␣does␣not␣satisfy␣predicate␣P[%s]␣=␣%s"
        $(toStr\ pp\_value\ v)\ (toStr\ pp\_attr\ a)\ (toStr\ pp\_phrase\ pA))$
  else
    $(SetofAttr.remove\ a\ u, MapofAttr.remove\ a\ tenv, pRest, f')$

get part of drop lens
$v \nearrow (I) = I\backslash_R[S \mapsto I(R)[U - A]]$

let $get\_drop\ (a\ :\ attr)\ ((x, v)\ :\ SetofAttr.t\ \times\ value)\ (r\ :\ rname)\ (s\ :\ rname)\ (dbI\ :\ database)\ :\ database\ =$
  let $(sortR, iR)\ =\ MapofRname.find\ r\ dbI$ in
  let $(u, \_, \_, \_)\ =\ sortR$ in
  let $sortS\ =\ typeinf\_drop\ a\ (x, v)\ sortR$ in
    $MapofRname.add\ s$
     $(sortS, restrict\_relation\ (SetofAttr.remove\ a\ u)\ iR)$
      $(MapofRname.remove\ r\ dbI)$

put part of the drop lens

$$\begin{array}{rcl} v \searrow (J, I) & = & J\backslash_S[R \mapsto M \leftarrow_{X \to A} I(R)] \\ \text{where}\quad M & = & (I(R) \bowtie J(S)) \cup (N_+ \bowtie \{\{A = a\}\}) \\ N_+ & = & J(S) \setminus I(R)[U - A] \\ U & = & dom(R) \end{array}$$

let $put\_drop\ (a : attr)\ ((x, v) : SetofAttr.t\ \times\ value)\ (r : rname)\ (s : rname)$
  $((dbJ, dbI) : database \times database)\ :\ database\ =$
  let $(sortR, iR)\ =\ MapofRname.find\ r\ dbI$ in
  let $(u, \_, \_, \_)\ =\ sortR$ in
  let $\_\ =\ typeinf\_drop\ a\ (x, v)\ sortR$ in (* for typecheck *)

$\quad \mathsf{let}\ jS\ =\ instance\ s\ dbJ\ \mathsf{in}$
$\quad \mathsf{let}\ nPlus\ =\ SetofRecord.diff\ jS\ (restrict\_relation\ (SetofAttr.remove\ a\ u)\ iR)\ \mathsf{in}$
$\quad \mathsf{let}\ mR\ =\ SetofRecord.union$
$\qquad\quad (nat\_join\ iR\ jS)$
$\qquad\quad (nat\_join\ nPlus\ (SetofRecord.singleton\ (MapofAttr.singleton\ a\ v)))\ \mathsf{in}$
$\quad MapofRname.add\ r$
$\qquad (sortR, relation\_revision\ mR\ (SetofFD.singleton\ (x, SetofAttr.singleton\ a))\ iR)$
$\qquad\quad (MapofRname.remove\ s\ dbJ)$

key(l) compute tentative keys from sort

$\mathsf{let}\ key\ (l : ilens)\ :\ SetofAttr.t\ =$
$\quad \mathsf{let}\ ((\_, \_, \_, fds), \_)\ =\ l\ \mathsf{in}$
$\quad \mathsf{let}\ s\ =\ roots\ fds\ \mathsf{in}$
$\quad\ PSetofAttr.fold\ (\mathsf{fun}\ s\ \rightarrow\ SetofAttr.union\ s)\ s\ SetofAttr.empty$

dom(l)

$\mathsf{let}\ idom\ (l : ilens)\ :\ SetofAttr.t\ =$
$\quad \mathsf{let}\ ((u, \_, \_, \_), \_)\ =\ l\ \mathsf{in}\ u$

$\mathsf{let}\ irel\ (l : ilens)\ :\ relation\ =$
$\quad \mathsf{let}\ ((\_, \_, \_, \_), r)\ =\ l\ \mathsf{in}\ r$

$\mathsf{let}\ is\_neutral\ ((t, m)\ :\ change\_entry)\ : bool = (m\ =\ Keep)$

$m \neq 0 \wedge \exists (t', m') \in \Delta R, m' = 0 \wedge t[\mathtt{key}(l_2)] = t'[\mathtt{key}(l_2)]$

$\mathsf{let}\ ntrl\_exists\_right\ ((t, m) : change\_entry)\ (dR : SetofChange.t)\ (l2 : ilens)\ : bool =$
$\quad (\neg\ (is\_neutral\ (t, m)))\ \wedge\ SetofChange.exists$
$\quad\ (\mathsf{fun}\ (t', m')\ \rightarrow\ m'\ =\ Keep\ \wedge\ match\_on\ t\ t'\ (key\ l2))\ dR$

$m \neq 0 \wedge \exists (t', m') \in \Delta R, m' = -m \wedge t[\mathtt{key}(l_1)] = t'[\mathtt{key}(l_1)]$

$\mathsf{let}\ compl\_exists\_left\ ((t, m) : change\_entry)\ (dR : SetofChange.t)\ (l1 : ilens)\ : bool =$
$\quad (\neg\ (is\_neutral\ (t, m)))\ \wedge\ SetofChange.exists$
$\quad\ (\mathsf{fun}\ (t', m')\ \rightarrow\ is\_compl\ m\ m'\ \wedge\ match\_on\ t\ t'\ (key\ l1))\ dR$

$m \neq 0 \wedge \exists (t', m') \in \Delta R, m' = -m \wedge t[\mathtt{key}(l_2)] = t'[\mathtt{key}(l_2)]$

$\mathsf{let}\ compl\_exists\_right\ ((t, m) : change\_entry)\ (dR : SetofChange.t)\ (l2 : ilens)\ : bool =$
$\quad (\neg\ (is\_neutral\ (t, m)))\ \wedge\ SetofChange.exists$
$\quad\ (\mathsf{fun}\ (t', m')\ \rightarrow\ is\_compl\ m\ m'\ \wedge\ match\_on\ t\ t'\ (key\ l2))\ dR$

$\mathtt{compl\_exists\_left}\ (t, m)\ \Delta R\ l_1$
$\wedge \exists (t', m') \in \Delta R, m' = -m \wedge t[\mathtt{key}(l_1)] = t'[\mathtt{key}(l_1)] \wedge t[\mathtt{dom}(l_1)] \neq t'[\mathtt{dom}(l_1)]$

let *upd_left* $((t, m) : change\_entry)$ $(dR : SetofChange.t)$ $(l1 : ilens)$: *bool* =
  compl_exists_left (t, m) *dR l1* $\wedge$
   SetofChange.exists (fun (t', m') $\rightarrow$
     match_on t t' (*key l1*) $\wedge$ ($\neg$ (*match_on* t t' (*idom l1*)))) *dR*

$\texttt{compl\_exists\_right}\ (t, m)\ \Delta R\ l_2$
$\wedge\ \exists (t', m') \in \Delta R, m' = -m \wedge t[\texttt{key}(l_2)] = t'[\texttt{key}(l_2)] \wedge t[\texttt{dom}(l_2)] \neq t'[\texttt{dom}(l_2)]$

let *upd_right* $((t, m) : change\_entry)$ $(dR : SetofChange.t)$ $(l2 : ilens)$: *bool* =
  compl_exists_right (t, m) *dR l2* $\wedge$
   SetofChange.exists (fun (t', m') $\rightarrow$
     match_on t t' (*key l2*) $\wedge$ ($\neg$ (*match_on* t t' (*idom l2*)))) *dR*

let *non_upd_left* $((t, m) : change\_entry)$ $(dR : SetofChange.t)$ $(l1 : ilens)$: *bool* =
  compl_exists_left (t, m) *dR l1* $\wedge$ ($\neg$ (*upd_left* (t, m) *dR l1*))

let *non_upd_right* $((t, m) : change\_entry)$ $(dR : SetofChange.t)$ $(l2 : ilens)$: *bool* =
  compl_exists_right (t, m) *dR l2* $\wedge$ ($\neg$ (*upd_right* (t, m) *dR l2*))

let *found_any_right* $((t, m) : change\_entry)$ $(l2 : ilens)$: *bool* =
    SetofRecord.exists
    (fun r $\rightarrow$ *eval_bool* r (*match_on_expr* t (*key l2*))) (*irel l2*)

let *found_same_right* $((t, m) : change\_entry)$ $(l2 : ilens)$: *bool* =
    SetofRecord.exists
    (fun r $\rightarrow$ *eval_bool* r (*match_on_expr* t (*idom l2*))) (*irel l2*)

let *found_upd_right* $((t, m) : change\_entry)$ $(l2 : ilens)$: *bool* =
    found_any_right (t, m) *l2* $\wedge$ ($\neg$ (*found_same_right* (t, m) *l2*))

$\langle\!\langle \bigvee [\![\{\texttt{match\_on\_expr}\ t\ key \mid (t, m) \in \Delta R, m = -1, \texttt{compl\_rows}\ (t, m)\ \Delta R\ key = \emptyset\}]\!] \rangle\!\rangle$

let *removed_row_expr* $(dR : SetofChange.t)$ $(key : SetofAttr.t)$: *phrase* =
  uf_or
    (*SetofChange.fold*
      (fun (t, m) *ps* $\rightarrow$
        if m = *Delete* $\wedge$ *SetofChange.is_empty* (*compl_rows* (t, m) *dR key*)
        then (*match_on_expr* t *key*) :: *ps* else *ps*) *dR* [])

$\{(t, t') \mid (t, m) \in \Delta R, (t', m') \in \texttt{compl\_rows}\ (t, m)\ \Delta R\ key, m = -1\}$

let *find_changed_rows* $(dR : SetofChange.t)$ $(key : SetofAttr.t)$: *SetofPRecord.t* =
  SetofChange.fold (fun (t, m) *ss* $\rightarrow$
    if m = *Delete* then
     SetofPRecord.union
     (*SetofChange.fold* (fun (t', m') $\rightarrow$ (*SetofPRecord.add* (t, t')))
       (*compl_rows* (t, m) *dR key*) *SetofPRecord.empty*) *ss* else *ss*)
       dR SetofPRecord.empty

$\langle\!\langle \bigvee\{[\![\texttt{match\_on\_expr}\ t\ X \mid (X \to Y) \in \mathit{fds}, t[X] = t'[X], t[Y] \neq t'[Y]\}]\!]\rangle\!\rangle$

let $\mathit{match\_changes\_expr}$ $((t, t') : \mathit{record} \times \mathit{record})(\mathit{fds} : \mathit{SetofFD.t}) : \mathit{phrase} =$
  $\mathit{uf\_or}$
    $(\mathit{SetofFD.fold}$ (fun (x, y) $\mathit{ss} \to$
      if $(\mathit{match\_on}\ t\ t'\ x \wedge \neg (\mathit{match\_on}\ t\ t'\ y))$ then
        $(\mathit{match\_on\_expr}\ t\ x) :: \mathit{ss}$ else ss) fds [ ])

let $\mathit{fwd\_fds}$ $(s : \mathit{SetofAttr.t})$ $(\mathit{fds} : \mathit{SetofFD.t}) : \mathit{SetofAttr.t} =$
  let $\mathit{edges} = \mathit{fds}$ in
  $\mathit{SetofFD.fold}$ (fun (x, y) $\mathit{ss} \to$
   if $\mathit{SetofAttr.subset}\ x\ s$ then $\mathit{SetofAttr.union}\ y\ \mathit{ss}$
   else ss) edges $\mathit{SetofAttr.empty}$

let lefts $(\mathit{fds} : \mathit{SetofFD.t}) : \mathit{PSetofAttr.t} =$
   $\mathit{SetofFD.fold}$ (fun $(x, y) \to$
     $\mathit{PSetofAttr.add}\ x)$ fds $\mathit{PSetofAttr.empty}$

let $\mathit{closure}$ $(\mathit{fd} : \mathit{fd})$ $(\mathit{fds} : \mathit{SetofFD.t}) : \mathit{SetofAttr.t} =$
  let rec cl $(\mathit{acc} : \mathit{SetofAttr.t})$ $(\mathit{frontier} : \mathit{SetofAttr.t}) : \mathit{SetofAttr.t} =$
   let $\mathit{frontier}' = \mathit{fwd\_fds}\ \mathit{frontier}\ \mathit{fds}$ in
   if $\mathit{SetofAttr.subset}\ \mathit{frontier}'\ \mathit{acc}$ then (* no more new attr *) acc else
   cl $(\mathit{SetofAttr.union}\ \mathit{acc}\ \mathit{frontier}')$ $\mathit{frontier}'$ in
 cl $(\mathit{snd}\ \mathit{fd})$ $(\mathit{snd}\ \mathit{fd})$

$\{\mathit{fd} \mid \mathit{fd} \in \mathit{fds}, \mathit{key} \subset \texttt{closure}(\mathit{fd}, \mathit{fds})\}$

let $\mathit{defining\_fds}$ $(\mathit{key} : \mathit{SetofAttr.t})$ $(\mathit{fds} : \mathit{SetofFD.t}) : \mathit{SetofFD.t} =$
 $\mathit{SetofFD.fold}$ (fun $\mathit{fd}\ \mathit{ss} \to$
   if $\mathit{SetofAttr.subset}\ \mathit{key}$ $(\mathit{closure}\ \mathit{fd}\ \mathit{fds})$
   then $\mathit{SetofFD.add}\ \mathit{fd}\ \mathit{ss}$
   else ss) fds $\mathit{SetofFD.empty}$

$\langle\!\langle \bigvee\{[\![\texttt{match\_on\_expr}\ \mathit{chl}\ \texttt{left}(f) \mid (\mathit{chl}, \mathit{chr}) \in \mathit{changes}\}]\!]\rangle\!\rangle$

let $\mathit{fd\_changes\_expr}$ $(\mathit{changes} : \mathit{SetofPRecord.t})$ $(f : \mathit{fd}) : \mathit{phrase} =$
 $\mathit{uf\_or}$
   $(\mathit{SetofPRecord.fold}$ (fun $(\mathit{chl}, \mathit{chr})$ $\mathit{ss} \to$
     $(\mathit{match\_on\_expr}\ \mathit{chl}\ (\mathit{fst}\ f)) :: \mathit{ss})$ $\mathit{changes}$ [ ])

$\langle\!\langle \bigvee\{[\![\texttt{fd\_changes\_expr}\ \mathit{changes}\ f \mid f \in \texttt{defining\_fds}\ J\ F,$
$(f, \mathit{changes}) \in \texttt{get\_updateset}\ \Delta R\ F\}]\!]\rangle\!\rangle$

```
let changes_expr (dR : SetofChange.t) (fS : SetofFD.t) (j : SetofAttr.t) : phrase =
  uf_or
    (SetofFD.fold (fun f ss →
      (SetofFDSPRecord.fold (fun (f′, changes) ss′ →
        if 0 = compare_fd f f′ then
         (fd_changes_expr changes f) :: ss′
        else ss′) (get_updateset dR fS) []) @ ss) (defining_fds j fS) [])
```

$\langle\!\langle [\![\mathtt{match\_on\_expr}\ t\ J]\!] \wedge \neg[\![\mathtt{removed\_row\_expr}\ \Delta R\ P_d]\!] \wedge \neg[\![\mathtt{changes\_expr}\ \Delta R\ F\ J]\!] \rangle\!\rangle$

```
let create_query_expr (t : record) (dR : SetofChange.t) (pd : SetofAttr.t)
    (j : SetofAttr.t) (fS : SetofFD.t) : phrase =
  PAnd (match_on_expr t j,
        PAnd (PNot (removed_row_expr dR pd),
              PNot (changes_expr dR fS j)))
```

```
let remove_entry_left ((t, m) : change_entry) (dR : SetofChange.t) (l1 : ilens) : bool =
  m = Delete ∧ (¬ (compl_exists_left (t, m) dR l1))
```

join with colums j

$\bowtie_J$

```
let join_with_columns (j : SetofAttr.t) (l1 : ilens) (l2 : ilens) : ilens =
  let ((u, tenv1, p, fds1), m) = l1 in
  let ((v, tenv2, q, fds2), n) = l2 in
  if (¬ (SetofAttr.subset j u)) then
    failwith (Format.sprintf "join_with_columns: left domain %s does not contain join colum %
  if (¬ (SetofAttr.subset j v)) then
    failwith (Format.sprintf "join_with_columns: right domain %s does not contain join colum
  ((SetofAttr.union u v,
    merge_tenv tenv1 tenv2,
    PAnd (p, q),
    SetofFD.union fds1 fds2),
    SetofRecord.fold (fun rm rms →
      SetofRecord.union
        (let rm′ = restrict j rm in
        (SetofRecord.fold (fun rn rns →
          let rn′ = restrict j rn in
          if record_equal rm′ rn′
            then SetofRecord.add (MapofAttr.merge
                                        (fun k a b → match (a, b) with
                                           (Some a, Some b) → if a = b then Some a else failwith "misma
                                         | (Some a, None ) → Some a
```

```
                                        | (None, Some b) → Some b
                                        | (None, None) → None) rm rn) rns
          else rns) n rms)) rms) m SetofRecord.empty)

let remove_entry_right ((t, m) : change_entry) (dR : SetofChange.t)
    (l1 : ilens) (l2 : ilens) (j : SetofAttr.t) (fds : SetofFD.t) : bool =
  (m = Delete) ∧ (¬ (compl_exists_right (t, m) dR l2))
    ∧ (¬ (ntrl_exists_right (t, m) dR l2))
    ∧ (¬
         (let (_, join_l1_l2) = (join_with_columns j l1 l2) in
          SetofRecord.exists
          (fun r → eval_bool r (create_query_expr t dR (key l1) j fds )) join_l1_l2))

let join_left_row ((t, m) : change_entry) (dR : SetofChange.t)
    (l1 : ilens) (l2 : ilens) (j : SetofAttr.t) (fds : SetofFD.t) (pd : phrase) =
  let pi_u_t = restrict (idom l1) t in
  if is_neutral (t, m) then SetofChange.singleton (pi_u_t, Keep)
  else
    if compl_exists_left (t, m) dR l1 then
      if upd_left (t, m) dR l1 then SetofChange.singleton (pi_u_t, m)
      else
        if non_upd_left (t, m) dR l1 then
          if m = Delete then SetofChange.singleton (pi_u_t, m)
          else SetofChange.empty
        else
          SetofChange.empty
    else
      if m = Delete then
        if remove_entry_left (t, m) dR l1 then
          if remove_entry_right (t, m) dR l1 l2 j fds then
            if eval_bool t pd then
              SetofChange.singleton (pi_u_t, Delete)
            else
              SetofChange.empty
          else
            SetofChange.singleton (pi_u_t, Delete)
        else
          SetofChange.empty (* OK? *)
      else (* (m = +1) *)
        SetofChange.singleton (pi_u_t, Insert)
```

```
let join_right_row ((t, m) : change_entry) (dR : SetofChange.t)
    (l1 : ilens) (l2 : ilens) (j : SetofAttr.t) (fds : SetofFD.t) (qd : phrase) =
  let pi_v_t = restrict (idom l2) t in
  if is_neutral (t, m) then SetofChange.singleton (pi_v_t, Keep)
  else
    if compl_exists_right (t, m) dR l2 then
      if ntrl_exists_right (t, m) dR l2 then SetofChange.empty
      else
        if upd_right (t, m) dR l2 then SetofChange.singleton (pi_v_t, m)
        else
          if non_upd_right (t, m) dR l2 then
            if m = Delete then SetofChange.singleton (pi_v_t, Keep)
            else SetofChange.empty
          else SetofChange.empty (* undocumented *)
    else
      if m = Delete then
        if remove_entry_right (t, m) dR l1 l2 j fds then
          if remove_entry_left (t, m) dR l1 then
            if eval_bool t qd then SetofChange.singleton (pi_v_t, Delete)
            else SetofChange.empty
          else SetofChange.empty
        else SetofChange.empty (* undocumented *)
      else (* (m = Insert) *)
        if found_same_right (t, m) l2 then
          SetofChange.singleton (pi_v_t, Keep)
        else if (¬ (found_any_right (t, m) l2)) then SetofChange.singleton (pi_v_t, Insert)
        else if found_upd_right (t, m) l2 then
          let ((_, _, _, _), rel_l2) = l2 in
          let ts =
            SetofRecord.filter (fun r → eval_bool r (match_on_expr t (key l2)))
              rel_l2 in
          let t' =
            match (SetofRecord.cardinal ts) with
              0 → failwith "no␣record␣found"
            | 1 → SetofRecord.choose ts
            | _ → failwith (Format.sprintf "found_upd_right:␣multiple␣record␣%s␣found"
                        (toStr pp_relation ts)) in
          SetofChange.of_list [(pi_v_t, Insert); (t', Delete)]
        else SetofChange.empty (* undocjmented *)
```

$$\left(\bigcup_{(t,m)\in\Delta R} \texttt{join\_left\_row}(t,m)\ \Delta R\ l_1\ l_2\ J\ F\ P_d, \right.$$
$$\left. \bigcup_{(t,m)\in\Delta R} \texttt{join\_right\_row}(t,m)\ \Delta R\ l_1\ l_2\ J\ F\ Q_d \right)$$

```
let delta_put_join (l1 : ilens) (l2 : ilens) (j : SetofAttr.t) (pd : phrase)
      (qd : phrase) (fS : SetofFD.t) (dR : SetofChange.t) : (SetofChange.t × SetofChange.t) =
   (SetofChange.fold (fun (t, m) →
      SetofChange.union
         (join_left_row (t, m) dR l1 l2 j fS pd)) dR SetofChange.empty,
      SetofChange.fold (fun (t, m) →
      SetofChange.union
         (join_right_row (t, m) dR l1 l2 j fS qd)) dR SetofChange.empty)
```

$v \nearrow \in \Sigma \to \Delta$

```
let rec get (l : lens) (dbI : database) : database = match l with
      Select (r, p, s) → get_select r p s dbI
   | JoinDL ((r, s), t) → get_join_dl r s t dbI
   | Drop (a, (attr_list, v), r, s) → get_drop a (SetofAttr.of_list attr_list, v) r s dbI
   | Compose (l1, l2) →
         let dbJ = get l1 dbI in
         get l2 dbJ
(* v ↘∈ Δ × Σ → Σ *)
and put (l : lens) ((dbJ, dbI) : database × database) : database = match l with
   Select (r, p, s) → put_select r p s (dbJ, dbI)
| JoinDL ((r, s), t) → put_join_dl r s t (dbJ, dbI)
| Drop (a, (attr_list, v), r, s) → put_drop a (SetofAttr.of_list attr_list, v) r s (dbJ, dbI)
| Compose (l1, l2) →
      let dbK = get l1 dbI in
      put l1 (put l2 (dbJ, dbK), dbI)
and delta_put (l : lens) (dbI : database) (dR : SetofChange.t) : SetofChange.t = match l with
      Select (r, p, s) → delta_put_select r dbI p dR
   | JoinDL ((r, s), t) → failwith "delta_put: binary operator join does not fit here"
   | Drop (a, (attr_list, v), r, s) →
         delta_put_drop r dbI a (SetofAttr.of_list attr_list) v dR
   | Compose (l1, l2) →
     let dbK = get l1 dbI in
     delta_put l1 dbI (delta_put l2 dbK dR)
(* because of binary operator, functional delta should return map from relation to its change
```

set ∗)

```
and delta_put_map (l : lens) (dbI : database) (dRm : SetofChange.t MapofRname.t) :
    SetofChange.t MapofRname.t = match l with
    Select (r, p, s) →
    let dS = MapofRname.find s dRm in
    let dR = delta_put_select r dbI p dS in
    MapofRname.add r dR (MapofRname.remove s dRm)
  | JoinDL ((r, s), t ) →
    let dT = MapofRname.find t dRm in
    let l1 = MapofRname.find r dbI in
    let l2 = MapofRname.find s dbI in
    let ((u, _, p, fds1), _) = l1 in
    let ((v, _, q, fds2), _) = l2 in
    let j = SetofAttr.inter u v in
    let (dR, dS) = delta_put_join l1 l2 j (PCns (Bol true)) (PCns (Bol false))
        (SetofFD.union fds1 fds2) dT in
    MapofRname.add r dR (MapofRname.add s dS (MapofRname.remove t dRm))
   | Drop (a, (attr_list, v), r, s) →
       let dS = MapofRname.find s dRm in
       let dR = delta_put_drop r dbI a (SetofAttr.of_list attr_list) v dS in
       MapofRname.add r dR (MapofRname.remove s dRm)
   | Compose (l1, l2) →
   let dbK = get l1 dbI in
   delta_put_map l1 dbI (delta_put_map l2 dbK dRm)
```

```
let select r p s = Select (r, p, s)
let join_dl (r, s) t = JoinDL ((r, s), t)
let drop a (attr_list, v) r s = Drop (a, (attr_list, v), r, s)
```

$$\frac{v \in \Sigma \Leftrightarrow \Sigma' \qquad w \in \Sigma' \Leftrightarrow \Delta}{v; w \in \Sigma \Leftrightarrow \Delta}\ (\text{T-Compose})$$

```
let compose l1 l2 = Compose (l1, l2)
```

composition lens in infix form

```
let (& :) (l1 : lens) (l2 : lens) : lens = compose l1 l2
```

type inference of lens, given sort on the left

```
let rec gt_lens (l : lens) (srt : sort MapofRname.t) : sort MapofRname.t = match l with
  Select (r, p, s) →
   let sortR = MapofRname.find r srt in
     MapofRname.add s (typeinf_select p sortR)
       (MapofRname.remove r srt)
```

| *JoinDL* $((r, s), t) \rightarrow$
let *sortR* = *MapofRname.find r srt* in
let *sortS* = *MapofRname.find s srt* in
let *sortT* = *typeinf_join_dl sortR sortS* in
MapofRname.add t sortT (*MapofRname.remove r* (*MapofRname.remove s srt*))
| *Drop* $(a, (attr\_list, v), r, s) \rightarrow$
let *sortR* = *MapofRname.find r srt* in
let *sortS* = *typeinf_drop a* ((*SetofAttr.of_list attr_list*), *v*) *sortR* in
MapofRname.add s sortS (*MapofRname.remove r srt*)
| *Compose* $(l1, l2) \rightarrow$
let *sort1* = *qt_lens l1 srt* in
qt_lens l2 sort1

Simple instantiation of `join_template`$_{\overset{\cup ?}{\longleftarrow}_{F,\Phi}}$ with

$$\overset{\cup ?}{\longleftarrow}_F = \overset{\cup}{\longleftarrow}_F$$
$$\Phi(U, P, F) = (F \text{ is tree-form) and } (P \text{ ignores } outputs(F))$$

TODO : statically (at least dynamically) check $P_d \cup Q_d = \top_{UV}$

let *put_join_template* ((*r* : *rname*), (*pd* : *phrase*)) ((*s* : *rname*), (*qd* : *phrase*)) (*t* : *rname*)
((*dbJ*, *dbI*) : (*database* × *database*)) : *database* =
let (*sortR*, *iR*) = *MapofRname.find r dbI* in
let (*u*, _, _, *f*) = *sortR* in
let (*sortS*, *iS*) = *MapofRname.find s dbI* in
let (*v*, _, _, *g*) = *sortS* in
let (*sortT*, *jT*) = *MapofRname.find t dbJ* in
let *m0* = *relational_merge iR f* (*restrict_relation u jT*) in
let *n0* = *relational_merge iS g* (*restrict_relation v jT*) in
let *l* = *SetofRecord.diff* (*nat_join m0 n0*) *jT* in
let *ll* = *nat_join l* (*restrict_relation* (*SetofAttr.inter u v*) *jT*) in
let *la* = *SetofRecord.diff l ll* in
let *m* =
SetofRecord.diff
(*SetofRecord.diff m0*
(*restrict_relation u* (*SetofRecord.filter* (fun *r* → *eval_bool r pd*) *la*)))
(*restrict_relation u ll*) in
let *n* =
SetofRecord.diff n0
(*restrict_relation v* (*SetofRecord.filter* (fun *r* → *eval_bool r qd*) *la*)) in
let *dbI′* = *MapofRname.remove t dbJ* in
let *dbI′* = *MapofRname.add r* (*sortR*, *m*) *dbI′* in
let *dbI′* = *MapofRname.add s* (*sortS*, *n*) *dbI′* in

Index