Relational Lenses as Libraries

Rudi Horn – Haskell Symposium 2020

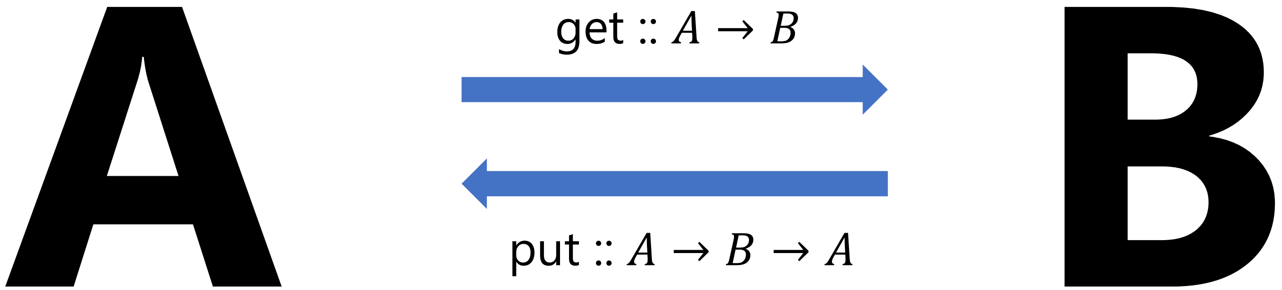
European Commission
Horizon 2020
European Union funding
for Research & Innovation

EPSRC Centre for Doctoral Training in
Pervasive Parallelism

THE UNIVERSITY of EDINBURGH
informatics

Lenses

A form of **bidirectional transformations** [1, 2]

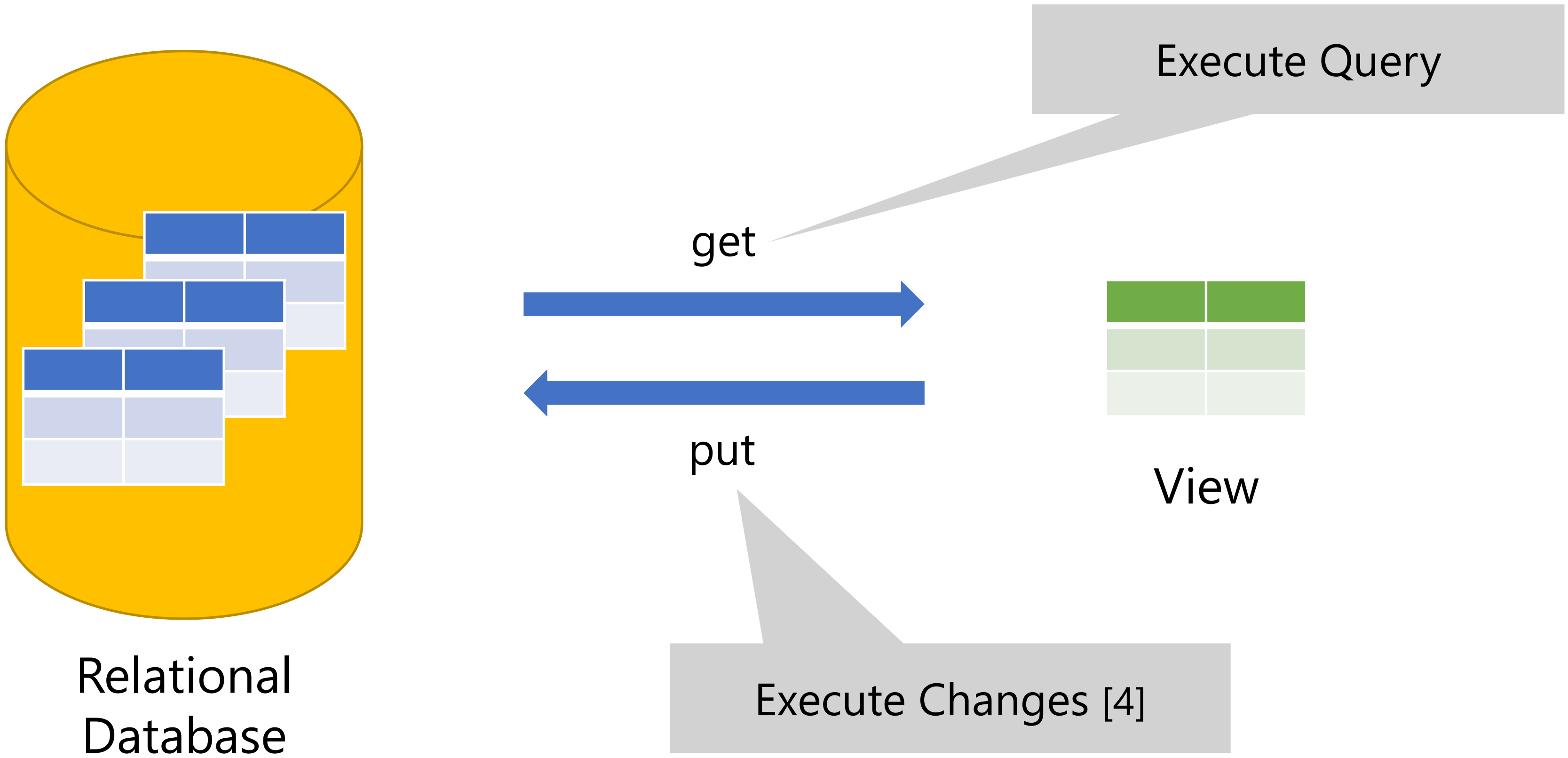


Example: Point { x :: Double; y :: Double }

get_x :: Point → Double
put_x :: Point → Double → Point

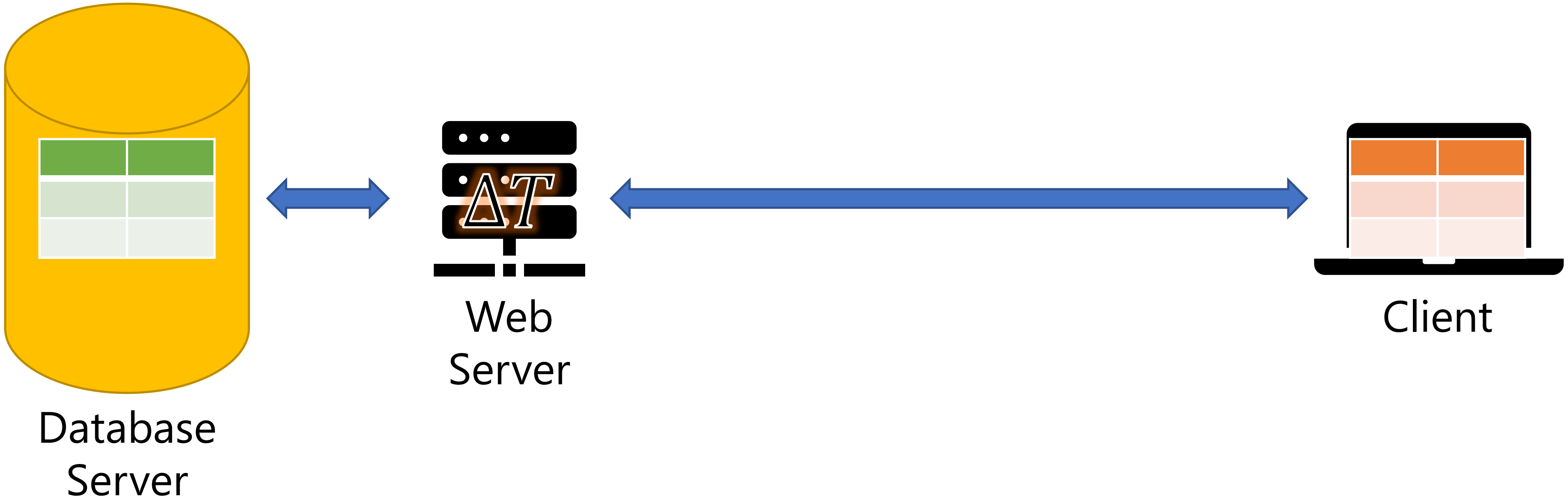
[1] Foster et al. "Combinators for bidirectional tree transformations: A linguistic approach to the view-update problem."
[2] Bohannon, Aaron, et al. "Boomerang: resourceful lenses for string data."

Relational Lenses [3]



[3] Bohannon, Pierce, and Vaughan. "Relational lenses: a language for updatable views."
[4] Horn, Perera, and Cheney. "Incremental relational lenses."

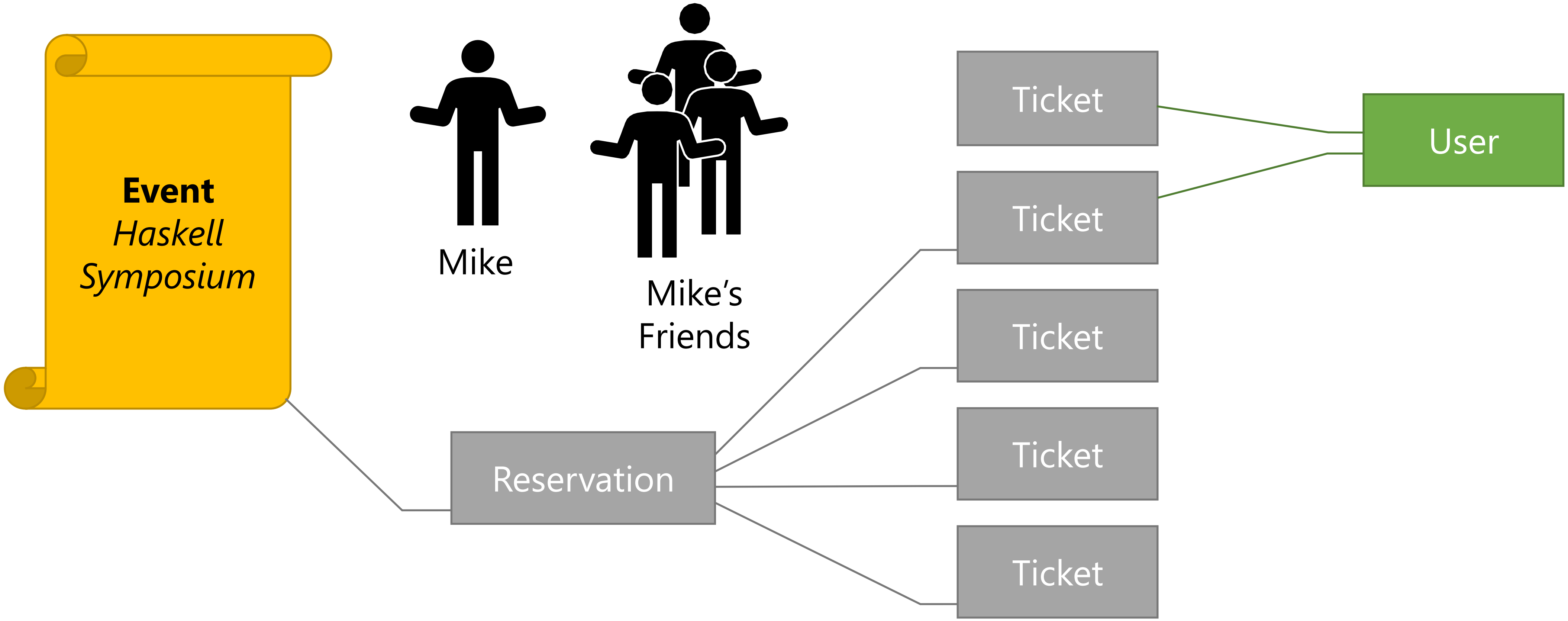
Relational Lenses



Typical Model-View-Controller workflow

Relational Lenses by Example

Lets build an event ticketing system!



Relational Lenses by Example

Lets build an event ticketing system!

```
db_connect = connect defaultConnectInfo {
    connectDatabase = "events",
    connectUser = "me",
    connectPassword = "password"
  }

-- in our IO Monad
conn <- db_connect
```

Relational Lenses by Example

Lets build an event ticketing system!

```
type Event = '[ '("event_id", Int), '("event_name", String) ]
events = prim @"events" @Event
  @'[ '["event_id"] --> '["event_name"]]

-- in our IO Monad
setup conn events
```

event_id	event_name

events

Relational Lenses by Example

```haskell
-- in our IO Monad
put conn events $ rows [(1, "haskell symposium"), (2, "icfp")]

get conn events
-- yields:
--    [{ event_id = 1, event_name = "haskell symposium" }
--    ,{ event_id = 2, event_name = "icfp" }]
```

event_id	event_name
1	haskell symposium
2	icfp

events

Relational Lenses by Example

```
type Reservation = '[ '("res_id", Int), '("event_id", Int) ]
reservations = prim @"reservations" @Reservation
  @'[ '["res_id"] --> '["event_id"] ]
```

res_id	event_id
1	1
2	1
3	2

reservations

Relational Lenses by Example

```
type Ticket = '[ '("res_id", Int), '("email", String) ]
tickets = prim @"tickets" @Ticket @'[]

type User =
  '[ '("email", String), '("title", String), '("name", String) ]
users = prim @"users" @User @'[ '["email"] --> '["title", "name"]]

user_tickets = join tickets users
res_view id =
  select (#res_id @= di id) user_tickets

-- in our IO Monad
setup conn user_tickets
put conn (res_view 1) $
  rows [(1, "mike@mail.com", "Mr.", "Mike")]
```

Relational Lenses by Example

```
type DefaultUser = '[
  '("name", 'P.String "<unknown>"),
  '("title", 'P.String "Mx.") ]
users_dr = dropl @DefaultUser @'["email"] users

res_view_simple id =
  select (#res_id @= di id) (join tickets users_dr)

set_res_emails conn id emails =
  put conn (res_view_simple id) $
    rows (map (\email -> (id, email)) emails)

-- in our IO Monad
set_res_emails conn 2 ["me@example.com", "mike@mail.com"]
```

Relational Lenses by Example

```
-- in our IO Monad
get conn (res_view 2)
```

res_id	email	title	name
2	mike@mail.com	Mr.	Mike
2	me@example.com	Mx.	<unknown >

Correctness of Relational Lenses

Should be **well-behaved**

For bidirectional transformations:

- $get\left(put(a, b)\right) = b$
- $put\left(a, get(a)\right) = a$

Correctness of Relational Lenses

Linearity of Tables

Functional Dependencies
$\boldsymbol{X \to Y}$

Correctness

Restrictions on Predicates

Consistency

[3] Bohannon, Pierce, and Vaughan. "Relational lenses: a language for updatable views."
[5] Horn, Fowler, and Cheney. “Language-Integrated Updatable Views"

Existing Relational Lenses

Implemented in **Links**

- Extended compiler to support Relational Lenses
  - Implements **Incremental** Relational Lenses [4]
  - Demonstrates **language integration** in functional setting [5]
- Difficult to maintain
- Compatibility with other language features (e.g. continuation serialisation)

https://links-lang.org

Links

[4] Horn, Perera, and Cheney. "Incremental relational lenses."
[5] Horn, Fowler, and Cheney. "Language-Integrated Updatable Views"

Lenses as Library

Better approach: Implement feature as a **Library**

- How to verify correctness statically?
- Implement with reusable language features
- Cleaner abstractions for intended feature

➢ Haskell type system sufficient for Relational Lenses

Type-level computation

Type Level Programming

Type Literals

```
type Hello = "Hello"
-- Hello has kind Symbol

type Five = 5
-- Five has kind Nat
```

Type Families

```
type family Length (l :: [k]) :: Nat where
  Length '[] = 0
  Length (_ ': xs) = 1 + Len xs

type Five = Length '[1, 2, 3, 4, 5]
```

Type Classes

```haskell
class Recoverable i t where
  recover :: Proxy i -> t

instance KnownSymbol s => Recoverable (s :: Symbol) String where
  recover p = symbolVal p

instance Recoverable 'True Bool where
  recover Proxy = True

instance Recoverable 'False Bool where
  recover Proxy = False
```

Constraints

```
type OnlyTwo l = (Length l ~ 2, Recoverable l [String])

only_two' :: forall l. OnlyTwo l => Proxy l -> [String]
only_two' Proxy = recover @l @[String] Proxy
```

Equality Constraint
$\tau_1 \sim \tau_2$

Equality Constraint
`Recoverable a String`

Lens Building Blocks

Tables

User tries to construct lens:

```
mylens = join (join tickets users) tickets
```

uses same table twice!

Need a constraint:

```
TablesDisjoint t1 t2 => ...
```

Tables

```
type Tables = [Symbol]

type family IsDisjoint (l :: [k]) (r :: [k]) :: Bool where
  IsDisjoint '[] _ = 'True
  IsDisjoint (x ': xs) ys =
    Not (IsElement x ys) && IsDisjoint xs ys

type TablesDisjoint l ls = IsDisjoint l ls ~ 'True
```

Record Types

Relational Lenses require representation for records / views

`res_id`	`email`	`title`	`name`
2	mike@mail.com	Mr.	Mike
2	me@example.com	Mx.	<unknown >

Record Types

```
type Env = [(Symbol, *)]

-- examples
type R = '[ '("A", String), '("B", String) ]
rec = toRow @R ("hello", "world")
    = Cons @"A" "hello" (Cons @"B" "world" Empty)

fetch @"A" rec -- "hello"
update @"A" "test" rec -- { A = "test", B = "world" }
update @"C" "test" rec -- { C = "test", A = "hello", B = "world" }
```

Record Projection

```
type family LookupType (env :: Env) (s :: Symbol) :: * where
  LookupType ('(key, val) ': xs) key = val
  LookupType (_ ': xs) key = LookupType xs key

type family ProjectEnv (s :: [Symbol]) (e :: Env) where
  ProjectEnv '[] _ = '[]
  ProjectEnv (x ': l) e = '(x, LookupType e x) ': ProjectEnv l e

class Project (s :: [Symbol]) (e :: Env) where
  project :: Row e -> Row (ProjectEnv s e)

instance Project '[] env where
  project _ = Empty
instance (Project xs env, Fetchable x env t evid)
  => Project (x ': xs) (env) where
  project r = Cons (fetch @x r) (project @xs @env r)
```

Record Set

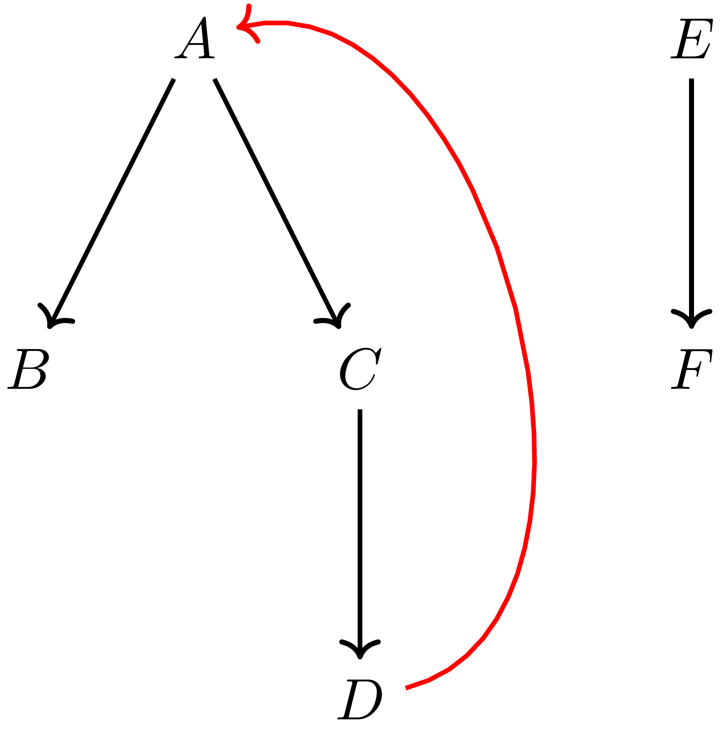
```haskell
type RecordsSet rt = Set (Row rt)

rows :: forall rt vals. (vals ~ R.TupleType rt, ToRow rt vals)
  => [vals] -> RecordsSet rt
rows vals = Set.fromList $ map (toRow @rt) vals

project :: forall s rt. (Project s rt)
  => RecordsSet rt -> RecordsSet (ProjectEnv s rt)
project rs = Set.map (R.project @s) rs
```

Functional Dependencies

Example: $email \rightarrow title, name$

Tree form:

Forrest with disjoint nodes



Functional Dependencies

```
data FunDep where
  FunDep :: [Symbol] -> [Symbol] -> FunDep

type family (-->) (l :: [Symbol]) (r :: [Symbol]) :: FunDep where
  xs --> ys = ('FunDep (SymAsSet xs) (SymAsSet ys) :: FunDep)

-- Example
type MyFds = '[ '["A", "B"] --> '["C"], '["C"] --> '["D", "E"]]
```

Functional Dependencies

```
type family IsInTreeForm (fds :: [FunDep]) where
  IsInTreeForm fds =
    AllDisjoint (Rights fds)
    && AllDisjoint (SLAsSet
      (Rights fds :++ Lefts fds))
    && IsAcyclic fds

type InTreeForm fds =
  OkOrError (IsInTreeForm fds)
  ('Text "Not in tree form.")
```

Predicates

Domain Specific Language (**DSL**) for predicates

- Predicate information retained in type
- Statically typable

Example for *`test = 5`*:

```
p :: HPhrase ('P.Var "quantity" :> 'P.Constant ('P.Int 5))
p = #test @= i @5
```

Predicates

Static predicates

- not always flexible enough,
- however not always necessary!

```
type PredRow = '[ '("quantity", Int), '("album", String)]

dynamic_pred
  :: Bool -> Int -> String -> HPhrase ('P.Dynamic PredRow Bool)
dynamic_pred b i s =
  if b
  then (dynamic @PredRow @Bool (#quantity @> di i))
  else (dynamic @PredRow @Bool (#album @= ds s))
```

Lens Constructors

Lens Sorts

Refinement type for lenses

```
data Sort where
  Sort :: Tables -> Env -> SPhrase -> [FunDep] -> Sort

-- get tables
type family Ts (s :: Sort) :: Tables where
  Ts ('Sort ts _ _ _) = ts

type QueryRow s = Row (Rt s)
```

Table Primitive Lens

```
-- data type definition
data Lens (s :: Sort) where
  Prim :: Lensable ('Sort '[table] rt p fds) snew => Lens snew
  -- ...

-- public constructor
prim :: forall table rt fds p fdsnew s snew.
  (s ~ 'Sort '[table] rt p fds,
  Lensable s snew)
  => Lens snew
prim = Prim @table @rt @p @fds

-- example
type Event = '[ '("event_id", Int), '("event_name", String) ]
events = prim @"events" @Event
  @'[ '["event_id"] --> '["event_name"]]
```

Select / Filter Lens

```
-- data type definition
data Lens (s :: Sort) where
  Select :: (Selectable p s snew) =>
    HPhrase p -> Lens s -> Lens snew
  -- ...

-- public constructor
select :: forall p s snew.
  (Selectable p s snew) => HPhrase p -> Lens s -> Lens snew
select pred l = Select pred l

-- example
res_view id =
  select (#res_id @= di id) user_tickets
```

Select / Filter Lens

```
type SelectableExp p rt pred fds =
  (TypesBool rt p,
   IgnoresOutputs pred fds,
   InTreeForm fds,
   SelectImplConstraints rt p pred fds)

type Selectable p s snew =
  (snew ~ 'Sort (Ts s) (Rt s) (Simplify (p :& (P s))) (Fds s),
   SelectableExp p (Rt s) (P s) (Fds s),
   LensCommon s,
   LensCommon snew)
```

Further Lenses

Drop / Join lenses also supported:

```
-- data type definition
data Lens (s :: Sort) where
  Drop :: Droppable env (key :: [Symbol]) s snew =>
    Proxy key -> Proxy env -> Lens s -> Lens snew
  Join :: Joinable s1 s2 snew joincols =>
    Lens s1 ->
    Lens s2 ->
    Lens snew
  -- ...
```

Conclusion

Type-level Computation

No extensions to Haskell necessary!

Lots of similar work on row types:

- https://hackage.haskell.org/package/CTRex-0.6
- https://hackage.haskell.org/package/row-types

Also type level sets:
https://hackage.haskell.org/package/type-level-sets-0.8.0.0

Future Work

Better error handling:

Haskell supports: `TypeError msg`

Lots of large / messy constraints

- Tend to be unavoidable without fully dependent types
- Better abstractions?

Other database server support (shouldn't be too difficult)

Serial column / auto incrementing column support

Results

- Haskell Lens Library
- Supports Incremental Relational Lens Semantics [4]
- Roughly ~3k of code
- Easy to use view-update for relational databases

[4] Horn, Rudi, Roly Perera, and James Cheney. "Incremental relational lenses."

Questions?