How to use the SIP

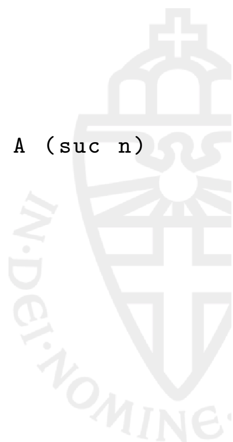
Simcha van Collem   Alex van der Hulst

Radboud University Nijmegen

21 December 2022

Matrices (as vectors)

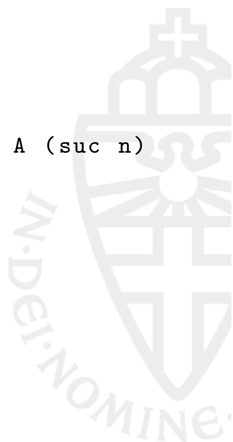
```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

VecMatrix : (A : Type) (m n : ℕ) → Type
VecMatrix A m n = Vec (Vec A n) m
```



Matrices (as vectors)

```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

VecMatrix : (A : Type) (m n : ℕ) → Type
VecMatrix A m n = Vec (Vec A n) m
```

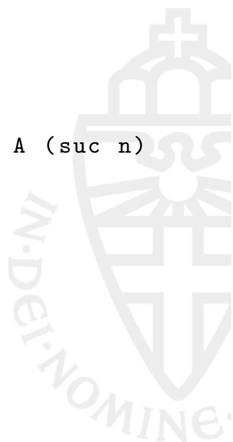
- Useful for computation



Matrices (as vectors)

```
data Vec (A : Type) : ℕ → Type where
  [] : Vec A zero
  _::_ : ∀ {n} (x : A) (xs : Vec A n) → Vec A (suc n)

VecMatrix : (A : Type) (m n : ℕ) → Type
VecMatrix A m n = Vec (Vec A n) m
```

- Useful for computation
- Difficult to prove properties about



Matrices (as pointwise function)

```
data Fin : ℕ → Type where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} (i : Fin n) → Fin (suc n)

FinMatrix : (A : Type) (m n : ℕ) → Type
FinMatrix A m n = Fin m → Fin n → A
```

Matrices (as pointwise function)

```
data Fin : ℕ → Type where
  zero : {n : ℕ} → Fin (suc n)
  suc : {n : ℕ} (i : Fin n) → Fin (suc n)

FinMatrix : (A : Type) (m n : ℕ) → Type
FinMatrix A m n = Fin m → Fin n → A
```

- Easy to prove for instance commutativity

Matrices (as pointwise function)

```
addFinMatrix : FinMatrix G m n → FinMatrix G m n →
               FinMatrix G m n
addFinMatrix M N i j = M i j + N i j

addFinMatrixComm : (M N : FinMatrix G m n) →
                    addFinMatrix M N ≡ addFinMatrix N M
addFinMatrixComm M N i k l = comm (M k l) (N k l) i
```

VecMatrix as Abelian group

- Addition on FinMatrix'es is commutative

VecMatrix as Abelian group

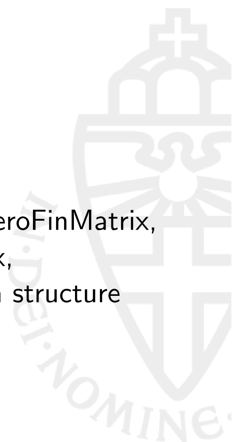
- Addition on FinMatrix'es is commutative
- Commutative addition via FinMatrix using $FinMatrix \simeq VecMatrix$ and addFinMatrixComm

$$\begin{array}{ccc} VecMatrix \times VecMatrix & \dashrightarrow & VecMatrix \times VecMatrix \\ \simeq \downarrow & & \uparrow \simeq \\ FinMatrix \times FinMatrix & \xrightarrow{addFinMatrixComm} & FinMatrix \times FinMatrix \end{array}$$

VecMatrix as Abelian group

- Addition on FinMatrix'es is commutative
- Commutative addition via FinMatrix (Very naive)
- Rather have commutative addition addVecMatrix

VecMatrix as Abelian group



- Addition on FinMatrix'es is commutative
- Commutative addition via FinMatrix (Very naive)
- Rather have commutative addition addVecMatrix
- Transfer via SIP if FinMatrix≃VecMatrix sends zeroFinMatrix, negFinMatrix and addFinMatrix to zeroVecMatrix, negVecMatrix and addVecMatrix i.e. if we have a structure preserving equivalence.

VecMatrix as Abelian group (using SIP)

- Structure preserving equivalence e

$$\begin{array}{ccc} VecMatrix \times VecMatrix & \xrightarrow{addVecMatrixComm} & VecMatrix \times VecMatrix \\ \downarrow e & & \uparrow e^{-1} \\ FinMatrix \times FinMatrix & \xrightarrow{addFinMatrixComm} & FinMatrix \times FinMatrix \end{array}$$

Queues

$$\text{RawQueueStructure } X = \underbrace{X}_{\text{empty queue}} \times \underbrace{(A \to X \to X)}_{\text{enqueue}} \times \underbrace{(X \to \text{Maybe}(X \times A))}_{\text{dequeue}}$$

```
ListQueue : (A : Type) → Queue A
ListQueue A = queue (List A) [] _::_ last

BatchedQueue : (A : Type) → Queue A
BatchedQueue A =
queue (List A × List A) ([] , [])
(λ x (xs , ys) → fastcheck (x :: xs , ys))
(λ { (_ , []) → nothing ; (xs , x :: ys) →
just (fastcheck (xs , ys) , x) })
where
fastcheck : {A : Type} → List A × List A → List A × List A
fastcheck (xs , ys) = if isEmpty ys then ([] , reverse xs)
else (xs , ys)
```

Queues

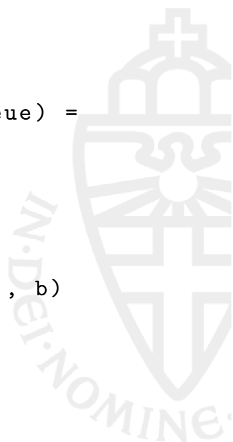
$$\text{RawQueueStructure } X = \underbrace{X}_{\text{empty queue}} \times \underbrace{(A \to X \to X)}_{\text{enqueue}} \times \underbrace{(X \to \text{Maybe}(X \times A))}_{\text{dequeue}}$$

```
ListQueue : (A : Type) → Queue A
ListQueue A = queue (List A) [] _::_ last

BatchedQueue : (A : Type) → Queue A
BatchedQueue A =
queue (List A × List A) ([] , [])
(λ x (xs , ys) → fastcheck (x :: xs , ys))
(λ { (_ , []) → nothing ; (xs , x :: ys) →
just (fastcheck (xs , ys) , x) })
where
fastcheck : {A : Type} → List A × List A → List A × List A
fastcheck (xs , ys) = if isEmpty ys then ([] , reverse xs)
else (xs , ys)
```

Note: There is no logical structure-preserving equivalence

A Queue axiom

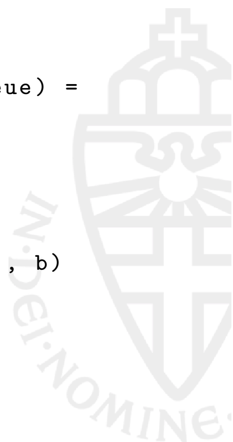
```
dequeueEnqueueAxiom Q (empty, enqueue, dequeue) =
∀ a q → dequeue (enqueue a q)
≡ just (returnOrEnq a (dequeue q))
where
returnOrEnq : A → Maybe (Q × A) → Q × A
returnOrEnq a nothing = (empty , a)
returnOrEnq a (just (q , b)) = (enqueue a q , b)
```



A Queue axiom

```
dequeueEnqueueAxiom Q (empty, enqueue, dequeue) =
∀ a q → dequeue (enqueue a q)
≡ just (returnOrEnq a (dequeue q))
where
returnOrEnq : A → Maybe (Q × A) → Q × A
returnOrEnq a nothing = (empty , a)
returnOrEnq a (just (q , b)) = (enqueue a q , b)
```

dequeue (enqueue c ([b, a], [])) ≡ just (([], [b, c]), a)



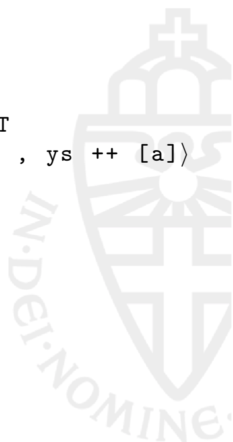
A Queue axiom

```
dequeueEnqueueAxiom Q (empty, enqueue, dequeue) =
∀ a q → dequeue (enqueue a q)
≡ just (returnOrEnq a (dequeue q))
where
returnOrEnq : A → Maybe (Q × A) → Q × A
returnOrEnq a nothing = (empty , a)
returnOrEnq a (just (q , b)) = (enqueue a q , b)
```

dequeue (enqueue c ([b, a], [])) ≡ just (([], [b, c]), a)
just (returnOrEnq c (dequeue ([b, a], []))) ≡ just (([c], [b]), a)

HIT

```
data BatchedQueueHIT : Type where
  Q⟨_,_⟩ : List A → List A → BatchedQueueHIT
  tilt : ∀ xs ys a → Q⟨xs ++ [a], ys⟩ ≡ Q⟨xs , ys ++ [a]⟩
  squash : isSet BatchedQueueHIT
```



HIT

```
data BatchedQueueHIT : Type where
  Q⟨_,_⟩ : List A → List A → BatchedQueueHIT
  tilt : ∀ xs ys a → Q⟨xs ++ [a], ys⟩ ≡ Q⟨xs , ys ++ [a]⟩
  squash : isSet BatchedQueueHIT

appendReverse : {A : Type} → (BatchedQueue A).Q
                           → (ListQueue A).Q
appendReverse (xs, ys) = xs ++ reverse ys
```

HIT

```
data BatchedQueueHIT : Type where
  Q⟨_,_⟩ : List A → List A → BatchedQueueHIT
  tilt : ∀ xs ys a → Q⟨xs ++ [a], ys⟩ ≡ Q⟨xs , ys ++ [a]⟩
  squash : isSet BatchedQueueHIT

appendReverse : {A : Type} → (BatchedQueue A).Q
                           → (ListQueue A).Q
appendReverse (xs, ys) = xs ++ reverse ys
```

- Extend this function to BatchedQueueHIT

HIT

```
data BatchedQueueHIT : Type where
  Q⟨_,_⟩ : List A → List A → BatchedQueueHIT
  tilt : ∀ xs ys a → Q⟨xs ++ [a], ys⟩ ≡ Q⟨xs , ys ++ [a]⟩
  squash : isSet BatchedQueueHIT

appendReverse : {A : Type} → (BatchedQueue A).Q
                           → (ListQueue A).Q
appendReverse (xs, ys) = xs ++ reverse ys
```

- Extend this function to BatchedQueueHIT
- Obtained S-structure preserving equivalence

HIT

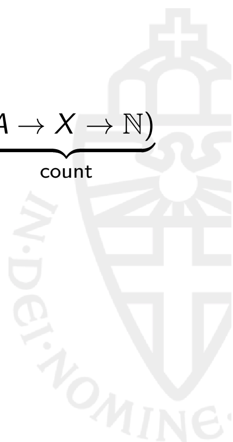
```
data BatchedQueueHIT : Type where
  Q⟨_,_⟩ : List A → List A → BatchedQueueHIT
  tilt : ∀ xs ys a → Q⟨xs ++ [a], ys⟩ ≡ Q⟨xs , ys ++ [a]⟩
  squash : isSet BatchedQueueHIT

appendReverse : {A : Type} → (BatchedQueue A).Q
                → (ListQueue A).Q
appendReverse (xs, ys) = xs ++ reverse ys
```

- Extend this function to BatchedQueueHIT
- Obtained S-structure preserving equivalence
- Are able to use SIP

Multisets

RawMultisetStructure $X =$

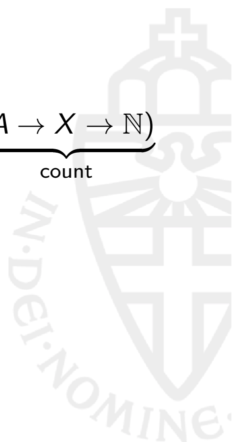
$$\underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$



Multisets

$$\text{RawMultisetStructure } X = \underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$

- As `List A`



Multisets

RawMultisetStructure $X =$

$$\underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$

- As `List A`
- As AssocList: `List (A` $\times$ `ℕ)`

Multisets

RawMultisetStructure $X =$

$$\underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$

- As `List A`
- As AssocList: `List (A` $\times$ `ℕ)`
- Both do not satisfy:

```
insert 1 (insert 2 xs) ≡ insert 2 (insert 1 xs)
```

Multisets

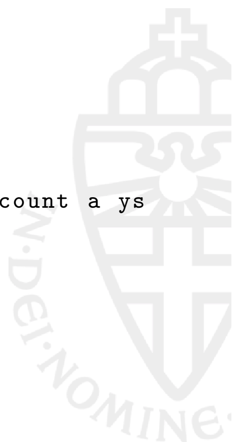
$$\text{RawMultisetStructure } X = \underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$

- As `List A`
- As AssocList: `List (A` $\times$ $\mathbb{N}$`)`
- Both do not satisfy:

```
insert 1 (insert 2 xs) ≡ insert 2 (insert 1 xs)
```

- No structure preserving equivalence

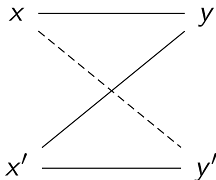
Multiset extensionality



```
R_multiset : List A → AssocList A → Type
R_multiset xs ys = (a : A) → L.count a xs ≡ AL.count a ys
```

Zigzag-complete relation

Definition: A prop-valued relation $R\colon X \to Y \to \mathrm{Type}$ is *zigzag-complete* when for any $r_0\colon R\ x\ y, r_1\colon R\ x'\ y$ and $r_2\colon R\ x'\ y'$, we have $R\ x\ y'$.



Zigzag-complete relation

- Relation $R\colon X \to Y \to \text{Type}$ is prop-valued when $R\ x\ y$ is a proposition for all $x : X, y : Y$
- For $R\colon X \to Y \to \text{Type}$, we define its inverse $R^{-1}\colon Y \to X \to \text{Type}$ as $R^{-1}\ y\ x = R\ x\ y$
- For $R\colon X \to Y \to \text{Type}$ prop-valued is R^{-1} also prop valued

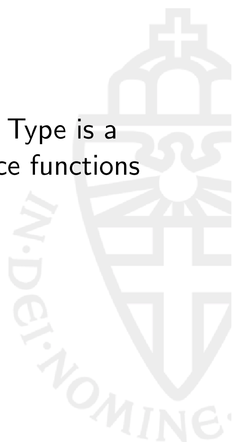
Zigzag-complete relation

- Relation $R\colon X \to Y \to \text{Type}$ is prop-valued when $R\ x\ y$ is a proposition for all $x : X, y : Y$
- For $R\colon X \to Y \to \text{Type}$, we define its inverse $R^{-1}\colon Y \to X \to \text{Type}$ as $R^{-1}\ y\ x = R\ x\ y$
- For $R\colon X \to Y \to \text{Type}$ prop-valued is R^{-1} also prop valued
- For $R\colon X \to Y \to \text{Type}$ and $S\colon Y \to Z \to \text{Type}$, we define their prop-valued composite as $(R \cdot S)\ x\ z = \| \sum[y \in Y] R\ x\ y \times S\ y\ z \|$

Zigzag-complete relation

- Relation $R \colon X \to Y \to \text{Type}$ is prop-valued when $R\ x\ y$ is a proposition for all $x : X, y : Y$
- For $R \colon X \to Y \to \text{Type}$, we define its inverse $R^{-1} \colon Y \to X \to \text{Type}$ as $R^{-1}\ y\ x = R\ x\ y$
- For $R \colon X \to Y \to \text{Type}$ prop-valued is R^{-1} also prop valued
- For $R \colon X \to Y \to \text{Type}$ and $S \colon Y \to Z \to \text{Type}$, we define their prop-valued composite as $(R \cdot S)\ x\ z = \| \sum[y \in Y] R\ x\ y \times S\ y\ z \|$
- Zigzag-complete relations induce two partial equivalence relations: $R \cdot R^{-1}$ on X and $R^{-1} \cdot R$ on Y.
- We denote these as $R^{\leftarrow}$ and $R^{\rightarrow}$

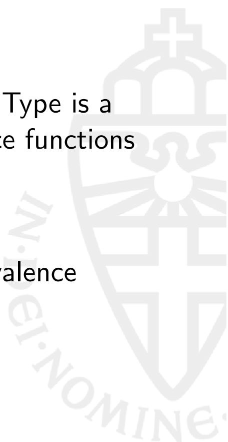
Quasi-equivalence relation (QER)



Definition: A zigzag-complete relation $R\colon X \to Y \to \text{Type}$ is a *quasi–equivalence relation* (QER) when there are choice functions

1. $(x\colon X) \to \| \sum[y \in Y] R\ x\ y \|$
2. $(y\colon Y) \to \| \sum[x \in X] R\ x\ y \|$

Quasi-equivalence relation (QER)



Definition: A zigzag-complete relation $R\colon X \to Y \to \text{Type}$ is a *quasi–equivalence relation* (QER) when there are choice functions

1. $(x\colon X) \to \| \sum[y \in Y] R\ x\ y \|$
2. $(y\colon Y) \to \| \sum[x \in X] R\ x\ y \|$

Note: R is a QER if and only if $R^{\leftarrow}$ and $R^{\rightarrow}$ are equivalence relations

Quasi-equivalence relation (QER)

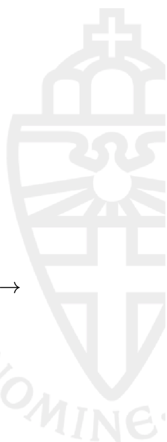
Definition: A zigzag-complete relation $R\colon X \to Y \to \text{Type}$ is a *quasi–equivalence relation* (QER) when there are choice functions

1. $(x\colon X) \to \| \sum[y \in Y] R\ x\ y \|$
2. $(y\colon Y) \to \| \sum[x \in X] R\ x\ y \|$

Note: R is a QER if and only if $R^{\leftarrow}$ and $R^{\rightarrow}$ are equivalence relations

Lemma: Given a QER R, there is an equivalence $e\colon X/R^{\leftarrow} \simeq Y/R^{\rightarrow}$ such that for every $x\colon X$ and $y\colon Y$, we have equivFun e [x] $\equiv$ [y] if and only if $R\ x\ y$.

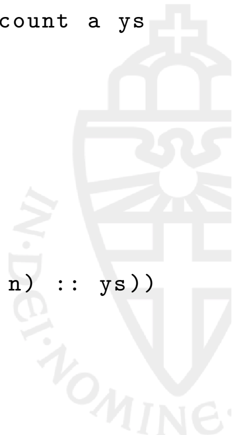
Proof sketch



- Fix an $x : A$
- $[_] \circ \mathsf{fst} : \sum[y \in Y] R\ x\ y \to Y/R^{\to}$ is constant
- This induces $f_x : \| \sum[y \in Y] R\ x\ y \| \to Y/R^{\to}$, with $f_x \circ [_] \equiv [_] \circ \mathsf{fst}$
- Compose with choice function to get $X \to Y/R^{\to}$
- After checking well-definedness we get $X/R^{\leftarrow} \to Y/R^{\to}$
- $Y/R^{\to} \to X/R^{\leftarrow}$ is constructed in a analog way

R_{multiset} is a QER

```
R_multiset : List A → AssocList A → Type
R_multiset xs ys = (a : A) → L.count a xs ≡ AL.count a ys

φ : List A → AssocList A
φ [] = []
φ (x :: xs) = AL.insert x (φ xs)

ψ : AssocList A → List A
ψ [] = []
ψ ((x , 0) :: ys) = ψ ys
ψ ((x , suc n) :: ys) = L.insert x (ψ ((x , n) :: ys))
```



R_{multiset} is a QER

```
Rmultiset : List A → AssocList A → Type
Rmultiset xs ys = (a : A) → L.count a xs ≡ AL.count a ys

φ : List A → AssocList A
φ [] = []
φ (x :: xs) = AL.insert x (φ xs)

ψ : AssocList A → List A
ψ [] = []
ψ ((x , 0) :: ys) = ψ ys
ψ ((x , suc n) :: ys) = L.insert x (ψ ((x , n) :: ys))
```

- R_{multiset} is zigzag-complete

R_{multiset} is a QER

```
Rmultiset : List A → AssocList A → Type
Rmultiset xs ys = (a : A) → L.count a xs ≡ AL.count a ys

φ : List A → AssocList A
φ [] = []
φ (x :: xs) = AL.insert x (φ xs)

ψ : AssocList A → List A
ψ [] = []
ψ ((x , 0) :: ys) = ψ ys
ψ ((x , suc n) :: ys) = L.insert x (ψ ((x , n) :: ys))
```

- R_{multiset} is zigzag-complete
- φ, ψ induce choice functions as they preserve count

R_{multiset} is a QER

```
Rmultiset : List A → AssocList A → Type
Rmultiset xs ys = (a : A) → L.count a xs ≡ AL.count a ys

φ : List A → AssocList A
φ [] = []
φ (x :: xs) = AL.insert x (φ xs)

ψ : AssocList A → List A
ψ [] = []
ψ ((x , 0) :: ys) = ψ ys
ψ ((x , suc n) :: ys) = L.insert x (ψ ((x , n) :: ys))
```

- R_{multiset} is zigzag-complete
- φ, ψ induce choice functions as they preserve count
- R_{multiset} is a thus a QER, giving an equivalence List $A/R^{\leftarrow}_{\text{multiset}} \simeq$ AssocList $A/R^{\rightarrow}_{\text{multiset}}$

Structured Relations

```
StrRel : (S : Type → Type) → Type₁
StrRel S = {X Y : Type} → (X → Y → Type)
           → (S X → S Y → Type)
```

Definition: A candidate ρ: StrRel S is *suitable* when the following hold:

- Set- and prop-preservation: If X is a set then $S\ X$ is a set. If R is a prop-valued-relation then $\rho\ R$ is a prop-valued relation.
- Symmetry: For any prop-valued relation R, if $(\rho\ R)\ s\ t$, then $(\rho\ R^{-1})\ t\ s$.
- Transitivity: : For any prop-valued relations $R : X \to Y, R' : Y \to Z$, if $(\rho\ R)\ s\ t$ and $(\rho\ R')\ t\ u$, then $(\rho\ R \cdot R')\ s\ u$.
- Descent to quotients: If $R\colon X \to X \to \text{Type}$ is a prop-valued relation and $(\rho\ R)\ s\ s$ for some $s : S\ X$, then there is a *unique* $\bar{s}\colon S\ (X/R)$ such that we have some $r\colon \rho\ (\text{graph}\ [_])\ s\ \bar{s}$

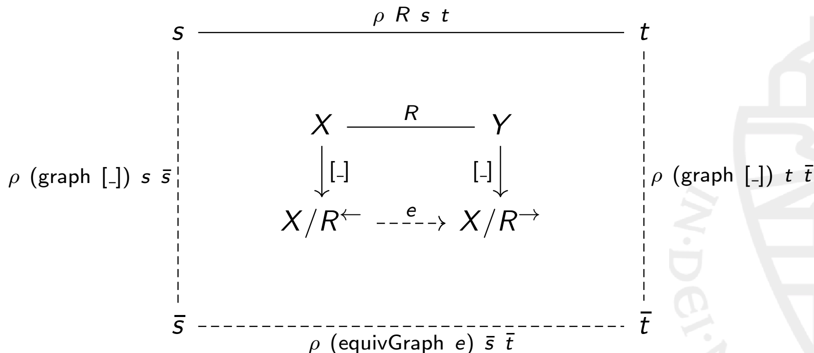
Relational Structure Identity Principle

$$s \overset{\rho\ R\ s\ t}{\longrule} t$$

$$\begin{array}{ccc} X & \overset{R}{\longrule} & Y \\ \downarrow{[_]} & & {[_]}\downarrow \\ X/R^{\leftarrow} & \overset{e}{\dashrightarrow} & Y/R^{\rightarrow} \end{array}$$

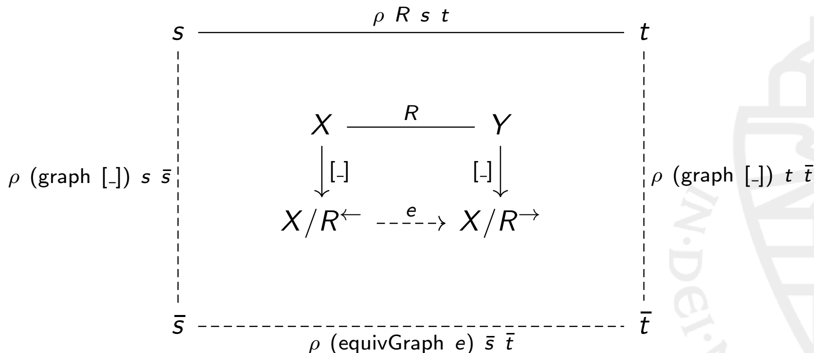
- e is the equivalence given by the previous lemma

Relational Structure Identity Principle



- e is the equivalence given by the previous lemma

Relational Structure Identity Principle



- e is the equivalence given by the previous lemma
- If ρ restricted to equivalences is univalent notion of structured equivalence, we obtain $(X/R^{\leftarrow}, \bar{s}) \equiv (Y/R^{\rightarrow}, \bar{t})$

Building structures

Suitable univalent relational structures are closed under the following type formers:

$$P\ X, Q\ X := X \mid A \mid P\ X \times Q\ X \mid \text{Maybe}\ (P\ X)$$
$$S\ X, T\ X := P\ X \mid S\ X \times T\ X \mid P\ X \to S\ X \mid \text{Maybe}\ (S\ X)$$

Differences with previous presentation (equivalent structures):

- A should be a set
- Positivity restriction for $\to$ type
- Prevents $\lambda\ X \to (X \to X) \to X$

Multiset structured relation

$$\text{RawMultisetStructure } X = \underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$

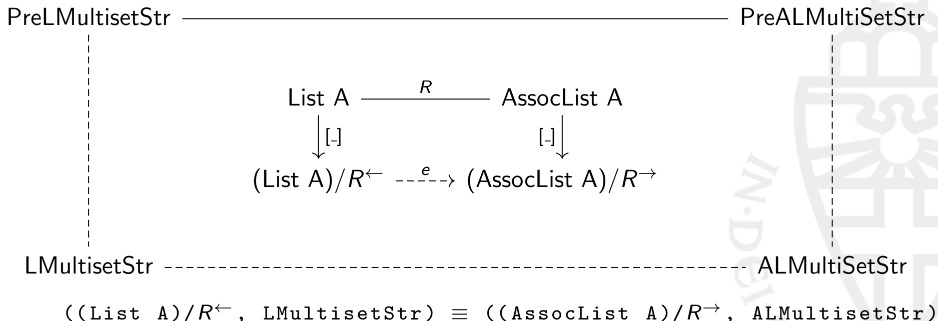
```
R : List A → AssocList A → Type
R xs ys = (a : A) → L.count a xs ≡ AL.count a ys
```

Multiset structured relation

$$\text{RawMultisetStructure } X = \underbrace{X}_{\text{empty Multiset}} \times \underbrace{(A \to X \to X)}_{\text{insert}} \times \underbrace{(X \to X \to X)}_{\text{union}} \times \underbrace{(A \to X \to \mathbb{N})}_{\text{count}}$$

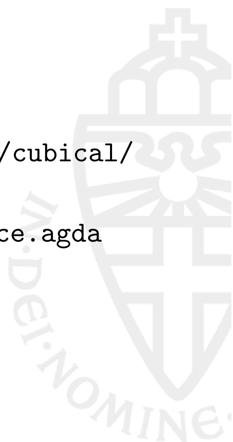
```
R : List A → AssocList A → Type
R xs ys = (a : A) → L.count a xs ≡ AL.count a ys

preservesEmp : R [] []
preservesCount : ∀ x {xs ys} → R xs ys
                 → L.count x xs ≡ AL.count x ys
preservesInsert : ∀ x {xs ys} → R xs ys
                 → R (L.insert x xs) (AL.insert x ys)
preservesUnion : ∀ {xs ys} → R xs ys
                 → ∀ {xs' ys'} → R xs' ys'
                 → R (L.union xs xs') (AL.union ys ys')
```

SIP on multisets



((List A)/$R^{\leftarrow}$, LMultisetStr) ≡ ((AssocList A)/$R^{\rightarrow}$, ALMultisetStr)

Closing remarks



- Cubical Agda repo: `https://github.com/agda/cubical/`
- Paper located at `Cubical/Papers/RepresentationIndependence.agda`