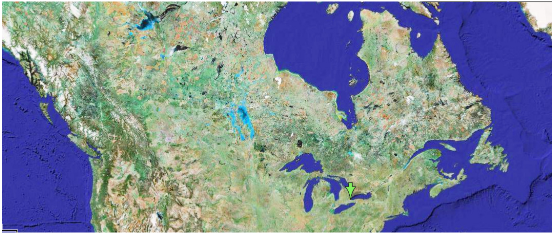
Towards Certifiable Implementation of Graph Transformation via Relation Categories

Wolfram Kahl

Software Quality Research Laboratory

McMaster University

Hamilton, Ontario, Canada



17 September 2012 — RAMiCS 13 — Cambridge, UK

This research is supported by the National Science and Engineering Research Council of Canada, NSERC.

The Categoric Approach to Graph Transformation

- There are various approaches to graph transformation/grammars:
  - Node-label controlled
  - Programmed
  - Hyperedge replacement
  - Categoric approach
    - **Double pushout-approach (DPO)**
    - Single pushout-approach
    - Single/double pullback-approach
    - …
  - Relation-algebraic approach
    - "Double pullout-approach"
    - …
- Graphs can be considered as unary algebras: "Algebraic Approach"
- Currently fashionable framework: "adhesive HLR categories"

Double-Pushout Transformations

$$\begin{tikzcd}
\mathcal{L} \arrow[d, "M"'] & \mathcal{G} \arrow[l, "\Phi_{\mathrm{L}}"'] \arrow[r, "\Phi_{\mathrm{R}}"] \arrow[d, "E"'] & \mathcal{R} \arrow[d, "N"'] \\
\mathcal{A} & \mathcal{H} \arrow[l, "\Psi_{\mathrm{L}}"'] \arrow[r, "\Psi_{\mathrm{R}}"] & \mathcal{B}
\end{tikzcd}$$

- Given a **rule** $\mathcal{L} \xleftarrow{\Phi_{\mathrm{L}}} \mathcal{G} \xrightarrow{\Phi_{\mathrm{R}}} \mathcal{R}$
- and a **matching** M into an **application graph** $\mathcal{A}$
- **construct pushout complement** $\xrightarrow{E} \mathcal{H} \xrightarrow{\Psi_{\mathrm{L}}}$
- **construct pushout** $\xrightarrow{\Psi_{\mathrm{R}}} \mathcal{B} \xleftarrow{N}$
- $\mathcal{B}$ is the **result graph**

How do we implement all that?

How do we go about
implementing transformation mechanisms like that?

How to Implement Categoric Graph Transformation?

Requirement 1 — Graph Category:

Represent graphs and graph homomorphisms as data, and implement pushouts and other categoric operations.

- Graphs structure categories
  are categories of unary algebras, or of diagrams
- Base category pushouts produce diagram category pushouts

Design Decision 1:

Implement graph structure categories abstractly on top of a base category

Diagram Pushouts from Base Pushouts

```
DiagMorPushout : HasPushouts compOp → {A B C : Diagram} (F : DiagMor A B) (G : DiagMor A C) → Pushout DiagMorCompOp F G
DiagMorPushout HP {A} {B} {C} FF GG = record
  {obj = mkDiagram (record
    {mapN = PN.D₀
    ; mapE = PE.eD
    ; congE = λ {n₁} {n₂} {e₁} {e₂} e₁≈e₂ → let open PE in ≈-sym (eDunique e₁
                 (eDcommute-R e₂ ⟨≈≈˘⟩ ⨾-cong₁ (DMor-cong B e₁≈e₂))
                 (eDcommute-S e₂ ⟨≈≈˘⟩ ⨾-cong₁ (DMor-cong C e₁≈e₂)))
    })
  ; left  = record {transform = PN.R; commute = PE.eDcommute-R}
  ; right = record {transform = PN.S; commute = PE.eDcommute-S}
  ; prf = record
    {commutes = PN.commutes
    ; universal = λ {Z} {P} {Q} F⨾P≈G⨾Q → let
      open PN
      trU = λ n → PO.univMor n (F⨾P≈G⨾Q n)
      U-left : (n : Node) → R n ⨾ trU n ≈ transform P n
      U-left n = PO.univMor-factors-left n (F⨾P≈G⨾Q n)
      U-right : (n : Node) → S n ⨾ trU n ≈ transform Q n
      U-right n = PO.univMor-factors-right n (F⨾P≈G⨾Q n)
      U = record {transform = trU
                 ; commute = λ {n₁} {n₂} e → let
                   P₁ = transform P n₁
                   Q₁ = transform Q n₁
                   P₂ = transform P n₂
                   Q₂ = transform Q n₂
                   open PE e
                   eZ = DMor Z e
                   V = trU n₁ ⨾ eZ
                   R₁⨾V≈P₁⨾eZ : R₁ ⨾ V ≈ P₁ ⨾ eZ
                   R₁⨾V≈P₁⨾eZ = ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (U-left n₁)
                   S₁⨾V≈Q₁⨾eZ : S₁ ⨾ V ≈ Q₁ ⨾ eZ
                   S₁⨾V≈Q₁⨾eZ = ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ (U-right n₁)
                   V′ = eD ⨾ trU n₂
                   R₁⨾V′≈P₁⨾eZ : R₁ ⨾ V′ ≈ P₁ ⨾ eZ
                   R₁⨾V′≈P₁⨾eZ = ≈-begin
                       R₁ ⨾ eD ⨾ trU n₂
                     ≈⟨ ⨾-cong₁&₂₁ eDcommute-R ⟩
                       eB ⨾ R₂ ⨾ trU n₂
                     ≈⟨ ⨾-cong₂ (U-left n₂) ⟩
                       eB ⨾ P₂
                     ≈˘⟨ commute P e ⟩
                       P₁ ⨾ eZ
                     □
                   S₁⨾V′≈Q₁⨾eZ : S₁ ⨾ V′ ≈ Q₁ ⨾ eZ
                   S₁⨾V′≈Q₁⨾eZ = ≈-begin
                       S₁ ⨾ eD ⨾ trU n₂
                     ≈⟨ ⨾-cong₁&₂₁ eDcommute-S ⟩
                       eC ⨾ S₂ ⨾ trU n₂
                     ≈⟨ ⨾-cong₂ (U-right n₂) ⟩
                       eC ⨾ Q₂
                     ≈˘⟨ commute Q e ⟩
                       Q₁ ⨾ eZ
                     □
                   eZ-commutes : F n₁ ⨾ P₁ ⨾ eZ ≈ G n₁ ⨾ Q₁ ⨾ eZ
                   eZ-commutes = ⨾-cong₁&₂₁ (F⨾P≈G⨾Q n₁)
                   ∃!U′ : PushoutUniv compOp (F n₁) (G n₁) R₁ S₁ eZ-commutes
                   ∃!U′ = PO.universal n₁ eZ-commutes
                   V≈U′ = PushoutUniv.univMor-unique compOp
                     ∃!U′ R₁⨾V≈P₁⨾eZ S₁⨾V≈Q₁⨾eZ
                   V′≈U′ = PushoutUniv.univMor-unique compOp
                     ∃!U′ R₁⨾V′≈P₁⨾eZ S₁⨾V′≈Q₁⨾eZ
                   in V≈U′ ⟨≈≈˘⟩ V′≈U′
                   }
      in record
        {univMor = U
        ; univMor-factors-left = U-left
        ; univMor-factors-right = U-right
        ; univMor-unique = λ {V} R⨾V≈P S⨾V≈Q n →
          PushoutUniv.univMor-unique compOp
            (PO.universal n (F⨾P≈G⨾Q n))
          {transform V n} (R⨾V≈P n) (S⨾V≈Q n)
        }
    }
  }
  where
    open Diagram
    open DiagMor
    open HasPushouts compOp HP
      -- open Pushout compOp
    module PN (n : Node) where
      F = transform FF n
      G = transform GG n
      PO : Pushout compOp F G
      PO = pushout F G
      module PO = Pushout compOp PO
      open PO public using (commutes) renaming
        (obj to D₀; left to R; right to S; universal to PO-universal)
    module PE {n₁ n₂ : Node} (e : Edge n₁ n₂) where
      open PN n₁ public using () renaming
        (F to F₁; G to G₁; R to R₁; S to S₁
        ; PO-universal to PO₁-universal)
      open PN n₂ public using () renaming (commutes to commutes₂
        ; F to F₂; G to G₂; R to R₂; S to S₂; D₀ to D₂)
      eB = DMor B e
      eC = DMor C e
      R′ = eB ⨾ R₂
      S′ = eC ⨾ S₂
      F⨾R′≈G⨾S′ : F₁ ⨾ R′ ≈ G₁ ⨾ S′
      F⨾R′≈G⨾S′ = ≈-begin
          F₁ ⨾ DMor B e ⨾ R₂
        ≈⟨ ⨾-cong₁&₂₁ (commute FF e) ⟩
          DMor A e ⨾ F₂ ⨾ R₂
        ≈⟨ ⨾-cong₂ commutes₂ ⟩
          DMor A e ⨾ (G₂ ⨾ S₂)
        ≈˘⟨ ⨾-cong₁&₂₁ (commute GG e) ⟩
          G₁ ⨾ DMor C e ⨾ S₂
        □
      eU : PushoutUniv compOp F₁ G₁ R₁ S₁ F⨾R′≈G⨾S′
      eU = PO₁-universal {D₂} {R′} {S′} F⨾R′≈G⨾S′
      eD = PushoutUniv.univMor compOp eU
      eDcommute-R : R₁ ⨾ eD ≈ R′
      eDcommute-R = PushoutUniv.univMor-factors-left compOp eU
      eDcommute-S : S₁ ⨾ eD ≈ S′
      eDcommute-S = PushoutUniv.univMor-factors-right compOp eU
      eDunique : ∀ {V} → R₁ ⨾ V ≈ R′ → S₁ ⨾ V ≈ S′ → V ≈ eD
      eDunique = PushoutUniv.univMor-unique compOp eU
```

$$
\begin{aligned}
\mathsf{DiagMorHasPushouts} &: \mathsf{HasPushouts\ compOp} \\
&\quad \to \mathsf{HasPushouts\ DiagMorCompOp} \\
\mathsf{DiagMorHasPushouts\ HP} &= \mathbf{record}\ \{\mathsf{pushout} = \mathsf{DiagMorPushout\ HP}\}
\end{aligned}
$$

My Choice for Formalisation and Implementation: Agda

- Agda is a dependently typed functional programming language
- Agda is a proof assistant based on Per Martin-Löf's intuitionistic type theory
- Syntactically and "culturally" close to Haskell
- Different semantics: strongly normalising, no $\perp$ values
- Dependently typed: No distinction between terms, types, and kinds
- **"Just a mechanised mathematical notation"**

  that lets me write the mathematics in a natural way
- **Normalisation provides execution:**

  $\Longrightarrow$ **Programming inside mathematics**

Diagram Pushouts from Base Pushouts

```
DiagMorPushout : HasPushouts compOp → {A B C : Diagram} (F : DiagMor A B) (G : DiagMor A C) → Pushout DiagMorCompOp F G
DiagMorPushout HP {A} {B} {C} FF GG = record
  {obj = mkDiagram (record
    {mapN = PN.D0
    ; mapE = PE.eD
    ; congE = λ {n1} {n2} {e1} {e2} e1≈e2 → let open PE in ≈-sym (eDunique e1
                 (eDcommute-R e2 ⟨≈≈˘⟩ ⨾-cong1 (DMor-cong B e1≈e2))
                 (eDcommute-S e2 ⟨≈≈˘⟩ ⨾-cong1 (DMor-cong C e1≈e2)))
    })
  ; left  = record {transform = PN.R; commute = PE.eDcommute-R}
  ; right = record {transform = PN.S; commute = PE.eDcommute-S}
  ; prf = record
    {commutes = PN.commutes
    ; universal = λ {Z} {P} {Q} F⨾P≈G⨾Q → let
      open PN
      trU = λ n → PO.univMor n (F⨾P≈G⨾Q n)
      U-left : (n : Node) → R n ⨾ trU n ≈ transform P n
      U-left n = PO.univMor-factors-left n (F⨾P≈G⨾Q n)
      U-right : (n : Node) → S n ⨾ trU n ≈ transform Q n
      U-right n = PO.univMor-factors-right n (F⨾P≈G⨾Q n)
      U = record {transform = trU
                 ; commute = λ {n1} {n2} e → let
                   P1 = transform P n1
                   Q1 = transform Q n1
                   P2 = transform P n2
                   Q2 = transform Q n2
                   open PE e
                   eZ = DMor Z e
                   V = trU n1 ⨾ eZ
                   R1⨾V≈P1⨾eZ : R1 ⨾ V ≈ P1 ⨾ eZ
                   R1⨾V≈P1⨾eZ = ⨾-assocL ⟨≈≈⟩ ⨾-cong1 (U-left n1)
                   S1⨾V≈Q1⨾eZ : S1 ⨾ V ≈ Q1 ⨾ eZ
                   S1⨾V≈Q1⨾eZ = ⨾-assocL ⟨≈≈⟩ ⨾-cong1 (U-right n1)
                   V′ = eD ⨾ trU n2
R1⨾V′≈P1⨾eZ : R1 ⨾ V′ ≈ P1 ⨾ eZ
R1⨾V′≈P1⨾eZ = ≈-begin
     R1 ⨾ eD ⨾ trU n2
  ≈⟨ ⨾-cong1&21 eDcommute-R ⟩
     eB ⨾ R2 ⨾ trU n2
  ≈⟨ ⨾-cong2 (U-left n2) ⟩
     eB ⨾ P2
  ≈˘⟨ commute P e ⟩
     P1 ⨾ eZ
  □
S1⨾V′≈Q1⨾eZ : S1 ⨾ V′ ≈ Q1 ⨾ eZ
S1⨾V′≈Q1⨾eZ = ≈-begin
     S1 ⨾ eD ⨾ trU n2
  ≈⟨ ⨾-cong1&21 eDcommute-S ⟩
     eC ⨾ S2 ⨾ trU n2
  ≈⟨ ⨾-cong2 (U-right n2) ⟩
     eC ⨾ Q2
  ≈˘⟨ commute Q e ⟩
     Q1 ⨾ eZ
  □

eZ-commutes : F n1 ⨾ P1 ⨾ eZ ≈ G n1 ⨾ Q1 ⨾ eZ
eZ-commutes = ⨾-cong1&21 (F⨾P≈G⨾Q n1)

∃!U′ : PushoutUniv compOp (F n1) (G n1) R1 S1 eZ-commutes
∃!U′ = PO.universal n1 eZ-commutes

V≈U′ = PushoutUniv.univMor-unique compOp
  ∃!U′ R1⨾V≈P1⨾eZ S1⨾V≈Q1⨾eZ
V′≈U′ = PushoutUniv.univMor-unique compOp
  ∃!U′ R1⨾V′≈P1⨾eZ S1⨾V′≈Q1⨾eZ
in V≈U′ ⟨≈≈˘⟩ V′≈U′
}
in record
  {univMor = U
  ; univMor-factors-left = U-left
  ; univMor-factors-right = U-right
  ; univMor-unique = λ {V} R⨾V≈P S⨾V≈Q n →
    PushoutUniv.univMor-unique compOp
      (PO.universal n (F⨾P≈G⨾Q n))
    {transform V n} (R⨾V≈P n) (S⨾V≈Q n)
  }
}
}
where
  open Diagram
  open DiagMor
  open HasPushouts compOp HP
    -- open Pushout compOp
  module PN (n : Node) where
    F = transform FF n
    G = transform GG n
    PO : Pushout compOp F G
    PO = pushout F G
    module PO = Pushout compOp PO
    open PO public using (commutes) renaming
      (obj to D0; left to R; right to S; universal to PO-universal)
  module PE {n1 n2 : Node} (e : Edge n1 n2) where
    open PN n1 public using () renaming
      (F to F1; G to G1; R to R1; S to S1
      ; PO-universal to PO1-universal)
    open PN n2 public using () renaming (commutes to commutes2
      ; F to F2; G to G2; R to R2; S to S2; D0 to D2)
    eB = DMor B e
    eC = DMor C e
    R′ = eB ⨾ R2
    S′ = eC ⨾ S2
    F⨾R′≈G⨾S′ : F1 ⨾ R′ ≈ G1 ⨾ S′
    F⨾R′≈G⨾S′ = ≈-begin
        F1 ⨾ DMor B e ⨾ R2
      ≈⟨ ⨾-cong1&21 (commute FF e) ⟩
        DMor A e ⨾ F2 ⨾ R2
      ≈⟨ ⨾-cong2 commutes2 ⟩
        DMor A e ⨾ (G2 ⨾ S2)
      ≈˘⟨ ⨾-cong1&21 (commute GG e) ⟩
        G1 ⨾ DMor C e ⨾ S2
      □
    eU : PushoutUniv compOp F1 G1 R1 S1 F⨾R′≈G⨾S′
    eU = PO1-universal {D2} {R′} {S′} F⨾R′≈G⨾S′
    eD = PushoutUniv.univMor compOp eU
    eDcommute-R : R1 ⨾ eD ≈ R′
    eDcommute-R = PushoutUniv.univMor-factors-left compOp eU
    eDcommute-S : S1 ⨾ eD ≈ S′
    eDcommute-S = PushoutUniv.univMor-factors-right compOp eU
    eDunique : ∀ {V} → R1 ⨾ V ≈ R′ → S1 ⨾ V ≈ S′ → V ≈ eD
    eDunique = PushoutUniv.univMor-unique compOp eU
```

DiagMorHasPushouts : HasPushouts compOp
→ HasPushouts DiagMorCompOp

DiagMorHasPushouts HP = **record** {pushout = DiagMorPushout HP}

Both a theorem and a parameterised implementation

How to Implement Categoric Graph Transformation?

Requirement 1 — Graph Category:

Represent graphs and graph homomorphisms as data, and implement pushouts and other categoric operations.

Design Decision 1:

Implement graph structure categories abstractly on top of a base category

Requirement 2 — Base Category:

Represent sets and total functions as data, and implement pushouts and other categoric operations.

How to Implement Pushouts in Base Category?

Requirement 2 — Base Category:

Represent sets and total functions as data, and implement pushouts and other categoric operations.

Pushouts from Co-Equalisers

```
constructPushout₁ : {A B C : Obj} (F : Mor A B) (G : Mor A C)
                  → {S : Obj} {ι : Mor B S} {κ : Mor C S}
                  → IsCoproduct ι κ → HasCoEqualisers → Pushout F G
```

```
constructPushout₁ F G {S} {ι} {κ} IsDSum HasCoEqu = let
    ce = HasCoEqualisers.coequaliser HasCoEqu (F ⨾ ι) (G ⨾ κ)
    open CoEqualiser ce using (obj; mor)
  in record
    {obj = obj
    ; left = ι ⨾ mor
    ; right = κ ⨾ mor
    ; prf = record
      {commutes = on-⨾-assocL (CoEqualiser.prop ce)
      ; universal = λ {Z} {P} {Q} F⨾P≈G⨾Q → let
          su = IsDSum P Q
          V = proj₁ su
          ι⨾V≈P, κ⨾V≈Q : (ι ⨾ V ≈ P) × (κ ⨾ V ≈ Q)
          ι⨾V≈P, κ⨾V≈Q = proj₁ (proj₂ su)
          ceu = CoEqualiser.universal ce {_} {V} (≈-begin
              (F ⨾ ι) ⨾ V
            ≈⟨ ≈-trans ⨾-assoc (⨾-cong₂ (proj₁ ι⨾V≈P, κ⨾V≈Q)) ⟩
              F ⨾ P   -- CoSpan.left PQ
            ≈⟨ F⨾P≈G⨾Q ⟩
              G ⨾ Q   -- CoSpan.right PQ
            ≈⟨ ≈-trans (⨾-cong₂ (≈-sym (proj₂ ι⨾V≈P, κ⨾V≈Q))) ⨾-assocL ⟩
              (G ⨾ κ) ⨾ V
            □)
          U = proj₁ ceu
          V≈m⨾U : V ≈ mor ⨾ U
          V≈m⨾U = proj₁ (proj₂ ceu)
          ι⨾m⨾U≈P = ≈-trans ⨾-assoc (≈-trans (⨾-cong₂ (≈-sym V≈m⨾U)) (proj₁ ι⨾V≈P, κ⨾V≈Q))
          κ⨾m⨾U≈Q = ≈-trans ⨾-assoc (≈-trans (⨾-cong₂ (≈-sym V≈m⨾U)) (proj₂ ι⨾V≈P, κ⨾V≈Q))
```

Co-Equalisers from Kleene Star

For two mappings F and G from A to B, given a splitting for equClos (F ˘ ⨾ G), we can construct a co-equaliser (in the mapping category) for F and G.

```
mappingCoEqualiser : {A B : Obj} (F G : Mapping A B)
                   → let V = mor F ˘ ⨾ mor G;        W = equClos V
                   in {C : Obj} {H : Mor B C}
                   → IsSymSplitting W H
                   → Category.CoEqualiser (MapCat occ) F G
mappingCoEqualiser {A} {B} F G {C} {H} HsplitsW = record
  {obj = C
  ; mor = H′
  ; prop = ≈-begin
        F₀ ⨾ H
      ≈˘⟨ ⨾-cong₂ HsplitsW.leftClosed ⟩
        F₀ ⨾ W ⨾ H
      ≈⟨ ⨾-cong₁&₂₁ F⨾W≈G⨾W ⟩
        G₀ ⨾ W ⨾ H
      ≈⟨ ⨾-cong₂ HsplitsW.leftClosed ⟩
        G₀ ⨾ H
```

□

Co-Equalisers from Kleene Star (2)

```
; universal = λ {Z} {R} F⨾R≈G⨾R → let
    H⨾H˘⨾R⊑R : H ⨾ H ˘ ⨾ mor R ⊑ mor R
    H⨾H˘⨾R⊑R = ⊑-begin
          H ⨾ H ˘ ⨾ mor R
       ≈⟨ ⨾-assocL ⟨≈≈⟩ ⨾-cong₁ HsplitsW.factors ⟩
          W ⨾ mor R
       ⊑⟨ *-leftInd (⊑-begin
                (V ⊔ V ˘) ⨾ mor R
             ⊑⟨ ⨾-⊔-subdistribL ⟨⊑≈⟩
                ⊔-cong ⨾-assoc (⨾-cong₁ ˘-involutionLeftConv ⟨≈≈⟩ ⨾-assoc) ⟩
                F₀ ˘ ⨾ G₀ ⨾ mor R ⊔ G₀ ˘ ⨾ F₀ ⨾ mor R
             ≈⟨ ⊔-cong (⨾-cong₂ (≈-sym F⨾R≈G⨾R)) (⨾-cong₂ F⨾R≈G⨾R) ⟩
                F₀ ˘ ⨾ F₀ ⨾ mor R ⊔ G₀ ˘ ⨾ G₀ ⨾ mor R
             ⊑⟨ ⊔-monotone (⨾-assocL ⟨≈⊑⟩ proj₁ (unival F))
                           (⨾-assocL ⟨≈⊑⟩ proj₁ (unival G)) ⟩
                mor R ⊔ mor R
             ≈⟨ ⊔-idempotent ⟩
                mor R
             □) ⟩
          mor R                                                    □
```

Co-Equalisers from Kleene Star (3)

```
H⨾H˘⨾R≈R : H ⨾ H ˘ ⨾ mor R ≈ mor R
H⨾H˘⨾R≈R = ⊑-antisym H⨾H˘⨾R⊑R (proj₁ (reflexiveIsSuperidentity
   (≈-isReflexive HsplitsW.factors *-isReflexive)) ⟨⊑≈⟩ ⨾-assoc)
in mkMapping (H ˘ ⨾ mor R)
   (⊑-isSubidentity (⊑-begin
         (H ˘ ⨾ mor R) ˘ ⨾ H ˘ ⨾ mor R
      ≈⟨ ⨾-cong₁ ˘-involutionLeftConv ⟨≈≈⟩ ⨾-assoc ⟩
         mor R ˘ ⨾ H ⨾ H ˘ ⨾ mor R
      ⊑⟨ ⨾-monotone₂ H⨾H˘⨾R⊑R ⟩
         mor R ˘ ⨾ mor R
      □) (unival R)
   , ⊑-isSuperidentity (⊑-begin
         H ˘ ⨾ H
      ⊑⟨ ⨾-monotone₂ (proj₁ (total R) ⟨⊑≈⟩ ⨾-assoc) ⟩
         H ˘ ⨾ mor R ⨾ mor R ˘ ⨾ H
      ≈˘⟨ ⨾-cong₂ ˘-involutionLeftConv ⟨≈≈⟩ ⨾-assoc ⟩
         (H ˘ ⨾ mor R) ⨾ (H ˘ ⨾ mor R) ˘
      □) (isIdentity-super HsplitsW.splitId)
   )
, ≈-sym H⨾H˘⨾R≈R
, (λ {U′} R≈H′⨾U′ → ≈-begin    H ˘ ⨾ mor R
                             ≈⟨ ⨾-cong₂ R≈H′⨾U′ ⟩
                                H ˘ ⨾ H ⨾ mor U′
                             ≈⟨ ⨾-assocL ⟨≈≈⟩ proj₁ (HsplitsW.splitId) ⟩
                                mor U′                                    □)
}
```

Co-Equalisers from Kleene Star (4)

```
where
  module HsplitsW = IsSymSplitting HsplitsW
  F0 = mor F
  G0 = mor G
  V = F0 ˘ ⨾ G0
  W = equClos V
  H′ : Mapping B C
  H′ = mkMapping H (isIdentity-sub HsplitsW.splitId
                            , reflexiveIsSuperidentity (≈-isReflexive HsplitsW.factors *-isReflexive))
  F⨾W≈G⨾W : F0 ⨾ W ≈ G0 ⨾ W
  F⨾W≈G⨾W = ⊑-antisym
    (⊑-begin
        F0 ⨾ W
      ⊑⟨ proj1 (total G) ⟨⊑≈⟩ ⨾-assoc ⟩
        G0 ⨾ G0 ˘ ⨾ F0 ⨾ W
      ⊑⟨ ⨾-monotone2 (⨾-assocL ⟨≈⊑⟩ ⨾-monotone1 (˘-involutionLeftConv ⟨≈˘⊑⟩ ⊔-upper2)) ⟩
        G0 ⨾ (V ⊔ V ˘) ⨾ W
      ⊑⟨ ⨾-monotone2 *-stepL ⟩
        G0 ⨾ W
      □)
    (⊑-begin
        G0 ⨾ W
      ⊑⟨ proj1 (total F) ⟨⊑≈⟩ ⨾-assoc ⟩
        F0 ⨾ F0 ˘ ⨾ G0 ⨾ W
      ⊑⟨ ⨾-monotone2 (⨾-assocL ⟨≈⊑⟩ ⨾-monotone1 ⊔-upper1) ⟩
        F0 ⨾ (V ⊔ V ˘) ⨾ W
      ⊑⟨ ⨾-monotone2 *-stepL ⟩
        F0 ⨾ W
      □)
```

How to Implement Pushouts in Base Category?

Requirement 2 — Base Category:
: Represent sets and total functions as data, and implement pushouts and other categoric operations.

Design Decision 2:
: Implement complex categoric constructions abstractly on top of relation categories

Requirement 3 — Relation Category:
: Represent sets and relations as data, and implement equivalence closure and other relational operations.

How to Derive Kleene Star for Implementations?

$E = \begin{pmatrix} a & b \\ c & d \end{pmatrix}$ as a morphism on the direct sum $\mathsf{A} \boxplus \mathsf{B}$:

$$\textbf{module}\ \mathsf{Square}\ (\mathsf{a} : \mathsf{Mor\ A\ A})\ (\mathsf{b} : \mathsf{Mor\ A\ B})\ (\mathsf{c} : \mathsf{Mor\ B\ A})\ (\mathsf{d} : \mathsf{Mor\ B\ B})$$

$$\textbf{where}$$

$$\mathsf{E} : \mathsf{Mor}\ \mathsf{A} \boxplus \mathsf{B}\ \mathsf{A} \boxplus \mathsf{B}$$

$$\mathsf{E} = (\mathsf{a} \supsetplus \mathsf{c}) \subsetplus (\mathsf{b} \supsetplus \mathsf{d})$$

$$\begin{array}{lll}
\mathsf{f} = \mathsf{a} \sqcup \mathsf{b} ⨾ \mathsf{d}^* ⨾ \mathsf{c} & \mathsf{h} = \mathsf{d}^* ⨾ \mathsf{c} ⨾ \mathsf{f}^* & \mathsf{E}^* : \mathsf{Mor}\ \mathsf{A} \boxplus \mathsf{B}\ \mathsf{A} \boxplus \mathsf{B} \\
\mathsf{g} = \mathsf{f}^* ⨾ \mathsf{b} ⨾ \mathsf{d}^* & \mathsf{k} = \mathsf{d}^* \sqcup \mathsf{d}^* ⨾ \mathsf{c} ⨾ \mathsf{g} & \mathsf{E}^* = (\mathsf{f}^* \supsetplus \mathsf{h}) \subsetplus (\mathsf{g} \supsetplus \mathsf{k})
\end{array}$$

If A is a (partial) unit object, we use $\mathsf{f}^* = \mathsf{Id}$:

$$\begin{array}{lll}
 & \mathsf{h}' = \mathsf{d}^* ⨾ \mathsf{c} & \mathsf{E}^{*\prime} : \mathsf{Mor}\ \mathsf{A} \boxplus \mathsf{B}\ \mathsf{A} \boxplus \mathsf{B} \\
\mathsf{g}' = \mathsf{b} ⨾ \mathsf{d}^* & \mathsf{k}' = \mathsf{d}^* \sqcup \mathsf{h}' ⨾ \mathsf{g}' & \mathsf{E}^{*\prime} = (\mathsf{Id} \supsetplus \mathsf{h}') \subsetplus (\mathsf{g}' \supsetplus \mathsf{k}')
\end{array}$$

$$\mathsf{UnitSumStarOp} : \mathsf{IsUnit\ A} \rightarrow \mathsf{LocalStarOp\ B} \rightarrow \mathsf{LocalStarOp}\ \mathsf{A} \boxplus \mathsf{B}$$

This enables simple recursive definition of Kleene star for base relations
— hope to derive algorithm similar to [Berghamme 2011]

Direct Sum Kleene Star Proof

```
E*-recDef₁ : Id ⊔ E ⨾ E* ⊑ E*
E*-recDef₁ = ⊑-begin
    Id ⊔ E ⨾ E*
  ≈˘⟨ ⊔-cong (isIdentity-≈Id Id⊞-isIdentity) (⨾-cong₁ ⟃-⟄) ⟩
    (ι ⟄ κ) ⊔ ((a ⟃ b) ⟄ (c ⟃ d)) ⨾ E*
  ≈⟨ ⊔-cong₂ ⟄-⨾ ⟩
    (ι ⟄ κ) ⊔ ((a ⟃ b) ⨾ E* ⟄ (c ⟃ d) ⨾ E*)
  ≈⟨ ⟄-⊔-⟄ ⟩
    (ι ⊔ (a ⟃ b) ⨾ E*) ⟄ (κ ⊔ (c ⟃ d) ⨾ E*)
  ⊑⟨ ⟄-monotone
    (⊑-begin
          ι ⊔ (a ⟃ b) ⨾ E*
      ≈⟨ ⊔-cong to-⟃ (⨾-⟃ ⟨≈≈⟩ ⟃-cong ⟃-⨾-⟄ ⟃-⨾-⟄) ⟩
          (ι ⨾ ι ˘ ⟃ ι ⨾ κ ˘) ⊔ ((a ⨾ f* ⊔ b ⨾ h) ⟃ (a ⨾ g ⊔ b ⨾ k))
      ≈⟨ ⟃-⊔-⟃ ⟩
          (ι ⨾ ι ˘ ⊔ a ⨾ f* ⊔ b ⨾ h) ⟃ (ι ⨾ κ ˘ ⊔ a ⨾ g ⊔ b ⨾ k)
      ⊑⟨ ⟃-monotone
          (⊑-begin
                ι ⨾ ι ˘ ⊔ a ⨾ f* ⊔ b ⨾ d* ⨾ c ⨾ f*
            ≈⟨ ⊔-cong             (isIdentity-≈Id leftKernel)
                                  (⊔-cong₂ ⨾-assoc₃₊₁ ⟨≈˘≈˘⟩ ⨾-⊔-distribL) ⟩
                Id ⊔ f ⨾ f*
            ⊑⟨ *-recDef₁⊑ StarA ⟩
                f*
            □)
          (⊑-begin
                ι ⨾ κ ˘ ⊔ a ⨾ g ⊔ b ⨾ (d* ⊔ d* ⨾ c ⨾ g)
            ⊑⟨ ⊔-universal (commutes ⟨≈⊑⟩ ⊥-⊑) (⊔-universal
                    (⊑-begin
                          a ⨾ f* ⨾ b ⨾ d*
                      ⊑⟨ ⨾-assocL ⟨≈⊑⟩ ⨾-monotone₁
                          (⨾-monotone₁ ⊔-upper₁ ⟨⊑⊑⟩ *-stepL StarA)⟫ ⟩
                          f* ⨾ b ⨾ d*
                      □)
                    (⊑-begin
                          b ⨾ (d* ⊔ d* ⨾ c ⨾ g)
(⊑-begin
          κ ⊔ (c ⟃ d) ⨾ E*
      ≈⟨ ⊔-cong to-⟃ (⨾-⟃ ⟨≈≈⟩ ⟃-cong ⟃-⨾-⟄ ⟃-⨾-⟄) ⟩
          (κ ⨾ ι ˘ ⟃ κ ⨾ κ ˘) ⊔ ((c ⨾ f* ⊔ d ⨾ h) ⟃ (c ⨾ g ⊔ d ⨾ k
      ≈⟨ ⟃-⊔-⟃ ⟩
          (κ ⨾ ι ˘ ⊔ c ⨾ f* ⊔ d ⨾ h) ⟃ (κ ⨾ κ ˘ ⊔ c ⨾ g ⊔ d ⨾ k)
      ⊑⟨ ⟃-monotone
          (⊑-begin
                κ ⨾ ι ˘ ⊔ c ⨾ f* ⊔ d ⨾ d* ⨾ c ⨾ f*
            ⊑⟨ ⊔-universal (commutes˘ ⟨≈⊑⟩ ⊥-⊑)
                  (⊔-universal
                      (proj₁ (*-isSuperidentity StarB))
                      (⨾-assocL ⟨≈⊑⟩
                        ⨾-monotone₁ (*-stepL StarB)))
            ⟩
                d* ⨾ c ⨾ f*
            □)
          (⊑-begin
                κ ⨾ κ ˘ ⊔ c ⨾ g ⊔ d ⨾ (d* ⊔ d* ⨾ c ⨾ g)
            ⊑⟨ ⊔-assocL ⟨≈⊑⟩ ⊔-universal
                (⊔-monotone
                      (isIdentity-≈Id rightKernel
                        ⟨≈⊑⟩ *-isReflexive StarB)
                      (proj₁ (*-isSuperidentity StarB)))
                (⨾-⊔-distribR ⟨≈⊑⟩ ⊔-monotone
                        (*-stepL StarB)
                        (⨾-assocL ⟨≈⊑⟩
                          ⨾-monotone₁ (*-stepL StarB)))
            ⟩
                d* ⊔ d* ⨾ c ⨾ g
            □)
          h ⟃ k
      □)
```

How to Relate Functions and Relations?

Requirement 3 — Relation Category:
: Represent sets and relations as data, and implement equivalence closure and other relational operations.

Design Non-Decision 1:
: Functions do not need to be implemented using the same data structures as relations.

Requirement 4 — Interoperability:
: Constructively prove equivalence of the base category with the category of mappings in the relation category.

Reflecting Co-Equalisers via a Full&Faithful Functor

```
reflectCoEqualiser : {A B : Obj} {f g : Mor₁ A B}
  → CoEqualiser SG₂ (mor f) (mor g) → CoEqualiser SG₁ f g
reflectCoEqualiser {A} {B} {f₁} {g₁} CoEq = record
  {obj = Q
  ; mor = p₁
  ; prop = ≈₁-begin
        f₁ ⨾₁ p₁
     ≈₁˘⟨ ⨾-cong₁ SG₁ mor⁻¹-mor ⟩
        mor⁻¹ (mor f₁) ⨾₁ mor⁻¹ p₂
     ≈₁˘⟨ mor⁻¹-⨾ ⟩
        mor⁻¹ (mor f₁ ⨾₂ p₂)
     ≈₁⟨ mor⁻¹-cong f₂⨾p₂≈g₂⨾p₂ ⟩
        mor⁻¹ (mor g₁ ⨾₂ p₂)
     ≈₁⟨ mor⁻¹-⨾ ⟩
        mor⁻¹ (mor g₁) ⨾₁ mor⁻¹ p₂
     ≈₁⟨ ⨾-cong₁ SG₁ mor⁻¹-mor ⟩
        g₁ ⨾₁ p₁                                   □₁
  ; universal = univ
  }
```

Reflecting Co-Equalisers via a Full&Faithful Functor (2)

```
where
  open CoEqualiser SG₂ CoEq renaming
      (obj to Q; mor to p₂; prop to f₂⨾p₂≈g₂⨾p₂)
  p₁ : Mor₁ B Q
  p₁ = mor⁻¹ p₂
  univ : {Z : Obj} {r₁ : Mor₁ B Z} (f₁⨾r₁≈g₁⨾r₁ : f₁ ⨾₁ r₁ ≈₁ g₁ ⨾₁ r₁)
      → ∃! _≈₁_ (λ u₁ → r₁ ≈₁ p₁ ⨾₁ u₁)
  univ {Z} {r₁} f₁⨾r₁≈g₁⨾r₁ with universal {Z} {mor r₁}
    (≈₂-begin
        mor f₁ ⨾₂ mor r₁
      ≈₂˘⟨ mor-⨾ ⟩
        mor (f₁ ⨾₁ r₁)
      ≈₂⟨ mor-cong f₁⨾r₁≈g₁⨾r₁ ⟩
        mor (g₁ ⨾₁ r₁)
      ≈₂⟨ mor-⨾ ⟩
        mor g₁ ⨾₂ mor r₁
      □₂)
```

Reflecting Co-Equalisers via a Full&Faithful Functor (3)

```
... | u2 , r2≈p2⨾u2 , u2-unique  =  u1 , r1≈p1⨾u1 , u1-unique
  where
    u1  : Mor1 Q Z
    u1  = mor⁻¹ u2
    r2  : Mor2 B Z
    r2  = mor r1
    r1≈p1⨾u1  : r1 ≈1 p1 ⨾1 u1
    r1≈p1⨾u1  = ≈1-begin
          r1
      ≈1˘⟨ mor⁻¹-mor ⟩
          mor⁻¹ (mor r1)
      ≈1⟨ ≈1-refl ⟩
          mor⁻¹ r2
      ≈1⟨ mor⁻¹-cong r2≈p2⨾u2 ⟩
          mor⁻¹ (p2 ⨾2 u2)
      ≈1⟨ mor⁻¹-⨾ ⟩
          mor⁻¹ p2 ⨾1 mor⁻¹ u2
      ≈1⟨ ≈1-refl ⟩
          p1 ⨾1 u1
      □1
    u1-unique : {v1 : Mor1 Q Z} (r1≈p1⨾v1 : r1 ≈1 p1 ⨾1 v1) → u1 ≈1 v1
    u1-unique {v1} r1≈p1⨾v1  = ≈1-begin
          u1
      ≈1⟨ ≈1-refl ⟩
          mor⁻¹ u2
      ≈1⟨ mor⁻¹-cong (u2-unique {mor v1}
        (≈2-begin
            mor r1
          ≈2⟨ mor-cong r1≈p1⨾v1 ⟩
            mor (p1 ⨾1 v1)
          ≈2⟨ mor-⨾ ⟩
            mor p1 ⨾2 mor v1
          ≈2⟨ ⨾-cong1 SG2 mor-mor⁻¹ ⟩
            p2 ⨾2 mor v1
```

How to Relate Subsets and Relations?

Requirement 3 — Relation Category:
: Represent sets and relations as data, and implement equivalence closure and other relational operations.

Design Non-Decision 2:
: Subsets do not need to be implemented as vectors or subidentity relations.

Requirement 5 — Support for heterogeneous Peirce-algebras:
: Provide reasoning support for "relation categories with tests".

Constructing PER-Quotients of Finite Sets

```
FinLSM-splitSymIdempot : {n : ℕ} {E : Mor n n}
                        → IsSymIdempot E → SymSplitting E
FinLSM-splitSymIdempot {n} {E} isSId-E = record
  {obj = q
  ; mor = E₁ ⨾ J ˘
  ; proof = splitting-from-univalentI {n} {n} {q} {E} {E₁} {J}
      {-isUnivalentI E₁ -}  (unival-to-⊆Id n n {E₁} (λ _ _ _ → chooseFst-unival (ρ
      {-E₁ ⨾ E₁ ˘ ≈ E -}    (chooseFst-quotProp E E-SId.symmetric E-SId.idempot
      {-isMappingI J -}     (unival-to-⊆Id q n (λ _ _ _ → enumerate-univalent)
                            , total-to-Id⊆ q n {J} (λ a₀ → _, enumerate-total))
      {-isInjectiveI J -}   (injective-to-⊆Id q n (λ _ _ _ → enumerate-injective ≡-r
      {-ran′ J ≈ ran′ E₁ -} (≈-begin
          Id ⊓ J ˘ ⨾ J
        ≈˘⟨ SubId-ran q n {J} ⟩
          SubId {ρ n} (Ran (ρ q) (ρ n) J)
        ≈⟨ SubId-cong {ρ n} (Ran-enumerate r) ⟩
          SubId {ρ n} r
        ≈⟨ SubId-ran n n {E₁} ⟩
          Id ⊓ E₁ ˘ ⨾ E₁
        □)
  } where E₁ = chooseFst (ρ n) (ρ n) E; J = enumerate (Ran _ _ E₁)
```

How to Choose the Base Category Objects? Set? Setoid?

Requirement 3 — Relation Category:

Represent **(certain)** sets and relations as data, and implement equivalence closure and other relational operations.

- Set does not have quotients in Agda
- Setoid only gives us equality
  — no sorted container structures possible
  — subsets have more complex elements
  — quotients only replace the equality
- StrictTotalOrder permits sorted container structures
- ... but still no useful subset and quotient constructions
- For most purposes, we are only interested in finite sets
- Each finite set is isomorphic to Fin n for some $n : \mathbb{N}$
- Restricting to Fin n for all $n : \mathbb{N}$ is one useful choice of base sets
- **Not fixing this choice: Base category as parameter**

First Base Category Implementation: SUList

- Sorted unique lists
- Elements “somehow” contain Key
- Key of minimal element is part of the type
- Invariant proofs are required for list construction

```
module Data.SUList.Core
  {ℓK ℓk₁ ℓk₂ : Level} (Key : StrictTotalOrder ℓK ℓk₁ ℓk₂)
  {ℓE : Level} (Elem : Set ℓE)
  (key : Elem → StrictTotalOrder.Carrier Key) where
```

```
data SUList : Maybe K → Set (ℓE ⊍ ℓK ⊍ ℓk) where
  []              :                                                   SUList nothing
  _:≈_:<_<:_ : (e : Elem) → {k : K} → k ≈K key e
     → {m : Maybe K} → (k<es : k <M m) → (es : SUList m) → SUList (just k)
```

SUList for Sets and for Relations

In SULists representing sets,

- elements **are** keys,
- there is **always** a first element

open module Core = Data.SUList.Core Key K id

$\mathsf{ListSet}_1 : \mathsf{Set}\ (\ell K \uplus \ell k)$
$\mathsf{ListSet}_1 = \Sigma\,[\,k : K\,]\ \mathsf{SUList}\ (\mathsf{just}\ k)$

In SULists representing relations from A to B,

- elements are key successor-set pairs.

open ListSet1 B **using** () **renaming** ($\mathsf{ListSet}_1$ to $\mathbb{P}B_1, \ldots$)
$\mathsf{Elem}_0 = A_0 \times \mathbb{P}B_1$
open module Map = Data.SUList.Core A Elem_0 proj_1

$\mathsf{ListSetMap} : \mathsf{Set}\ (\ell A \uplus \ell a \uplus \ell B \uplus \ell b)$
$\mathsf{ListSetMap} = \Sigma\,[\,m : \mathsf{Maybe}\ A_0\,]\ \mathsf{SUList}\ m$

Membership semantics:

$\_\in\_ : A_0 \times B_0 \to \mathsf{ListSetMap} \to \mathsf{Set}\ (\ell A \uplus \ell a \uplus \ell B \uplus \ell b)$
$(a, b) \in (\_, R) = \Sigma\,[\,a{\in}R : a\ \mathsf{Map}.{\in}\ R\,]\ b\ \mathsf{SetB}.{\in}\ \mathsf{proj}_2\ (\mathsf{Map}.{\in}\text{-}\mathsf{Elem}'\ a{\in}R)$

Re-Use of Relation-Algebraic Properties

- SUList.ListSetMap implements relations between types of aribitrary Levels
- Categoric.KleeneCollagory only talks about properties of relations between types of the same Level
- The proofs that SUList implements KleeneCollagory don't rely on Level homogeneity

$\Longrightarrow$ **Factor these proofs out!**

- Separation of concerns
- Proofs become re-usable for different implementations
- Generalisation of direct formalisation of concrete relation properties
- (High declaration overhead)

ElemSet

```
module ElemSubset {ℓa₀ ℓa₁ j ℓ : Level} (Elem : Setoid ℓa₀ ℓa₁)
                  {SetRepr : Set j} (_∈_ : ⌊ Elem ⌋ → SetRepr → Set ℓ)
  where
  infix 4 _⇒_
  _⇒_ : Rel SetRepr (ℓ ⊍ ℓa₀)
  _⇒_ R S = (a : Elem₀) → a ∈ R → a ∈ S
  record IsElemSet {k₁ k₂ : Level} (_≈_ : Rel SetRepr k₁)
                   (_⊆_ : Rel SetRepr k₂)
                 : Set (j ⊍ k₁ ⊍ k₂ ⊍ ℓ ⊍ ℓa) where
    field
      ∈-subst₁ : {R : SetRepr} {a₁ a₂ : Elem₀} → a₁ ~ a₂ → a₁ ∈ R → a₂ ∈ R
      ≈-to-⇒    : {R S : SetRepr} → R ≈ S → R ⇒ S
      ≈-to-⇔    : {R S : SetRepr} → R ≈ S → (R ⇒ S) × (S ⇒ R)
      ⊆-to-⇒    : {R S : SetRepr} → R ⊆ S → R ⇒ S
      ⊆-from-⇒ : {R S : SetRepr} → R ⇒ S → R ⊆ S
      ≈-from-⇔ : {R S : SetRepr} → (R ⇒ S) × (S ⇒ R) → R ≈ S
    isUniversal : SetRepr → Set (ℓ ⊍ ℓa₀)
    isUniversal S = (a : Elem₀) → a ∈ S
```

ElemRel-Dedekind (Declaration Overhead 1)

```
module ElemRel-Dedekind
  {ℓa₀ ℓa₁ : Level}         {A : Setoid ℓa₀ ℓa₁}
  {ℓb₀ ℓb₁ : Level}         {B : Setoid ℓb₀ ℓb₁}
  {ℓc₀ ℓc₁ : Level}         {C : Setoid ℓc₀ ℓc₁}
  {ℓq₀ ℓq₁ ℓq₂ ℓq∈ : Level} (AB : ElemRel A B ℓq₀ ℓq₁ ℓq₂ ℓq∈)
  {ℓr₀ ℓr₁ ℓr₂ ℓr∈ : Level} (BC : ElemRel B C ℓr₀ ℓr₁ ℓr₂ ℓr∈)
  {ℓs₀ ℓs₁ ℓs₂ ℓs∈ : Level} (AC : ElemRel A C ℓs₀ ℓs₁ ℓs₂ ℓs∈)
  {ℓt₀ ℓt₁ ℓt₂ ℓt∈ : Level} (BA : ElemRel B A ℓt₀ ℓt₁ ℓt₂ ℓt∈)
  {ℓu₀ ℓu₁ ℓu₂ ℓu∈ : Level} (CB : ElemRel C B ℓu₀ ℓu₁ ℓu₂ ℓu∈)
  {convAB : RelRepr₀ AB → RelRepr₀ BA}
  {convBC : RelRepr₀ BC → RelRepr₀ CB}
  (EConv-AB : ElemRelConv A B (_∈_ AB) (_∈_ BA) convAB)
  (EConv-BC : ElemRelConv B C (_∈_ BC) (_∈_ CB) convBC)
  {compABC : RelRepr₀ AB → RelRepr₀ BC → RelRepr₀ AC}
  {compACB : RelRepr₀ AC → RelRepr₀ CB → RelRepr₀ AB}
  {compBAC : RelRepr₀ BA → RelRepr₀ AC → RelRepr₀ BC}
  (EComp-ABC : ElemRelComp A B C (_∈_ AB) (_∈_ BC) (_∈_ AC) compAB
  (EComp-ACB : ElemRelComp A C B (_∈_ AC) (_∈_ CB) (_∈_ AB) compAC
  (EComp-BAC : ElemRelComp B A C (_∈_ BA) (_∈_ AC) (_∈_ BC) compBA
  {meetAB : RelRepr₀ AB → RelRepr₀ AB → RelRepr₀ AB}
  {meetBC : RelRepr₀ BC → RelRepr₀ BC → RelRepr₀ BC}
```

ElemRel-Dedekind (Declaration Overhead 2)

```
where
  open SetoidA A
  open SetoidB B
  open SetoidB C
  private
    module AB where
      open ElemSetMeet (A ×× B) (_∈_ AB) EM-AB public
      open ElemRel AB public
      open ElemRelConv EConv-AB public
    module BC where
      open ElemSetMeet (B ×× C) (_∈_ BC) EM-BC public
      open ElemRel BC public
      open ElemRelConv EConv-BC public
    module AC where
      open ElemSetMeet (A ×× C) (_∈_ AC) EM-AC public
      open ElemRel AC public
    module ABC = ElemRelComp EComp-ABC
    module ACB = ElemRelComp EComp-ACB
    module BAC = ElemRelComp EComp-BAC
```

ElemRel-Dedekind (Declaration Overhead 3, and Proof)

```
Dedekind-⇒ : {Q : AB.RelRepr₀} {R : BC.RelRepr₀} {S : AC.RelRepr₀}
             → meetAC (compABC Q R) S
               AC.⇒
               compABC (meetAB Q (compACB S (convBC R)))
                       (meetBC R (compBAC (convAB Q) S))
Dedekind-⇒ {S} {Q} {R} (a, c) aQR∩Sc
   with AC.from-∈-intersection _ _ (a, c) aQR∩Sc
... | aQRc, aSc with ABC.from-∈-comp _ _ a c aQRc
... | b, aQb, bRc = ABC.to-∈-comp _ _ a b c
      (AB.to-∈-intersection _ _ (a, b) aQb
         (ACB.to-∈-comp _ _ a c b aSc (BC.to-∈-conv _ b c bRc)))
      (BC.to-∈-intersection _ _ (b, c) bRc
         (BAC.to-∈-comp _ _ b a c (AB.to-∈-conv _ a b aQb) aSc))

Dedekind :     {Q : AB.RelRepr₀} {R : BC.RelRepr₀} {S : AC.RelRepr₀}
          →    meetAC (compABC Q R) S
               AC.⊆
               compABC (meetAB Q (compACB S (convBC R)))
                       (meetBC R (compBAC (convAB Q) S))
Dedekind = AC.⊆-from-⇒ Dedekind-⇒
```

Current State

- Everything for graph pushouts is there

```
46537 kahl       20   0  262G  259G  8056 R 54.0 25.7 43h43:18 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46538 kahl       20   0  262G  259G  8056 R 61.0 25.7 43h39:05 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46539 kahl       20   0  262G  259G  8056 R 70.0 25.7 43h40:50 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46540 kahl       20   0  262G  259G  8056 R 74.0 25.7 43h40:38 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46541 kahl       20   0  262G  259G  8056 R 74.0 25.7 43h38:13 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46542 kahl       20   0  262G  259G  8056 R 70.0 25.7 43h41:29 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46543 kahl       20   0  262G  259G  8056 S  0.0 25.7  0:00.00 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46544 kahl       20   0  262G  259G  8056 S  0.0 25.7  9:43.12 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46545 kahl       20   0  262G  259G  8056 R 67.0 25.7 43h29:20 agda +RTS -N8 -S -K256M -H256G -M256G -RTS
46536 kahl       20   0  262G  259G  8056 R 544. 25.7     365h agda +RTS -N8 -S -K256M -H256G -M256G -RTS
```

On facilities of the Shared Hierarchical Academic Research Computing Network (SHARCNET: www.sharcnet.ca) and Compute/Calcul Canada.

- Agda runs out of ressources for connecting the mapping category of the concrete relation KleeneCollagory with the concrete function Category

- Pushout complements require (pseudo-/semi-)complements

- I/O still missing

Conclusion

- **"Categorical interfaces"**
  - semigroupoids, categories, allegories
  - Kleene categories, action lattice categories
  - Dedekind/Schröder categories

  are **useful for modular programming** of high-level transformation systems

- **"Messier interfaces**
  - KAT, Peirce categories, ...

  help **abstract from implementation details**

- **Fully verified graph transformation is within reach**
- **New graph transformation concepts will follow**
- URL: http://RelMiCS.McMaster.ca/~kahl/RATH/Agda/ **(soon)**

Terminology Questions

- A good name that encompasses allegories and Kleene categories?
  - "relation categories"? — No
  - "locally-ordered categories"? — No
  - "relation-algebraic categories"?
  - **There must be something better...**
- A good name for "tests" (as in KAT) in the more general case?
  - Abstracting in particular from "sets" of Peirce algebra
  - Possibly abstracting also from (partial/fuzzy) equivalences
  - **?**