Coupled Transformation of Schemas, Data, and Queries

Joost Visser

Joint work with

Pablo Berdaguer   Alcino Cunha   José Nuno Oliveira   Hugo Pacheco

Universidade do Minho, Portugal

CIC 2006

What?

A ***two-level data transformation*** consists of:

a type-level transformation of a ***data format***

coupled with

value-level transformations of ***data instances***

and

program transformations of ***data operations***

What?

A ***two-level data transformation*** consists of:

a type-level transformation of a ***data format***

coupled with

value-level transformations of ***data instances***

and

program transformations of ***data operations***

Examples:
XML schema evolution + document, query migration
SQL schema evolution + data, query migration
Data mappings (e.g. hierarchical-relational)

Challenge 1/2

Transform format **A** into format **B**
T : Type → Type

$A \xrightarrow{T} B$

Challenge 1/2

$$\begin{tikzcd} A \arrow[r, "T"] \arrow[r, bend left=60, "\text{to}"] & B \arrow[l, bend left=60, "\text{from}"] \end{tikzcd}$$

Transform format **A** into format **B**

  T : Type → Type

Format transformation T

  induces / **is witnessed by**

instance conversions:

  to :: A → B

  from :: B → A

Challenge 1/2

$$\begin{tikzcd} A \arrow[rr, "T"] \arrow[rr, bend left=60, "\text{to}"] & & B \arrow[ll, bend left=60, "\text{from}"] \end{tikzcd}$$

Transform format **A** into format **B**
  T : Type → Type

Format transformation T
  induces / **is witnessed by**
instance conversions:
  to :: A → B
  from :: B → A

Challenge: capture **type-changing** transformations in a **type-safe** rewrite system (types and rewrite steps are **unknown statically!**).

Challenge 2/2

$$\begin{tikzcd} A \arrow[r, "T"] \arrow[r, bend left=60, "to"] & B \arrow[l, bend left=60, "from"] \end{tikzcd}$$

Challenge 2/2

$$\begin{tikzcd}
X \arrow[d, "p"'] & \\
A \arrow[r, "T"] \arrow[r, bend left=60, "to"] \arrow[d, "q"'] & B \arrow[l, bend left=60, "from"] \\
Y &
\end{tikzcd}$$

query **q** : $A \rightarrow Y$
producer **p** : $X \rightarrow A$

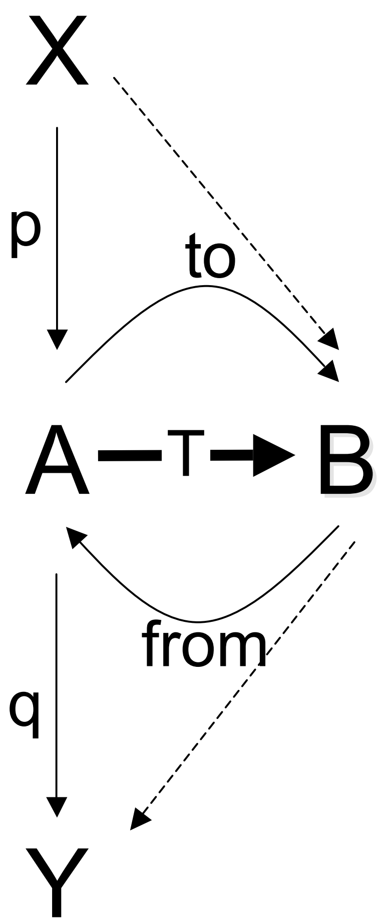
Challenge 2/2

$$\begin{tikzcd}
X \arrow[d, "p"'] \arrow[dr, dashed] & \\
A \arrow[r, "T"] \arrow[r, bend left=60, "to"] \arrow[d, "q"'] & B \arrow[l, bend left=60, "from"] \arrow[dl, dashed] \\
Y &
\end{tikzcd}$$

query **q** : A → Y
producer **p** : X → A

<u>Challenge</u>:
From the composition ***q.from*** or ***to.p*** compute optimized queries and producers not involving type **A** and original **p** or **q**.

Challenge 2/2



query **q** : A → Y
producer **p** : X → A

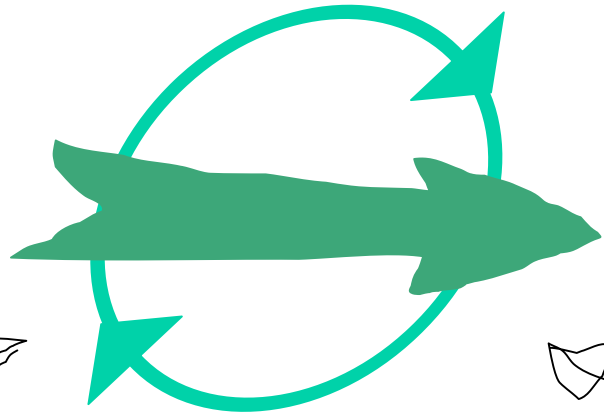
Challenge:
From the composition ***q.from*** or ***to.p*** compute optimized queries and producers not involving type **A** and original **p** or **q**.

Apply **program calculus** laws for fusion, deforestation, specialization, generalization.

Ingredients

Data refinement

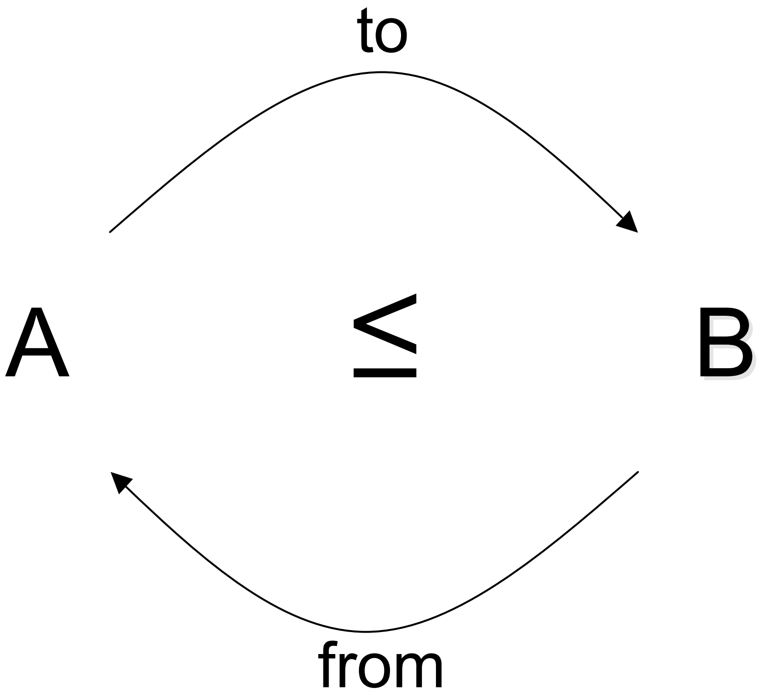
Generalized algebraic datatypes



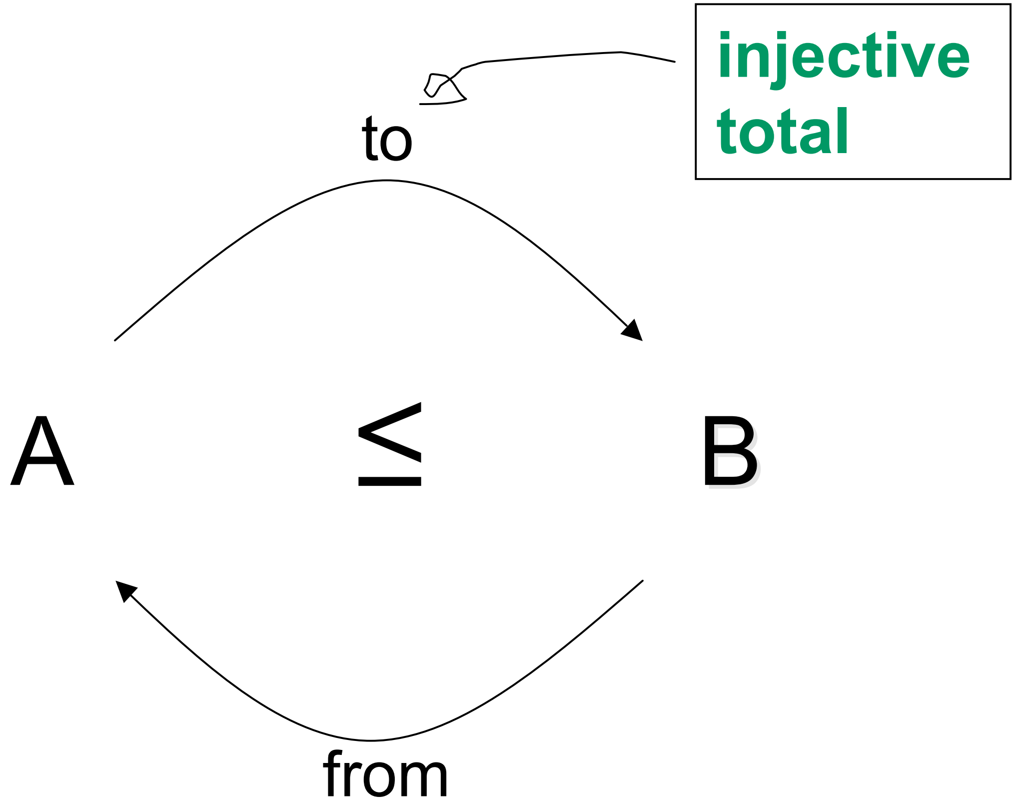
Point-free program transformation

Strategic term rewriting

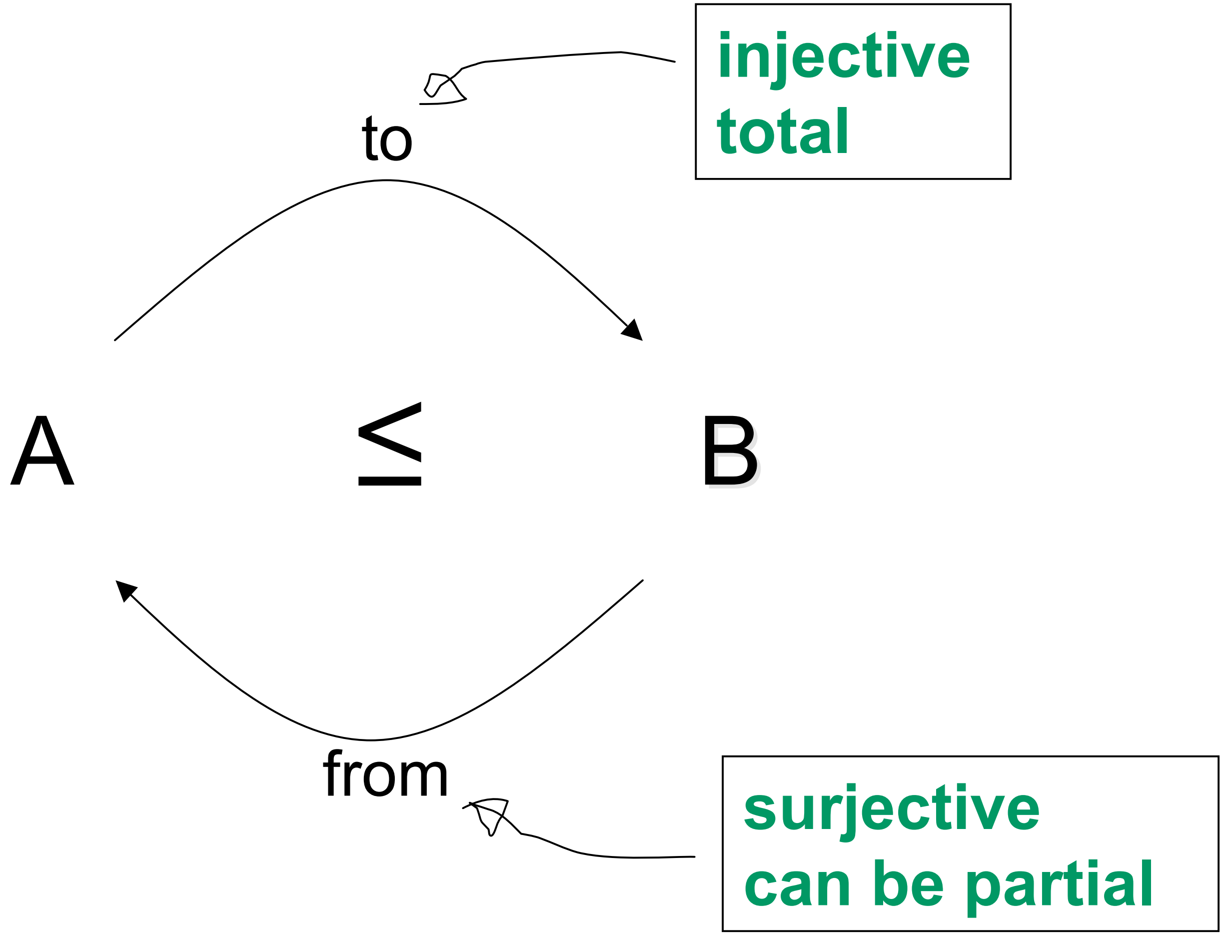
Data refinement



Data refinement



Data refinement



Data refinement

injective
total

$$\begin{tikzcd} A \arrow[rr, bend left, "\text{to}"] & \leq & B \arrow[ll, bend left, "\text{from}"] \end{tikzcd}$$

surjective
can be partial

$$\boxed{\text{from} \;.\; \text{to} = \text{id}_A}$$

Data refinement

$$
\begin{tikzcd}
A \arrow[rr, bend left, "\lambda x.(x,b)"] \arrow[rr, phantom, "\leq"] & & A \times B \arrow[ll, bend left, "\pi_1"]
\end{tikzcd}
$$

addField(B,b)

Format evolution (user-driven)

Data refinement

$$\begin{tikzcd}[column sep=huge] A\to(B\times(C\to D)) \arrow[r, phantom, "\leq"] \arrow[r, bend left, "\text{un-nested-join}"] & (A\to B)\times(A\times C\to D) \arrow[l, bend left, "\text{nested-join}"] \end{tikzcd}$$

Hierarchical-relational data mapping (automatic)

Data refinement

if $\begin{tikzcd} A \arrow[rr, bend left, "\text{to}"] & \leq & B \arrow[ll, bend left, "\text{from}"] \end{tikzcd}$ **and** $\begin{tikzcd} B \arrow[rr, bend left, "\text{to'}"] & \leq & C \arrow[ll, bend left, "\text{from'}"] \end{tikzcd}$ **then** $\begin{tikzcd} A \arrow[rr, bend left, "\text{to'.to}"] & \leq & C \arrow[ll, bend left, "\text{from.from'}"] \end{tikzcd}$

if $\begin{tikzcd} A \arrow[rr, bend left, "\text{to}"] & \leq & B \arrow[ll, bend left, "\text{from}"] \end{tikzcd}$ **then** $\begin{tikzcd} FA \arrow[rr, bend left, "\text{F to}"] & \leq & FB \arrow[ll, bend left, "\text{F from}"] \end{tikzcd}$

GADTs

Traditional algebraic data type (ADT):

```
data F = Id | Comp F F | …
```

In syntax of **generalized** ADT:

```
data F where
  Id :: F
  Comp :: F → F → F
```

Exploiting generalization:

```
data F f where
  Id :: F (a→a)
  Comp :: F (b→c) → F (a→b) → F (a→c)
```

GADTs

Proof-carrying code:

```
data Equal a b where
    Eq :: Equal a a
```

Type-safe value-level **type representations**:

```
data Type a where
     Int      :: Type Int
     List     :: Type a -> Type [a]
     .><.     :: Type a -> Type b -> Type (a,b)
     .--\.    :: Type a -> Type b -> Type (Map a b)
```

Type-safe **dynamics**:

```
data Dynamic where
     Dyn     :: Type a -> a -> Dynamic
```

Data refinement in Haskell

Masquerade **changes** as **views**:

```
data Rep a b = Rep { to :: a→b, from :: b→a }

data View a where
   View :: Rep a b → Type b → View (Type a)

type RULE = ∀a . Type a → Maybe (View (Type a))
```

$$\begin{tikzcd} A \arrow[r, phantom, "\leq"] \arrow[r, bend left, "to"] & B \arrow[l, bend left, "from"] \end{tikzcd}$$

Data refinement in Haskell

Strategic combinators:

nop :: RULE
($\succ$) :: RULE → RULE → RULE
($\oplus$) :: RULE → RULE → RULE
everywhere :: RULE → RULE *etc.*

Basic **type-changing** rewrite steps:

addField :: Type b → b → RULE
addField b y a = return (View (Rep (λx.(x,y)) fst) (a,b))

etc.

Data refinement in Haskell

Compose basic rules and combinators to obtain a full rewrite system for two-level data transformation.

Hierarchical-relational mapping:

```
toDB :: RULE
toDB = …
```

Evolution:

```
addTracks :: RULE
addTracks = …
```

Data refinement in Haskell

Compose basic rules and combinators to obtain a full rewrite system for two-level data transformation.

Hierarchical-relational mapping:

toDB :: RULE
toDB = …

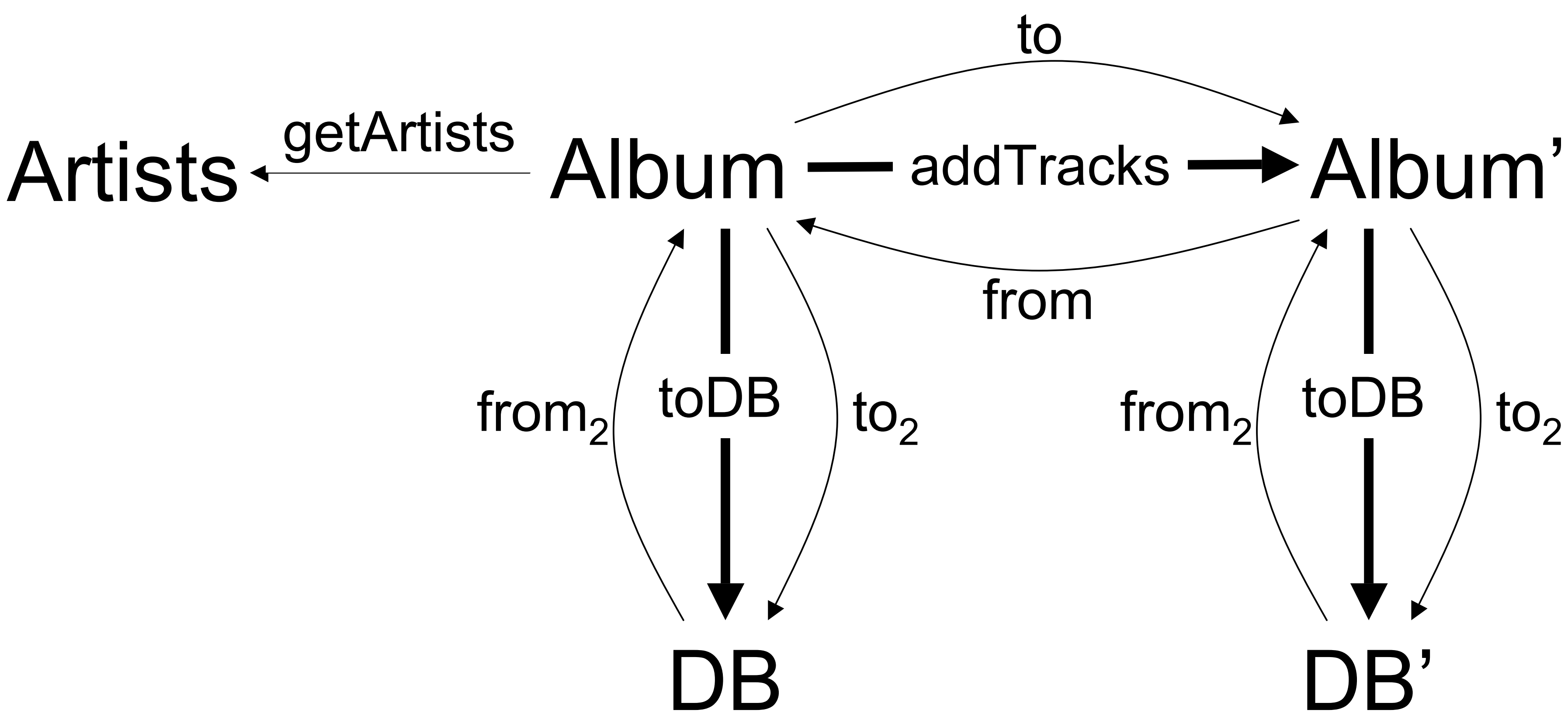
Evolution:

addTracks :: RULE
addTracks = …

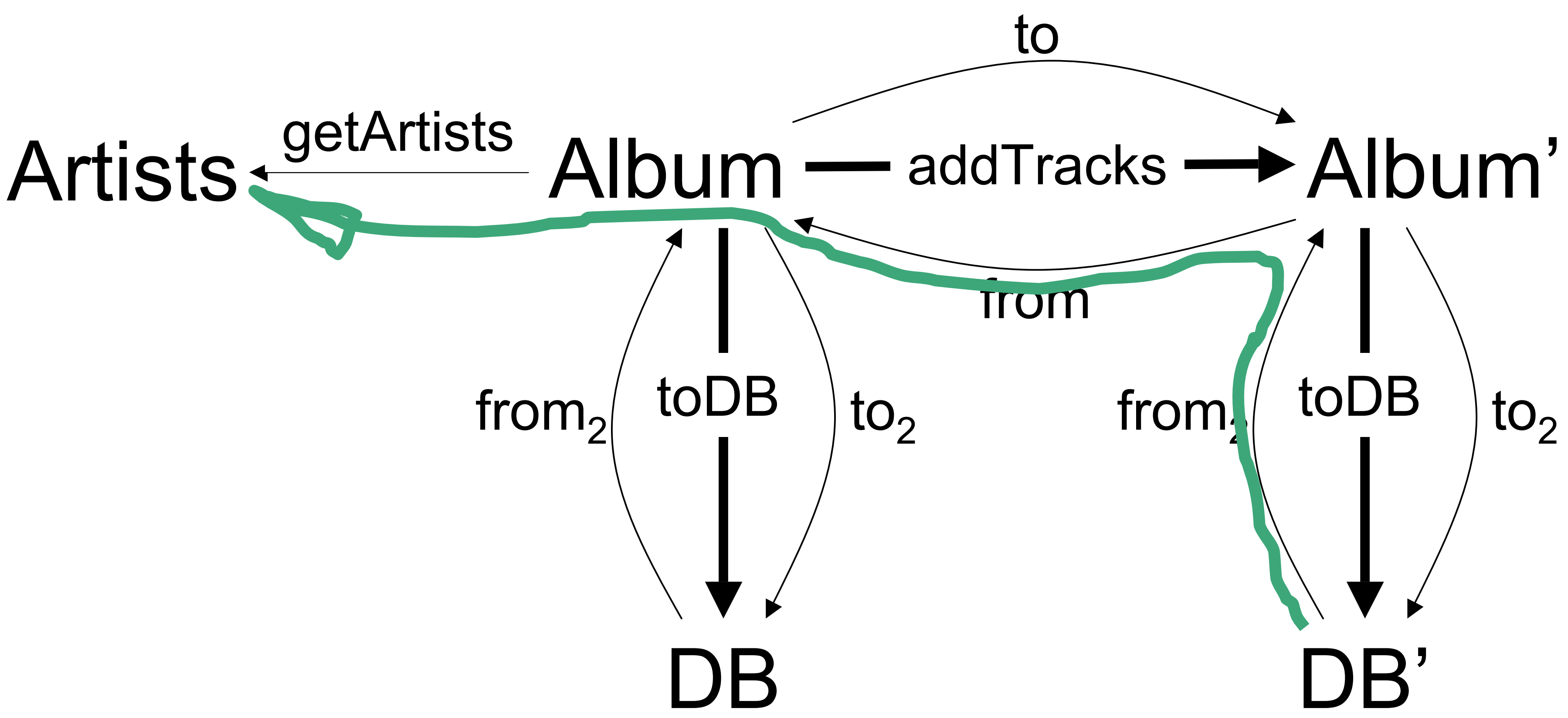
Helpers for **staged** application:

showType :: View (Type a) → String
unView :: View (Type a) → Type b → Maybe (a→b, b→a)

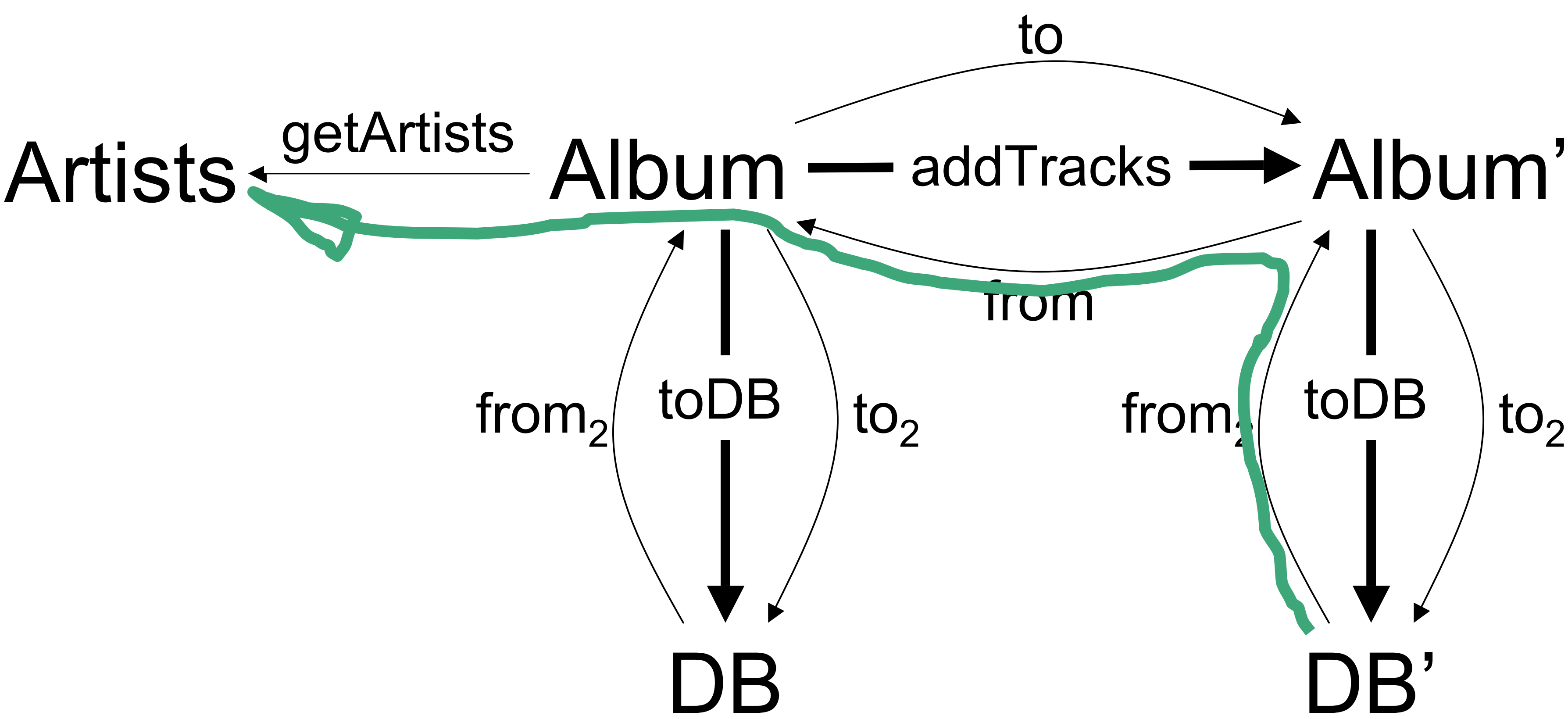
Two-Level Transformation



Two-Level Transformation



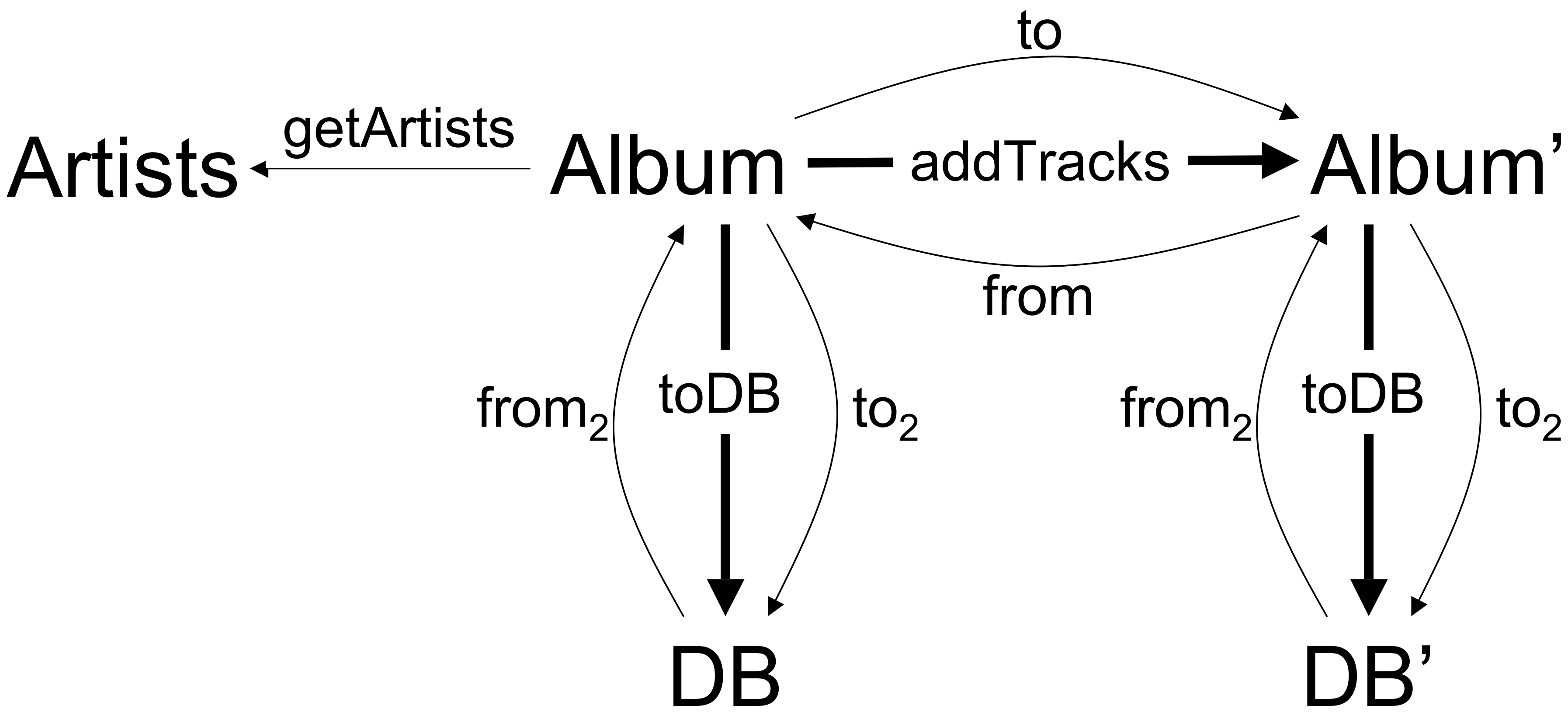
Two-Level Transformation



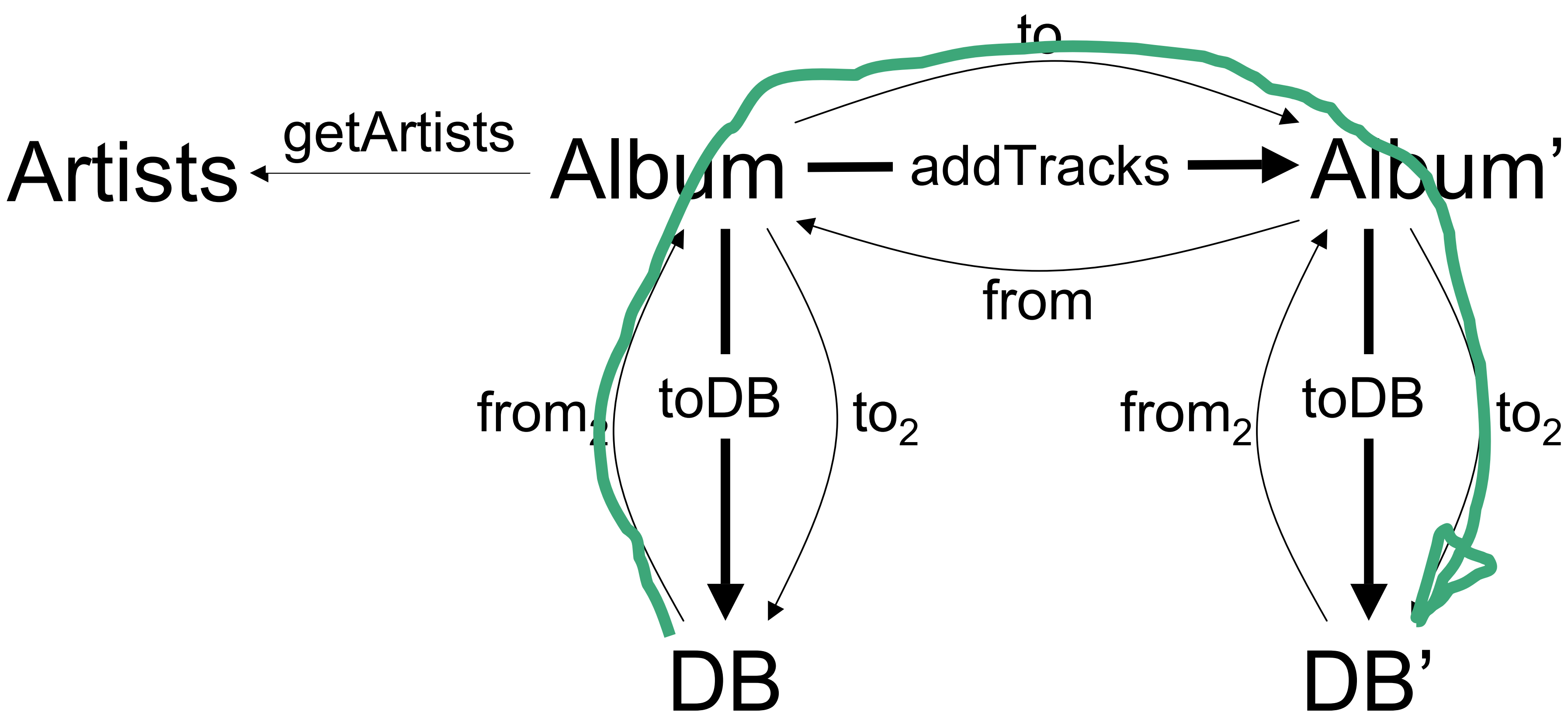
Query migration

getArtists . from . from$_2$

Two-Level Transformation



Two-Level Transformation



Compute concrete data migration from abstract migration

to_2 . to . $from_1$

Challenge 2/2

$$
\begin{tikzcd}
X \arrow[d, "p"'] \arrow[dr, dashed] & \\
A \arrow[r, "T"] \arrow[r, bend left, "to"] \arrow[d, "q"'] & B \arrow[l, bend left, "from"] \arrow[dl, dashed] \\
Y &
\end{tikzcd}
$$

query **q** : A → Y
producer **p** : X → A

<u>Challenge</u>:
From the composition ***q.from*** or ***to.p*** compute optimized queries and producers not involving type **A** and original **p** or **q**.

Apply **program calculus** laws for fusion, deforestation, specialization, generalization.

Program transformation

Not **functions**:

data Rep a b = Rep { to :: **a→b**, from :: **b→a** }

Program transformation

Not **functions**, but function **representations**:

```
data Rep a b = Rep { to :: F (a→b), from :: F (b→a) }
```

```
data F f where
  Id :: F (a→a)
  Comp :: Type b → F (b→c) → F (a→b) → F (a → c)
  Split :: F (a→b) → F (a→c) → F (a→(b,c))
  Fst :: F ((a,b)→a)
  …

eval :: F f -> f
eval Id = id
eval (Comp b f g) = f . g
eval …
```

$$\begin{tikzcd} A \arrow[rr, bend left, "\mathbf{to}"] \arrow[r, phantom, "\leq"] & {} & B \arrow[ll, bend left, "\mathbf{from}"] \end{tikzcd}$$

Program transformation

Type-directed, type-safe rewriting of point-free functions:

```
type Rule = ∀a . Type a → F a → M (F a)
```

Strategic combinators:

```
nop :: Rule
(≻)  :: Rule → Rule → Rule
(⊕)  :: Rule → Rule → Rule
everywhere :: Rule → Rule                    etc.
```

Basic rewrite steps, e.g. associativity of composition:

$$f \,.\, (g \,.\, h) = (f \,.\, g) \,.\, h$$

```
comp_assocr :: Rule
comp_assocr _ (Comp a (Comp b f g) h)
              = return (Comp b f (Comp a g h))
comp_assocr _ _ = mzero
```

Program transformation

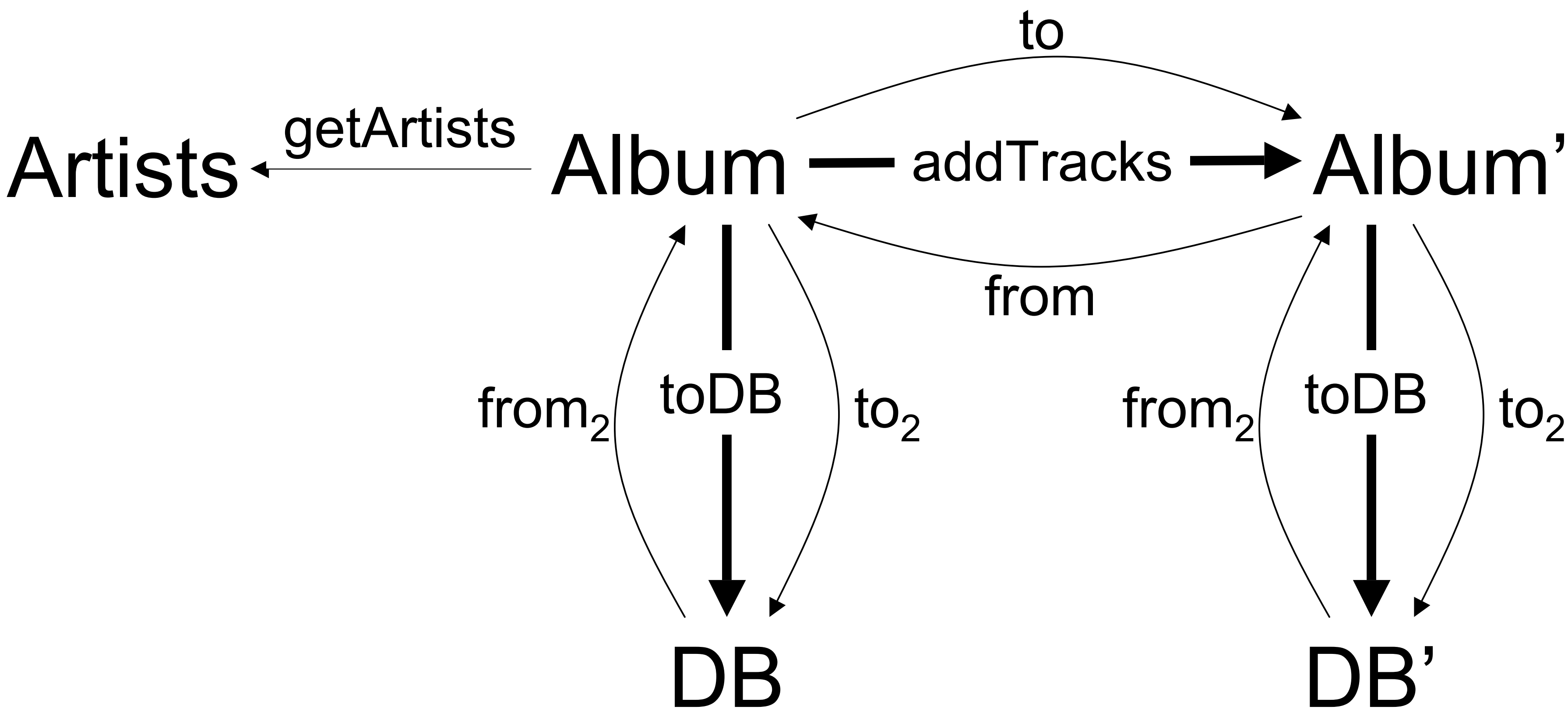
Compose basic rules and combinators to obtain a full rewrite system for simplification / optimization of point-free functions.

```
optimize :: Rule
optimize = many (prods ⊕ maps ⊕ sums)
  where
    prods :: Rule
    prods = … (everywhere comp_assocr) …
    …
```

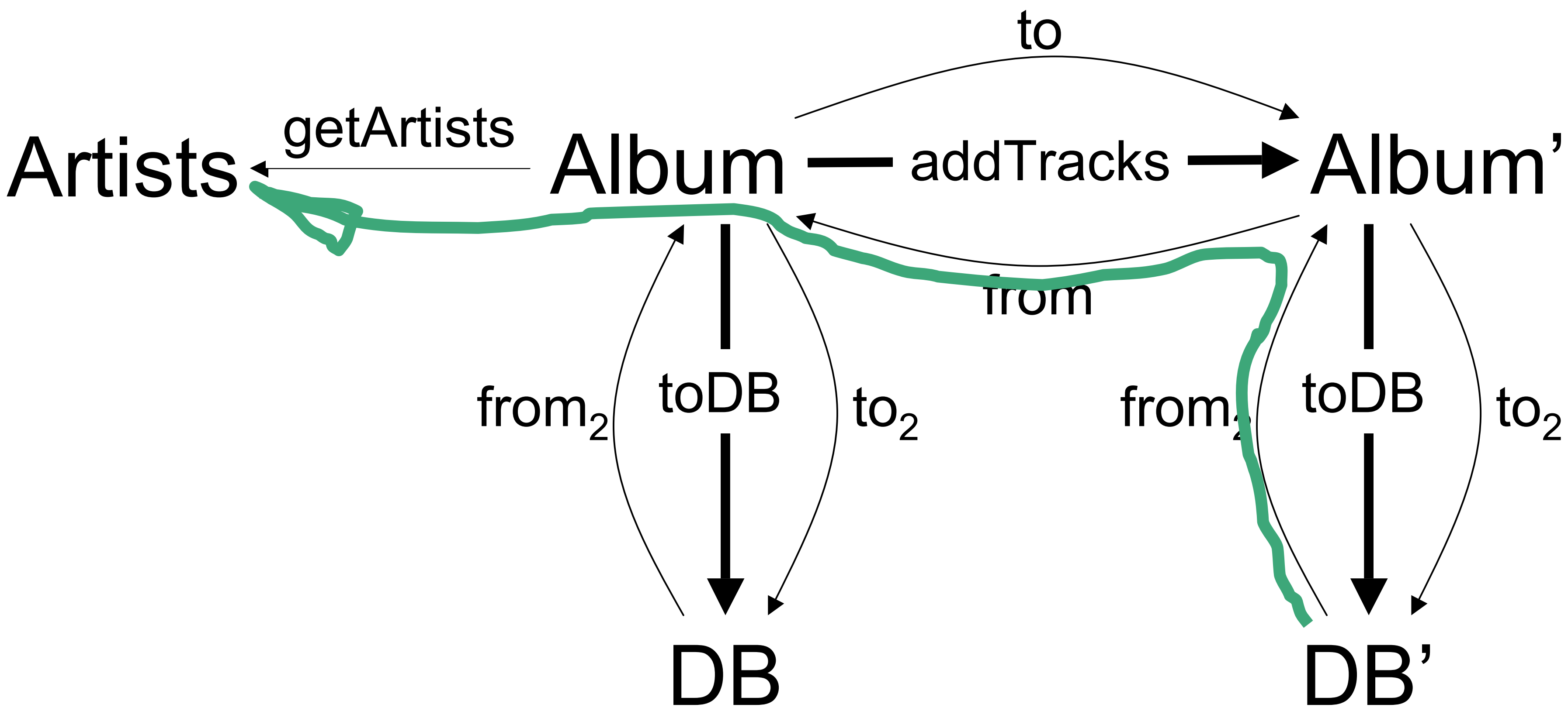
For example:

```
> rewrite optimize (Comp Albums getArtists from) ListMap (Comp .. Fst (Comp .. Snd Snd))
```

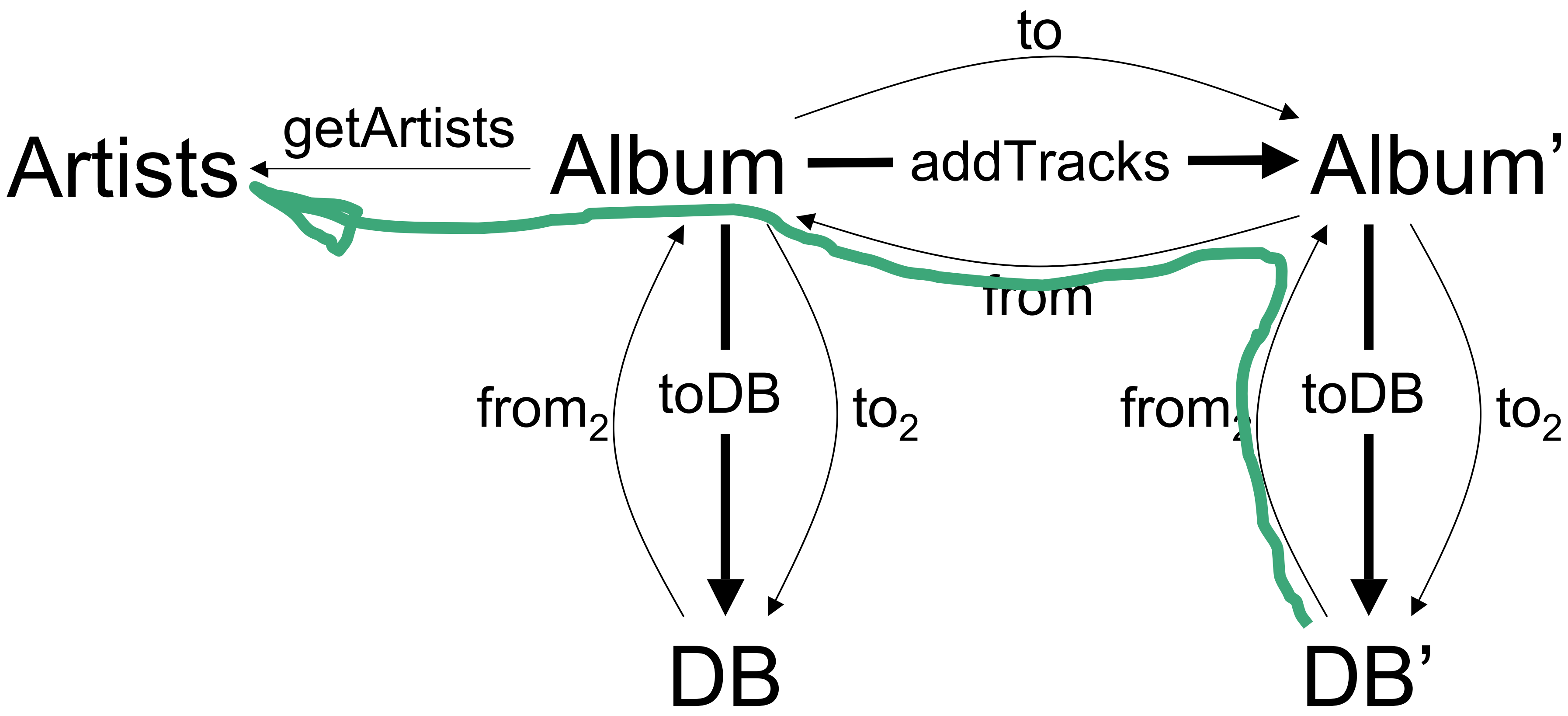
Coupled Transformation



Coupled Transformation



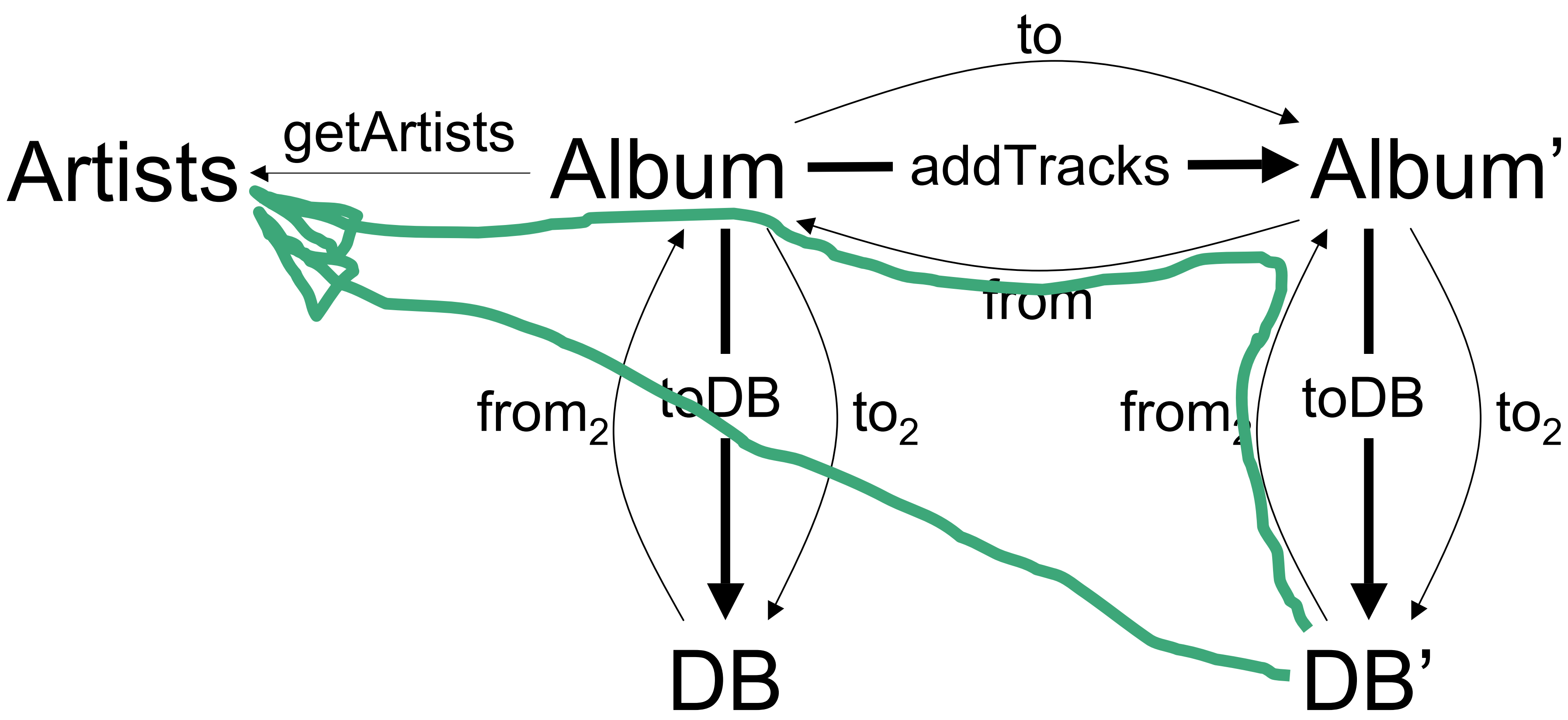
Coupled Transformation



Query migration

getArtists . from . from$_2$

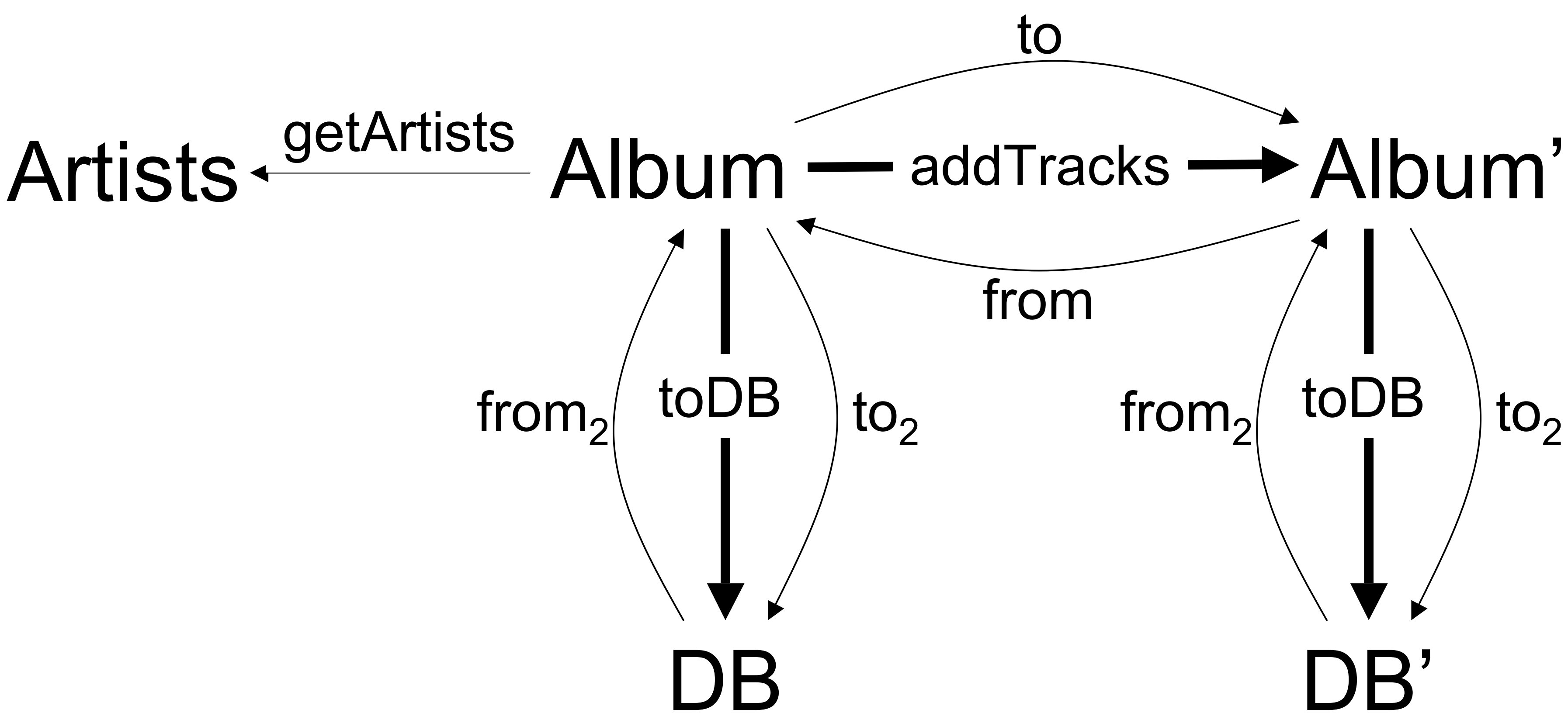
Coupled Transformation



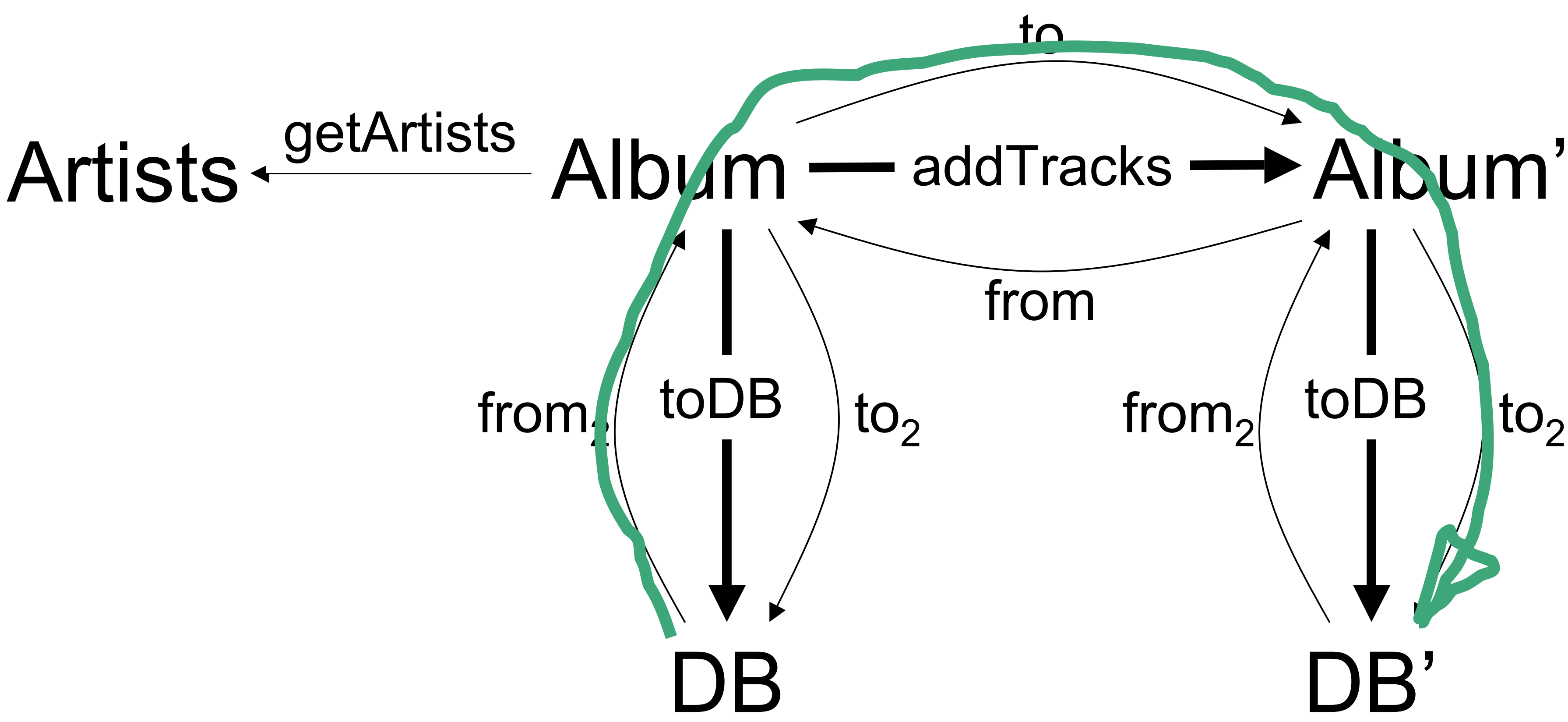
Query migration

getArtists . from . from$_2$

Coupled Transformation



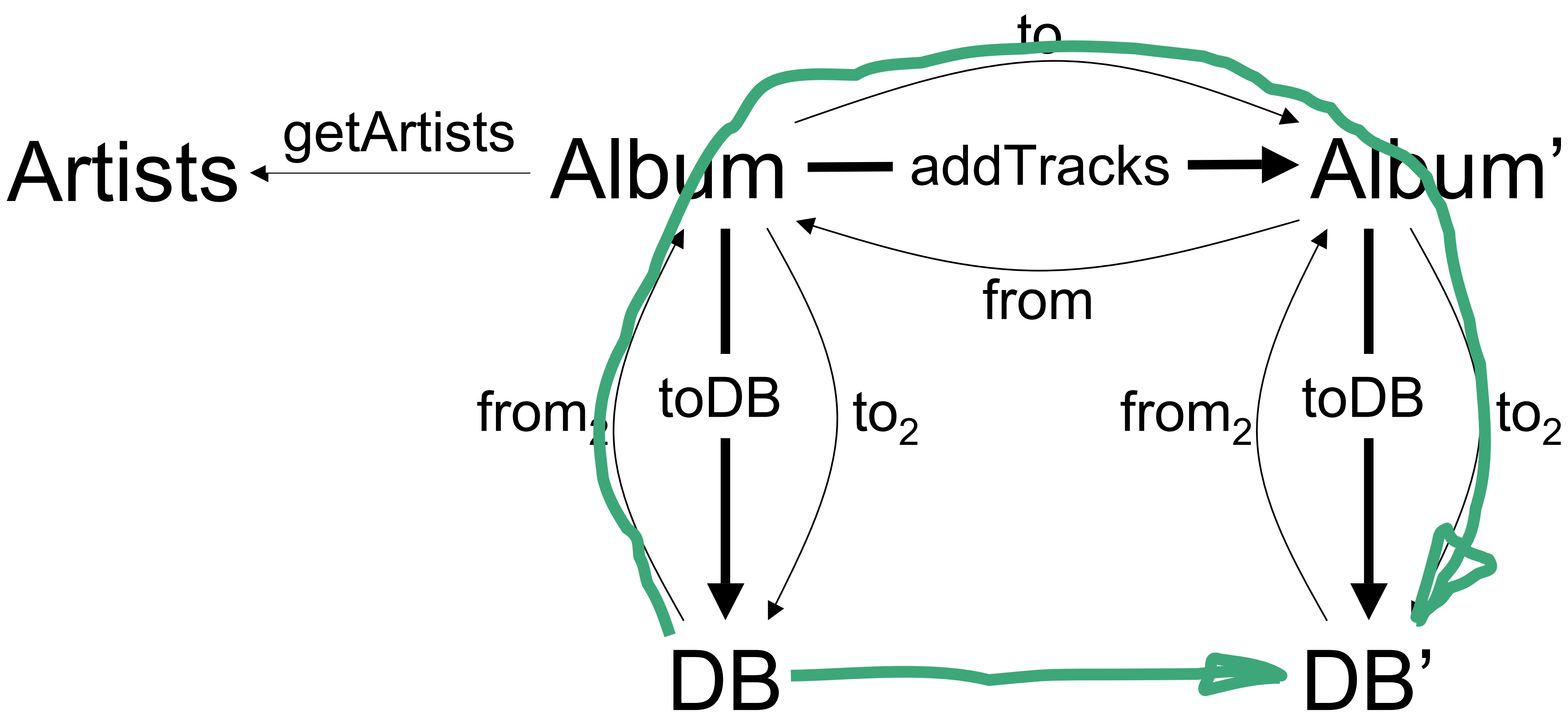
Coupled Transformation



Compute concrete data migration from abstract migration

to_2 . to . $from_1$

Coupled Transformation



Compute concrete data migration from abstract migration

to_2 . to . $from_1$

More (1/2)

Front-ends (schemas+data) for:

XML Schema, SQL, Haskell itself (done)

VDM (underway)

Type-directed optimization of **structure-shy programs**, such as XML queries and transformations, or functional strategic programs (SYB,Strafunski).

Transformation of types with **invariants**. Carrying constructive proofs through rewrite steps.

Front-ends (programs) for:

XPath, SYB, SQL

More (2/2)

Generalize to **lenses**, a.k.a. bi-directional programming, applicable to the classical view-update problem, data synchronization.

Model transformation -- think UML, etc.
Object-relational data mappings.
Refinements with effects (time, mutable state).

Schema/grammar **matching**.
Data synchronization. Interoperability.

Reverse direction: abstraction rather than refinement.

Papers

Type-safe Two-level Data Transformation. FM 2006.
Alcino Cunha, José Nuno Oliveira, Joost Visser.

Strongly Typed Rewriting For Coupled Software Transformation. RULE 2006.
Alcino Cunha, Joost Visser.

Coupled Schema Transformation and Data Conversion For XML and SQL. PADL 2007
Pablo Berdaguer, Alcino Cunha, Hugo Pacheco, Joost Visser.

http://wiki.di.uminho.pt/wiki/bin/view/PURe/2LT