Bi-directional Transformations @ UMinho

Alcino Cunha,José Nuno Oliveira,Joost Visser
Pablo Berdaguer,Flávio Ferreira,Miguel Marques,Hugo Pacheco

Universidade do Minho, Portugal

CIC'07, October 23th

Bi-directional Transformations

Definition

By specifying a transformation on the type-level between A and B we get migration functions between values of A and B and vice-versa.

Bi-directional Transformations

Definition

By specifying a transformation on the type-level between A and B we get migration functions between values of A and B and vice-versa.

Scenarios

User-driven XML schema evolution coupled with document migration. Views of databases where updates to the view induce updates to the database.

Automated Data mappings for storing XML in relational databases.

Bi-directional Transformations

Definition

By specifying a transformation on the type-level between A and B we get migration functions between values of A and B and vice-versa.

Scenarios

User-driven XML schema evolution coupled with document migration. Views of databases where updates to the view induce updates to the database.

Automated Data mappings for storing XML in relational databases.

Examples

Refinements Going from abstract to concrete.

Lenses Going from concrete to abstract.

Refinements

An abstract type A is mapped to a concrete type B

Representation Injective and total.

Abstraction Surjective and possibly partial.

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

Examples of Refinements

Format evolution

$$\begin{tikzcd} A \arrow[r, bend left, "inject"] \arrow[r, phantom, "\leq"] & A + B \arrow[l, bend left, "isleft"] \end{tikzcd} \qquad \begin{tikzcd} A \arrow[r, bend left, "pairwith(b)"] \arrow[r, phantom, "\leq"] & A \times B \arrow[l, bend left, "project"] \end{tikzcd}$$

Examples of Refinements

Format evolution

$$A \underset{isleft}{\overset{inject}{\rightleftarrows}} A + B \qquad (\leq)$$

$$A \underset{project}{\overset{pairwith(b)}{\rightleftarrows}} A \times B \qquad (\leq)$$

Hierarchical to relational mappings

$$A \rightharpoonup (B \times (C \rightharpoonup D)) \underset{njoin}{\overset{unnjoin}{\rightleftarrows}} (A \rightharpoonup B) \times (A \times C \rightharpoonup D) \qquad (\leq)$$

Composition of Refinements

Sequential composition

$$
\begin{tikzcd}
A \arrow[rr, bend left, "to"] \arrow[rr, phantom, "\leq"] & & B \arrow[ll, bend left, "from"]
\end{tikzcd}
\;\wedge\;
\begin{tikzcd}
B \arrow[rr, bend left, "to'"] \arrow[rr, phantom, "\leq"] & & C \arrow[ll, bend left, "from'"]
\end{tikzcd}
\;\Rightarrow\;
\begin{tikzcd}
A \arrow[rr, bend left, "to' \cdot to"] \arrow[rr, phantom, "\leq"] & & C \arrow[ll, bend left, "from \cdot from'"]
\end{tikzcd}
$$

Composition of Refinements

Sequential composition

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

Nesting

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

Motivation

Two-level Type-level transformation of a data format coupled with the corresponding value-level transformation of data instances.

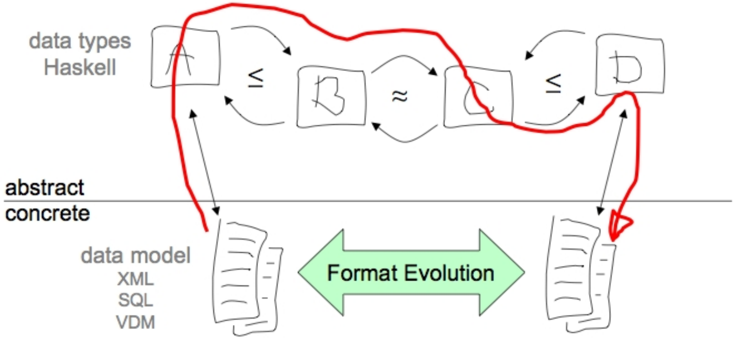
Motivation

Two-level Type-level transformation of a data format coupled with the corresponding value-level transformation of data instances.

Type-safe Type-checking guarantees that the data migration functions are well-formed with respect to the type-level transformation.

Ingredients

- Concrete data models are abstracted as Haskell data types.
- Type-level transformations are data refinements.
- Strategic programming to compose flexible rewrite systems.



Strategic Programming

- Apply refinement steps ...
  - in what order?
  - how often?
  - at what depth?
  - under which conditions?
- Compose rewrite systems from:
  - basic rewrite rules and
  - combinators for traversal construction.

Combinators

```
(>>>) :: Rule -> Rule -> Rule
(|||) :: Rule -> Rule -> Rule
nop :: Rule
many :: Rule -> Rule
once :: Rule -> Rule
```

Some Implementation Details

Representation of Types

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

Some Implementation Details

Representation of Types

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

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)
```

Some Implementation Details

Representation of Types

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

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)
```

The Type of Rules

```
type Rule = forall a . Type a -> Maybe (View (Type a))
```

Examples of Rules

Refine lists by maps

$$\begin{tikzcd} A^\star \arrow[rr, bend left, "seq2index"] \arrow[rr, phantom, "\leq"] & & \mathbb{N} \rightharpoonup A \arrow[ll, bend left, "list"] \end{tikzcd}$$

Examples of Rules

Refine lists by maps

$$\begin{tikzcd} A^\star \arrow[rr, bend left, "seq2index"] \arrow[rr, phantom, "\leq"] & & N \rightharpoonup A \arrow[ll, bend left, "list"] \end{tikzcd}$$

Rule Implementation

```
listmap :: Rule
listmap (List a) = Just (View rep (Map Int a))
    where rep = Rep {to = seq2index, from = list}
listmap _ = Nothing
```

Examples of Rules

Refine lists by maps

$$
\begin{tikzcd}
A^\star \arrow[rr, bend left, "seq2index"] \arrow[rr, phantom, "\leq"] & & N \rightharpoonup A \arrow[ll, bend left, "list"]
\end{tikzcd}
$$

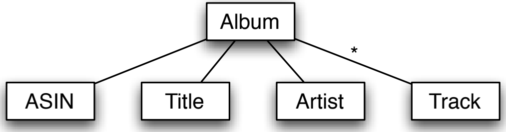
Rule Implementation

```
listmap :: Rule
listmap (List a) = Just (View rep (Map Int a))
    where rep = Rep {to = seq2index, from = list}
listmap _ = Nothing
```

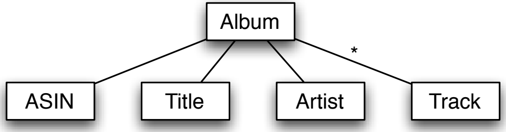
Rewrite system for hierarchical-to-relational mapping

```
flatten :: Rule
flatten = many (once (listmap ||| mapprodmap ||| ...))
```

Example



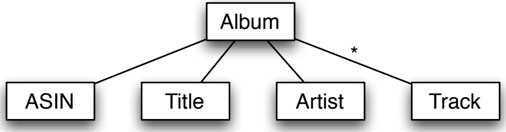
Example



Abstract representation

```
type Album  = (String, (String, (String, [String])))
```

Example



Abstract representation

```
type Album  = (String, (String, (String, [String])))
```

Converting to a relational database

```
> let (Just vw) = flatten (typeoff :: [Album])
> showType vw
(Map Int (String,(String,String)), Map (Int,Int) String)
```

Motivation

Constrains Rules must take into account constrain information such as primary and secondary keys.

Motivation

Constrains Rules must take into account constrain information such as primary and secondary keys.

Names Names are being discarded. They should also be evolved so that resulting tables have meaningful names.

Motivation

Constrains Rules must take into account constrain information such as primary and secondary keys.

Names Names are being discarded. They should also be evolved so that resulting tables have meaningful names.

Front-ends To apply 2LT to real-world examples we need front-ends for popular data description languages, namely XML and SQL.

Type Annotations

Types can be tagged

```
data Type a where
    ...
    Tag :: Tag -> Type a -> Type a

data Tag = Info {name :: Maybe Name,
                 key  :: Maybe Key,
                 refs :: [Key]}
```

Type Annotations

Types can be tagged

```
data Type a where
    ...
    Tag :: Tag -> Type a -> Type a

data Tag = Info {name :: Maybe Name,
                 key  :: Maybe Key,
                 refs :: [Key]}
```

Rules manipulate tags

$$({}_kA_r \rightharpoonup (B \times (C \rightharpoonup D)^o))^m$$
$$\cong$$
$$({}_kA_r \rightharpoonup B)^m \times ({}_{\_}A_{kr} \times C \rightharpoonup D)^o$$

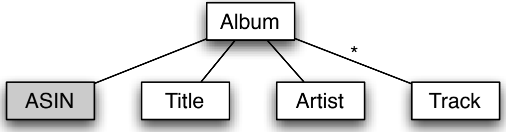
Front-ends

Group them in a class

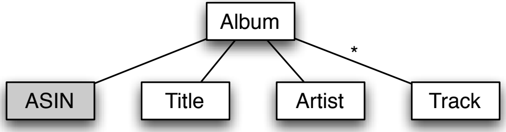
```
class FrontEnd t v | t -> v, v -> t where
    parsetype  :: t -> Maybe DynType
    printtype  :: Type a -> Maybe t
    parsevalue :: Type a -> v -> Maybe a
    printvalue :: Type a -> a -> Maybe v

instance FrontEnd XSD XML where ...
instance FrontEnd DDL DML where ...
instance FrontEnd VDM VDM where ...
```

Example



Example



Converting to a relational database

$$({}_1\mathit{String}^{Asin} \rightharpoonup \mathit{String}^{Title} \times \mathit{String}^{Artist})^{Albums} \times$$
$$(\mathit{String}_1^{Asin} \times \mathit{Int} \rightharpoonup \mathit{String}^{Track})^{Tracks}$$

Motivation

Performance The migration functions are very inefficient because of strategic combinators.

Motivation

Performance The migration functions are very inefficient because of strategic combinators.

3LT Coupled transformation should encompass not only types and values, but other artifacts like queries and producers.

$$
\begin{tikzcd}
a : A \arrow[rr, Rightarrow, "T"] \arrow[rr, bend left, "to"] \arrow[d, "q"'] & & a' : A' \arrow[ll, bend left, "from"] \arrow[dll, dashed, "q'"] \\
B & &
\end{tikzcd}
$$

Point-free Functional Programming

Some combinators

$$
\begin{array}{l}
id :: A \rightarrow A \\
(\cdot) :: (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\
fst :: A \times B \rightarrow A \\
snd :: A \times B \rightarrow B \\
(\vartriangle) :: (A \rightarrow B) \rightarrow (A \rightarrow C) \rightarrow (A \rightarrow B \times C)
\end{array}
$$

Point-free Functional Programming

Some combinators

$$
\begin{aligned}
& id :: A \rightarrow A \\
& (\cdot) :: (B \rightarrow C) \rightarrow (A \rightarrow B) \rightarrow (A \rightarrow C) \\
& fst :: A \times B \rightarrow A \\
& snd :: A \times B \rightarrow B \\
& (\triangle) :: (A \rightarrow B) \rightarrow (A \rightarrow C) \rightarrow (A \rightarrow B \times C)
\end{aligned}
$$

Some laws

$$
\begin{aligned}
& f \cdot id = f \wedge id \cdot f = f \\
& f \cdot (g \cdot h) = (f \cdot g) \cdot h \\
& fst \cdot (f \triangle g) = f \wedge snd \cdot (f \triangle g) = g \\
& fst \triangle snd = id \\
& (f \triangle g) \cdot h = (f \cdot h) \triangle (g \cdot h)
\end{aligned}
$$

Rewriting Point-free Expressions

Type-safe representation of functions

```
data PF f where
    Id    :: PF (a -> a)
    Fst   :: PF ((a,b) -> a)
    (/\)  :: PF (a -> b) -> PF (a -> c) -> PF (a -> (b,c))
    ...
```

Rewriting Point-free Expressions

Type-safe representation of functions

```
data PF f where
    Id    :: PF (a -> a)
    Fst   :: PF ((a,b) -> a)
    (/\) :: PF (a -> b) -> PF (a -> c) -> PF (a -> (b,c))
    ...
```

Another strategic rewrite system

```
type RULE = forall f . Type f -> PF f -> RewriteM (PF f)
```

Rewriting Point-free Expressions

Type-safe representation of functions

```
data PF f where
    Id   :: PF (a -> a)
    Fst  :: PF ((a,b) -> a)
    (/\) :: PF (a -> b) -> PF (a -> c) -> PF (a -> (b,c))
    ...
```

Another strategic rewrite system

```
type RULE = forall f . Type f -> PF f -> RewriteM (PF f)
```

Migration functions represented in point-free

```
data Rep a b = Rep {to :: PF (a -> b), from :: PF (b -> a)}

data View a where
    View :: Rep a b -> Type b -> View (Type a)

type Rule = forall a . Type a -> Maybe (View (Type a))
```

Example

Type refinement

```
type Album    = (String, (String, (String, [String])))
type AlbumsDB = (Map Int (String,(String,String)),
                 Map (Int,Int) String)
```

Example

Type refinement

```
type Album    = (String, (String, (String, [String])))
type AlbumsDB = (Map Int (String,(String,String)),
                 Map (Int,Int) String)
```

Query migration

```
getArtists :: [Album] -> [String]
getArtists = List.map (fst . snd . snd)

getArtists' :: AlbumsDB -> [String]
getArtists' = elems . Map.map (snd . snd) . fst
```

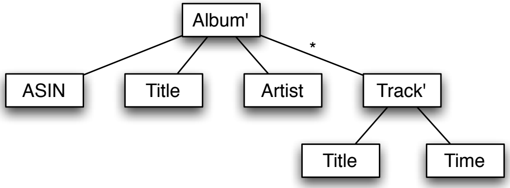
Motivation

Partiality To deal correctly with query migration the abstraction function must be explicitly partial.

Motivation

Partiality To deal correctly with query migration the abstraction function must be explicitly partial.

Generics How to deal with structure-shy languages like XPath? Does `//Title` returns the expected results after refining tracks to have both title and duration?



Composition With Partiality

Sequential composition

$$A+1 \xleftarrow{from} B \ \wedge\ B+1 \xleftarrow{from'} C$$
$$\Rightarrow$$
$$A+1 \xleftarrow{id \triangledown inr} (A+1)+1 \xleftarrow{from+id} B+1 \xleftarrow{from'} C$$

Nesting

$$A+1 \xleftarrow{from} B$$
$$\Rightarrow$$
$$[A]+1 \xleftarrow{inl \cdot concat} [[A]] \xleftarrow{map\ (wrap \triangledown nil)} [A+1] \xleftarrow{map\ \ from} [B]$$

Structure-Shy Languages

SYB

$$
\begin{aligned}
(\triangleright) &:: \mathrm{T} \to \mathrm{T} \to \mathrm{T} \\
mapT &:: \mathrm{T} \to \mathrm{T} \\
mkT_A &:: (A \to A) \to \mathrm{T} \\
apT_A &:: \mathrm{T} \to (A \to A) \\
mapQ &:: \mathrm{Q}\ R \to \mathrm{Q}\ R \\
mkQ_A &:: (A \to R) \to \mathrm{Q}\ R \\
apQ_A &:: \mathrm{Q}\ R \to (A \to R)
\end{aligned}
$$

XPath

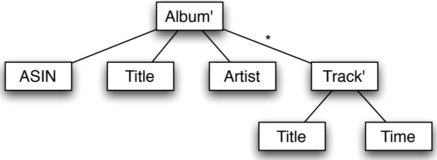
$$
\begin{aligned}
child &:: \mathrm{Q}\ [*] \\
descorself &:: \mathrm{Q}\ [*] \\
name &:: String \to \mathrm{Q}[*] \\
(/) &:: \mathrm{Q}\ [*] \to \mathrm{Q}\ R \to \mathrm{Q}\ R
\end{aligned}
$$

Specializing Structure-Shy Programs

Some Laws

$$
\begin{aligned}
& apT_A\ (f \triangleright g) = apT_A\ f \cdot apT_A\ g \\
& apT_A\ (mkT_A\ f) = f \\
& apT_A\ (mkT_B\ f) = id, \mathbf{if} A \neq B \\
& apT_(A \times B)\ (mapT\ \ f) = apT_A\ f \times apT_B\ f \\
& apQ_A\ (mkQ_A\ f) = f \\
& apQ_A\ (mkQ_B\ f) = zero, \mathbf{if} A \neq B \\
& apQ_(A \times B)\ (mapQ\ \ f) = plus \cdot (apQ_A\ f \times apQ_B\ f) \\
& descendant\ = everything\ \ child \\
& apQ_A\ (f / g) = fold \cdot map\ (apQ_*\ g) \cdot apQ_A\ f
\end{aligned}
$$

Example



Specialization of //Title to Album'

```
cat . (map (mkAny . title) /\
       concat . map (map (mkAny . track_title) . tracks))
    where title  = fst . snd . unAlbum'
          tracks = snd . snd . snd . unAlbum'
          track_title = fst . unTrack'
```

Specialization of //Title to Album composed with abstraction

```
map (mkAny . title)
    where title  = fst . snd . unAlbum'
```

Lenses

A concrete type A is abstracted into a view B

$$\begin{tikzcd} A \arrow[r, bend left, "get"] \arrow[r, phantom, "\geq"] & B \times A \arrow[l, bend left, "putback"] \end{tikzcd}$$

Properties

Acceptability *putback* must be injective on first argument.

$$get \cdot putback \sqsubseteq fst$$

Stability If the target does not change, neither should the source.

$$putback \cdot (get \vartriangle id) \sqsubseteq id$$

Example of Lenses

Drop field

$$\begin{tikzcd} A \times B \arrow[rr, bend left, "fst"] \arrow[rr, phantom, "\geq"] & & A \times A \times B \arrow[ll, bend left, "id \times snd"] \end{tikzcd}$$

Composition of Lens

Sequential composition

$$
\begin{tikzcd}
A \arrow[rr, bend left, "get"] \arrow[rr, phantom, "\geq"] & & B \times A \arrow[ll, bend left, "putback"]
\end{tikzcd}
\quad \wedge \quad
\begin{tikzcd}
B \arrow[rr, bend left, "get'"] \arrow[rr, phantom, "\geq"] & & C \times B \arrow[ll, bend left, "putback'"]
\end{tikzcd}
$$

$$\Rightarrow$$

$$
\begin{tikzcd}
A \arrow[rr, bend left, "get' \cdot get"] \arrow[rr, phantom, "\geq"] & & C \times A \arrow[ll, bend left]
\end{tikzcd}
$$

$$A \xleftarrow{putback} B \times A \xleftarrow{putback' \times id} (C \times B) \times A \xleftarrow{(id \times get) \triangle id} C \times A$$

The Future of 2LT

- We want a single version of 2LT but...
  - Recursion and tags are not well supported in the `rule` release.
  - Haskell's type system is not compatible.

The Future of 2LT

- We want a single version of 2LT but. . .
  - Recursion and tags are not well supported in the `rule` release.
  - Haskell's type system is not compatible.
- We will reimplement 2LT from scratch:
  - With a dedicated syntax more appealing to non-Haskell users.
  - With a proper type-system.

The Future of 2LT

- We want a single version of 2LT but. . .
  - Recursion and tags are not well supported in the `rule` release.
  - Haskell's type system is not compatible.
- We will reimplement 2LT from scratch:
  - With a dedicated syntax more appealing to non-Haskell users.
  - With a proper type-system.
- We want more features:
  - Proper invariants to deal with constrains.
  - Front-ends to other query languages, namely SQL.
  - Support for mutually recursive types.

The Future of 2LT

- We want a single version of 2LT but...
  - Recursion and tags are not well supported in the `rule` release.
  - Haskell's type system is not compatible.
- We will reimplement 2LT from scratch:
  - With a dedicated syntax more appealing to non-Haskell users.
  - With a proper type-system.
- We want more features:
  - Proper invariants to deal with constrains.
  - Front-ends to other query languages, namely SQL.
  - Support for mutually recursive types.
- And we would love to support lenses!