Type-Safe Two-Level Data Transformation

Alcino Cunha, José Nuno Oliveira, and Joost Visser

Universidade do Minho, Portugal

FM'06, August 24th

Outline

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.

Motivation

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

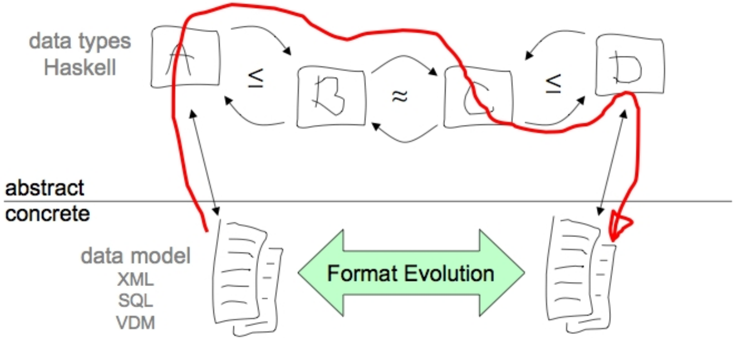
Type-safe Type-checking guarantees that the data migration functions are well-formed with respect to the type-level transformation.

User-driven XML schema evolution coupled with document migration.

Automated Data mappings for storing XML in relational databases.

Ingredients

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

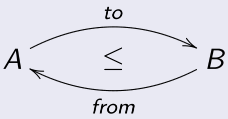


Data Refinement

An abstract type A is mapped to a concrete type B

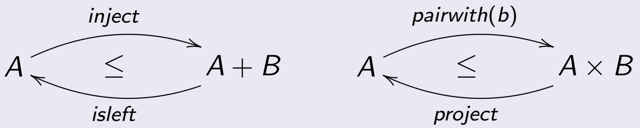
Representation Injective and total.

Abstraction Surjective and possibly partial.



Examples of Refinements

Format evolution



Examples of Refinements

Format evolution

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

Hierarchical to relational mappings

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

Composition of Refinements

Sequential composition

$$A \underset{from}{\overset{to}{\underset{\longleftarrow}{\overset{\longrightarrow}{\leq}}}} B \;\wedge\; B \underset{from'}{\overset{to'}{\underset{\longleftarrow}{\overset{\longrightarrow}{\leq}}}} C \;\Rightarrow\; A \underset{from \cdot from'}{\overset{to' \cdot to}{\underset{\longleftarrow}{\overset{\longrightarrow}{\leq}}}} C$$

Composition of Refinements

Sequential composition

$$A \;\underset{\mathit{from}}{\overset{\mathit{to}}{\leq}}\; B \;\wedge\; B \;\underset{\mathit{from}'}{\overset{\mathit{to}'}{\leq}}\; C \;\Rightarrow\; A \;\underset{\mathit{from}\cdot\mathit{from}'}{\overset{\mathit{to}'\cdot\mathit{to}}{\leq}}\; C$$

Nesting

$$A \;\underset{\mathit{from}}{\overset{\mathit{to}}{\leq}}\; B \;\Rightarrow\; \mathsf{F}\,A \;\underset{\mathsf{F}\,\mathit{from}}{\overset{\mathsf{F}\,\mathit{to}}{\leq}}\; \mathsf{F}\,B$$

Strategic Programming

- Apply refinement steps . . .
  - in what order?
  - how often?
  - at what depth?
  - under which conditions?

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.

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
```

Representation of Types

The Type of Types

```
data Type a where
    Int :: Type Int
    String :: Type String
    One :: Type ()
    List :: Type a -> Type [a]
    Map :: Type a -> Type b -> Type (Map a b)
    Either :: Type a -> Type b -> Type (Either a b)
    Prod :: Type a -> Type b -> Type (a,b)
    Tag :: String -> Type a -> Type a
```

Type-Changing Rewrite Rules

How to combine strategic programming with type-changing rules?

Type-Changing Rewrite Rules

How to combine strategic programming with type-changing rules?

Masquerade Changes as Views

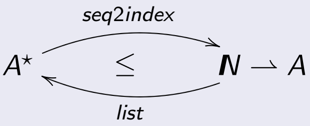
```
data Rep a b = Rep {to :: a -> b, from :: b -> a}

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))
```

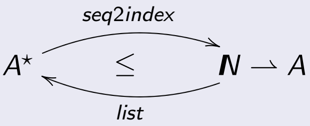
Examples of Rules

Refine Lists by Maps



Examples of Rules

Refine Lists by Maps

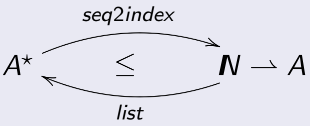


Rule Implementation

```
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



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 ||| ...))
```

Unleashing the Migration Functions

- The target type is existentially quantified in a view.
- Since its not known statically we can use a staged approach:
  1. Apply the intended transformation to compute it dynamically and get its string representation using `showType`.
  2. Incorporate that string in the source and unleash the migration functions.

```
forth :: View (Type a) -> Type b -> a -> Maybe b
back  :: View (Type a) -> Type b -> b -> Maybe a

data Equal a b where Eq :: Equal a a
teq :: Type a -> Type b -> Maybe (Equal a b)
```

Evolution of a Music Album Format

Concrete XML Schema

```
<element name="Album" type="AlbumType"/>
<complexType name="AlbumType">
  <attribute name="ASIN" type="string"/>
  <attribute name="Title" type="string"/>
  <attribute name="Artist" type="string"/>
  <attribute name="Format"><simpleType base="string">
    <enumeration value="LP"/><enumeration value="CD"/>
  </simpleType></attribute>
</complexType>
```

Evolution of a Music Album Format

Abstract Haskell Type

```
albumFormat = Tag "Album" (
   Prod (Tag "ASIN" String) (
   Prod (Tag "Title" String) (
   Prod (Tag "Artist" String) (
         Tag "Format" (Either (Tag "LP" One)
                              (Tag "CD" One)))))
```

Evolution of a Music Album Format

Abstract Haskell Type

```
albumFormat = Tag "Album" (
   Prod (Tag "ASIN" String) (
   Prod (Tag "Title" String) (
   Prod (Tag "Artist" String) (
         Tag "Format" (Either (Tag "LP" One)
                              (Tag "CD" One)))))))
```

evolve =

```
once (inside "Format" (addalt (Tag "DVD" One))) >>>
once (inside "Album" (addfield (List String) query))
```

Mapping Albums to Relational Tables

```
tordb =
once enum2int >>> removetags >>> flatten
```

Mapping Albums to Relational Tables

```
tordb =
```

```
once enum2int >>> removetags >>> flatten
```

Computing the Target Type

```
> let (Just vw) = evolve >>> tordb (List albumFormat)
> showType vw
Prod (Map Int
          (Prod (Prod (Prod String String) String) Int))
      (Map (Prod Int Int) String)
```

Data Migration

Sample

```
lp = ("B000002UB2",("Abbey Road",("Beatles",Left ())))
cd = ("B000002HCO",("Debut",("Bjork",Right ())))
```

Data Migration

Sample

```
lp = ("B000002UB2",("Abbey Road",("Beatles",Left ())))
cd = ("B000002HCO",("Debut",("Bjork",Right ())))
```

Migrating Data

```
> let dbs = Prod (Map ...) (Map (Prod Int Int) String)
> let (Just db) = forth vw dbs [lp,cd]
> db
({0 := ((("B000002UB2","Abbey Road"),"Beatles"),0),
  1 := ((("B000002HCO","Debut"),"Bjork"),1)},
 {(0,0) := "Come Together",
  (0,1) := "Something",
  ...})
```

Conclusion

- Conclusions:
  - Type-safe formalization of two-level data transformations.
  - Haskell's type system, namely GADTs, allows a direct and elegant implementation.
  - Allows flexible rewrite systems but termination and confluence is not guaranteed.
  - Restricted to single-recursive data types.

Conclusion

- Conclusions:
  - Type-safe formalization of two-level data transformations.
  - Haskell's type system, namely GADTs, allows a direct and elegant implementation.
  - Allows flexible rewrite systems but termination and confluence is not guaranteed.
  - Restricted to single-recursive data types.
- Current status:
  - Coupled transformation of data processing programs, such as queries expressed in a point-free notation.
  - Front-ends for XML and SQL database schemas.

Conclusion

- Conclusions:
  - Type-safe formalization of two-level data transformations.
  - Haskell's type system, namely GADTs, allows a direct and elegant implementation.
  - Allows flexible rewrite systems but termination and confluence is not guaranteed.
  - Restricted to single-recursive data types.
- Current status:
  - Coupled transformation of data processing programs, such as queries expressed in a point-free notation.
  - Front-ends for XML and SQL database schemas.
- Future work:
  - Bi-directional programming.
  - Data types with invariants.
  - Mutually-recursive data types.