Scrapping Your Dependently-Typed Boilerplate is Hard

Ahmad Salim Al-Sibahi
Supervised by:
Dr. Peter Sestoft
David R. Christiansen
IT University of Copenhagen

Disposition

- Introduction
- Uniplate in 10 minutes
- The Hard Part
- Related Work

Introduction

- Outline for a problem
- Example-oriented approach
- Based on my work on described types

Motivation

- Generic traversal frameworks aims to reduce repetitive programs using practical interfaces
- Dependently-typed programming is getting increasingly popular as paradigm

A Framework to Scrap Your Boilerplate

UNIPLATE IN 10 MINUTES

Background

- Popular framework created by Neil Mitchell
- Based on Scrap Your Boilerplate (SYB) by Ralf Lämmel and Simon Peyton-Jones

Background

- Type-directed querying and transformation
- Automatically derivable using generic programming techniques

Interface - Uniplate


```
children   ::  Uniplate on =>
               on -> [on]
universe   ::  Uniplate on =>
               on -> [on]


descend    ::  Uniplate on =>
               (on -> on) -> on -> on
transform  ::   Uniplate on =>
               (on -> on) -> on -> on
```

Interface - Biplate


```
childrenBi  :: Biplate from to =>
               from -> [to]
universeBi  :: Biplate from to =>
               from -> [to]

descendBi   :: Biplate from to =>
               (to -> to) -> from -> from
transformBi :: Biplate from to =>
               (to -> to) -> from -> from
```

Example: Blog Post


```
type Title = String
type Timestamp = Int

data Post = Single Title Timestamp
          | Aggregate [Post]

timestamps :: Post -> [Timestamp]
timestamps = universeBi

capitaliseTitles :: Post -> Post
capitaliseTitles = transformBi capitalise
```

Adapted Realistic Case


```
data  TT =
      Ref Name TT
    | Var Int
    | Bind Name TT TT
    | App TT TT
    | ConstantStr String
    | ConstantI Int
    | Proj TT Int
    | Type
    | Erased
```

Adapted Realistic Case


```
freeNames :: TT -> [Name]
freeNames (Ref n _)       =[n]
freeNames (Bind n ty sc) =
      freeNames ty ++ (freeNames sc \\ [n])
freeNames (App tf ta) =
      freeNames tf ++ freeNames ta
freeNames (Proj tm _) =
       freeNames tm
freeNames _ = []
```

Adapted Realistic Case


```
freeNames :: TT -> [Name]
freeNames (Ref n _)       = [n]
freeNames (Bind n ty sc) =
   freeNames ty ++ (freeNames sc \\ [n])
freeNames tm =
   concat [freeNames t | t <- children tm]
```

Uniplate type class


```
class Uniplate on where
  uniplate :: on ->
              ([on], [on] -> on)
```

Instance of Uniplate


```
instance Uniplate Post where
  uniplate (Single ttl ts)  =
        ([],
        \_      -> Single ttl ts)
  uniplate (Aggregate psts) =
        (psts,
        \psts' -> Aggregate psts')
```

Biplate type class


```
class Uniplate to => Biplate from to where
  biplate :: from -> ([to], [to] -> from)
```

Instance of Biplate


```
instance Biplate Post Timestamp where
  biplate (Single ttl ts)  =
   ([ts],
   \ts' -> Single ttl (head ts'))

  biplate (Aggregate psts) =
   ([],
   \_   -> Aggregate psts)
```

How I tried and failed to specify a correct interface for a dependent version of Uniplate

THE HARD PART

Automatic Deriving

- Significantly simplifies usage of the library
- Depends on structure of datatypes (like Eq or Show)
- Works only on monomorphic types

Automatic Deriving


```
data List_Nat : Set where
  nil  : List_Nat
  cons : (head : Nat) (tail : List_Nat)
         -> List_Nat
```

Automatic Deriving


```
data Vec_Nat : Nat -> Set where
  nil  : Vec_Nat zero
  cons : {n : Nat}
         (head : Nat) (tail : Vec_Nat n)
         -> Vec_Nat (suc n)
```

Index indistinguishable from ordinary data

Automatic Deriving


```
data OList : Nat -> Set where
  nil  : {n : Nat} -> OList n
  cons : {n : Nat}
         (head : Nat) (ok : n <= head)
         (tail : OList head)
          -> OList n
```

Index is based on ordinary data

Generic Querying


```
childrenBi : {from to : Set}
             {{bip : Biplate from to}}
             -> from -> List to

universeBi : {from to : Set}
             {{bip : Biplate from to}}
             -> from -> List to
```

List Nat may be interesting
Vec 0 Nat is definitely not

Generic Querying


```
childrenBi : {from ix : Set}
             {to : ix -> Set}
             {{bip : Biplate from to}}
              -> from -> List (Sigma ix to)

universeBi : {from ix : Set}
             {to : ix -> Set}
             {{bip : Biplate from to}}
              -> from -> List (Sigma ix to)
```

Is all data we are getting useful?

Generic Traversal


```
descendBi : {from to : Set}
            {{bip : Biplate from to}}
            -> (to -> to) -> from -> from


transformBi : {from to : Set}
              {{bip : Biplate from to}}
            -> (to -> to) -> from -> from
```

Generic Traversal

```
descendBi  : {from ix : Set}
             {to : ix -> Set}
             {{bip : Biplate from to}}
           -> (Sigma ix to -> Sigma ix to)
           -> from -> from
transformBi : {from ix : Set}
              {to : ix -> Set}
              {{bip : Biplate from to}}
            -> (Sigma ix to -> Sigma ix to)
            -> from -> from
```

Is this OK?

Generic Traversal


```
data TwoVec : Nat -> Set where
  two_vec : {n : Nat}
            -> Vec_Nat n -> Vec_Nat (S n)
            -> TwoVec n

double : Sigma Nat Vec_Nat
      -> Sigma Nat Vec_Nat
double (proj1 , proj2) =
       plus proj1 proj1 , append proj2 proj2
```

What if we do "`descendBi double xs`"?

Generic Traversal

- Transformations can break dependent datatype invariants
- Transformations must be constrained to only allow creation of valid structures

Generic Traversal


```
data OList : Nat -> Set where
  nil  : {n : Nat} -> OList n
  cons : {n : Nat}
         (head : Nat) (ok : n <= head)
         (tail : OList head)
          -> OList n
```

OList is the best to describe the constraints of OList

Generic Traversal


```
data Vec (A : Set) (n : Nat) : Set where
  nil  :  {{ix : n == zero}} -> Vec A n
  _::_  : {m : Nat} {{ix : n == suc m}}
          (x : A) (xs : Vec A m) -> Vec A n
```

Index-like restrictions can be added to any datatype

Fret not! There is still hope.

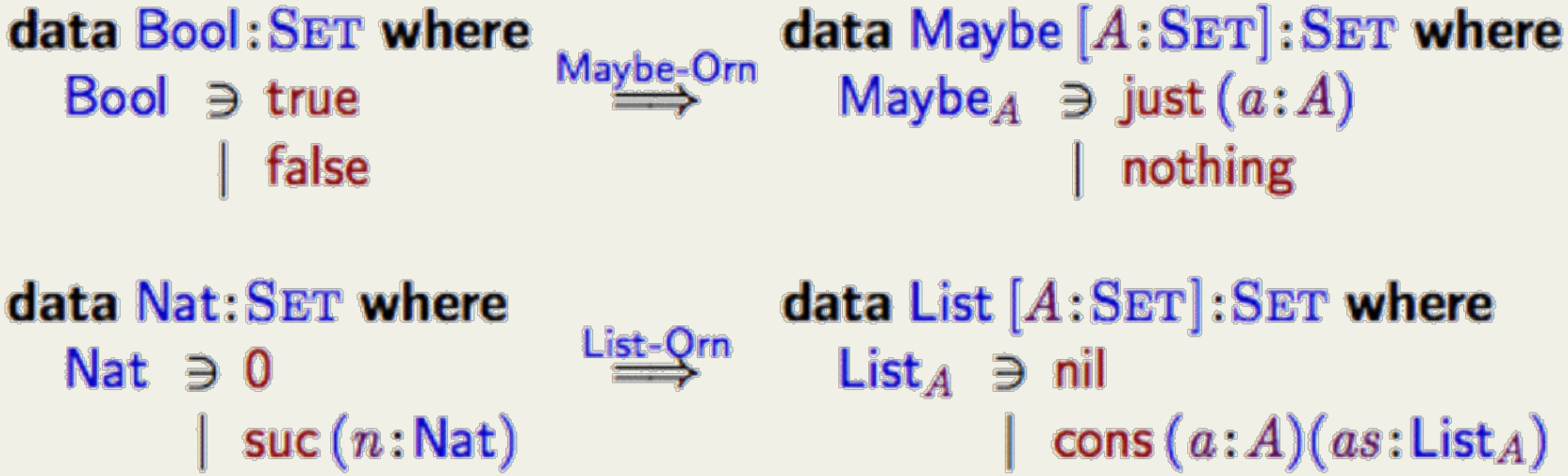
RELATED WORK

SYB in a closed universe

- Work by Larry Diehl
- Typecasing on universes using described types
- Traversal change the type of input

Transporting functions across ornaments

- Work by Conor McBride and Piérre-Evariste Dagand



Transporting function across ornaments

Just 3 more minutes…

DISCUSSION

Conclusion

- Dependent types allow us to constraint our datatypes
  - However we lose structural genericity
- It is hard to define a correct generalized interface for SYB-style programming
  - Automatic Deriving: Hard
  - Generic Querying: Doable
  - Generic Traversal: Hard
- Extrinsic vs. Intrinsic proving

Contribution

- An example-based problem definition of why it can be hard to implement a SYB-style generics library using dependent types

Questions⁉

And feedback too!