Datatype-generic programming meets elaborator reflection

Josh Ko, Liang-Ting Chen, and Tzu-Chi Lin
Institute of Information Science, Academia Sinica, Taiwan

International Conference on Functional Programming (ICFP), 13 September 2022, Ljubljana, Slovenia

Libraries for dependently typed programming

Libraries for dependently typed programming (in Agda)

```
data List (A : Set) : Set where
  []   : List A
  _∷_  : A → List A → List A
```

Libraries for dependently typed programming (in Agda)

```
data List (A : Set) : Set where
  []    : List A
  _∷_   : A → List A → List A
```

Libraries for dependently typed programming (in Agda)

```
data List (A : Set) : Set where
  []    : List A
  _∷_   : A → List A → List A


data ListAny (P : A → Set) : List A → Set where
  here     : P a             → ListAny P (a ∷ as)
  there    : ListAny P as → ListAny P (a ∷ as)
```

Libraries for dependently typed programming (in Agda)

```
data List (A : Set) : Set where
  []    : List A
  _∷_   : A → List A → List A


data ListAny (P : A → Set) : List A → Set where
  here     : P a           → ListAny P (a ∷ as)
  there    : ListAny P as → ListAny P (a ∷ as)


lookupListAny : ListAny P as → Σ A P
lookupListAny (here p)  = _ , p
lookupListAny (there i) = lookupListAny i
```

Libraries for dependently typed programming (in Agda)

```
data List (A : Set) : Set where
  []    : List A
  _∷_   : A → List A → List A
```

primary

derived

```
data ListAny (P : A → Set) : List A → Set where
  here     : P a          → ListAny P (a ∷ as)
  there    : ListAny P as → ListAny P (a ∷ as)


lookupListAny : ListAny P as → Σ A P
lookupListAny (here p)  = _ , p
lookupListAny (there i) = lookupListAny i
```

Definitions derived from datatype structure

```
data Tree (A : Set) : Set where
  leaf : A → Tree A
  bin  : Tree A → Tree A → Tree A
```

primary

derived

```
data ListAny (P : A → Set) : List A → Set where
  here    : P a          → ListAny P (a ∷ as)
  there   : ListAny P as → ListAny P (a ∷ as)


lookupListAny : ListAny P as → Σ A P
lookupListAny (here p)  = _ , p
lookupListAny (there i) = lookupListAny i
```

Definitions derived from datatype structure

```
data Tree (A : Set) : Set where
  leaf : A → Tree A
  bin  : Tree A → Tree A → Tree A
```

primary

derived

```
data TreeAny (P : A → Set) : Tree A → Set where
  here    : P a              → TreeAny P (leaf a)
  there-l : TreeAny P t  → TreeAny P (bin t u)
  there-r : TreeAny P u  → TreeAny P (bin t u)

lookupTreeAny : TreeAny P t → Σ A P
lookupTreeAny (here p)    = _ , p
lookupTreeAny (there-l i) = lookupTreeAny i
lookupTreeAny (there-r i) = lookupTreeAny i
```

Write your own datatype

```
data Lam : Set where
  var  : Nat         → Lam
  app  : Lam → Lam → Lam
  lam  : Lam         → Lam
```

Write your own datatype

```
data Lam                :       Set    where
  var   : Nat                        → Lam
  app   : Lam        → Lam           → Lam
  lam   : Lam                        → Lam
```

Write your own datatype

```
data Myst              :       Set    where
  var  : Nat                        → Myst
  app  : Myst      → Myst           → Myst
  lam  : Myst                       → Myst
```

Write your own datatype

```
data Myst                 : Nat → Set    where
  var  : Fin n                         → Myst   n
  app  : Myst   n → Myst   n           → Myst   n
  lam  : Myst   (suc n)                → Myst   n
```

Write your own datatype

```
data Myst             : Nat → Set   where
  var  : Fin n                     → Myst   n
  app  : Myst   n → Myst   n       → Myst   n
  lam  : Myst   (suc n)            → Myst   n
  łet  : Myst   n → Myst   (suc n) → Myst   n
```

Write your own datatype

```
data Myst                   : Nat → Set   where
  var  : Fin n                          → Myst   n
  app  : Myst   n → Myst   n            → Myst   n
  lam  : Myst   (suc n)                 → Myst   n
  łet  : Myst   n → Myst   (suc n)      → Myst   n
  myst : A                              → Myst   n
```

Write your own datatype

```
data Myst (A : Set  ) : Nat → Set   where
  var  : Fin n                       → Myst A n
  app  : Myst A n → Myst A n         → Myst A n
  lam  : Myst A (suc n)              → Myst A n
  łet  : Myst A n → Myst A (suc n) → Myst A n
  myst : A                           → Myst A n
```

Write your own datatype

```
data Myst (A : Set ℓ) : Nat → Set ℓ where
  var  : Fin n                        → Myst A n
  app  : Myst A n → Myst A n          → Myst A n
  lam  : Myst A (suc n)               → Myst A n
  łet  : Myst A n → Myst A (suc n) → Myst A n
  myst : A                            → Myst A n
```

Write your own datatype

```
data Myst (A : Set ℓ) : Nat → Set ℓ where
  var  : Fin n                          → Myst A n
  app  : Myst A n → Myst A n            → Myst A n
  lam  : Myst A (suc n)                 → Myst A n
  łet  : Myst A n → Myst A (suc n) → Myst A n
  myst : A                              → Myst A n
```

Write your own datatype

```
data Myst (A : Set ℓ) : Nat → Set ℓ where
  var  : Fin n                         → Myst A n
  app  : Myst A n → Myst A n           → Myst A n
  lam  : Myst A (suc n)                → Myst A n
  łet  : Myst A n → Myst A (suc n) → Myst A n
  myst : A                             → Myst A n
```

primary

Derive definitions from your own datatype

derived

```
data MystAny {A : Set ℓ} (P : A → Set ℓ')
  : {n : Nat} → Myst A n → Set (ℓ ⊔ ℓ') where
  here         : P a          → MystAny P (myst a)
  there-app-l  : MystAny P t → MystAny P (app t u)
  there-app-r  : MystAny P u → MystAny P (app t u)
  there-lam    : MystAny P t → MystAny P (abs t)
  there-łet-l  : MystAny P t → MystAny P (łet t u)
  there-łet-r  : MystAny P u → MystAny P (łet t u)

lookupMystAny : MystAny P t → Σ A P
lookupMystAny (here p)        = _ , p
lookupMystAny (there-app-l i) = lookupMystAny i
lookupMystAny (there-app-r i) = lookupMystAny i
lookupMystAny (there-lam   i) = lookupMystAny i
lookupMystAny (there-łet-l i) = lookupMystAny i
lookupMystAny (there-łet-r i) = lookupMystAny i
```

Use derived definitions ‘natively’

```
some-theorem : (t : Myst A n) → WellTyped t
             → (i : MystAny P t) → Nice (lookupMystAny i) t
some-theorem t wt i = {!   !}
```

Use derived definitions ‘natively’

```
some-theorem : (t : Myst A n) → WellTyped t
             → (i : MystAny P t) → Nice (lookupMystAny i) t
some-theorem t wt i = {! i !}
```

interactive case splitting

Use derived definitions ‘natively’

```
some-theorem : (t : Myst A n) → WellTyped t
             → (i : MystAny P t) → Nice (lookupMystAny i) t
some-theorem .(myst _)  wt (here p)         = {!   !}
some-theorem .(app _ _) wt (there-app-l i) = {!   !}
some-theorem .(app _ _) wt (there-app-r i) = {!   !}
some-theorem .(lam _)   wt (there-lam   i) = {!   !}
some-theorem .(łet _ _) wt (there-łet-l i) = {!   !}
some-theorem .(łet _ _) wt (there-łet-r i) = {!   !}
```

Use derived definitions ‘natively’

```
some-theorem : (t : Myst A n) → WellTyped t
             → (i : MystAny P t) → Nice (lookupMystAny i) t
some-theorem .(myst _)  wt (here p)         = {!   !}
some-theorem .(app _ _) wt (there-app-l i) = {!   !}
some-theorem .(app _ _) wt (there-app-r i) = {!   !}
some-theorem .(lam _)   wt (there-lam   i) = {!   !}
some-theorem .(łet _ _) wt (there-łet-l i) = {!   !}
some-theorem .(łet _ _) wt (there-łet-r i) = {!   !}
```

↑ **goal type**

```
Nice (lookupMystAny (there-łet-r i)) (łet t u)
```

Use derived definitions ‘natively’

```
some-theorem : (t : Myst A n) → WellTyped t
             → (i : MystAny P t) → Nice (lookupMystAny i) t
some-theorem .(myst _)  wt (here p)         = {!   !}
some-theorem .(app _ _) wt (there-app-l i) = {!   !}
some-theorem .(app _ _) wt (there-app-r i) = {!   !}
some-theorem .(lam _)   wt (there-lam   i) = {!   !}
some-theorem .(łet _ _) wt (there-łet-l i) = {!   !}
some-theorem .(łet _ _) wt (there-łet-r i) = {!   !}
```

↑ **goal type**

```
  Nice (lookupMystAny (there-łet-r i)) (łet t u)
= Nice (lookupMystAny i) (łet t u)
```

Want:

Generic constructions for deriving native definitions from our own datatypes

First try:

Elaborator reflection

Elaborator reflection

Reflected syntax trees

```
data Term : Set where

  var : Nat → (args : List Term) → Term
  lam : …
  lit : …
  ⋮

  pi  : Term → Term → Term
  ⋮
```

Elaborator reflection

Reflected syntax trees

```
data Term : Set where

  var : Nat → (args : List Term) → Term
  lam : …
  lit : …
  ⋮

  pi  : Term → Term → Term                        Type : Set
  ⋮                                               Type = Term
```

Elaborator reflection

Type-checking monad

```
declareDef  : Name → Type → TC ⊤
defineFun   : Name → List Clause → TC ⊤

declareData : Name → (nParams : ℕ) → Type → TC ⊤
defineData  : Name → List (Name × Type) → TC ⊤

unify       : Term → Term → TC ⊤
normalise   : Term → TC Term
⋮
```

Elaborator reflection

Type-checking monad

```
declareDef   : Name → Type → TC ⊤
defineFun    : Name → List Clause → TC ⊤

declareData  : Name → (nParams : ℕ) → Type → TC ⊤
defineData   : Name → List (Name × Type) → TC ⊤

unify        : Term → Term → TC ⊤
normalise    : Term → TC Term
⋮
```

pull request merged recently

Don’t like: (very) imprecisely typed representations

```
pi (lam …) (… (var 666 []) …) : Type
```

```
data Term : Set where

  var : Nat → (args : List Term) → Term
  lam : …
  ⋮
                                              Type : Set
  pi  : Term → Term → Term                    Type = Term
  ⋮
```

Don't like: (very) imprecisely typed representations

```
pi (lam …) (… (var 666 []) …) : Type
```

not a type expression

```
data Term : Set where

  var : Nat → (args : List Term) → Term
  lam : …
  ⋮
                                        Type : Set
  pi  : Term → Term → Term              Type = Term
  ⋮
```

Don’t like: (very) imprecisely typed representations

out-of-bounds de Bruijn index

```
pi (lam …) (… (var 666 []) …) : Type
```

not a type expression

```
data Term : Set where

  var : Nat → (args : List Term) → Term
  lam : …
  ⋮
                                          Type : Set
  pi  : Term → Term → Term                Type = Term
  ⋮
```

Second try:

Datatype-generic programming

Datatype-generic programming

Precisely typed representations

```
record DataD : Setω where
⋮

data ConD : List (Level ⊎ …) → Setω where
  ⋮
  σ : (A : Set ℓ) → (A → ConD cb) → ConD (inl ℓ ∷ cb)
  ⋮
```

Datatype-generic programming

Precisely typed representations

```
σ Nat (λ n → … n …) : ConD (inl 0ℓ ∷ …)
```

```
data ConD : List (Level ⊎ …) → Setω where
  ⋮
  σ : (A : Set ℓ) → (A → ConD cb) → ConD (inl ℓ ∷ cb)
  ⋮
```

Datatype-generic programming

Precisely typed representations

```
σ Nat (λ n → … n …) : ConD (inl 0ℓ ∷ …)
```

has to be a type

```
data ConD : List (Level ⊎ …) → Setω where
  ⋮
  σ : (A : Set ℓ) → (A → ConD cb) → ConD (inl ℓ ∷ cb)
  ⋮
```

Datatype-generic programming

Precisely typed representations

ordinary variable of the right type

```
σ Nat (λ n → … n …) : ConD (inl 0ℓ ∷ …)
```

has to be a type

```
data ConD : List (Level ⊍ …) → Setω where
  ⋮
  σ : (A : Set ℓ) → (A → ConD cb) → ConD (inl ℓ ∷ cb)
  ⋮
```

Datatype-generic programming

Precisely typed representations

ordinary variable of the right type

```
σ Nat (λ n → … n …) : ConD (inl 0ℓ ∷ …)
```

has to be a type

internal reasoning about universe level–correctness

```
data ConD : List (Level ⊎ …) → Setω where
  ⋮
  σ : (A : Set ℓ) → (A → ConD cb) → ConD (inl ℓ ∷ cb)
  ⋮
```

Don’t like: ‘non-native’ datatypes (and functions)

```
data μ (D : DataD) (ℓs : D .Levels) (ps : ⟦ D .applyL ℓs .Param ⟧ᵗ)
      : let D' = D .applyL ℓs
        in  ⟦ D' .Index ps ⟧ᵗ → Set (D' .dlevel) where
  con : let D' = D .applyL ℓs
        in  rewriteLevel (level-pre-fixed-point D')
              (Lift (D' .dlevel) (⟦ D ⟧ᵈ (μ D ℓs ps) is))
          → μ D ℓs ps is
```

Precisely typed representations
× native definitions

Our framework

user code

library

```
data Myst …
  ⋮
```

`AnyD`

`lookupAny`

Our framework

user code | **interfacing code** | **library**

```
data Myst …
   ⋮
```

```
AnyD

lookupAny
```

Our framework

user code

```
data Myst …
  ⋮
```

interfacing code

```
MystD : DataD
MystD = genDataD Myst
```

library

```
AnyD

lookupAny
```

Our framework

user code | **interfacing code** | **library**

```
data Myst …
  ⋮
```

```
MystD : DataD
MystD = genDataD Myst
```

```
AnyD

lookupAny
```

Our framework

user code

```
data Myst …
  ⋮
```

interfacing code

```
MystC : DataC …
MystC = genDataC …
MystD : DataD
MystD = genDataD Myst
```

library

```
AnyD

lookupAny
```

Our framework

user code

```
data Myst …
  ⋮
```

interfacing code

```
MystC : DataC …
MystC = genDataC …

        MystD : DataD
        MystD = genDataD Myst

MystS : SimpleContainer MystD
MystS = …
```

library

```
AnyD

lookupAny
```

Our framework

user code

```
data Myst …
  ⋮
```

interfacing code

```
MystC : DataC …
MystC = genDataC …

          MystD : DataD
          MystD = genDataD Myst

MystS : SimpleContainer MystD
MystS = …

          MystAnyD : DataD
          MystAnyD = AnyD MystC MystS
```

library

```
AnyD

lookupAny
```

Our framework

user code

```
data Myst …
  ⋮

unquoteDecl
  data MystAny …
  = defineByDataD
      MystAnyD …
```

interfacing code

```
MystC : DataC …
MystC = genDataC …

MystD : DataD
MystD = genDataD Myst

MystS : SimpleContainer MystD
MystS = …

MystAnyD : DataD
MystAnyD = AnyD MystC MystS
```

library

```
AnyD

lookupAny
```

Our framework

user code

```
data Myst …
  ⋮

unquoteDecl
  data MystAny …
  = defineByDataD
      MystAnyD …
```

interfacing code

```
MystC : DataC …
MystC = genDataC …

                MystD : DataD
                MystD = genDataD Myst

MystS : SimpleContainer MystD
MystS = …

                MystAnyD : DataD
                MystAnyD = AnyD MystC MystS

MystAnyC : DataC …
MystAnyC = genDataC …
```

library

```
AnyD

lookupAny
```

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
```

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
unquoteDecl lookupMystAny = defineFold alg …
```

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
unquoteDecl lookupMystAny = defineFold alg …
```

```
lookupMystAny : FoldNT alg …
lookupMystAny i = fold-base alg lookupMystAny i
```

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
unquoteDecl lookupMystAny = defineFold alg …
```

type of the fold function ⦇ alg ⦈

```
lookupMystAny : FoldNT alg …
lookupMystAny i = fold-base alg lookupMystAny i
```

body of the fold function ⦇ alg ⦈

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
unquoteDecl lookupMystAny = defineFold alg …
```

type of the fold function ⦇alg⦈

```
lookupMystAny : FoldNT alg …
lookupMystAny i = fold-base alg lookupMystAny i
```

body of the fold function ⦇alg⦈

```
normalise : Term → TC Term
```

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
unquoteDecl lookupMystAny = defineFold alg …
```

```
lookupMystAny : FoldNT alg …
lookupMystAny (here p)        = fold-base alg lookupMystAny (here p)
lookupMystAny (there-app-l i) = fold-base alg lookupMystAny (there-app-l i)
lookupMystAny (there-app-r i) = fold-base alg lookupMystAny (there-app-r i)
lookupMystAny (there-lam   i) = fold-base alg lookupMystAny (there-lam   i)
lookupMystAny (there-łet-l i) = fold-base alg lookupMystAny (there-łet-l i)
lookupMystAny (there-łet-r i) = fold-base alg lookupMystAny (there-łet-r i)
```

```
normalise : Term → TC Term
```

Instantiation of a generic function

```
alg = lookupAny MystC MystS MystAnyC
unquoteDecl lookupMystAny = defineFold alg …
```

```
lookupMystAny : MystAny P t → Σ A P
lookupMystAny (here p)        = _ , p
lookupMystAny (there-app-l i) = lookupMystAny i
lookupMystAny (there-app-r i) = lookupMystAny i
lookupMystAny (there-lam   i) = lookupMystAny i
lookupMystAny (there-łet-l i) = lookupMystAny i
lookupMystAny (there-łet-r i) = lookupMystAny i
```

So, can we use the generic library now?

So, can we use the generic library now?

Not yet...

Towards a generic library for the practical Agda programmer

- Identify and implement a practically useful range of generic library components
  - In the paper: fold operator & fusion theorem; algebraic ornamentation; All & Any
  - ‘Syntax-generic operations, reflectively reified’ (TyDe 2022)
- Support more datatype features and more liberal function definitions (e.g., case trees)
- Provide interaction/automation to reduce/replace interfacing code
- Test and tweak the macros (difficult to reason about macro correctness)
- Improve the efficiency of the macros and, in general, type checking