Generic Programming With Dependent Types: II

Generic Haskell in Agda

Stephanie Weirich

University of Pennsylvania

March 24–26, 2010 – SSGIP

Generic-Haskell style generic programming in Agda

Dependently-typed languages are expressive enough to *embed* generic-haskell style genericity.

Goals for this part:

1. Foundations of Generic Haskell, in a framework that is easy to explore variations
2. Examples of dependently-typed programming used for metaprogramming, including typeful representations and tagless interpreters.

Challenge problem: Kind-indexed, type-generic functions

Can we make a generic version of these functions?

$$
\begin{aligned}
\text{eq-nat} &: \mathbb{N} \to \mathbb{N} \to \text{Bool} \\
\text{eq-bool} &: \text{Bool} \to \text{Bool} \to \text{Bool} \\
\text{eq-list} &: \forall \{A\} \to (A \to A \to \text{Bool}) \\
&\to \text{List } A \to \text{List } A \to \text{Bool} \\
\text{eq-choice} &: \forall \{A\ B\} \to (A \to A \to \text{Bool}) \\
&\to (B \to B \to \text{Bool}) \\
&\to \text{Choice } A\ B \to \text{Choice } A\ B \to \text{Bool}
\end{aligned}
$$

where

$$
\begin{aligned}
\text{Choice} &: \text{Set} \to \text{Set} \to \text{Set} \\
\text{Choice} &= \lambda\, A\ B \to (A \times B) \uplus A \uplus B \uplus \top
\end{aligned}
$$

Challenge problem: Kind-indexed, type-generic functions

What about these?

$$
\begin{aligned}
\text{size-nat} &: \mathbb{N} \rightarrow \mathbb{N} \\
\text{size-bool} &: \text{Bool} \rightarrow \mathbb{N} \\
\text{size-list} &: \forall \{A\} \rightarrow (A \rightarrow \mathbb{N}) \rightarrow \text{List}\ A \rightarrow \mathbb{N} \\
\text{size-choice} &: \forall \{A\ B\} \rightarrow (A \rightarrow \mathbb{N}) \rightarrow (B \rightarrow \mathbb{N}) \\
&\rightarrow \text{Choice}\ A\ B \rightarrow \mathbb{N}
\end{aligned}
$$

Challenge problem: Kind-indexed, type-generic functions

What about these?

$$
\begin{aligned}
\text{size-nat} &: \mathbb{N} \to \mathbb{N} \\
\text{size-bool} &: \text{Bool} \to \mathbb{N} \\
\text{size-list} &: \forall \{A\} \to (A \to \mathbb{N}) \to \text{List } A \to \mathbb{N} \\
\text{size-choice} &: \forall \{A\ B\} \to (A \to \mathbb{N}) \to (B \to \mathbb{N}) \\
&\quad \to \text{Choice } A\ B \to \mathbb{N}
\end{aligned}
$$

or these

$$
\begin{aligned}
\text{arb-nat} &: \mathbb{N} \\
\text{arb-bool} &: \text{Bool} \\
\text{arb-list} &: \forall \{A\} \to A \to \text{List } A \\
\text{arb-choice} &: \forall \{A\ B\} \to A \to B \to \text{Choice } A\ B
\end{aligned}
$$

Challenge problem: Kind-indexed, type-generic functions

or these

$$
\begin{aligned}
\text{map-list} \; &: \forall \, \{A_1 \; A\} \to (A_1 \to A_2) \\
&\to \text{List } A_1 \to \text{List } A_2 \\
\text{map-choice} \; &: \forall \, \{A_1 \; A_2 \; B_1 \; B_2\} \to (A_1 \to A_2) \to (B_2 \to B_2) \\
&\to \text{Choice } A_1 \; B_1 \to \text{Choice } A_2 \; B_2
\end{aligned}
$$

Recall: "universes" for generic programming

- Start with a "code" for types:

```
data Type : Set where
  nat  : Type
  bool : Type
  pair : Type → Type → Type
```

- Define an "interpretation" as an Agda type

```
⟦_⟧ : Type → Set
⟦ nat ⟧         = ℕ
⟦ bool ⟧        = Bool
⟦ pair t₁ t₂ ⟧  = ⟦ t₁ ⟧ × ⟦ t₂ ⟧
```

- Then define generic op by "interpreting" as Agda function

```
eq : (t : Type) → ⟦ t ⟧ → ⟦ t ⟧ → Bool
eq nat          x        y        = eq-nat x y
eq bool         x        y        = eq-bool x y
eq (pair t₁ t₂) (x₁,x₂) (y₁,y₂)  = eq t₁ x₁ y₁ ∧ eq t₂ x₂ y₂
```

Today's discussion

We'll do the same thing, except for more types.

Today's discussion

We'll do the same thing, except for more types.
By representing all types structurally, we can define functions that are generic in the structure of types.

Today's discussion

We'll do the same thing, except for more types.
By representing all types structurally, we can define functions that are generic in the structure of types.

Generic Haskell Universe

Types are described by the simply-typed lambda calculus, using type constants $\top$, $\uplus$, $\times$ and recursion.

Structural types

Must make recursion explicit in type definitions. Recursive type definitions are a good way to make the Agda type checker diverge. No fun!

```
data μ : (Set → Set) → Set where
  roll : ∀ {A} → A (μ A) → μ A

unroll : ∀ {A} → μ A → A (μ A)
unroll (roll x) = x
```

Structural types

Must make recursion explicit in type definitions. Recursive type definitions are a good way to make the Agda type checker diverge. No fun!

```
data μ : (Set → Set) → Set where
  roll : ∀ {A} → A (μ A) → μ A
unroll : ∀ {A} → μ A → A (μ A)
unroll (roll x) = x
```

Recursive sum-of-product types:

```
Bool   = ⊤ ⊎ ⊤
Maybe  = λ A → ⊤ ⊎ A
Choice = λ A → λ B → (A × B) ⊎ A ⊎ B ⊎ ⊤
ℕ      = μ (λ A → ⊤ ⊎ A)
List   = λ A → μ (λ B → ⊤ ⊎ A × B)
```

Example of structural type definition

Structural definition of lists

$$
\begin{aligned}
&\mathsf{List} &&: \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{List}\ \mathsf{A} &&= \mu\, (\lambda\, \mathsf{B} \to \top \uplus (\mathsf{A} \times \mathsf{B})) \\
&\mathsf{nil} &&: \forall \{\mathsf{A}\} \to \mathsf{List}\ \mathsf{A} \\
&\mathsf{nil} &&= \mathsf{roll}\ (\mathsf{inj}_1\ \mathsf{tt}) \\
&\_{:}\_ &&: \forall \{\mathsf{A}\} \to \mathsf{A} \to \mathsf{List}\ \mathsf{A} \to \mathsf{List}\ \mathsf{A} \\
&\mathsf{x} : \mathsf{xs} &&= \mathsf{roll}\ (\mathsf{inj}_2\ (\mathsf{x},\mathsf{xs}))
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{example\text{-}list} &&: \mathsf{List}\ \mathsf{Bool} \\
&\mathsf{example\text{-}list} &&= \mathsf{true} : \mathsf{false} : \mathsf{nil}
\end{aligned}
$$

Universe is Structure

Generic Haskell Universe

Types are described by the simply-typed lambda calculus, using type constants $\top$, $\uplus$, $\times$ and recursion.

Universe is Structure

Generic Haskell Universe

Types are described by the simply-typed lambda calculus, using type constants $\top$, $\uplus$, $\times$ and recursion.

Plan:

1. Repesent 'universe' as datatype for STLC + constants + recursion.
2. Define interpretation of this universe as Agda types
3. Define type-generic functions as interpretations of this universe as Agda terms with dependent types.

Representing STLC

First, we define datatypes for kinds and type constants:

```
data Kind  :  Set where
  ⋆       :  Kind
  _⇒_     :  Kind → Kind → Kind
```

```
data Const  :  Kind → Set where
  Unit  :  Const ⋆
  Sum   :  Const (⋆ ⇒ ⋆ ⇒ ⋆)
  Prod  :  Const (⋆ ⇒ ⋆ ⇒ ⋆)
```

Note that the constants are indexed by their kinds.

Simply-typed lambda calculus

Represent variables with deBrujin indices.

```
data Ctx : Set where
  []    : Ctx
  _::_  : Kind → Ctx → Ctx
```

Variables are indexed by their kind and context.

```
data V : Kind → Ctx → Set where
  VZ : ∀ {Γ k} → V k (k :: Γ)
  VS : ∀ {Γ k' k} → V k Γ → V k (k' :: Γ)
```

Simply-typed lambda calculus

```
data Typ : Ctx → Kind → Set where
  Var : ∀ {Γ k} → V k Γ → Typ Γ k
  Lam : ∀ {Γ k₁ k₂} → Typ (k₁ :: Γ) k₂
      → Typ Γ (k₁ ⇒ k₂)
  App : ∀ {Γ k₁ k₂} → Typ Γ (k₁ ⇒ k₂) → Typ Γ k₁
      → Typ Γ k₂
  Con : ∀ {Γ k} → Const k → Typ Γ k
  Mu  : ∀ {Γ} → Typ Γ (⋆ ⇒ ⋆) → Typ Γ ⋆
```

Note: closed types type check in the empty environment.

```
Ty : Kind → Set
Ty k = Typ [] k
```

Interpreting kinds and constants

Interpreting kinds and constants

A simple recursive function interprets Kinds as Agda "kinds".

$$
\begin{aligned}
&[\![\_]\!] && : \ \mathsf{Kind} \rightarrow \mathsf{Set} \\
&[\![\star]\!] && = \ \mathsf{Set} \\
&[\![\mathsf{a} \Rightarrow \mathsf{b}]\!] && = \ [\![\mathsf{a}]\!] \rightarrow [\![\mathsf{b}]\!]
\end{aligned}
$$

Interpreting kinds and constants

A simple recursive function interprets Kinds as Agda "kinds".

```
⟦_⟧         : Kind → Set
⟦ ⋆ ⟧       = Set
⟦ a ⇒ b ⟧   = ⟦ a ⟧ → ⟦ b ⟧
```

We need to know the kind of a constructor to know the type of its interpretation.

```
interp-c : ∀ {k} → Const k → ⟦ k ⟧
interp-c Unit  = ⊤       -- has kind Set
interp-c Sum   = _⊎_     -- has kind Set → Set → Set
interp-c Prod  = _×_
```

Interpreting codes as types

Environment stores the interpretation of free variables, indexed by the context.

```
data Env : Ctx → Set where
  []    : Env []
  _::_  : ∀ {k Γ} → ⟦ k ⟧ → Env Γ → Env (k :: Γ)
```

```
sLookup : ∀ {k Γ} → V k Γ → Env Γ → ⟦ k ⟧
sLookup VZ     (v :: Γ) = v
sLookup (VS x) (v :: Γ) = sLookup x Γ
```

Interpreting codes as types

Interpretation of codes is a 'tagless' lambda-calculus interpreter.

```
interp : ∀ {k Γ} → Typ Γ k → Env Γ → ⟦ k ⟧
interp (Var x)      e  =  sLookup x e
interp (Lam t)      e  =  λ y → interp t (y :: e)
interp (App t1 t2) e  =  (interp t1 e) (interp t2 e)
interp (Mu t)       e  =  μ (interp t e)
interp (Con c)      e  =  interp-c c
```

Interpreting codes as types

Interpretation of codes is a 'tagless' lambda-calculus interpreter.

```
interp : ∀ {k Γ} → Typ Γ k → Env Γ → ⟦ k ⟧
interp (Var x)       e  =  sLookup x e
interp (Lam t)       e  =  λ y → interp t (y :: e)
interp (App t1 t2) e  =  (interp t1 e) (interp t2 e)
interp (Mu t)        e  =  μ (interp t e)
interp (Con c)       e  =  interp-c c
```

Special notation for closed types.

```
⌊_⌋ : ∀ {k} → Ty k → ⟦ k ⟧
⌊ t ⌋  =  interp t []
```

Example

Recall the structural type List

$$\begin{aligned} \mathsf{List} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{List} &= \lambda\, \mathsf{A} \to \mu\, (\lambda\, \mathsf{B} \to \top \uplus (\mathsf{A} \times \mathsf{B})) \end{aligned}$$

Example

Recall the structural type List

$$\begin{aligned} \mathsf{List} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{List} &= \lambda\, A \to \mu\, (\lambda\, B \to \top \uplus (A \times B)) \end{aligned}$$

Represent with the following code:

```
list : Ty (⋆ ⇒ ⋆)
list =
  Lam (Mu (Lam
    (App (App (Con Sum) (Con Unit))
    (App (App (Con Prod) (Var (VS VZ))) (Var VZ)))))
```

The Agda type checker can see that $\lfloor$ list $\rfloor$ normalizes to List, so it considers these two types equal.

Kind-indexed types

The kind of a type determines the type of a generic function.

Kind-indexed types

The kind of a type determines the type of a generic function.

$$
\begin{aligned}
&\_\langle\_\rangle\_ \;:\; (\mathsf{Set} \to \mathsf{Set}) \to (k \;:\; \mathsf{Kind}) \to [\![\, k \,]\!] \to \mathsf{Set} \\
&b \,\langle\, \star \,\rangle\, t \;=\; b\; t \\
&b \,\langle\, k1 \Rightarrow k2 \,\rangle\, t \;=\; \forall\, \{A\} \to b \,\langle\, k1 \,\rangle\, A \to b \,\langle\, k2 \,\rangle\, (t\; A)
\end{aligned}
$$

Kind-indexed types

The kind of a type determines the type of a generic function.

$$
\begin{aligned}
&\_\langle\_\rangle\_ \;:\; (\mathsf{Set} \to \mathsf{Set}) \to (k \;:\; \mathsf{Kind}) \to [\![\, k \,]\!] \to \mathsf{Set} \\
&b \,\langle\, \star \,\rangle\, t \;=\; b\; t \\
&b \,\langle\, k1 \Rightarrow k2 \,\rangle\, t \;=\; \forall\, \{A\} \to b \,\langle\, k1 \,\rangle\, A \to b \,\langle\, k2 \,\rangle\, (t\; A)
\end{aligned}
$$

Equality example

```
Eq : Set → Set
Eq A = A → A → Bool

eq-bool   : Eq ⟨ ⋆ ⟩ Bool
  -- Bool → Bool → Bool
eq-list   : Eq ⟨ ⋆ ⇒ ⋆ ⟩ List
  -- ∀ A → (A → A → Bool) → (List A → List A → Bool)
eq-choice : Eq ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ Choice
  -- ∀ A B → (A → A → Bool) → (B → B → Bool)
  -- → (Choice A B → Choice A B → Bool)
```

Defining generic functions

A generic function is an interpretation of the Typ universe as an Agda term with a kind-indexed type.

Generic equality

$$\mathsf{geq} : \forall \{k\} \rightarrow (t : \mathsf{Ty}\ k) \rightarrow \mathsf{Eq} \langle\, k \,\rangle \lfloor\, t \,\rfloor$$

Defining generic functions

A generic function is an interpretation of the Typ universe as an Agda term with a kind-indexed type.

Generic equality

$$\mathsf{geq} \ : \ \forall \, \{k\} \rightarrow (t \ : \ \mathsf{Ty}\ k) \rightarrow \mathsf{Eq}\ \langle\ k\ \rangle\ \lfloor\ t\ \rfloor$$

... however, because of λ, must generalize to types with free variables.

Variables

Variables are interpreted with an environment.

```
data VarEnv (b : Set → Set) : Ctx → Set where
  []    : VarEnv b []
  _::_  : {k : Kind} {Γ : Ctx} {a : ⟦ k ⟧}
          → b ⟨ k ⟩ a
          → VarEnv b Γ
          → VarEnv b (k :: Γ)
```

What is the type of the lookup function?

```
vLookup : ∀ {Γ k} {b : Set → Set}
  → (v : V k Γ) → (ve : VarEnv b Γ)
  → b ⟨ k ⟩ ?
vLookup VZ      (v :: ve)  =  v
vLookup (VS x) (v :: ve)  =  vLookup x ve
```

Variables

Variables are interpreted with an environment.

```
data VarEnv (b : Set → Set) : Ctx → Set where
  []   : VarEnv b []
  _::_ : {k : Kind} {Γ : Ctx} {a : ⟦ k ⟧}
         → b ⟨ k ⟩ a
         → VarEnv b Γ
         → VarEnv b (k :: Γ)
```

What is the type of the lookup function?

```
vLookup : ∀ {Γ k} {b : Set → Set}
  → (v : V k Γ) → (ve : VarEnv b Γ)
  → b ⟨ k ⟩ (sLookup v (toEnv ve))
vLookup VZ     (v :: ve) = v
vLookup (VS x) (v :: ve) = vLookup x ve
```

Aux function toEnv converts a VarEnv to an Env.

Another interpreter

```
Eq : Set → Set
Eq A  =  A → A → Bool
```

```
geq-open : {Γ : Ctx} {k : Kind}
  → (ve : VarEnv Eq Γ)
  → (t : Typ Γ k) → Eq ⟨ k ⟩ (interp t (toEnv ve))
geq-open ve (Var v)       =  vLookup v ve
geq-open ve (Lam t)       =  λ y → geq-open (y :: ve) t
geq-open ve (App t1 t2)   =  (geq-open ve t1) (geq-open ve t2)
geq-open ve (Mu t)        =
  λ x y → geq-open ve (App t (Mu t)) (unroll x) (unroll y)
geq-open ve (Con c)       =  geq-c c
```

Interpretation of constants

```
geq-sum  : ∀ {A} → (A → A → Bool)
           → ∀ {B} → (B → B → Bool)
           → (A ⊎ B) → (A ⊎ B) → Bool
geq-sum ra rb (inj₁ x₁) (inj₁ x₂)  =  ra x₁ x₂
geq-sum ra rb (inj₂ x₁) (inj₂ x₂)  =  rb x₁ x₂
geq-sum _ _ _ _  =  false

geq-prod  : ∀ {A} → (A → A → Bool)
            → ∀ {B} → (B → B → Bool)
            → (A × B) → (A × B) → Bool
geq-prod ra rb (x₁,x₂) (y₁,y₂)  =  ra x₁ y₁ ∧ rb x₂ y₂


geq-c : {k : Kind} → (c : Const k) → Eq ⟨ k ⟩ ⌊ Con c ⌋
geq-c Unit  =  λ t1 t2 → true
geq-c Sum   =  geq-sum
geq-c Prod  =  geq-prod
```

Constants

Only the interpretation of constants and the rolling/unrolling in the Mu case changes with each generic function.

Constants

Only the interpretation of constants and the rolling/unrolling in the Mu case changes with each generic function.

Interpretation of constants

$$\mathsf{ConstEnv} \;:\; (\mathsf{Set} \rightarrow \mathsf{Set}) \rightarrow \mathsf{Set}$$
$$\mathsf{ConstEnv}\; b \;=\; \forall \{k\} \rightarrow (c \;:\; \mathsf{Const}\; k) \rightarrow b \,\langle\, k \,\rangle \,\lfloor\, \mathsf{Con}\; c \,\rfloor$$

Constants

Only the interpretation of constants and the rolling/unrolling in the Mu case changes with each generic function.

Interpretation of constants

$$\mathsf{ConstEnv} : (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set}$$
$$\mathsf{ConstEnv}\; b = \forall \{k\} \to (c : \mathsf{Const}\; k) \to b \langle k \rangle \lfloor \mathsf{Con}\; c \rfloor$$

Conversion for Mu case

$$\mathsf{MuGen} : (\mathsf{Set} \to \mathsf{Set}) \to \mathsf{Set}$$
$$\mathsf{MuGen}\; b = \forall \{A\} \to b\,(A\,(\mu\, A)) \to b\,(\mu\, A)$$

Generic polytypic interpreter

```
gen-open : {b : Set → Set} {Γ : Ctx} {k : Kind}
   → ConstEnv b → (ve : VarEnv b Γ) → MuGen b
   → (t : Typ Γ k) → b ⟨ k ⟩ (interp t (toEnv ve))
gen-open ce ve d (Var v)       = vLookup v ve
gen-open ce ve d (Lam t)       = λ y → gen-open ce (y :: ve) d t
gen-open ce ve d (App t1 t2)   =
   (gen-open ce ve d t1) (gen-open ce ve d t2)
gen-open ce ve d (Con c)       = ce c
gen-open ce ve d (Mu t)        =
   d (gen-open ce ve d (App t (Mu t)))
```

Generic polytypic interpreter

```
gen-open : {b : Set → Set} {Γ : Ctx} {k : Kind}
  → ConstEnv b → (ve : VarEnv b Γ) → MuGen b
  → (t : Typ Γ k) → b ⟨ k ⟩ (interp t (toEnv ve))
gen-open ce ve d (Var v)      = vLookup v ve
gen-open ce ve d (Lam t)      = λ y → gen-open ce (y :: ve) d t
gen-open ce ve d (App t1 t2)  =
  (gen-open ce ve d t1) (gen-open ce ve d t2)
gen-open ce ve d (Con c)      = ce c
gen-open ce ve d (Mu t)       =
  d (gen-open ce ve d (App t (Mu t)))
```

Specialized to closed types

```
gen : {b : Set → Set} {k : Kind} → ConstEnv b → MuGen b
  → (t : Ty k) → b ⟨ k ⟩ ⌊ t ⌋
gen c b t = gen-open c [] b t
```

Equality example

```
geq : {k : Kind} → (t : Ty k) → Eq ⟨ k ⟩ ⌊ t ⌋
geq  =  gen geq-c eb where
   eb  : ∀ {A} → Eq (A (μ A)) → Eq (μ A)
   eb f  =  λ x y → f (unroll x) (unroll y)
```

```
eq-list  : List Nat → List Nat → Bool
eq-list  =  geq (App list nat)
```

Count example

```
Count : Set → Set
Count A = A → ℕ

gcount : {k : Kind} → (t : Ty k) → Count ⟨ k ⟩ ⌊ t ⌋
gcount = gen ee eb where
  ee : ConstEnv Count
  ee Unit = λ t → 0
  ee Sum = g where
    g : ∀ {A} → _ → ∀ {B} → _ → (A ⊎ B) → ℕ
    g ra rb (inj₁ x) = ra x
    g ra rb (inj₂ x) = rb x
  ee Prod = g where
    g : ∀ {A} → _ → ∀ {B} → _ → (A × B) → ℕ
    g ra rb (x₁,x₂) = ra x₁ + rb x₂
  eb : MuGen Count
  eb f = λ x → f (unroll x)
```

Count example

Count shows why it is important to make the type parameters explicit in the representation.

$$\mathsf{gsize} : (\mathsf{t} : \mathsf{Ty}\,(\star \Rightarrow \star)) \rightarrow \forall\,\{\mathsf{A}\} \rightarrow \lfloor\,\mathsf{t}\,\rfloor\,\mathsf{A} \rightarrow \mathbb{N}$$
$$\mathsf{gsize}\;\mathsf{t} = \mathsf{gcount}\;\mathsf{t}\;(\lambda\,\mathsf{x} \rightarrow 1)$$

$$\mathsf{gsum} : (\mathsf{t} : \mathsf{Ty}\,(\star \Rightarrow \star)) \rightarrow \lfloor\,\mathsf{t}\,\rfloor\,\mathbb{N} \rightarrow \mathbb{N}$$
$$\mathsf{gsum}\;\mathsf{t} = \mathsf{gcount}\;\mathsf{t}\;(\lambda\,\mathsf{x} \rightarrow \mathsf{x})$$

Count example

Count shows why it is important to make the type parameters explicit in the representation.

$$\mathsf{gsize} : (\mathsf{t} : \mathsf{Ty}\ (\star \Rightarrow \star)) \rightarrow \forall\ \{A\} \rightarrow \lfloor\ \mathsf{t}\ \rfloor\ A \rightarrow \mathbb{N}$$
$$\mathsf{gsize}\ \mathsf{t} = \mathsf{gcount}\ \mathsf{t}\ (\lambda\ x \rightarrow 1)$$

$$\mathsf{gsum} : (\mathsf{t} : \mathsf{Ty}\ (\star \Rightarrow \star)) \rightarrow \lfloor\ \mathsf{t}\ \rfloor\ \mathbb{N} \rightarrow \mathbb{N}$$
$$\mathsf{gsum}\ \mathsf{t} = \mathsf{gcount}\ \mathsf{t}\ (\lambda\ x \rightarrow x)$$

$$\mathsf{exlist2} : \mathsf{List}\ \mathbb{N}$$
$$\mathsf{exlist2} = 1 : 2 : 3 : \mathsf{nil}$$

$$\mathsf{gsize}\ \mathsf{list}\ \mathsf{exlist2} \equiv 3$$
$$\mathsf{gsum}\ \mathsf{list}\ \mathsf{exlist2} \equiv 6$$

What about map?

$$
\begin{aligned}
\text{map-list} &: \forall \{A_1\ A_2\} \to (A_1 \to A_2) \\
&\quad \to \text{List}\ A_1 \to \text{List}\ A_2 \\
\text{map-maybe} &: \forall \{A_1\ A_2\} \to (A_1 \to A_2) \\
&\quad \to \text{Maybe}\ A_1 \to \text{Maybe}\ A_2 \\
\text{map-choice} &: \forall \{A_1\ A_2\ B_1\ B_2\} \to (A_1 \to A_2) \to (B_2 \to B_2) \\
&\quad \to \text{Choice}\ A_1\ B_1 \to \text{Choice}\ A_2\ B_2
\end{aligned}
$$

Want something like:

$$
\begin{aligned}
&\text{Map}\ \langle \star \rangle\ T\ =\ T \to T \\
&\text{Map}\ \langle \star \Rightarrow \star \rangle\ T\ =\ \forall \{A\ B\} \to (A \to B) \to (T\ A \to T\ B) \\
&\text{Map}\ \langle \star \Rightarrow \star \Rightarrow \star \rangle\ T\ =\ \forall \{A_1\ B_1\ A_2\ B_2\} \\
&\quad \to (A_1 \to B_1) \to (A_2 \to B_2) \to (T\ A_1\ A_2 \to T\ B_1\ B_2)
\end{aligned}
$$

Can't define Map as a kind-indexed type.

Arities in Kind-indexed types

Solution is an 'arity-2' kind-indexed type:

$$\_\langle\_\rangle_2 \;:\; (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}) \rightarrow (\mathsf{k} \;:\; \mathsf{Kind}) \rightarrow [\![\, \mathsf{k} \,]\!] \rightarrow [\![\, \mathsf{k} \,]\!] \rightarrow \mathsf{Set}$$
$$\mathsf{b} \,\langle\, \star \,\rangle_2 \;=\; \lambda\, \mathsf{t}_1\ \mathsf{t}_2 \rightarrow \mathsf{b}\ \mathsf{t}_1\ \mathsf{t}_2$$
$$\mathsf{b} \,\langle\, \mathsf{k}_1 \Rightarrow \mathsf{k}_2 \,\rangle_2 \;=\; \lambda\, \mathsf{t}_1\ \mathsf{t}_2 \rightarrow \forall\, \{\mathsf{a}_1\ \mathsf{a}_2\} \rightarrow$$
$$\quad (\mathsf{b} \,\langle\, \mathsf{k}_1 \,\rangle_2)\ \mathsf{a}_1\ \mathsf{a}_2 \rightarrow (\mathsf{b} \,\langle\, \mathsf{k}_2 \,\rangle_2)\ (\mathsf{t}_1\ \mathsf{a}_1)\ (\mathsf{t}_2\ \mathsf{a}_2)$$

$$\mathsf{Map} \;:\; \mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}$$
$$\mathsf{Map}\ \mathsf{A}\ \mathsf{B} \;=\; \mathsf{A} \rightarrow \mathsf{B}$$

$$\mathsf{gmap} \;:\; \forall\, \{\mathsf{k}\} \rightarrow (\mathsf{t} \;:\; \mathsf{Ty}\ \mathsf{k}) \rightarrow \mathsf{Map} \,\langle\, \mathsf{k} \,\rangle_2 \,\lfloor\, \mathsf{t} \,\rfloor\, \lfloor\, \mathsf{t} \,\rfloor$$

Arities in Kind-indexed types

Solution is an 'arity-2' kind-indexed type:

$$\_\langle\_\rangle_2 \;:\; (\mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}) \rightarrow (k \;:\; \mathsf{Kind}) \rightarrow [\![\, k \,]\!] \rightarrow [\![\, k \,]\!] \rightarrow \mathsf{Set}$$
$$b \,\langle \star \rangle_2 \;=\; \lambda\, t_1\; t_2 \rightarrow b\; t_1\; t_2$$
$$b \,\langle\, k_1 \Rightarrow k_2 \,\rangle_2 \;=\; \lambda\, t_1\; t_2 \rightarrow \forall\, \{a_1\; a_2\} \rightarrow$$
$$\quad (b \,\langle\, k_1 \,\rangle_2)\; a_1\; a_2 \rightarrow (b \,\langle\, k_2 \,\rangle_2)\; (t_1\; a_1)\; (t_2\; a_2)$$

$$\mathsf{Map} \;:\; \mathsf{Set} \rightarrow \mathsf{Set} \rightarrow \mathsf{Set}$$
$$\mathsf{Map}\; A\; B \;=\; A \rightarrow B$$

$$\mathsf{gmap} \;:\; \forall\, \{k\} \rightarrow (t \;:\; \mathsf{Ty}\; k) \rightarrow \mathsf{Map} \,\langle\, k \,\rangle_2 \,\lfloor\, t \,\rfloor\, \lfloor\, t \,\rfloor$$

To make a general framework, need to define $\mathsf{ConstEnv}_2$, VarEnv_2, $\mathsf{gen\text{-}open}_2$, gen_2, etc.

Or, arbitrary-arity kind indexed type

$$
\begin{aligned}
&\_\langle\_\rangle\_ \;:\; \{n : \mathbb{N}\} \rightarrow (\mathsf{Vec\ Set}\ n \rightarrow \mathsf{Set}) \rightarrow (k : \mathsf{Kind}) \\
&\quad \rightarrow \mathsf{Vec}\ [\![\ k\ ]\!]\ n \rightarrow \mathsf{Set} \\
&b\ \langle \star \rangle\ v \qquad\quad = \ b\ v \\
&b\ \langle\ k1 \Rightarrow k2\ \rangle\ v \ = \ \{a : \mathsf{Vec}\ [\![\ k1\ ]\!]\ \_\} \rightarrow \\
&\quad b\ \langle\ k1\ \rangle\ a \rightarrow b\ \langle\ k2\ \rangle\ (v \circledast a)
\end{aligned}
$$

(Recall: $v \circledast a$ applies vector of functions to vector of arguments pointwise.)

Or, arbitrary-arity kind indexed type

$$
\begin{aligned}
&\_\langle\_\rangle\_ \ : \ \{n \ : \ \mathbb{N}\} \to (\text{Vec Set } n \to \text{Set}) \to (k \ : \ \text{Kind}) \\
&\quad \to \text{Vec } [\![\, k \,]\!] \ n \to \text{Set} \\
&b \ \langle \, \star \, \rangle \ v \qquad\qquad = \ b \ v \\
&b \ \langle \, k1 \Rightarrow k2 \, \rangle \ v \ = \ \{a \ : \ \text{Vec } [\![\, k1 \,]\!] \ \_\} \to \\
&\quad b \ \langle \, k1 \, \rangle \ a \to b \ \langle \, k2 \, \rangle \ (v \circledast a)
\end{aligned}
$$

(Recall: $v \circledast a$ applies vector of functions to vector of arguments pointwise.)

Single framework for all arities of generic functions.

Or, arbitrary-arity kind indexed type

$$
\begin{aligned}
&\_\langle\_\rangle\_ \ : \ \{n \ : \ \mathbb{N}\} \rightarrow (\mathsf{Vec\ Set}\ n \rightarrow \mathsf{Set}) \rightarrow (k \ : \ \mathsf{Kind}) \\
&\quad \rightarrow \mathsf{Vec}\ [\![\ k\ ]\!]\ n \rightarrow \mathsf{Set} \\
&b\ \langle\ \star\ \rangle\ v \qquad\qquad = \ b\ v \\
&b\ \langle\ k1 \Rightarrow k2\ \rangle\ v \ = \ \{a \ : \ \mathsf{Vec}\ [\![\ k1\ ]\!]\ \_\} \rightarrow \\
&\quad b\ \langle\ k1\ \rangle\ a \rightarrow b\ \langle\ k2\ \rangle\ (v \circledast a)
\end{aligned}
$$

(Recall: $v \circledast a$ applies vector of functions to vector of arguments pointwise.)

Single framework for all arities of generic functions. Also allows arity-generic, type-generic code. More next time.

Discussion and variations

- Problem: What about datatypes like List and Maybe?

Discussion and variations

- Problem: What about datatypes like List and Maybe?
- Can extend this solution to work with built-in datatypes with *datatype isomorphisms*.

Discussion and variations

- Problem: What about datatypes like List and Maybe?
- Can extend this solution to work with built-in datatypes with *datatype isomorphisms*.
- Problem: What about indexed datatypes like Vec?

Discussion and variations

- Problem: What about datatypes like List and Maybe?
- Can extend this solution to work with built-in datatypes with *datatype isomorphisms*.
- Problem: What about indexed datatypes like Vec?
- Structural types that depend on indices.

```
Vec' : ℕ → Set → Set
Vec' zero    = λ A → ⊤
Vec' (suc n) = λ A → A × (Vec' n A)
```

Discussion and variations

- Problem: What about datatypes like List and Maybe?
- Can extend this solution to work with built-in datatypes with *datatype isomorphisms*.
- Problem: What about indexed datatypes like Vec?
- Structural types that depend on indices.

```
Vec' : ℕ → Set → Set
Vec' zero    = λ A → ⊤
Vec' (suc n) = λ A → A × (Vec' n A)
```

- Problem: The types of some generic functions depend on what they are instantiated with.

Discussion and variations

- Problem: What about datatypes like List and Maybe?
- Can extend this solution to work with built-in datatypes with *datatype isomorphisms*.
- Problem: What about indexed datatypes like Vec?
- Structural types that depend on indices.

$$
\begin{aligned}
&\text{Vec'} : \mathbb{N} \to \text{Set} \to \text{Set} \\
&\text{Vec'}\ \text{zero} = \lambda\, A \to \top \\
&\text{Vec'}\ (\text{suc}\ n) = \lambda\, A \to A \times (\text{Vec'}\ n\ A)
\end{aligned}
$$

- Problem: The types of some generic functions depend on what they are instantiated with.
- Change the type of b from $\text{Set} \to \text{Set}$ to $\text{Ty} \star \to \text{Set}$.

Discussion and variations

- Problem: Not all generic functions make sense for all type constructors.

Discussion and variations

- Problem: Not all generic functions make sense for all type constructors.
- Can generalize over the constants to allow generic functions to pick the constants that they are valid for.

Discussion and variations

- Problem: Not all generic functions make sense for all type constructors.
- Can generalize over the constants to allow generic functions to pick the constants that they are valid for.
- Problem: what about Agda's termination checker?

Discussion and variations

- Problem: Not all generic functions make sense for all type constructors.
- Can generalize over the constants to allow generic functions to pick the constants that they are valid for.
- Problem: what about Agda's termination checker?
- Can use coinduction for recursive types, partiality monad for potentially divergent function(?)

Discussion and variations

- Problem: Not all generic functions make sense for all type constructors.
- Can generalize over the constants to allow generic functions to pick the constants that they are valid for.
- Problem: what about Agda's termination checker?
- Can use coinduction for recursive types, partiality monad for potentially divergent function(?)
- Problem: No easy way to use generic functions. Have to explicitly supply the universe.

Discussion and variations

- Problem: Not all generic functions make sense for all type constructors.
- Can generalize over the constants to allow generic functions to pick the constants that they are valid for.
- Problem: what about Agda's termination checker?
- Can use coinduction for recursive types, partiality monad for potentially divergent function(?)
- Problem: No easy way to use generic functions. Have to explicitly supply the universe.
- ???

References

1. Ralf Hinze. Polytypic values possess polykinded types. *Science of Computer Programming*, 43(2–3):129–159, 2002. MPC Special Issue
2. Thorsten Altenkirch and Conor McBride. Generic programming within dependently typed programming. In *Proceedings of the IFIP TC2 Working Conference on Generic Programming*, Dagstuhl, Germany, July 2003
3. Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. Polytypic programming in Coq. In *WGP '08: Proceedings of the ACM SIGPLAN Workshop on Generic Programming*, pages 49–60, New York, NY, USA, 2008. ACM