

Arity-Generic Datatype-Generic Programming

Chris Casinghino
Stephanie Weirich

University of Pennsylvania

January 19, 2010 – PLPV, Madrid



Outline

- 1** Motivation
- 2 Arities in Datatype-Generic Programming
- 3 A Universe for Generic Programming
- 4 The Framework
- 5 Arity-Generic Map
- 6 Conclusion

Datatype-Genericity

Datatype-generic functions can be implemented for many datatypes.

`list-map` : $(a \rightarrow b) \rightarrow [a] \rightarrow [b]$

`maybe-map` : $(a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b$

`pair-map` : $(a_1 \rightarrow a_2) \rightarrow (b_1 \rightarrow b_2) \rightarrow (a_1, b_1) \rightarrow (a_2, b_2)$

Arity-Genericity

Arity-generic functions can be implemented at many arities.

At higher arities, `map` is often called `zip`. The Haskell standard library includes:

```
repeat    : a                → [a]
map       : (a → b)          → [a] → [b]
zipWith   : (a → b → c)      → [a] → [b] → [c]
zipWith3  : (a → b → c → d) → [a] → [b] → [c] → [d]
```

Doubly Generic Functions

- Some functions, like `map`, are **doubly generic**.

`pair-map1` : $a_1 \rightarrow a_2 \rightarrow (a_1, a_2)$

`maybe-map2` : $(a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b$

`list-map3` : $(a \rightarrow b \rightarrow c) \rightarrow [a] \rightarrow [b] \rightarrow [c]$

...

- Goal: one function, `ngmap`, which produces any of these definitions from an arity and a type. For example:

`ngmap 2 Maybe` : $(a \rightarrow b) \rightarrow \text{Maybe } a \rightarrow \text{Maybe } b$

- **Importantly**, the notion of arity already appears in datatype-generic programming (but not arity-genericity).

Motivation

Why study doubly generic functions?

- They can reduce boilerplate.
- We may be able to prove properties generally for all instances of doubly generic functions.
- New insight into the nature of operations like `map`.
- Formally specifying a framework for them helps us find more.

Outline

- 1 Motivation
- 2 Arities in Datatype-Generic Programming**
- 3 A Universe for Generic Programming
- 4 The Framework
- 5 Arity-Generic Map
- 6 Conclusion

Different Types at Different Kinds

- Consider equality functions for various types:

$\text{nat-eq} : \mathbb{N} \rightarrow \mathbb{N} \rightarrow \text{Bool}$

$\text{list-eq} : \forall a . (a \rightarrow a \rightarrow \text{Bool}) \rightarrow [a] \rightarrow [a] \rightarrow \text{Bool}$

$\text{pair-eq} : \forall a b . (a \rightarrow a \rightarrow \text{Bool}) \rightarrow (b \rightarrow b \rightarrow \text{Bool})$
 $\rightarrow (a, b) \rightarrow (a, b) \rightarrow \text{Bool}$

- Generic Haskell style datatype-generic programs have **kind-indexed types**.

$\text{Eq} \langle \star \rangle \quad t = t \rightarrow t \rightarrow \text{Bool}$

$\text{Eq} \langle k_1 \Rightarrow k_2 \rangle t = \forall a . \text{Eq} \langle k_1 \rangle a \rightarrow \text{Eq} \langle k_2 \rangle (t a)$

- The kind-indexed type `Eq` has arity one, because it takes one type argument.

Kind-indexed Types for Maps

- On the other hand, datatype-generic `map` has a kind-indexed type of arity two.

$$\begin{aligned}\text{Map } \langle \star \rangle \quad t_1 \ t_2 &= t_1 \rightarrow t_2 \\ \text{Map } \langle k_1 \Rightarrow k_2 \rangle \ t_1 \ t_2 &= \forall a \ b . \text{Map } \langle k_1 \rangle \ a \ b \\ &\quad \rightarrow \text{Map } \langle k_2 \rangle \ (t_1 \ a) \ (t_2 \ b)\end{aligned}$$

- Datatype-generic `zipWith` has a kind-indexed type of arity three.

$$\begin{aligned}\text{Zip } \langle \star \rangle \quad t_1 \ t_2 \ t_3 &= t_1 \rightarrow t_2 \rightarrow t_3 \\ \text{Zip } \langle k_1 \Rightarrow k_2 \rangle \ t_1 \ t_2 \ t_3 &= \forall a \ b \ c . \text{Zip } \langle k_1 \rangle \ a \ b \ c \\ &\quad \rightarrow \text{Zip } \langle k_2 \rangle \ (t_1 \ a) \ (t_2 \ b) \ (t_3 \ c)\end{aligned}$$

- These two look very similar...

Summary

- Arities already show up in standard kind-indexed types.
- Some frameworks for datatype-generic programming support different arities (but not generalizing over the arity).
- New observation: when written in dependently-typed languages, they can support arity-genericity too.

Outline

- 1 Motivation
- 2 Arities in Datatype-Generic Programming
- 3 A Universe for Generic Programming**
- 4 The Framework
- 5 Arity-Generic Map
- 6 Conclusion

Universes

- In dependently typed languages (like Agda), generic programming frameworks are implemented using **universes**.
- A universe is:
 - A data structure to represent part of the type language
 - And an **interpretation function** to convert members of the datatype to actual Agda types.
- A generic program is defined by recursion over the universe.
- Our universe is based on the type language of F^ω —it's essentially STLC.

Kinds

- We define a data type to represent kinds, and a function to interpret its members (the **codes**) as Agda kinds.

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

```
[[_]]      : Kind → Set  
[[★]]     = Set  
[[a ⇒ b]] = [[a]] → [[b]]
```

- **Set** is the classifier of Agda types. We'll gloss over some details of the type hierarchy.

Type Constants

Type constants are indexed by their kinds.

data Const : Kind \rightarrow Set **where**

Unit : Const \star

Nat : Const \star

Sum : Const ($\star \Rightarrow \star \Rightarrow \star$)

Prod : Const ($\star \Rightarrow \star \Rightarrow \star$)

interp-c : $\forall \{k\} \rightarrow$ Const k \rightarrow $\llbracket k \rrbracket$

interp-c Unit = \top

interp-c Nat = \mathbb{N}

interp-c Sum = $_ \uplus _$

interp-c Prod = $_ \times _$

Types

- Types comprise variables, functions, applications, and constants.

data Ty : Ctx → Kind → Set **where**

Var : ...

Lam : ...

App : ...

Con : ...

- We have a function for interpreting closed Types:

$$[_] : \forall \{k\} \rightarrow \text{Ty } [] \ k \rightarrow \llbracket k \rrbracket$$

Example

- As an example, we might define an `Option` type in Agda, isomorphic to the standard datatype:

```
Option : Set → Set
Option = λ A → ⊤ ⊔ A
```

- We can represent it with a code:

```
option : Ty [] (★ ⇒ ★)
option =
  Lam (App (App (Con Sum) (Con Unit))
          (Var VZ))
```

- Agda can see that `[option]` normalizes to `Option`.

Outline

- 1 Motivation
- 2 Arities in Datatype-Generic Programming
- 3 A Universe for Generic Programming
- 4 The Framework**
- 5 Arity-Generic Map
- 6 Conclusion

Preliminaries

- We need just two more things.
- Agda's library has length indexed vectors:

```
data Vec (A : Set) : ℕ → Set where  
  []      : Vec A zero  
  _::__   : ∀ {n} (a : A) (xs : Vec A n) → Vec A (suc n)
```

- We define $_ \circledast _$, pronounced “zap”, a zipping application on vectors:

```
 $\_ \circledast \_ : \{A B : Set\} \{n : \mathbb{N}\}$   
 $\rightarrow Vec (A \rightarrow B) n \rightarrow Vec A n \rightarrow Vec B n$ 
```

- And `repeat`, which creates `n` copies of a term:

```
repeat : {A : Set} → (n : ℕ) → A → Vec A n
```

Kind-indexed Types revisited

- Observe that the kind-indexed types we saw before have a very regular form:

$$\begin{aligned}\text{Eq } \langle \star \rangle \quad t &= t \rightarrow t \rightarrow \text{Bool} \\ \text{Eq } \langle k_1 \Rightarrow k_2 \rangle t &= \forall a . \text{Eq } \langle k_1 \rangle a \rightarrow \text{Eq } \langle k_2 \rangle (t a)\end{aligned}$$

$$\begin{aligned}\text{Map } \langle \star \rangle \quad t_1 t_2 &= t_1 \rightarrow t_2 \\ \text{Map } \langle k_1 \Rightarrow k_2 \rangle t_1 t_2 &= \forall a b . \text{Map } \langle k_1 \rangle a b \\ &\quad \rightarrow \text{Map } \langle k_2 \rangle (t_1 a) (t_2 b)\end{aligned}$$

- At kind \star , they use the type argument(s) in an operation-specific way.
- At arrow kinds they are “logical” and completely regular.

Kind-indexed Types, formally

- So, we can derive instances of kind-indexed types automatically if provided with the definition at base kinds:

$$\begin{aligned} _ \langle _ \rangle _ &: \{n : \mathbb{N}\} \rightarrow (b : \text{Vec Set } n \rightarrow \text{Set}) \\ &\rightarrow (k : \text{Kind}) \rightarrow \text{Vec } \llbracket k \rrbracket n \\ &\rightarrow \text{Set} \\ b \langle \star \rangle v &= b v \\ b \langle k_1 \Rightarrow k_2 \rangle v &= \\ (a : \text{Vec } \llbracket k_1 \rrbracket _) &\rightarrow b \langle k_1 \rangle a \rightarrow b \langle k_2 \rangle (v \circledast a) \end{aligned}$$

- Since the number of type arguments is variable, we take them as a vector.
- In the development, this definition is then curried so users can supply the types individually.

Example: GMap

- Datatype-generic `map` has arity two. Its base type is:

$$\begin{aligned} \text{GMap} &: (\mathbf{v} : \text{Vec Set } 2) \rightarrow \text{Set} \\ \text{GMap } (A :: B :: []) &= A \rightarrow B \end{aligned}$$

- `GMap <★ ⇒ ★>` (`repeat 2 List`) normalizes to:

$$\forall \{A B : \text{Set}\} \rightarrow (A \rightarrow B) \rightarrow \text{List } A \rightarrow \text{List } B$$

- `GMap <★ ⇒ ★ ⇒ ★>` (`repeat 2 _ × _`) normalizes to:

$$\begin{aligned} \forall \{A_1 B_1 : \text{Set}\} &\rightarrow (A_1 \rightarrow B_1) \\ &\rightarrow \{A_2 B_2 : \text{Set}\} \rightarrow (A_2 \rightarrow B_2) \\ &\rightarrow (A_1 \times A_2) \rightarrow (B_1 \times B_2) \end{aligned}$$

The Framework

- Users implement the operation for type constants by constructing a `TyConstEnv b`

$$\begin{aligned} \text{TyConstEnv } b &= \{k : \text{Kind}\} (c : \text{Const } k) \\ &\rightarrow b \langle k \rangle (\text{repeat } _ \lfloor \text{Con } c \rfloor) \end{aligned}$$

- Then they call `ngen`:

$$\begin{aligned} \text{ngen} &: \{n : \mathbb{N}\} \{b : \text{Vec Set } n \rightarrow \text{Set}\} \{k : \text{Kind}\} \\ &\rightarrow (\text{TyConstEnv } b) \\ &\rightarrow (t : \text{Ty } [] k) \\ &\rightarrow b \langle k \rangle (\text{repeat } n \lfloor t \rfloor) \end{aligned}$$

Map's TyConstEnv

- We've already defined datatype-generic map's type.
- Now, its TyConstEnv.

```
gmap-const : TyConstEnv GMap
gmap-const Nat  =  $\lambda x \rightarrow x$ 
gmap-const Unit =  $\lambda x \rightarrow x$ 
gmap-const Prod =  $\lambda f1 f2 x \rightarrow (f1 (\text{proj}_1 x), f2 (\text{proj}_2 x))$ 
gmap-const Sum  = g
  where g : {A1 B1 A2 B2 : Set}
          → (A1 → B1) → (A2 → B2)
          → A1  $\uplus$  A2 → B1  $\uplus$  B2
    g    fa fb (inj1 xa) = inj1 (fa xa)
    g    fa fb (inj2 xb) = inj2 (fb xb)
```

Datatype-Generic Map

- With `gmap-const`, we can define datatype-generic `map`:

```
gmap : {k : Kind} → (t : Ty [] k)
      → GMap ⟨ k ⟩ ([ t ] :: [ t ] :: [])
gmap t = ngen gmap-const t
```

- For example:

```
option-map : {A B : Set} → (A → B)
            → Option A → Option B
option-map = gmap option
```

- Using generic functions at actual Agda datatypes is discussed in the paper.

Outline

- 1 Motivation
- 2 Arities in Datatype-Generic Programming
- 3 A Universe for Generic Programming
- 4 The Framework
- 5 Arity-Generic Map**
- 6 Conclusion

Writing Arity-Generic Programs

- All the pieces of the framework are parameterized by arity:

$$\begin{aligned} _ \langle _ \rangle _ &: \{n : \mathbb{N}\} \rightarrow (b : \text{Vec Set } n \rightarrow \text{Set}) \\ &\rightarrow (k : \text{Kind}) \rightarrow \text{Vec } \llbracket k \rrbracket n \\ &\rightarrow \text{Set} \end{aligned}$$
$$\begin{aligned} \text{TyConstEnv} &: \{n : \mathbb{N}\} \rightarrow (b : \text{Vec Set } n \rightarrow \text{Set}) \\ &\rightarrow \text{Set} \end{aligned}$$
$$\begin{aligned} \text{ngen} &: \{n : \mathbb{N}\} \{b : \text{Vec Set } n \rightarrow \text{Set}\} \{k : \text{Kind}\} \\ &\rightarrow (\text{TyConstEnv } b) \\ &\rightarrow (t : \text{Ty } \llbracket k \rrbracket) \\ &\rightarrow b \langle k \rangle (\text{repeat } n \llbracket t \rrbracket) \end{aligned}$$

- So, if we define `b` and the `TyConstEnv` while leaving the arity general, we can get out an arity-generic definition.

Arity-Generic Map

- So, we want to define arity-generic `map`'s base type and `TyConstEnv` for any arity:

`NGmap` : $\{n : \mathbb{N}\} \rightarrow \text{Vec Set } n \rightarrow \text{Set}$

`NGmap Ts` = see the paper!

`ngmap-const` : $\{n : \mathbb{N}\} \rightarrow \text{TyConstEnv } \{n\} \text{ NGmap}$

`ngmap-const` $\{n\} c$ = see the paper!

- With these, we can get arity-generic `map` directly from `ngen`.

`ngmap` : $\{k : \text{Kind}\} \rightarrow (n : \mathbb{N}) \rightarrow (t : \text{Ty } [] k)$

$\rightarrow \text{NGmap } \langle k \rangle (\text{repeat } n \ [t])$

`ngmap` $n t = \text{ngen ngmap-const } t$

Examples with `ngmap`

For example, `maps` and `zips` on products are instances of `ngmap`:

$$\begin{aligned} \text{pair-map1} &: \{A\ B : \text{Set}\} \rightarrow A \rightarrow B \rightarrow A \times B \\ \text{pair-map1} &= \text{ngmap } 1 \text{ (Con Prod)} \end{aligned}$$
$$\begin{aligned} \text{pair-map2} &: \{A1\ B1\ A2\ B2 : \text{Set}\} \\ &\rightarrow (A1 \rightarrow B1) \rightarrow (A2 \rightarrow B2) \\ &\rightarrow A1 \times A2 \rightarrow B1 \times B2 \\ \text{pair-map2} &= \text{ngmap } 2 \text{ (Con Prod)} \end{aligned}$$
$$\begin{aligned} \text{pair-map3} &: \{A1\ B1\ C1\ A2\ B2\ C2 : \text{Set}\} \\ &\rightarrow (A1 \rightarrow B1 \rightarrow C1) \rightarrow (A2 \rightarrow B2 \rightarrow C2) \\ &\rightarrow A1 \times A2 \rightarrow B1 \times B2 \rightarrow C1 \times C2 \\ \text{pair-map3} &= \text{ngmap } 3 \text{ (Con Prod)} \end{aligned}$$

Outline

- 1 Motivation
- 2 Arities in Datatype-Generic Programming
- 3 A Universe for Generic Programming
- 4 The Framework
- 5 Arity-Generic Map
- 6 Conclusion**

Conclusion

Also in the paper:

- More arity-generic programs (equality, unzip, folds).
- How to interface with Agda datatypes.
- Discussion of programming in Agda.

Future work:

- More examples of doubly generic operations.
- Proving properties of doubly generic programs.
- Support for a bigger universe, like arbitrary indexed datatypes and corecursion.

Related Work

- Ralf Hinze. *Polytypic values possess polykinded types*. MPC 2000.
- Thorsten Altenkirch and Conor McBride. *Generic programming within dependently typed programming*. WCGP 2002.
- Wendy Verbruggen, Edsko de Vries, and Arthur Hughes. *Polytypic programming in Coq*. WGP 2008.
- Daniel Fridlender and Mia Indrika. *Do we need dependent types?* JFP 2000.

The End

Thanks for listening!