Arity-Generic Datatype-Generic Programming

Chris Casinghino
Stephanie Weirich

University of Pennsylvania

January 19, 2010 – PLPV, Madrid

Outline

Datatype-Genericity

Datatype-generic functions can be implemented for many datatypes.

$$
\begin{aligned}
&\text{list-map} &&: (a \to b) &&\to [a] \to [b] \\
&\text{maybe-map} &&: (a \to b) &&\to \text{Maybe } a \to \text{Maybe } b \\
&\text{pair-map} &&: (a_1 \to a_2) \to (b_1 \to b_2) &&\to (a_1, b_1) \to (a_2, b_2)
\end{aligned}
$$

Arity-Genericity

Arity-generic functions can be implemented at many arities.

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

$$
\begin{array}{lll}
\mathsf{repeat} & : & a \to [a] \\
\mathsf{map} & : & (a \to b) \to [a] \to [b] \\
\mathsf{zipWith} & : & (a \to b \to c) \to [a] \to [b] \to [c] \\
\mathsf{zipWith3} & : & (a \to b \to c \to d) \to [a] \to [b] \to [c] \to [d]
\end{array}
$$

Doubly Generic Functions

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

$$
\begin{aligned}
&\text{pair-map1} && : a_1 \to a_2 && \to (a_1, a_2) \\
&\text{maybe-map2} && : (a \to b) && \to \text{Maybe } a \to \text{Maybe } b \\
&\text{list-map3} && : (a \to b \to c) && \to [a] \to [b] \to [c] \\
&\ldots
\end{aligned}
$$

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

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

- **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

Different Types at Different Kinds

- Consider equality functions for various types:

$$
\begin{aligned}
\mathsf{nat\text{-}eq} &: \mathbb{N} \to \mathbb{N} \to \mathsf{Bool} \\
\mathsf{list\text{-}eq} &: \forall\, \mathsf{a} \,.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to [\mathsf{a}] \to [\mathsf{a}] \to \mathsf{Bool} \\
\mathsf{pair\text{-}eq} &: \forall\, \mathsf{a}\ \mathsf{b} \,.\, (\mathsf{a} \to \mathsf{a} \to \mathsf{Bool}) \to (\mathsf{b} \to \mathsf{b} \to \mathsf{Bool}) \\
&\qquad \to (\mathsf{a}, \mathsf{b}) \to (\mathsf{a}, \mathsf{b}) \to \mathsf{Bool}
\end{aligned}
$$

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

$$
\begin{aligned}
\mathsf{Eq}\ \langle \star \rangle \qquad\quad \mathsf{t} &= \mathsf{t} \to \mathsf{t} \to \mathsf{Bool} \\
\mathsf{Eq}\ \langle \mathsf{k}_1 \Rightarrow \mathsf{k}_2 \rangle\ \mathsf{t} &= \forall\, \mathsf{a} \,.\, \mathsf{Eq}\ \langle \mathsf{k}_1 \rangle\ \mathsf{a} \to \mathsf{Eq}\ \langle \mathsf{k}_2 \rangle\ (\mathsf{t}\ \mathsf{a})
\end{aligned}
$$

- 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.

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

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

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

- 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

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

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

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:

```
⌊_⌋ : ∀ {k} → Ty [] k → ⟦ k ⟧
```

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

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 _⊛_, pronounced "zap", a zipping application on vectors:

```
_⊛_ : {A B : Set} {n : ℕ}
    → Vec (A → B) n → Vec A n → 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:

$$\begin{aligned}
\mathsf{Eq}\ \langle\star\rangle\ \ \mathsf{t} &= \mathsf{t} \to \mathsf{t} \to \mathsf{Bool} \\
\mathsf{Eq}\ \langle \mathsf{k}_1 \Rightarrow \mathsf{k}_2\rangle\ \mathsf{t} &= \forall\ \mathsf{a}\ .\ \mathsf{Eq}\ \langle \mathsf{k}_1\rangle\ \mathsf{a} \to \mathsf{Eq}\ \langle \mathsf{k}_2\rangle\ (\mathsf{t}\ \mathsf{a})
\end{aligned}$$

$$\begin{aligned}
\mathsf{Map}\ \langle\star\rangle\ \ \mathsf{t}_1\ \mathsf{t}_2 &= \mathsf{t}_1 \to \mathsf{t}_2 \\
\mathsf{Map}\ \langle \mathsf{k}_1 \Rightarrow \mathsf{k}_2\rangle\ \mathsf{t}_1\ \mathsf{t}_2 &= \forall\ \mathsf{a}\ \mathsf{b}\ .\ \mathsf{Map}\ \langle \mathsf{k}_1\rangle\ \mathsf{a}\ \mathsf{b} \\
&\qquad \to \mathsf{Map}\ \langle \mathsf{k}_2\rangle\ (\mathsf{t}_1\ \mathsf{a})\ (\mathsf{t}_2\ \mathsf{b})
\end{aligned}$$

- 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:

$$
\begin{aligned}
\_\langle\_\rangle\_ \; &: \{n : \mathbb{N}\} \to (b : \mathsf{Vec}\ \mathsf{Set}\ n \to \mathsf{Set}) \\
&\to (k : \mathsf{Kind}) \to \mathsf{Vec}\ [\![\ k\ ]\!]\ n \\
&\to \mathsf{Set} \\
b \langle \star \rangle v &= b\ v \\
b \langle k_1 \Rightarrow k_2 \rangle v &= \\
&(a : \mathsf{Vec}\ [\![\ k1\ ]\!]\ \_) \to b \langle k_1 \rangle a \to b \langle k_2 \rangle (v \circledast a)
\end{aligned}
$$

- 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:

$$\begin{aligned}&\mathsf{GMap} \;:\; (\mathsf{v} \;:\; \mathsf{Vec\ Set\ 2}) \longrightarrow \mathsf{Set}\\ &\mathsf{GMap}\ (\mathsf{A} :: \mathsf{B} :: []) \;=\; \mathsf{A} \longrightarrow \mathsf{B}\end{aligned}$$

- $\mathsf{GMap}\ \langle \star \Rightarrow \star \rangle\ (\mathsf{repeat\ 2\ List})$ normalizes to:

$$\forall\, \{\mathsf{A}\ \mathsf{B} \;:\; \mathsf{Set}\} \longrightarrow (\mathsf{A} \longrightarrow \mathsf{B}) \longrightarrow \mathsf{List\ A} \longrightarrow \mathsf{List\ B}$$

- $\mathsf{GMap}\ \langle \star \Rightarrow \star \Rightarrow \star \rangle\ (\mathsf{repeat\ 2\ \_ \times \_})$ normalizes to:

$$\begin{aligned}&\forall\ \ \{\mathsf{A}_1\ \mathsf{B}_1 \;:\; \mathsf{Set}\} \longrightarrow (\mathsf{A}_1 \longrightarrow \mathsf{B}_1)\\ &\longrightarrow \{\mathsf{A}_2\ \mathsf{B}_2 \;:\; \mathsf{Set}\} \longrightarrow (\mathsf{A}_2 \longrightarrow \mathsf{B}_2)\\ &\longrightarrow (\mathsf{A}_1 \times \mathsf{A}_2) \longrightarrow (\mathsf{B}_1 \times \mathsf{B}_2)\end{aligned}$$

The Framework

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

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

- Then they call ngen:

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

Map's TyConstEnv

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

```
gmap-const : TyConstEnv GMap
gmap-const Nat  = λ x → x
gmap-const Unit = λ x → x
gmap-const Prod = λ f1 f2 x → (f1 (proj₁ x), f2 (proj₂ x))
gmap-const Sum  = g
  where g : {A1 B1 A2 B2 : Set}
            → (A1 → B1) → (A2 → B2)
            → A1 ⊎ A2 → B1 ⊎ 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:

$$
\begin{aligned}
&\mathsf{gmap} \;:\; \{\mathsf{k} \;:\; \mathsf{Kind}\} \to (\mathsf{t} \;:\; \mathsf{Ty}\,[]\,\mathsf{k}) \\
&\qquad \to \mathsf{GMap}\,\langle\,\mathsf{k}\,\rangle\,(\lfloor\,\mathsf{t}\,\rfloor :: \lfloor\,\mathsf{t}\,\rfloor :: []) \\
&\mathsf{gmap}\;\mathsf{t} \;=\; \mathsf{ngen}\;\mathsf{gmap\text{-}const}\;\mathsf{t}
\end{aligned}
$$

- For example:

$$
\begin{aligned}
&\mathsf{option\text{-}map} \;:\; \{\mathsf{A}\;\mathsf{B} \;:\; \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{B}) \\
&\qquad \to \mathsf{Option}\;\mathsf{A} \to \mathsf{Option}\;\mathsf{B} \\
&\mathsf{option\text{-}map} \;=\; \mathsf{gmap}\;\mathsf{option}
\end{aligned}
$$

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

Outline

Writing Arity-Generic Programs

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

$$
\begin{aligned}
\_\langle\_\rangle\_ \;&:\; \{n \;:\; \mathbb{N}\} \to (b \;:\; \mathsf{Vec}\ \mathsf{Set}\ n \to \mathsf{Set}) \\
&\to (k \;:\; \mathsf{Kind}) \to \mathsf{Vec}\ [\![\ k\ ]\!]\ n \\
&\to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
\mathsf{TyConstEnv} \;&:\; \{n \;:\; \mathbb{N}\} \to (b \;:\; \mathsf{Vec}\ \mathsf{Set}\ n \to \mathsf{Set}) \\
&\to \mathsf{Set}
\end{aligned}
$$

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

- So, if we define b and the $\mathsf{TyConstEnv}$ 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:

$$\begin{aligned}
&\mathsf{NGmap} : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{Vec\ Set\ n} \to \mathsf{Set} \\
&\mathsf{NGmap\ Ts} = \text{see the paper!} \\
&\mathsf{ngmap\text{-}const} : \{\mathsf{n} : \mathbb{N}\} \to \mathsf{TyConstEnv}\ \{\mathsf{n}\}\ \mathsf{NGmap} \\
&\mathsf{ngmap\text{-}const}\ \{\mathsf{n}\}\ \mathsf{c} = \text{see the paper!}
\end{aligned}$$

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

$$\begin{aligned}
&\mathsf{ngmap} : \{\mathsf{k} : \mathsf{Kind}\} \to (\mathsf{n} : \mathbb{N}) \to (\mathsf{t} : \mathsf{Ty}\ [\,]\ \mathsf{k}) \\
&\qquad \to \mathsf{NGmap}\ \langle\ \mathsf{k}\ \rangle\ (\mathsf{repeat\ n}\ \lfloor\ \mathsf{t}\ \rfloor) \\
&\mathsf{ngmap\ n\ t} = \mathsf{ngen\ ngmap\text{-}const\ t}
\end{aligned}$$

Examples with ngmap

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

$$
\begin{aligned}
&\text{pair-map1} : \{A\ B : \text{Set}\} \to A \to B \to A \times B \\
&\text{pair-map1} = \text{ngmap}\ 1\ (\text{Con}\ \text{Prod})
\end{aligned}
$$

$$
\begin{aligned}
&\text{pair-map2} : \{A1\ B1\ A2\ B2 : \text{Set}\} \\
&\qquad \to (A1 \to B1) \to (A2 \to B2) \\
&\qquad \to A1 \times A2 \to B1 \times B2 \\
&\text{pair-map2} = \text{ngmap}\ 2\ (\text{Con}\ \text{Prod})
\end{aligned}
$$

$$
\begin{aligned}
&\text{pair-map3} : \{A1\ B1\ C1\ A2\ B2\ C2 : \text{Set}\} \\
&\qquad \to (A1 \to B1 \to C1) \to (A2 \to B2 \to C2) \\
&\qquad \to A1 \times A2 \to B1 \times B2 \to C1 \times C2 \\
&\text{pair-map3} = \text{ngmap}\ 3\ (\text{Con}\ \text{Prod})
\end{aligned}
$$

Outline

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!