Generic Programming With Dependent Types: III

Arity-generic programming

Stephanie Weirich

University of Pennsylvania

March 24–26, 2010 – SSGIP

Challenge: Arity-generic map

Can we make a generic version of these functions?

repeat : ∀ {B} → B → List B

map : ∀ {A B} → (A → B) → List A → List B

zipWith : ∀ {A_1 A_2 B} → (A_1 → A_2 → B)
→ List A_1 → List A_2 → List B

zipWith3 : ∀ {A_1 A_2 A_3 B} → (A_1 → A_2 → A_3 → B)
→ List A_1 → List A_2 → List A_3 → List B

Pattern in both types and definition.

Challenge: Arity-generic map

Can we make a generic version of these functions?

$$\text{repeat} : \forall \{B\} \to B \to \text{List}\ B$$
$$\text{repeat}\ x = x :: \text{repeat}\ x$$

$$\text{map} : \forall \{A\ B\} \to (A \to B) \to \text{List}\ A \to \text{List}\ B$$
$$\text{map}\ f\ x = \text{repeat}\ f \odot x$$

$$\text{zipWith} : \forall \{A_1\ A_2\ B\} \to (A_1 \to A_2 \to B) \to \text{List}\ A_1 \to \text{List}\ A_2 \to \text{List}\ B$$
$$\text{zipWith}\ f\ x\ y = \text{repeat}\ f \odot x \odot y$$

$$\text{zipWith3} : \forall \{A_1\ A_2\ A_3\ B\} \to (A_1 \to A_2 \to A_3 \to B) \to \text{List}\ A_1 \to \text{List}\ A_2 \to \text{List}\ A_3 \to \text{List}\ B$$
$$\text{zipWith3}\ f\ x\ y\ z = \text{repeat}\ f \odot x \odot y \odot z$$

Pattern in both types and definition.

Do we need dependent types?

General pattern

$$\text{zipn } n\ f\ x_1\ x_2\ \ldots\ =\ \text{repeat } f \odot x_1 \odot x_2 \odot \ldots$$

Do we need dependent types?

General pattern

$$\text{zipn } n\ f\ x_1\ x_2\ \ldots\ =\ \text{repeat } f \odot x_1 \odot x_2 \odot \ldots$$

Inspiration for a solution: make n do all of the work.

$$\text{zipn } n\ f\ =\ n\ (\text{repeat } f)$$

Do we need dependent types?

General pattern

$$\text{zipn } n\ f\ x_1\ x_2\ \ldots\ =\ \text{repeat } f \odot x_1 \odot x_2 \odot \ldots$$

Inspiration for a solution: make n do all of the work.

$$\text{zipn } n\ f\ =\ n\ (\text{repeat } f)$$

Want encoding of natural numbers where

$$\begin{aligned}
0 &\Rightarrow \lambda\, f \rightarrow f \\
1 &\Rightarrow \lambda\, f\ a \rightarrow f \odot a \\
2 &\Rightarrow \lambda\, f\ a\ b \rightarrow f \odot a \odot b \\
3 &\Rightarrow \lambda\, f\ a\ b\ c \rightarrow f \odot a \odot b \odot c
\end{aligned}$$

Do we need dependent types?

General pattern

$$\text{zipn } n\ f\ x_1\ x_2\ \ldots\ =\ \text{repeat } f \odot x_1 \odot x_2 \odot \ldots$$

Inspiration for a solution: make n do all of the work.

$$\text{zipn } n\ f\ =\ n\ (\text{repeat } f)$$

Want encoding of natural numbers where

$$\begin{aligned} 0 &\Rightarrow \lambda\, f \rightarrow f \\ 1 &\Rightarrow \lambda\, f\ a \rightarrow f \odot a \\ 2 &\Rightarrow \lambda\, f\ a\ b \rightarrow f \odot a \odot b \\ 3 &\Rightarrow \lambda\, f\ a\ b\ c \rightarrow f \odot a \odot b \odot c \end{aligned}$$

General definition

$$\begin{aligned} z\ &= \lambda\, f \rightarrow f \\ s\ n &= \lambda\, f \rightarrow \lambda\, a \rightarrow n\ (f \odot a) \end{aligned}$$

What are the types?

$$
\begin{aligned}
z &: \\
z &= \lambda\, f \rightarrow f \\
s &: \\
\\
s\ n &= \lambda\, f \rightarrow \lambda\, a \rightarrow n\ (f \odot a)
\end{aligned}
$$

What are the types?

$$
\begin{aligned}
z &: \forall \{A\} \to A \to A \\
z &= \lambda\, f \to f \\
s &: \\
\\
s\; n &= \lambda\, f \to \lambda\, a \to n\; (f \odot a)
\end{aligned}
$$

What are the types?

$$
\begin{aligned}
z &: \forall \{A\} \to A \to A \\
z &= \lambda f \to f \\
s &: \forall \{A\ B\} \to (\text{List}\ A \to B) \\
&\quad \to \forall \{C\} \to \text{List}\ (C \to A) \to (\text{List}\ C \to B) \\
s\ n &= \lambda f \to \lambda a \to n\ (f \odot a)
\end{aligned}
$$

What are the types?

$$
\begin{aligned}
z &: \forall \{A\} \to A \to A \\
z &= \lambda\, f \to f \\
s &: \forall \{A\ B\} \to (\mathsf{List}\ A \to B) \\
&\quad \to \forall \{C\} \to \mathsf{List}\ (C \to A) \to (\mathsf{List}\ C \to B) \\
s\ n &= \lambda\, f \to \lambda\, a \to n\ (f \odot a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{one} : \forall \{A\ B\} \to \mathsf{List}\ (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B \\
&\mathsf{one} = s\ z
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{two} : \forall \{A\ B\ C\} \to \mathsf{List}\ (A \to B \to C) \\
&\qquad \to \mathsf{List}\ A \to \mathsf{List}\ B \to \mathsf{List}\ C \\
&\mathsf{two} = s\ (s\ z)
\end{aligned}
$$

What are the types?

$$
\begin{aligned}
\mathsf{z} &: \forall \{A\} \to A \to A \\
\mathsf{z} &= \lambda\, f \to f \\
\mathsf{s} &: \forall \{A\ B\} \to (\mathsf{List}\ A \to B) \\
&\quad \to \forall \{C\} \to \mathsf{List}\ (C \to A) \to (\mathsf{List}\ C \to B) \\
\mathsf{s}\ n &= \lambda\, f \to \lambda\, a \to n\ (f \odot a)
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{one} : \forall \{A\ B\} \to \mathsf{List}\ (A \to B) \to \mathsf{List}\ A \to \mathsf{List}\ B \\
&\mathsf{one} = \mathsf{s}\ \mathsf{z}
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{two} : \forall \{A\ B\ C\} \to \mathsf{List}\ (A \to B \to C) \\
&\qquad \to \mathsf{List}\ A \to \mathsf{List}\ B \to \mathsf{List}\ C \\
&\mathsf{two} = \mathsf{s}\ (\mathsf{s}\ \mathsf{z})
\end{aligned}
$$

$$
\begin{aligned}
&\mathsf{zipn} : \forall \{A\ B\} \to (\mathsf{List}\ A \to B) \to A \to B \\
&\mathsf{zipn}\ n\ f = n\ (\mathsf{repeat}\ f)
\end{aligned}
$$

Actually, no dependent types necessary

Entire example can be implemented in Haskell 98.
See Daniel Fridlender and Mia Indrika. Do we need dependent types? *Journal of Functional Programming*, 10(4):409–415, July 2000.

Discussion about this approach

Discussion about this approach

- Awfully clever. What about other arity-generic functions?

Discussion about this approach

- Awfully clever. What about other arity-generic functions?
- What about types other than lists?

Discussion about this approach

- Awfully clever. What about other arity-generic functions?
- What about types other than lists?
- Haven't we seen something about arities before?

Arity-indexed Generic Programming

Recall from last time:

Kind-indexed type

$$\_\langle\_\rangle\_ \;:\; \forall \{n : \mathbb{N}\}$$
$$\rightarrow (\mathsf{Vec}\ \mathsf{Set}\ n \rightarrow \mathsf{Set}) \rightarrow (k : \mathsf{Kind}) \rightarrow \mathsf{Vec}\ [\![\ k\ ]\!]\ n \rightarrow \mathsf{Set}$$
$$b\ \langle\ \star\ \rangle\ v \ = \ b\ v$$
$$b\ \langle\ k_1 \Rightarrow k_2\ \rangle\ v \ =$$
$$\{a : \mathsf{Vec}\ [\![\ k_1\ ]\!]\ \_\} \rightarrow b\ \langle\ k_1\ \rangle\ a \rightarrow b\ \langle\ k_2\ \rangle\ (v \circledast a)$$

Generator

$$\mathsf{ngen} \;:\; \forall \{n : \mathbb{N}\}\ \{b : \mathsf{Vec}\ \mathsf{Set}\ n \rightarrow \mathsf{Set}\}\ \{k : \mathsf{Kind}\} \rightarrow$$
$$(t : \mathsf{Ty}\ k) \rightarrow \mathsf{ConstEnv}\ b \rightarrow \mathsf{MuGen}\ b \rightarrow b\ \langle\ k\ \rangle\ (\iota\ \lfloor\ t\ \rfloor)$$

Specific example: Repeat

```
Repeat : Vec Set 1 → Set
Repeat (A :: []) = A

grepeat : {k : Kind} → (t : Ty k) → Repeat ⟨ k ⟩ (ι ⌊ t ⌋)
grepeat t = ngen t re (λ {As} → rb {As}) where
  re : ConstEnv Repeat
  re Unit = tt
  re Sum = g where
    g : Repeat ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (ι _⊎_)
    g {A :: []} ra {B :: []} rb = inj₁ ra
  re Prod = g where
    g : Repeat ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (ι _×_)
    g {A :: []} ra {B :: []} rb = (ra,rb)

  rb : MuGen Repeat
  rb {A :: []} = roll
```

Specific example: Map

```
Map : Vec Set 2 → Set
Map (A :: B :: []) = A → B
gmap : ∀ {k : Kind} → (t : Ty k) → Map ⟨ k ⟩ (ι ⌊ t ⌋)
gmap t = ngen t re rb where
  re : ConstEnv Map
  re Unit = λ x → x
  re Sum = g where
    g : Map ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (ι _⊎_)
    g {_ :: _ :: []} ra {_ :: _ :: []} rb = map-sum ra rb
  re Prod = g where
    g : Map ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (ι _×_)
    g {_ :: _ :: []} ra {_ :: _ :: []} rb = map-prod ra rb
  rb : ∀ {As} → Map (As ⊛ (ι μ ⊛ As)) → Map (ι μ ⊛ As)
  rb {_ :: _ :: []} = λ x y → roll (x (unroll y))
```

Specific example: ZipWith

```
ZW : Vec Set 3 → Set
ZW (A :: B :: C :: []) = A → B → C
gzipWith : ∀ {k} → (t : Ty k) → ZW ⟨ k ⟩ (ι ⌊ t ⌋)
gzipWith t = ngen t re rb where
  re : ConstEnv ZW
  re Unit = λ x y → x
  re Sum = g where
    g : ZW ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (ι _⊎_)
    g {_ :: _ :: _ :: []} ra {_ :: _ :: _ :: []} rb = zip-sum ra rb
  re Prod = g where
    g : ZW ⟨ ⋆ ⇒ ⋆ ⇒ ⋆ ⟩ (ι _×_)
    g {_ :: _ :: _ :: []} ra {_ :: _ :: _ :: []} rb = zip-prod ra rb
  rb : ∀ {As} → ZW (As ⊛ (ι μ ⊛ As)) → ZW (ι μ ⊛ As)
  rb {_ :: _ :: _ :: []} = λ x y z → roll (x (unroll y) (unroll z))
```

General version: Arity-generic type-generic map

General version: Arity-generic type-generic map

Start with the type

$$
\begin{aligned}
&\mathsf{NGmap} \;:\; \{\mathsf{n} \;:\; \mathbb{N}\} \to \mathsf{Vec}\;\mathsf{Set}\;(\mathsf{suc}\;\mathsf{n}) \to \mathsf{Set} \\
&\mathsf{NGmap}\;(\mathsf{A} :: [\,]) \;=\; \mathsf{A} \\
&\mathsf{NGmap}\;(\mathsf{A} :: \mathsf{B} :: \mathsf{As}) \;=\; \mathsf{A} \to \mathsf{NGmap}\;(\mathsf{B} :: \mathsf{As})
\end{aligned}
$$

General version: Arity-generic type-generic map

Start with the type

$$
\begin{array}{l}
\mathsf{NGmap} : \{n : \mathbb{N}\} \to \mathsf{Vec\ Set}\ (\mathsf{suc}\ n) \to \mathsf{Set} \\
\mathsf{NGmap}\ (A :: [])\ =\ A \\
\mathsf{NGmap}\ (A :: B :: As)\ =\ A \to \mathsf{NGmap}\ (B :: As)
\end{array}
$$

Then define cases for constants and mu-coercion

$$
\begin{array}{l}
\mathsf{ngmap} : (n : \mathbb{N}) \to \{k : \mathsf{Kind}\} \to (e : \mathsf{Ty}\ k) \\
\qquad \to \mathsf{NGmap}\ \{n\}\ \langle\, k\, \rangle\ (\iota\ \lfloor\, e\, \rfloor) \\
\mathsf{ngmap}\ n\ e\ =\ \mathsf{ngen}\ e\ \mathsf{ngmap\text{-}const} \\
\qquad\qquad\qquad (\lambda\ \{As\} \to \mathsf{ngmap\text{-}mu}\ \{n\}\ \{As\})
\end{array}
$$

Unit case

```
defUnit : (n : ℕ) → NGmap {n} ⟨ ⋆ ⟩ (ι ⊤)
  -- (n : ℕ) → ⊤ → ⊤ → ... → ⊤
defUnit zero     = tt
defUnit (suc n)  = λ x → (defUnit n)
```

Product case

```
defPair : (n : ℕ)
        → {As : Vec Set (suc n)} → NGmap As
        → {Bs : Vec Set (suc n)} → NGmap Bs
        → NGmap (ι _×_ ⊛ As ⊛ Bs)
  -- (n : ℕ) → (A1 → A2 → .... An)
  -- → (B1 → B2 → .... Bn)
  -- → (A1 × B1 → A2 × B2 → .... An × Bn)
defPair zero    {A :: []}         a {B :: []}         b = (a,b)
defPair (suc n) {A1 :: A2 :: As} a {B1 :: B2 :: Bs} b =
λ p →
  defPair n {A2 :: As} (a (proj1 p))
            {B2 :: Bs} (b (proj2 p))
```

Sum Case

```
defSum : (n : ℕ)
  → {As : Vec Set (suc n)} → NGmap As
  → {Bs : Vec Set (suc n)} → NGmap Bs
  → NGmap (ι _⊎_ ⊛ As ⊛ Bs)
defSum zero    {(A :: [])}       a {B :: []}         b  =
  (inj₂ b)
defSum (suc n) {A₁ :: A₂ :: As} a {B₁ :: B₂ :: Bs} b  =  f
  where
    f : A₁ ⊎ B₁ → NGmap (ι _⊎_ ⊛ (A₂ :: As) ⊛ (B₂ :: Bs))
    f (inj₁ a₁)  =  defSum n {A₂ :: As} (a a₁) {B₂ :: Bs} (b error)
    f (inj₂ b₁)  =  defSum n {A₂ :: As} (a error) {B₂ :: Bs} (b b₁)
```

As long as all arguments are the same branch, we never need error.
Note that because defUnit is not strict, we get the truncating behavior for lists.

Iso-recursive coercion

```
MuGen     : {n : ℕ} → (Vec Set (suc n) → Set) → Set
MuGen {b} = ∀ {As} → b (As ⊛ (ι μ ⊛ As)) → b (ι μ ⊛ As)
```

```
ngmap-mu : ∀ {n} → MuGen {n} NGmap
ngmap-mu {zero}  {A :: []}            = roll
ngmap-mu {suc n} {A₁ :: A₂ :: As}  = λ f x →
  ngmap-mu {n} {A₂ :: As} (f (unroll x))
```

Assemble

$$
\begin{aligned}
&\mathsf{ngmap} : (\mathsf{n} : \mathbb{N}) \to \{\mathsf{k} : \mathsf{Kind}\} \to (\mathsf{e} : \mathsf{Ty}\ \mathsf{k}) \\
&\qquad \to \mathsf{NGmap}\ \{\mathsf{n}\}\ \langle\ \mathsf{k}\ \rangle\ (\iota\ \lfloor\ \mathsf{e}\ \rfloor) \\
&\mathsf{ngmap}\ \mathsf{n}\ \mathsf{e} = \mathsf{ngen}\ \mathsf{e}\ \mathsf{ngmap\text{-}const} \\
&\qquad\qquad\qquad (\lambda\ \{\mathsf{As}\} \to \mathsf{ngmap\text{-}mu}\ \{\mathsf{n}\}\ \{\mathsf{As}\})
\end{aligned}
$$

Examples

$$
\begin{array}{lll}
\text{repeat-ml} & : & \forall \{B\} \to B \to \text{List } B \\
\text{repeat-ml} & = & \text{ngmap } 0 \text{ list } \{\_ :: []\} \\
\text{map-ml} & : & \forall \{A_1\ B\} \to (A_1 \to B) \to \text{List } A_1 \to \text{List } B \\
\text{map-ml} & = & \text{ngmap } 1 \text{ list } \{\_ :: \_ :: []\} \\
\text{zipWith-ml} & : & \forall \{A_1\ A_2\ B\} \to (A_1 \to A_2 \to B) \\
 & & \to \text{List } A_1 \to \text{List } A_2 \to \text{List } B \\
\text{zipWith-ml} & = & \text{ngmap } 2 \text{ list } \{\_ :: \_ :: \_ :: []\} \\
\text{zipWith3-ml} & : & \forall \{A_1\ A_2\ A_3\ B\} \to (A_1 \to A_2 \to A_3 \to B) \\
 & & \to \text{List } A_1 \to \text{List } A_2 \to \text{List } A_3 \to \text{List } B \\
\text{zipWith3-ml} & = & \text{ngmap } 3 \text{ list } \{\_ :: \_ :: \_ :: \_ :: []\}
\end{array}
$$

Other examples of arity-generic type-generic functions

Other examples of arity-generic type-generic functions

- n-ary unzip

$$\begin{aligned}
\text{unzip1} &: (A \to B_1) \to \text{List } A \to \text{List } B_1 \\
\text{unzip2} &: (A \to B_1 \times B_2) \to \text{List } A \to \text{List } B_1 \times \text{List } B_2 \\
\text{unzip3} &: (A \to B_1 \times B_2 \times B_3) \\
&\quad \to \text{List } A \to \text{List } B_1 \times \text{List } B_2 \times \text{List } B_3
\end{aligned}$$

Other examples of arity-generic type-generic functions

- n-ary unzip

$$\begin{aligned}
\text{unzip1} &: (A \to B_1) \to \text{List } A \to \text{List } B_1 \\
\text{unzip2} &: (A \to B_1 \times B_2) \to \text{List } A \to \text{List } B_1 \times \text{List } B_2 \\
\text{unzip3} &: (A \to B_1 \times B_2 \times B_3) \\
&\quad \to \text{List } A \to \text{List } B_1 \times \text{List } B_2 \times \text{List } B_3
\end{aligned}$$

- n-ary equality

$$\begin{aligned}
\text{eq1} &: (A_1 \to \text{Bool}) \to \text{List } A_1 \to \text{Bool} \\
\text{eq2} &: (A_1 \to A_2 \to \text{Bool}) \\
&\quad \to \text{List } A_1 \to \text{List } A_2 \to \text{Bool} \\
\text{eq3} &: (A_1 \to A_2 \to A_3 \to \text{Bool}) \\
&\quad \to \text{List } A_1 \to \text{List } A_2 \to \text{List } A_3 \to \text{Bool}
\end{aligned}$$

Other examples of arity-generic type-generic functions

- n-ary unzip

$$
\begin{aligned}
\mathsf{unzip1} &: (A \to B_1) \to \mathsf{List}\ A \to \mathsf{List}\ B_1 \\
\mathsf{unzip2} &: (A \to B_1 \times B_2) \to \mathsf{List}\ A \to \mathsf{List}\ B_1 \times \mathsf{List}\ B_2 \\
\mathsf{unzip3} &: (A \to B_1 \times B_2 \times B_3) \\
&\qquad \to \mathsf{List}\ A \to \mathsf{List}\ B_1 \times \mathsf{List}\ B_2 \times \mathsf{List}\ B_3
\end{aligned}
$$

- n-ary equality

$$
\begin{aligned}
\mathsf{eq1} &: (A_1 \to \mathsf{Bool}) \to \mathsf{List}\ A_1 \to \mathsf{Bool} \\
\mathsf{eq2} &: (A_1 \to A_2 \to \mathsf{Bool}) \\
&\quad \to \mathsf{List}\ A_1 \to \mathsf{List}\ A_2 \to \mathsf{Bool} \\
\mathsf{eq3} &: (A_1 \to A_2 \to A_3 \to \mathsf{Bool}) \\
&\quad \to \mathsf{List}\ A_1 \to \mathsf{List}\ A_2 \to \mathsf{List}\ A_3 \to \mathsf{Bool}
\end{aligned}
$$

- n-ary crushes, others

Discussion about arity-genericity

1. Can we get rid of those implicit lists?

Discussion about arity-genericity

1. Can we get rid of those implicit lists?
2. We used ι and $\circledast$ for vectors to define map, is that fair?

Discussion about arity-genericity

1. Can we get rid of those implicit lists?
2. We used ι and $\circledast$ for vectors to define map, is that fair?
3. Is there a connection between the two definitions?

Conclusion

Generic programming in Agda

- Not exactly simple to define or easy to use
- Easier to define the concrete instances when needed

Conclusion

Generic programming in Agda

- Not exactly simple to define or easy to use
- Easier to define the concrete instances when needed

Why do this?

- Prototyping generic functions shows that they make sense
- Knowing the general definition in Agda helps to understand how to implement the specific definition in other languages/generic frameworks

Future research

Where to next:

- Generic programs for dependently-typed data
- Generic proofs about generic programs
- Generic proofs about dependently-typed programs

Future research

Where to next:

- Generic programs for dependently-typed data
- Generic proofs about generic programs
- Generic proofs about dependently-typed programs

What language features would make this more practical?

- Stronger type inference (canonical structures)
- Better specified type inference
- Better reflection support (Automatic datatype reps...)
- Compile-time specialization, partial evaluation or staging

Additional References

- Stephanie Weirich and Chris Casinghino. Arity-generic type-generic programming. In *ACM SIGPLAN Workshop on Programming Languages Meets Program Verification (PLPV)*, pages 15–26, January 2010
- T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *ESOP '09: Proceedings of the Eighteenth European Symposium On Programming*, pages 32–46, March 2009
- Tim Sheard. Generic programming programming in omega. In Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, editors, *Datatype-Generic Programming*, volume 4719 of *Lecture Notes in Computer Science*, pages 258–284. Springer, 2006