Expanding the Universe

Andres Löh
with lots of inspiration from
José Pedro Magalhães and Conor McBride

Well-Typed

23 May 2011
Why datatype-generic programming?

Motivation (old story):

- capture behaviour that depends on the structure of types;
- capture types that are depend on the structure of types;
- avoid boilerplate, only write the interesting parts of functions;
- write code that is robust against changes in the datatypes.
Some DGP history
Haskell only, and incomplete

- PolyP (Jeuring and Jansson 1997)
- A new approach to generic FP (Hinze 1999)
- Derivable Type Classes (Hinze 2000)
- Generic Haskell (Hinze, Jeuring, Löh 2000–03)
- SYB . . . (Lämmel, Peyton Jones, Hinze, Oliveira, Löh 2003–06)
- . . . Generics for the Masses (Hinze, Oliveira, Löh 2004–06)
- RepLib (Weirich 2006)
- Regular (Noort, Rodriguez, Holdermans, Jeuring, Heeren 2008)
- Instant Generics (Chakravarty, Ditu, Leshchinskiy 2009)
- MultiRec (Rodriguez, Holdermans, Jeuring, Löh 2009)
- Generic deriving (Magalhães, Dijkstra, Jeuring, Löh 2010)
- . . .
Why so many approaches?

Many technical differences:

▸ Which Haskell constructs are used to encode certain concepts.
▸ Mainly a language extension, or mainly a library.

Some conceptual differences:

▸ How are datatypes being viewed?
▸ The view dictates which generic functions can easily be expressed and which not.
▸ The view also restricts the datatypes generic functions can operate on.
Why so many approaches?

Many technical differences:

▶ Which Haskell constructs are used to encode certain concepts.
▶ Mainly a language extension, or mainly a library.

Some conceptual differences:

▶ How are datatypes being viewed?
▶ The view dictates which generic functions can easily be expressed and which not.
▶ The view also restricts the datatypes generic functions can operate on.
Several attempts have been made to categorize approaches:

- by view, representation mechanism, overloading mechanism;
- by a large table of features.
Comparing DGP approaches

Several attempts have been made to categorize approaches:
  - by view, representation mechanism, overloading mechanism;
  - by a large table of features.

There is surprisingly little work on formally comparing different approaches.
Agda is a dependently typed programming language with Haskell-inspired syntax.

Very suitable for generic programming:

- universe constructions (see soon);
- no syntactic difference between terms and types, thus between generic functions and generic types;
- similarity with Haskell allows us to code in a similar style;
- we can prove properties of functions in Agda.
The (long-term) plan

- Implement (model) many approaches to GP in Haskell using Agda.
- Relate the approaches in Agda, by means of Agda functions and properties.
- Gain more understanding of the approaches.
- Fix remaining problems in Agda.
- Either port back to Haskell, or enjoy using GP in Agda.
This talk

- Look at regular, PolyP, multirec.
- Model these approaches as universes in Agda.
- Observe the similarities, and see how one extends the other.
- Generalize.
Universes

A type (Set) of codes:

```haskell
data Code : Set where
  ...
```

An interpretation function taking codes to types:

```haskell
⟦⟧ : Code → ... → Set
  ...
```
Codes (a familiar type):

```
data ℕ : Set where
  zero : ℕ
  suc  : ℕ → ℕ
```
Universes

Example

Codes (a familiar type):

```
data N : Set where
  zero : N
  suc  : N → N
```

Interpretation:

```
Vec : N → Set → Set
Vec (zero) A = ⊤ -- the “unit” type
Vec (suc n) A = A × Vec n A -- a pair
```

We have defined “vectors” of a given type.
Universes
A “generic” function

\[
\text{sum} : (n : \mathbb{N}) \rightarrow \text{Vec} n \ \mathbb{N} \rightarrow \mathbb{N} \\
\text{sum zero tt} = \text{zero} \\
\text{sum (suc n) (x, xs)} = x + \text{sum n xs}
\]
Universes

A “generic” function

```
sum : (n : ℕ) → Vec n ℕ → ℕ
sum zero    tt = zero
sum (suc n) (x, xs) = x + sum n xs
```

In general:

```
generic : (C : Code) → [ C ] → ...
...```

We parameterize over the code, and then do something with its interpretation.
Universes

Remarks

- Universes need not be unfamiliar types.
- One type of codes can admit several interpretations (e.g. Vec and Fin).
- Interpretations can also be defined as datatypes.
- Codes and interpretation functions are first-class.
- So we can do other things with codes than to interpret them; we can define generic functions over them, but also transform them, extend them, restrict them etc.
A more interesting universe

```haskell
data Code : Set where
  U    : Code
  K    : Set → Code
  I    : Code
  _ ⊕ _ : Code → Code → Code
  _ ⊗ _ : Code → Code → Code
  _ ⊙ _ : Code → Code → Code
```
A more interesting universe

data Code : Set where
   U   : Code
   K   : Set → Code
   I   : Code
   _ ⊕ _ : Code → Code → Code
   _ ⊗ _ : Code → Code → Code
   _ ⊙ _ : Code → Code → Code

[ ] : Code → Set → Set
[ U ] X = T
[ K A ] X = A
[ I ] X = X
[ F ⊙ G ] X = [ F ] ([ G ] X)
Encoding types

MaybeC : Code
MaybeC = U ⊕ I
Maybe : Set → Set
Maybe = [] MaybeC ]
nothing : { A : Set} → Maybe A
nothing = inj₁ tt
just : { A : Set} → A → Maybe A
just = inj₂

SquareC : Code
SquareC = I ⊗ I
Square : Set → Set
Square = [ SquareC ]
Example function: map

map : (F : Code) { A B : Set } →
(A → B) → \lfloor F \rfloor A → \lfloor F \rfloor B

map \textbf{U} f \texttt{tt} = \texttt{tt}
map (\textbf{K} A) f c = c
map \textbf{I} f x = f x
map (F \oplus G) f (\text{inj}_1 x) = \text{inj}_1 (map F f x)
map (F \oplus G) f (\text{inj}_2 x) = \text{inj}_2 (map G f x)
map (F \otimes G) f (x, y) = map F f x, map G f y
map (F \odot G) f x = map F (map G f) x
Examples

test₁ : map MaybeC (λ x → suc x) (just 7) ≡ just 8
test₁ = refl

test₂ : map SquareC (λ x → suc x) (2, 3) ≡ (3, 4)
test₂ = refl
Examples

\[
\begin{align*}
\text{test}_1 & : \text{map } \text{MaybeC} (\lambda x \to \text{suc } x) \text{ (just 7)} \equiv \text{just 8} \\
\text{test}_1 & = \text{refl} \\
\text{test}_2 & : \text{map } \text{SquareC} (\lambda x \to \text{suc } x) \text{ (2, 3)} \equiv (3, 4) \\
\text{test}_2 & = \text{refl}
\end{align*}
\]

Still, the universe isn’t particularly interesting, because we cannot describe recursive structures.
Adding fixed points

```haskell
data μ (F : Code) : Set where
  ⟨₀⟩ : [[ F ]] (μ F) → μ F

Nat : Set
Nat = μ MaybeC
nzero : Nat
nzero = ⟨ nothing ⟩
nsuc : Nat → Nat
nsuc n = ⟨ just n ⟩
```
Another datatype

\[
\text{Tree} : \text{Set} \\
\text{Tree} = \mu (\text{MaybeC} \odot \text{SquareC}) \\
\text{leaf} : \text{Tree} \\
\text{leaf} = \langle \text{nothing} \rangle \\
\text{node} : \text{Tree} \to \text{Tree} \to \text{Tree} \\
\text{node} \ l \ r = \langle \text{just} \ (l, r) \rangle
\]
Generic recursion schemes

cata : \{ F : Code \} \{ A : Set \} \rightarrow ([ F ] A \rightarrow A) \rightarrow \mu F \rightarrow A
cata \{ F \} \phi \langle x \rangle = \phi (map F (cata \phi) x)
Generic recursion schemes

cata : \{ F : \text{Code} \} \{ A : \text{Set} \} \to ([F] A \to A) \to \mu F \to A
cata \{ F \} \phi \langle x \rangle = \phi (\text{map} \ F \ (\text{cata} \ \phi) \ x)

plus : \text{Nat} \to \text{Nat} \to \text{Nat}
plus m = \text{cata} \ [\text{const} \ m, \text{nsuc}]

reverse : \text{Tree} \to \text{Tree}
reverse = \text{cata} \ [\text{const} \ \text{leaf}, \ \text{uncurry} \ \text{node} \ \circ \ \text{swap}]

Generic recursion schemes

cata : \{ F : \text{Code} \} \{ A : \text{Set} \} \to (\downarrow F \downarrow A \to A) \to \mu F \to A

cata \{ F \} \phi \langle x \rangle = \phi \left( \text{map} \ F \ (\text{cata} \ \phi) \ x \right)

plus : \text{Nat} \to \text{Nat} \to \text{Nat}

plus m = \text{cata} \ [\text{const} \ m, \text{nsuc}]

reverse : \text{Tree} \to \text{Tree}

reverse = \text{cata} \ [\text{const} \ \text{leaf}, \ \text{uncurry} \ \text{node} \circ \ \text{swap}]

[\_, \_] : \{ A B C : \text{Set} \} \to (A \to C) \to (B \to C) \to (A \uplus B) \to C
More generic recursion schemes

\[
\text{ana : } \{ F : \text{Code} \} \{ A : \text{Set} \} \to (A \to \mu F) \to A \to \mu F
\]

\[
\text{ana } \{ F \} \psi x = \langle \text{map } F (\text{ana } \psi) (\psi x) \rangle
\]
Observations

- Almost exact match with the Haskell library regular.
- We still cannot encode recursive structures with parameters.
- We also cannot encode mutually recursive structures.
From *regular* to *PolyP*

We move from codes of functors to codes of bifunctors.

```haskell
data Code₂ : Set where
  U    : Code₂
  K    : Set → Code₂
  Par  : Code₂
  I    : Code₂
  _ ⊕ _ : Code₂ → Code₂ → Code₂
  _ ⊗ _ : Code₂ → Code₂ → Code₂
```

- Instead of one variable, we have two.
- We ignore composition for now.
Interpretation

\[ [\_]_2 : \text{Code}_2 \to \text{Set} \to \text{Set} \to \text{Set} \]

\[
\begin{align*}
[\text{U}]_2 \, X \, Y & = \top \\
[\text{K} \, \text{A}]_2 \, X \, Y & = \text{A} \\
[\text{Par}]_2 \, X \, Y & = \text{X} \\
[\text{I}]_2 \, X \, Y & = \text{Y} \\
[\text{F} \oplus \text{G}]_2 \, X \, Y & = [\text{F}]_2 \, X \, Y \uplus [\text{G}]_2 \, X \, Y \\
[\text{F} \otimes \text{G}]_2 \, X \, Y & = [\text{F}]_2 \, X \, Y \times [\text{G}]_2 \, X \, Y
\end{align*}
\]
bimap : (F : Code₂) {A B C D : Set} →
    (A → B) → (C → D) → [[F]]₂ A C → [[F]]₂ B D
bimap U f g tt = tt
bimap (K A) f g c = c
bimap Par f g y = f y
bimap I f g x = g x
bimap (F ⊕ G) f g (inj₁ x) = inj₁ (bimap F f g x)
bimap (F ⊕ G) f g (inj₂ x) = inj₂ (bimap G f g x)
bimap (F ⊗ G) f g (x, y) = bimap F f g x, bimap G f g y
Fixed points

data \( \mu \) (F : Code_2) (A : Set) : Set where 
\( \langle \_ \rangle : \semantics{F}_2 A (\mu F A) \to \mu F A \)
**Fixed points**

\[
\textbf{data} \ \mu \ (F : \text{Code}_2) \ (A : \text{Set}) : \text{Set} \ \textbf{where} \\
\langle \_ \rangle : \mathllbracket F \mathrrbracket_2 A (\mu F A) \to \mu F A
\]

\[
cata : \{ F : \text{Code}_2 \} \ \{ A R : \text{Set} \} \to \\
(\mathllbracket F \mathrrbracket_2 A R \to R) \to (\mu F A \to R) \\
cata \{ F \} \ \phi \langle x \rangle = \phi \ (\text{bimap} \ F \ \text{id} \ (\text{cata} \ \phi) \ x)
\]
Examples

List : Set → Set
List = μ (U ⊕ (Par ⊗ I))
Examples

```
List : Set → Set
List = μ(U ⊕ (Par ⊗ I))

nil : {A : Set} → List A
nil = ⟨inj₁ tt⟩

cons : {A : Set} → A → List A → List A
cons x xs = ⟨inj₂ (x, xs)⟩
```
Examples

List : Set → Set
List = \( \mu (U \oplus (\text{Par} \otimes I)) \)

nil : \{ A : \text{Set}\} → \text{List A}
nil = \langle \text{inj}_1 \text{tt} \rangle
cons : \{ A : \text{Set}\} → A → \text{List A} → \text{List A}
cons x xs = \langle \text{inj}_2 (x, xs) \rangle

caseList : \{ A B : \text{Set}\} → \text{List A} → B → (A → \text{List A} → B) → B
caseList \langle \text{inj}_1 \text{tt} \rangle n c = n
caseList \langle \text{inj}_2 (x, xs) \rangle n c = c x xs
foldr : \{ A B : \text{Set}\} → (A → B → B) → B → \text{List A} → B
foldr c n = cata [\text{const} n, \text{uncurry} c]
Other types

\[\text{data} \ \text{Maybe} \ a = \text{Nothing} \mid \text{Just} \ a \quad \text{-- Haskell}\]

Maybe : Set \to Set
Maybe = \mu (U \oplus \text{Par})
Other types

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`data Maybe a = Nothing</td>
<td>Just a`</td>
</tr>
<tr>
<td><code>Maybe : Set \rightarrow Set</code></td>
<td></td>
</tr>
<tr>
<td><code>Maybe = \mu (U \oplus Par)</code></td>
<td></td>
</tr>
<tr>
<td>`data Tree a = Leaf a</td>
<td>Node (Tree a) (Tree a)`</td>
</tr>
<tr>
<td><code>Tree : Set \rightarrow Set</code></td>
<td></td>
</tr>
<tr>
<td><code>Tree = \mu (Par \oplus (I \otimes I))</code></td>
<td></td>
</tr>
</tbody>
</table>
Other types

\[
\text{data } \text{Maybe } a = \text{Nothing } | \text{ Just } a \quad \text{-- Haskell}
\]

Maybe : Set \rightarrow Set
Maybe = \mu (\mathbf{U} \oplus \text{Par})

\[
\text{data } \text{Tree } a = \text{Leaf } a \mid \text{ Node } (\text{Tree } a) (\text{Tree } a) \quad \text{-- Haskell}
\]

Tree : Set \rightarrow Set
Tree = \mu (\text{Par} \oplus (\mathbf{I} \otimes \mathbf{I}))

\[
\text{data } \text{Rose } a = \text{Fork } a [\text{Rose } a]
\]

Rose : Set \rightarrow Set
Rose = \mu (\text{Par} \otimes \{!!\})
What about composition?
Extending the codes

```haskell
data Code₂ : Set where
  U    : Code₂
  K    : Set → Code₂
  Par  : Code₂
  I    : Code₂
  _ ⊕ _ : Code₂ → Code₂ → Code₂
  _ ⊗ _ : Code₂ → Code₂ → Code₂
  _ ⊙ _ : Code₂ → Code₂ → Code₂
```
What about composition?
Extending the interpretation

<table>
<thead>
<tr>
<th>mutual</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\boxed{[_]} : \text{Code}_2 \rightarrow \text{Set} \rightarrow \text{Set} \rightarrow \text{Set}$</td>
</tr>
<tr>
<td>$\boxed{\text{U}}$</td>
</tr>
<tr>
<td>$\boxed{\text{K A}}$</td>
</tr>
<tr>
<td>$\boxed{\text{Par}}$</td>
</tr>
<tr>
<td>$\boxed{I}$</td>
</tr>
<tr>
<td>$\boxed{\text{F } \oplus \ G}$</td>
</tr>
<tr>
<td>$\boxed{\text{F } \otimes \ G}$</td>
</tr>
<tr>
<td>$\boxed{\text{F } \odot \ G}$</td>
</tr>
</tbody>
</table>

**data** $\mu \ (F : \text{Code}_2) \ (A : \text{Set}) : \text{Set}$ where

$\langle \_ \rangle : [\text{F}] \ A \ (\mu \ F \ A) \rightarrow \mu \ F \ A$

We now have the actual PolyP universe.
From the PolyP library

**mutual**

\[
\text{bimap} : (F : \text{Code}_2) \{A B C D : \text{Set}\} \rightarrow \\
\quad (A \rightarrow B) \rightarrow (C \rightarrow D) \rightarrow \mu F A \rightarrow \mu F B \\
\text{bimap } U \quad f \ g \ tt \quad = \quad tt \\
\text{bimap } (K A) \quad f \ g \ c \quad = \quad c \\
\text{bimap } \text{Par} \quad f \ g \ y \quad = \quad f \ y \\
\text{bimap } I \quad f \ g \ x \quad = \quad g \ x \\
\text{bimap } (F \oplus G) \quad f \ g \ (\text{inj}_1 \ x) \quad = \quad \text{inj}_1 (\text{bimap } F f g x) \\
\text{bimap } (F \oplus G) \quad f \ g \ (\text{inj}_2 \ x) \quad = \quad \text{inj}_2 (\text{bimap } G f g x) \\
\text{bimap } (F \otimes G) \quad f \ g \ (x, y) \quad = \quad \text{bimap } F f g x, \text{bimap } G f g y \\
\text{bimap } (F \circop G) \quad f \ g \ x \quad = \quad \text{pmap } \{F\} (\text{bimap } G f g) x \\
\]

**pmap**

\[
pmap : \{F : \text{Code}_2\} \{A B : \text{Set}\} \rightarrow \\
\quad (A \rightarrow B) \rightarrow \mu F A \rightarrow \mu F B \\
pmap \{F\} f \langle x \rangle = \langle \text{bimap } F f (\text{pmap } \{F\} f) x \rangle \\
\]
From the PolyP library

mutual

\[
\begin{align*}
\text{fsum} : (F : \text{Code}_2) & \to \llbracket F \rrbracket \mathbb{N} \mathbb{N} \to \mathbb{N} \\
\text{fsum} U \quad \text{tt} & = 0 \\
\text{fsum} (K A) \quad c & = 0 \\
\text{fsum} \text{Par} \quad x & = x \\
\text{fsum} I \quad x & = x \\
\text{fsum} (F \oplus G) \ (\text{inj}_1 x) & = \text{fsum} F \ x \\
\text{fsum} (F \oplus G) \ (\text{inj}_2 y) & = \text{fsum} G \ y \\
\text{fsum} (F \otimes G) \ (x, y) & = \text{fsum} F \ x + \text{fsum} G \ y \\
\text{fsum} (F \odot G) \ x & = \text{psum} \{ F \} \ (\text{pmap} (\text{fsum} G) \ x) \\
\text{psum} : \{ F : \text{Code}_2 \} & \to \mu F \mathbb{N} \to \mathbb{N} \\
\text{psum} \{ F \} & = \text{cata} \ (\text{fsum} F)
\end{align*}
\]
**From the PolyP library**

```
mutual
   fflatten : (F : Code₂) {A : Set} →
             [ F ] (List A) (List A) → List A
   fflatten U       tt       = []
   fflatten (K A)   c        = []
   fflatten Par x   = x
   fflatten I x     = x
   fflatten (F ⊕ G) (inj₁ x) = fflatten F x
   fflatten (F ⊕ G) (inj₂ x) = fflatten G x
   fflatten (F ⊗ G) (x, y)   = fflatten F x ++ fflatten G y
   fflatten (F ⊚ G) x      = concat (flatten {F}
                                (pmap (fflatten G) x))

   flatten : {F : Code₂} {A : Set} → μ F A → List A
   flatten {F} ⟨ x ⟩ = fflatten F (bimap F [_] flatten x)
```
Limitations of PolyP

- No mutually recursive datatypes.
- No nested (or other forms of indexed) datatypes.
Limitations of PolyP

- No mutually recursive datatypes.
- No nested (or other forms of indexed) datatypes.
- As a reaction, a large number of Haskell approaches without fixed points were introduced.
- Translating this to Agda, it means that inductive types get recursive (infinite) codes.
- We can model that with a coinductive type of codes (but not in this talk).
Recap: regular

data Code : Set where
   U    : Code
   K    : Set → Code
   I    : Code
   _ ⊕ _ : Code → Code → Code
   _ ⊗ _ : Code → Code → Code
   _ ⊙ _ : Code → Code → Code
Recap: regular

```haskell
data Code : Set where
  U     : Code
  K     : Set → Code
  I     : Code
  _ ⊕ _ : Code → Code → Code
  _ ⊗ _ : Code → Code → Code
  _ ⊙ _ : Code → Code → Code

[⟦⟧]  : Code → Set → Set
```
Recap: regular

data Code : Set where
  U : Code
  K : Set → Code
  I : Code
  _ ⊕ _ : Code → Code → Code
  _ ⊗ _ : Code → Code → Code
  _ ⊙ _ : Code → Code → Code

⟦⟧ : Code → Set → Set

data μ (F : Code) : Set where
  ⟨_⟩ : [ F ] (μ F) → μ F
Recap: PolyP

```
data Code₂ : Set where
    U   : Code₂
    K   : Set → Code₂
    Par : Code₂
    I   : Code₂
    _ ⊕ _ : Code₂ → Code₂ → Code₂
    _ ⊗ _ : Code₂ → Code₂ → Code₂
```
Recap: PolyP

```
data Code₂ : Set where
  U : Code₂
  K : Set → Code₂
  Par : Code₂
  I : Code₂
  _ ⊕ _ : Code₂ → Code₂ → Code₂
  _ ⊗ _ : Code₂ → Code₂ → Code₂

[ ] : Code₂ → Set → Set → Set
```
**Recap: PolyP**

\[
\textbf{data} \ \text{Code}_2 : \ \text{Set} \ \textbf{where}
\]
\[
U \quad : \ \text{Code}_2
\]
\[
K \quad : \ \text{Set} \rightarrow \ \text{Code}_2
\]
\[
\text{Par} \quad : \ \text{Code}_2
\]
\[
I \quad : \ \text{Code}_2
\]
\[
_\odot_\otimes : \ \text{Code}_2 \rightarrow \ \text{Code}_2 \rightarrow \ \text{Code}_2
\]
\[
\mu (F : \ \text{Code}_2) (A : \ \text{Set}) : \ \text{Set} \ \textbf{where}
\]
\[
\langle _\rangle : [ [ F ] ]_2 A (\mu F A) \rightarrow \mu F A
\]
Mutually recursive datatypes

Can we define a universe that describes many functors at once?
Mutually recursive datatypes

Can we define a universe that describes many functors at once?

```haskell
data Code (lx : Set) : Set where
  U   : Code lx
  K   : (A : Set) → Code lx
  I   : lx → Code lx
  _ ⊕ _ : Code lx → Code lx → Code lx
  _ ⊗ _ : Code lx → Code lx → Code lx
```

Well-Typed
Can we define a universe that describes many functors at once?

```haskell
data Code (Ix : Set) : Set where
  U      : Code Ix
  K      : (A : Set) → Code Ix
  I      : Ix → Code Ix
  _ ⊕ _  : Code Ix → Code Ix → Code Ix
  _ ⊗ _  : Code Ix → Code Ix → Code Ix

!      : Ix → Code Ix
```
Indexed : Set → Set
Indexed Ix = Ix → Set
Interpretation

Indexed : Set → Set
Indexed Ix = Ix → Set

[ ] : {Ix : Set} → Code Ix → Indexed Ix → Indexed Ix
[ U ] X i = T
[ K A ] X i = A
[ I j ] X i = X j
[ F ⊕ G ] X i = [ F ] X i ⊕ [ G ] X i
[ F ⊗ G ] X i = [ F ] X i × [ G ] X i
[ ! j ] X i = j ≡ i
Example

\[
_\triangleright_ : \{ Ix : \text{Set} \} \to \text{Code Ix} \to Ix \to \text{Code Ix} \\
F \triangleright i = F \otimes !i
\]
Example

\[ \_ \triangleright \_ : \{ \text{lx} : \text{Set} \} \rightarrow \text{Code lx} \rightarrow \text{lx} \rightarrow \text{Code lx} \]

\[ F \triangleright i = F \otimes !i \]

Haskell:

```haskell
data Zero = ZA Zero Zero | ZB One One | ZC Zero

data One = OA Zero One | OB One Zero | OC One
```

Agda encoding without fixed point:

```agda
ZeroOneC : Code (Fin 2)
ZeroOneC = ((I 0 ⊗ I 0) ⊕ (I 1 ⊗ I 1) ⊕ I 0) ▷ 0
             ⊗ ((I 0 ⊗ I 1) ⊕ (I 1 ⊗ I 0) ⊕ I 1) ▷ 1
```
Map

_ ⇒ _ : \{ l : Set \} → Indexed l → Indexed l → Set
R ⇒ S = (l : _) → R l → S l
Map

_ ⇒ _ : {Ix : Set} → Indexed Ix → Indexed Ix → Set
R ⇒ S = (ix : _) → R ix → S ix

map : {Ix : Set} (F : Code Ix) → {R S : Indexed Ix} →
(R ⇒ S) → [F] R ⇒ [F] S
map U  f i _    = tt
map (K X) f i x = x
map (I j) f i x = f j x
map (F ⊕ G) f i (inj₁ x) = inj₁ (map F f i x)
map (F ⊕ G) f i (inj₂ y) = inj₂ (map G f i y)
map (F ⊗ G) f i (x, y) = map F f i x, map G f i y
map (! j) f i x = x
Fixed points

data \( \mu \{ \text{Ix} : \text{Set} \} (F : \text{Code Ix}) (ix : \text{Ix}) : \text{Set} \) where

\[ \langle \_ \rangle : \llbracket F \rrbracket (\mu F) \text{ ix } \to \mu F \text{ ix} \]

cata : \{ \text{Ix} : \text{Set} \} \{ F : \text{Code Ix} \} \{ R : \text{Indexed Ix} \} \to

\((\llbracket F \rrbracket R \Rightarrow R) \to (\mu F \Rightarrow R)\)

cata \{ F = F \} \phi \text{ ix } \langle x \rangle = \phi \text{ ix } (\text{map} F (\text{cata} \phi) \text{ ix } x)\)
So far, we have seen:

- the regular universe: fixed points of functors
  (no parameters, one recursive position)
- the PolyP universe: fixed points of bifunctors
  (one parameter, one recursive position)
- the multirec universe: fixed points of indexed functors
  (no parameters, several recursive positions)
So far, we have seen:

- the *regular* universe: fixed points of functors (no parameters, one recursive position)
- the *PolyP* universe: fixed points of bifunctors (one parameter, one recursive position)
- the *multirec* universe: fixed points of indexed functors (no parameters, several recursive positions)
- Can we also have many parameters?
So far, we have seen:

- the *regular* universe: fixed points of functors
  (no parameters, one recursive position)
- the *PolyP* universe: fixed points of bifunctors
  (one parameter, one recursive position)
- the *multirec* universe: fixed points of indexed functors
  (no parameters, several recursive positions)
- Can we also have many parameters?
  Yes, by decoupling input from output positions.
data Code (Ix : Set) (Ox : Set) : Set where
  U    : Code Ix Ox
  K    : (A : Set) → Code Ix Ox
  I    : Ix → Code Ix Ox
  _ ⊕ _ : Code Ix Ox → Code Ix Ox → Code Ix Ox
  _ ⊗ _ : Code Ix Ox → Code Ix Ox
  _ ⊙ _ : {Mx : Set} →
         Code Mx Ox → Code Ix Mx → Code Ix Ox
  !    : Ox → Code Ix Ox

Composition becomes easier again.
Interpretation

Only the type changes.

\[
\begin{align*}
\mathcal{F} &: \{ \text{Ix Ox : Set} \} \to \text{Code Ix Ox} \to \text{Indexed Ix} \to \text{Indexed Ox} \\
\mathcal{U} &: \quad \text{X} i = \top \\
\mathcal{K} \mathcal{A} &: \quad \text{X} i = \text{A} \\
\mathcal{I} \mathcal{j} &: \quad \text{X} i = \text{X} j \\
\mathcal{F} \oplus \mathcal{G} &: \quad \text{X} i = \mathcal{F} \text{X} i \cup \mathcal{G} \text{X} i \\
\mathcal{F} \otimes \mathcal{G} &: \quad \text{X} i = \mathcal{F} \text{X} i \times \mathcal{G} \text{X} i \\
\mathcal{F} \odot \mathcal{G} &: \quad \text{X} i = \mathcal{F} (\text{G} \text{X}) i \\
\mathcal{!} \mathcal{j} &: \quad \text{X} i = j \equiv i
\end{align*}
\]
Again, only the type changes:

\[
\text{map} : \{ \text{Ix Ox : Set} \} (F : \text{Code Ix Ox}) \rightarrow \\
\quad \{ \text{R S : Indexed Ix} \} \rightarrow (R \Rightarrow S) \rightarrow [F] R \Rightarrow [F] S
\]

\[
\text{map } U \quad f \ i \ _ \ = \ \text{tt}
\]

\[
\text{map } (K \ X) \quad f \ i \ x \ = \ x
\]

\[
\text{map } (I \ j) \quad f \ i \ x \ = \ f \ j \ x
\]

\[
\text{map } (F \oplus G) \quad f \ i \ (\text{inj}_1 \ x) \ = \ \text{inj}_1 \ (\text{map } F \ f \ i \ x)
\]

\[
\text{map } (F \oplus G) \quad f \ i \ (\text{inj}_2 \ y) \ = \ \text{inj}_2 \ (\text{map } G \ f \ i \ y)
\]

\[
\text{map } (F \otimes G) \quad f \ i \ (x, y) \ = \ \text{map } F \ f \ i \ x, \text{map } G \ f \ i \ y
\]

\[
\text{map } (F \odot G) \quad f \ i \ x \ = \ \text{map } F \ (\text{map } G \ f) \ i \ x
\]

\[
\text{map } (! \ j) \quad f \ i \ x \ = \ x
\]
Indexed Bifunctors

To distinguish parameter positions from recursive positions, let us reintroduce bifunctors:

\[
\text{Code}_2 \ : \ (l x \ j x \ o x : \text{Set}) \rightarrow \text{Set}
\]
\[
\text{Code}_2 \ l x \ j x \ o x \ = \ \text{Code} \ (l x \uplus j x) \ o x
\]
Indexed Bifunctors

To distinguish parameter positions from recursive positions, let us reintroduce bifunctors:

\[
\text{Code}_2 : (\text{Ix Jx Ox : Set}) \rightarrow \text{Set}
\]
\[
\text{Code}_2 \text{ Ix Jx Ox} = \text{Code (Ix } \cup \text{ Jx) Ox}
\]

\[
\lbrack \_ \rbrack_2 : \{ \text{Ix Jx Ox : Set} \} \rightarrow \text{Code}_2 \text{ Ix Jx Ox} \rightarrow \text{Indexed Ix} \rightarrow \text{Indexed Jx} \rightarrow \text{Indexed Ox}
\]
\[
\lbrack F \rbrack_2 R S = \lbrack F \rbrack [R, S]
\]
Fixed points

\[ \text{data } \mu \{ \text{Ix Ox : Set} \} (F : \text{Code}_2 \text{ Ix Ox Ox}) \]
\[ (R : \text{Indexed Ix}) : \text{Indexed Ox} \] \text{where}
\[ \langle \_ \rangle : [\, F \,]_2 R (\mu F R) \Rightarrow \mu F R \]

Compare with the PolyP version:

\[ \text{data } \mu (F : \text{Code}_2) (A : \text{Set}) : \text{Set} \] \text{where}
\[ \langle \_ \rangle : [\, F \,]_2 A (\mu F A) \rightarrow \mu F A \]
Fixed points in universe

Actually, the universe can be made closed under fixed points:

```haskell
data Code (Ix : Set) (Ox : Set) : Set where
  ...
  Fix : (F : Code₂ Ix Ox Ox) → Code Ix Ox

...
[ Fix F ] R i  =  μ F R i
```
cata : \{ Ix Ox : Set \} \{ F : \text{Code}_2 Ix Ox Ox \} \\
\{ A : \text{Indexed Ix} \} \{ R : \text{Indexed Ox} \} \rightarrow \\
(\llbracket F \rrbracket_2 A R \Rightarrow R) \rightarrow (\mu F A \Rightarrow R) \\
cata \{ F = F \} \phi i \langle x \rangle = \phi i (\text{bimap} F \text{id} \Rightarrow (\text{cata} \phi) i x)
Special cases

Regular  $= \text{Code}_2 (\text{Fin } 0) (\text{Fin } 1) (\text{Fin } 1)$

PolyP    $= \text{Code}_2 (\text{Fin } 1) (\text{Fin } 1) (\text{Fin } 1)$

Multirec $\text{l}x$  $= \text{Code}_2 (\text{Fin } 0) \text{l}x \text{l}x$
Concluding remarks

- Playing with universes is easy and lots of fun.
- This is still just the beginning.
- Other GP approaches are more different from the ones presented here.
- Other things we can do in universes: abstraction, application, quantification, embedded isomorphisms.
- We should explore the relations between universes.
- Type-indexed datatypes just become other interpretations, or even functions from codes to codes.
- We can often automatically lift functions in one universe to functions in another.
The view/universe described in the paper “A generic deriving mechanism for Haskell” have been implemented in GHC and will hopefully be in GHC 7.2.*.

The mechanism is expressive enough to describe all but one of the currently derivable type classes in GHC.

There will thus be “official” support for generic programming with a sum-of-products view in GHC.