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.

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.

Comparing DGP approaches

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

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:

$\textbf{data}\ \mathsf{Code}\ :\ \mathsf{Set}\ \textbf{where}$

$\quad \ldots$

An interpretation function taking codes to types:

$[\![\_]\!]\ :\ \mathsf{Code} \rightarrow\ \ldots\ \rightarrow \mathsf{Set}$

$\ldots$

Universes

Example

Codes (a familiar type):

data $\mathbb{N}$: Set **where**
  zero : $\mathbb{N}$
  suc : $\mathbb{N} \to \mathbb{N}$

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

$$
\begin{array}{l}
\mathsf{sum}\ :\ (\mathsf{n}\ :\ \mathbb{N}) \to \mathsf{Vec}\ \mathsf{n}\ \mathbb{N} \to \mathbb{N} \\
\mathsf{sum}\ \mathsf{zero}\quad\ \mathsf{tt}\qquad\quad\ = \mathsf{zero} \\
\mathsf{sum}\ (\mathsf{suc}\ \mathsf{n})\ (\mathsf{x}, \mathsf{xs}) = \mathsf{x} + \mathsf{sum}\ \mathsf{n}\ \mathsf{xs}
\end{array}
$$

Universes

A "generic" function

$$
\begin{aligned}
&\mathsf{sum} : (\mathsf{n} : \mathbb{N}) \to \mathsf{Vec}\ \mathsf{n}\ \mathbb{N} \to \mathbb{N} \\
&\mathsf{sum}\ \mathsf{zero}\quad \mathsf{tt} \qquad\quad = \mathsf{zero} \\
&\mathsf{sum}\ (\mathsf{suc}\ \mathsf{n})\ (\mathsf{x}, \mathsf{xs}) = \mathsf{x} + \mathsf{sum}\ \mathsf{n}\ \mathsf{xs}
\end{aligned}
$$

In general:

$$
\begin{aligned}
&\mathsf{generic} : (\mathsf{C} : \mathsf{Code}) \to [\![\ \mathsf{C}\ ]\!] \to \ldots \\
&\ldots
\end{aligned}
$$

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

```
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 = ⊤
⟦ K A     ⟧ X = A
⟦ I       ⟧ X = X
⟦ F ⊕ G   ⟧ X = ⟦ F ⟧ X ⊎ ⟦ G ⟧ X
⟦ F ⊗ G   ⟧ X = ⟦ F ⟧ X × ⟦ G ⟧ X
⟦ F ⊚ G   ⟧ X = ⟦ F ⟧ (⟦ G ⟧ X)
```

Encoding types

$$\mathrm{MaybeC} \;:\; \mathrm{Code}$$
$$\mathrm{MaybeC} \;=\; \mathbf{U} \oplus \mathbf{I}$$

$$\mathrm{Maybe} \;:\; \mathrm{Set} \to \mathrm{Set}$$
$$\mathrm{Maybe} \;=\; [\![\, \mathrm{MaybeC} \,]\!]$$

$$\mathrm{nothing} \;:\; \{\mathrm{A} : \mathrm{Set}\} \to \mathrm{Maybe}\ \mathrm{A}$$
$$\mathrm{nothing} \;=\; \mathrm{inj}_1\ \mathrm{tt}$$

$$\mathrm{just} \;:\; \{\mathrm{A} : \mathrm{Set}\} \to \mathrm{A} \to \mathrm{Maybe}\ \mathrm{A}$$
$$\mathrm{just} \;=\; \mathrm{inj}_2$$

$$\mathrm{SquareC} \;:\; \mathrm{Code}$$
$$\mathrm{SquareC} \;=\; \mathbf{I} \otimes \mathbf{I}$$

$$\mathrm{Square} \;:\; \mathrm{Set} \to \mathrm{Set}$$
$$\mathrm{Square} \;=\; [\![\, \mathrm{SquareC} \,]\!]$$

Example function: map

$$
\begin{array}{l}
\mathsf{map} : (\mathsf{F} : \mathsf{Code})\ \{\mathsf{A}\ \mathsf{B} : \mathsf{Set}\} \rightarrow \\
\qquad (\mathsf{A} \rightarrow \mathsf{B}) \rightarrow [\![\ \mathsf{F}\ ]\!]\ \mathsf{A} \rightarrow [\![\ \mathsf{F}\ ]\!]\ \mathsf{B} \\
\begin{array}{lll}
\mathsf{map}\ \mathbf{U}\ \mathsf{f}\ \mathsf{tt} & = & \mathsf{tt} \\
\mathsf{map}\ (\mathbf{K}\ \mathsf{A})\ \mathsf{f}\ \mathsf{c} & = & \mathsf{c} \\
\mathsf{map}\ \mathbf{I}\ \mathsf{f}\ \mathsf{x} & = & \mathsf{f}\ \mathsf{x} \\
\mathsf{map}\ (\mathsf{F} \oplus \mathsf{G})\ \mathsf{f}\ (\mathsf{inj}_1\ \mathsf{x}) & = & \mathsf{inj}_1\ (\mathsf{map}\ \mathsf{F}\ \mathsf{f}\ \mathsf{x}) \\
\mathsf{map}\ (\mathsf{F} \oplus \mathsf{G})\ \mathsf{f}\ (\mathsf{inj}_2\ \mathsf{x}) & = & \mathsf{inj}_2\ (\mathsf{map}\ \mathsf{G}\ \mathsf{f}\ \mathsf{x}) \\
\mathsf{map}\ (\mathsf{F} \otimes \mathsf{G})\ \mathsf{f}\ (\mathsf{x}, \mathsf{y}) & = & \mathsf{map}\ \mathsf{F}\ \mathsf{f}\ \mathsf{x}, \mathsf{map}\ \mathsf{G}\ \mathsf{f}\ \mathsf{y} \\
\mathsf{map}\ (\mathsf{F} \circledcirc \mathsf{G})\ \mathsf{f}\ \mathsf{x} & = & \mathsf{map}\ \mathsf{F}\ (\mathsf{map}\ \mathsf{G}\ \mathsf{f})\ \mathsf{x}
\end{array}
\end{array}
$$

Examples

$$
\begin{aligned}
&\mathsf{test}_1 \;:\; \mathsf{map}\ \mathsf{MaybeC}\ \ (\lambda\ \mathsf{x} \to \mathsf{suc}\ \mathsf{x})\ (\mathsf{just}\ 7) \equiv \mathsf{just}\ 8\\
&\mathsf{test}_1 \;=\; \mathsf{refl}\\
&\mathsf{test}_2 \;:\; \mathsf{map}\ \mathsf{SquareC}\ (\lambda\ \mathsf{x} \to \mathsf{suc}\ \mathsf{x})\ (2,3) \quad\ \equiv (3,4)\\
&\mathsf{test}_2 \;=\; \mathsf{refl}
\end{aligned}
$$

Examples

$$
\begin{aligned}
&\mathrm{test}_1 \;:\; \mathrm{map}\ \mathrm{MaybeC}\ \ (\lambda\ \mathrm{x} \to \mathrm{suc}\ \mathrm{x})\ (\mathrm{just}\ 7) \equiv \mathrm{just}\ 8 \\
&\mathrm{test}_1 \;=\; \mathrm{refl} \\
&\mathrm{test}_2 \;:\; \mathrm{map}\ \mathrm{SquareC}\ (\lambda\ \mathrm{x} \to \mathrm{suc}\ \mathrm{x})\ (2, 3) \quad \equiv (3, 4) \\
&\mathrm{test}_2 \;=\; \mathrm{refl}
\end{aligned}
$$

Still, the universe isn't particularly interesting, because we cannot describe recursive structures.

Adding fixed points

$$
\begin{array}{l}
\textbf{data}\ \mu\ (\mathsf{F}\ :\ \mathsf{Code})\ :\ \mathsf{Set}\ \textbf{where} \\
\quad \langle\_\rangle\ :\ [\![\ \mathsf{F}\ ]\!]\ (\mu\ \mathsf{F}) \to \mu\ \mathsf{F}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{Nat}\ :\ \mathsf{Set} \\
\mathsf{Nat}\ =\ \mu\ \mathsf{MaybeC} \\
\mathsf{nzero}\ :\ \mathsf{Nat} \\
\mathsf{nzero}\ =\ \langle\ \mathsf{nothing}\ \rangle \\
\mathsf{nsuc}\ :\ \mathsf{Nat} \to \mathsf{Nat} \\
\mathsf{nsuc}\ \mathsf{n}\ =\ \langle\ \mathsf{just}\ \mathsf{n}\ \rangle
\end{array}
$$

Another datatype

$$
\begin{aligned}
&\mathsf{Tree} \;:\; \mathsf{Set}\\
&\mathsf{Tree} \;=\; \mu\,(\mathsf{MaybeC} \circledcirc \mathsf{SquareC})\\
&\mathsf{leaf} \;:\; \mathsf{Tree}\\
&\mathsf{leaf} \;=\; \langle\, \mathsf{nothing}\,\rangle\\
&\mathsf{node} \;:\; \mathsf{Tree} \to \mathsf{Tree} \to \mathsf{Tree}\\
&\mathsf{node}\;\mathsf{l}\;\mathsf{r} \;=\; \langle\, \mathsf{just}\;(\mathsf{l},\mathsf{r})\,\rangle
\end{aligned}
$$

Generic recursion schemes

$$\mathsf{cata} : \{\mathsf{F} : \mathsf{Code}\}\,\{\mathsf{A} : \mathsf{Set}\} \to (\llbracket\,\mathsf{F}\,\rrbracket\,\mathsf{A} \to \mathsf{A}) \to \mu\,\mathsf{F} \to \mathsf{A}$$
$$\mathsf{cata}\,\{\mathsf{F}\}\,\phi\,\langle\,\mathsf{x}\,\rangle = \phi\,(\mathsf{map}\;\mathsf{F}\;(\mathsf{cata}\;\phi)\;\mathsf{x})$$

Generic recursion schemes

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

$$\mathsf{plus} : \mathsf{Nat} \to \mathsf{Nat} \to \mathsf{Nat}$$
$$\mathsf{plus}\; m = \mathsf{cata}\,[\mathsf{const}\; m, \mathsf{nsuc}]$$

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

Generic recursion schemes

$$\mathsf{cata} : \{F : \mathsf{Code}\}\ \{A : \mathsf{Set}\} \to (\llbracket F \rrbracket\ A \to A) \to \mu\ F \to A$$
$$\mathsf{cata}\ \{F\}\ \phi\ \langle\ x\ \rangle\ =\ \phi\ (\mathsf{map}\ F\ (\mathsf{cata}\ \phi)\ x)$$

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

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

$$[\_,\_] : \{A\ B\ C : \mathsf{Set}\} \to (A \to C) \to (B \to C) \to (A \uplus B) \to C$$

More generic recursion schemes

$$\mathsf{ana} : \{\mathsf{F} : \mathsf{Code}\}\,\{\mathsf{A} : \mathsf{Set}\} \rightarrow (\mathsf{A} \rightarrow [\![\, \mathsf{F} \,]\!]\, \mathsf{A}) \rightarrow \mathsf{A} \rightarrow \mu\, \mathsf{F}$$
$$\mathsf{ana}\,\{\mathsf{F}\}\,\psi\,\mathsf{x} = \langle\, \mathsf{map}\, \mathsf{F}\, (\mathsf{ana}\, \psi)\, (\psi\, \mathsf{x}) \,\rangle$$

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.

$$
\begin{array}{lll}
\textbf{data}\ \mathrm{Code}_2 & : & \mathrm{Set}\ \textbf{where} \\
\quad \mathbf{U} & : & \mathrm{Code}_2 \\
\quad \mathbf{K} & : & \mathrm{Set} \rightarrow \mathrm{Code}_2 \\
\quad \mathrm{Par} & : & \mathrm{Code}_2 \\
\quad \mathbf{I} & : & \mathrm{Code}_2 \\
\quad \_ \oplus \_ & : & \mathrm{Code}_2 \rightarrow \mathrm{Code}_2 \rightarrow \mathrm{Code}_2 \\
\quad \_ \otimes \_ & : & \mathrm{Code}_2 \rightarrow \mathrm{Code}_2 \rightarrow \mathrm{Code}_2
\end{array}
$$

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

Interpretation

$$
\begin{aligned}
&[\![\_]\!]_2 : \mathsf{Code}_2 \to \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}\\
&[\![\ \mathbf{U}\ ]\!]_2\ X\ Y = \top\\
&[\![\ \mathbf{K}\ A\ ]\!]_2\ X\ Y = A\\
&[\![\ \mathsf{Par}\ ]\!]_2\ X\ Y = X\\
&[\![\ \mathbf{I}\ ]\!]_2\ X\ Y = Y\\
&[\![\ F \oplus G\ ]\!]_2\ X\ Y = [\![\ F\ ]\!]_2\ X\ Y \uplus [\![\ G\ ]\!]_2\ X\ Y\\
&[\![\ F \otimes G\ ]\!]_2\ X\ Y = [\![\ F\ ]\!]_2\ X\ Y \times [\![\ G\ ]\!]_2\ X\ Y
\end{aligned}
$$

Mapping

```
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 μ (F : Code₂) (A : Set) : Set where
  ⟨_⟩ : ⟦ F ⟧₂ A (μ F A) → μ F A
```

Fixed points

$$
\begin{array}{l}
\textbf{data}\ \mu\ (\mathsf{F} : \mathsf{Code}_2)\ (\mathsf{A} : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \langle\_\rangle : [\![\ \mathsf{F}\ ]\!]_2\ \mathsf{A}\ (\mu\ \mathsf{F}\ \mathsf{A}) \to \mu\ \mathsf{F}\ \mathsf{A}
\end{array}
$$

$$
\begin{array}{l}
\mathsf{cata} : \{\mathsf{F} : \mathsf{Code}_2\}\ \{\mathsf{A}\ \mathsf{R} : \mathsf{Set}\} \to \\
\qquad\quad ([\![\ \mathsf{F}\ ]\!]_2\ \mathsf{A}\ \mathsf{R} \to \mathsf{R}) \to (\mu\ \mathsf{F}\ \mathsf{A} \to \mathsf{R}) \\
\mathsf{cata}\ \{\mathsf{F}\}\ \phi\ \langle\ \mathsf{x}\ \rangle\ =\ \phi\ (\mathsf{bimap}\ \mathsf{F}\ \mathsf{id}\ (\mathsf{cata}\ \phi)\ \mathsf{x})
\end{array}
$$

Examples

$$\begin{aligned}\mathsf{List} &: \mathsf{Set} \to \mathsf{Set}\\ \mathsf{List} &= \mu\,(\mathbf{U} \oplus (\mathsf{Par} \otimes \mathbf{I}))\end{aligned}$$

Examples

$$\begin{aligned}
&\mathsf{List} : \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{List} = \mu\,(\mathbf{U} \oplus (\mathsf{Par} \otimes \mathbf{I}))
\end{aligned}$$

$$\begin{aligned}
&\mathsf{nil} : \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{List}\ \mathsf{A} \\
&\mathsf{nil} = \langle\, \mathsf{inj}_1\ \mathsf{tt}\, \rangle \\
&\mathsf{cons} : \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{List}\ \mathsf{A} \to \mathsf{List}\ \mathsf{A} \\
&\mathsf{cons}\ \mathsf{x}\ \mathsf{xs} = \langle\, \mathsf{inj}_2\ (\mathsf{x}, \mathsf{xs})\, \rangle
\end{aligned}$$

Examples

$$\begin{aligned}
\mathsf{List} &: \mathsf{Set} \to \mathsf{Set} \\
\mathsf{List} &= \mu\,(\mathbf{U} \oplus (\mathsf{Par} \otimes \mathbf{I}))
\end{aligned}$$

$$\begin{aligned}
\mathsf{nil} &: \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{List\ A} \\
\mathsf{nil} &= \langle\, \mathsf{inj}_1\ \mathsf{tt}\, \rangle \\
\mathsf{cons} &: \{\mathsf{A} : \mathsf{Set}\} \to \mathsf{A} \to \mathsf{List\ A} \to \mathsf{List\ A} \\
\mathsf{cons\ x\ xs} &= \langle\, \mathsf{inj}_2\ (\mathsf{x}, \mathsf{xs})\, \rangle
\end{aligned}$$

$$\begin{aligned}
&\mathsf{caseList} : \{\mathsf{A\ B} : \mathsf{Set}\} \to \mathsf{List\ A} \to \mathsf{B} \to (\mathsf{A} \to \mathsf{List\ A} \to \mathsf{B}) \to \mathsf{B} \\
&\mathsf{caseList}\ \langle\, \mathsf{inj}_1\ \mathsf{tt}\, \rangle\ \mathsf{n\ c} = \mathsf{n} \\
&\mathsf{caseList}\ \langle\, \mathsf{inj}_2\ (\mathsf{x}, \mathsf{xs})\, \rangle\ \mathsf{n\ c} = \mathsf{c\ x\ xs} \\
&\mathsf{foldr} : \{\mathsf{A\ B} : \mathsf{Set}\} \to (\mathsf{A} \to \mathsf{B} \to \mathsf{B}) \to \mathsf{B} \to \mathsf{List\ A} \to \mathsf{B} \\
&\mathsf{foldr\ c\ n} = \mathsf{cata}\ [\mathsf{const\ n}, \mathsf{uncurry\ c}]
\end{aligned}$$

Other types

$$\textbf{data}\ \text{Maybe a} \;=\; \text{Nothing} \;\mid\; \text{Just a} \qquad \text{-- Haskell}$$

$$\begin{aligned}
\text{Maybe} &: \text{Set} \to \text{Set} \\
\text{Maybe} &= \mu\,(\mathbf{U} \oplus \text{Par})
\end{aligned}$$

Other types

data Maybe a $=$ Nothing $|$ Just a -- Haskell

$$\begin{aligned} \mathsf{Maybe} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Maybe} &= \mu\,(\mathbf{U} \oplus \mathsf{Par}) \end{aligned}$$

data Tree a $=$ Leaf a $|$ Node (Tree a) (Tree a) -- Haskell

$$\begin{aligned} \mathsf{Tree} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Tree} &= \mu\,(\mathsf{Par} \oplus (\mathbf{I} \otimes \mathbf{I})) \end{aligned}$$

Other types

data Maybe a $=$ Nothing $|$ Just a -- Haskell

$$\begin{aligned}\mathsf{Maybe} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Maybe} &= \mu\,(\mathbf{U} \oplus \mathsf{Par})\end{aligned}$$

data Tree a $=$ Leaf a $|$ Node (Tree a) (Tree a) -- Haskell

$$\begin{aligned}\mathsf{Tree} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Tree} &= \mu\,(\mathsf{Par} \oplus (\mathbf{I} \otimes \mathbf{I}))\end{aligned}$$

data Rose a $=$ Fork a [Rose a]

$$\begin{aligned}\mathsf{Rose} &: \mathsf{Set} \to \mathsf{Set} \\ \mathsf{Rose} &= \mu\,(\mathsf{Par} \otimes \{!!\})\end{aligned}$$

What about composition?

Extending the codes

$$
\begin{array}{lcl}
\textbf{data}\ \mathsf{Code}_2 & : & \mathsf{Set}\ \textbf{where} \\
\quad \textbf{U} & : & \mathsf{Code}_2 \\
\quad \textbf{K} & : & \mathsf{Set} \to \mathsf{Code}_2 \\
\quad \mathsf{Par} & : & \mathsf{Code}_2 \\
\quad \textbf{I} & : & \mathsf{Code}_2 \\
\quad \_\oplus\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2 \\
\quad \_\otimes\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2 \\
\quad \_\circledcirc\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2
\end{array}
$$

What about composition?

Extending the interpretation

mutual

$$
\begin{array}{l}
[\![\_]\!] : \mathrm{Code}_2 \to \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set} \\
[\![\, \mathbf{U} \,]\!]\; X\; Y = \top \\
[\![\, \mathbf{K}\; A \,]\!]\; X\; Y = A \\
[\![\, \mathrm{Par} \,]\!]\; X\; Y = X \\
[\![\, \mathbf{I} \,]\!]\; X\; Y = Y \\
[\![\, F \oplus G \,]\!]\; X\; Y = [\![\, F \,]\!]\; X\; Y \uplus [\![\, G \,]\!]\; X\; Y \\
[\![\, F \otimes G \,]\!]\; X\; Y = [\![\, F \,]\!]\; X\; Y \times [\![\, G \,]\!]\; X\; Y \\
[\![\, F \circledcirc G \,]\!]\; X\; Y = \mu\, F\, ([\![\, G \,]\!]\; X\; Y)
\end{array}
$$

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

$\quad \langle\_\rangle : [\![\, F \,]\!]\; A\; (\mu\, F\, A) \to \mu\, F\, A$

We now have the actual PolyP universe.

From the PolyP library

mutual

$$
\begin{array}{l}
\mathsf{bimap} : (F : \mathsf{Code}_2)\ \{A\ B\ C\ D : \mathsf{Set}\} \rightarrow \\
\qquad (A \rightarrow B) \rightarrow (C \rightarrow D) \rightarrow [\![\, F \,]\!]\ A\ C \rightarrow [\![\, F \,]\!]\ B\ D \\
\mathsf{bimap}\ \mathbf{U}\ f\ g\ tt = tt \\
\mathsf{bimap}\ (\mathbf{K}\ A)\ f\ g\ c = c \\
\mathsf{bimap}\ \mathsf{Par}\ f\ g\ y = f\ y \\
\mathsf{bimap}\ \mathbf{I}\ f\ g\ x = g\ x \\
\mathsf{bimap}\ (F \oplus G)\ f\ g\ (\mathsf{inj}_1\ x) = \mathsf{inj}_1\ (\mathsf{bimap}\ F\ f\ g\ x) \\
\mathsf{bimap}\ (F \oplus G)\ f\ g\ (\mathsf{inj}_2\ x) = \mathsf{inj}_2\ (\mathsf{bimap}\ G\ f\ g\ x) \\
\mathsf{bimap}\ (F \otimes G)\ f\ g\ (x, y) = \mathsf{bimap}\ F\ f\ g\ x, \mathsf{bimap}\ G\ f\ g\ y \\
\mathsf{bimap}\ (F \circledcirc G)\ f\ g\ x = \mathsf{pmap}\ \{F\}\ (\mathsf{bimap}\ G\ f\ g)\ x
\end{array}
$$

$$
\begin{array}{l}
\mathsf{pmap} : \{F : \mathsf{Code}_2\}\ \{A\ B : \mathsf{Set}\} \rightarrow \\
\qquad (A \rightarrow B) \rightarrow \mu\ F\ A \rightarrow \mu\ F\ B \\
\mathsf{pmap}\ \{F\}\ f\ \langle\, x \,\rangle = \langle\, \mathsf{bimap}\ F\ f\ (\mathsf{pmap}\ \{F\}\ f)\ x \,\rangle
\end{array}
$$

From the PolyP library

$$
\begin{array}{l}
\textbf{mutual} \\
\quad \mathsf{fsum} : (F : \mathsf{Code}_2) \to [\![F]\!]\, \mathbb{N}\, \mathbb{N} \to \mathbb{N} \\
\quad \begin{array}{lll}
\mathsf{fsum}\ \mathbf{U} & \mathsf{tt} & = 0 \\
\mathsf{fsum}\ (\mathbf{K}\ A) & c & = 0 \\
\mathsf{fsum}\ \mathsf{Par} & x & = x \\
\mathsf{fsum}\ \mathbf{I} & x & = x \\
\mathsf{fsum}\ (F \oplus G) & (\mathsf{inj}_1\ x) & = \mathsf{fsum}\ F\ x \\
\mathsf{fsum}\ (F \oplus G) & (\mathsf{inj}_2\ y) & = \mathsf{fsum}\ G\ y \\
\mathsf{fsum}\ (F \otimes G) & (x, y) & = \mathsf{fsum}\ F\ x + \mathsf{fsum}\ G\ y \\
\mathsf{fsum}\ (F \circledcirc G) & x & = \mathsf{psum}\ \{F\}\ (\mathsf{pmap}\ (\mathsf{fsum}\ G)\ x)
\end{array} \\
\quad \mathsf{psum} : \{F : \mathsf{Code}_2\} \to \mu\, F\, \mathbb{N} \to \mathbb{N} \\
\quad \mathsf{psum}\ \{F\} = \mathsf{cata}\ (\mathsf{fsum}\ F)
\end{array}
$$

From the PolyP library

mutual

$$
\begin{array}{l}
\text{fflatten} : (F : \text{Code}_2)\ \{A : \text{Set}\} \rightarrow \\
\qquad [\![\, F \,]\!]\ (\text{List}\ A)\ (\text{List}\ A) \rightarrow \text{List}\ A
\end{array}
$$

$$
\begin{array}{lll}
\text{fflatten}\ \mathbf{U}\ \ \text{tt} & = & [\,] \\
\text{fflatten}\ (\mathbf{K}\ A)\ \ c & = & [\,] \\
\text{fflatten}\ \text{Par}\ \ x & = & x \\
\text{fflatten}\ \mathbf{I}\ \ x & = & x \\
\text{fflatten}\ (F \oplus G)\ (\text{inj}_1\ x) & = & \text{fflatten}\ F\ x \\
\text{fflatten}\ (F \oplus G)\ (\text{inj}_2\ x) & = & \text{fflatten}\ G\ x \\
\text{fflatten}\ (F \otimes G)\ (x, y) & = & \text{fflatten}\ F\ x + \text{fflatten}\ G\ y \\
\text{fflatten}\ (F \circledcirc G)\ x & = & \text{concat}\ (\text{flatten}\ \{F\} \\
 & & \qquad (\text{pmap}\ (\text{fflatten}\ G)\ x))
\end{array}
$$

$$
\begin{array}{l}
\text{flatten} : \{F : \text{Code}_2\}\ \{A : \text{Set}\} \rightarrow \mu\ F\ A \rightarrow \text{List}\ A \\
\text{flatten}\ \{F\}\ \langle\, x \,\rangle = \text{fflatten}\ F\ (\text{bimap}\ F\ [\_]\ \text{flatten}\ x)
\end{array}
$$

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`

```
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`

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Code}_2 & : & \mathsf{Set}\ \textbf{where} \\
\quad \mathbf{U} & : & \mathsf{Code}_2 \\
\quad \mathbf{K} & : & \mathsf{Set} \to \mathsf{Code}_2 \\
\quad \mathsf{Par} & : & \mathsf{Code}_2 \\
\quad \mathbf{I} & : & \mathsf{Code}_2 \\
\quad \_\oplus\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2 \\
\quad \_\otimes\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2
\end{array}
$$

Recap: `PolyP`

$$
\begin{array}{lll}
\textbf{data}\ \mathrm{Code}_2 & : & \mathrm{Set}\ \textbf{where} \\
\quad \mathbf{U} & : & \mathrm{Code}_2 \\
\quad \mathbf{K} & : & \mathrm{Set} \to \mathrm{Code}_2 \\
\quad \mathrm{Par} & : & \mathrm{Code}_2 \\
\quad \mathbf{I} & : & \mathrm{Code}_2 \\
\quad \_\oplus\_ & : & \mathrm{Code}_2 \to \mathrm{Code}_2 \to \mathrm{Code}_2 \\
\quad \_\otimes\_ & : & \mathrm{Code}_2 \to \mathrm{Code}_2 \to \mathrm{Code}_2
\end{array}
$$

$$
[\![\_]\!] : \mathrm{Code}_2 \to \mathrm{Set} \to \mathrm{Set} \to \mathrm{Set}
$$

Recap: `PolyP`

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Code}_2 & : & \mathsf{Set}\ \textbf{where} \\
\quad \mathbf{U} & : & \mathsf{Code}_2 \\
\quad \mathbf{K} & : & \mathsf{Set} \to \mathsf{Code}_2 \\
\quad \mathsf{Par} & : & \mathsf{Code}_2 \\
\quad \mathbf{I} & : & \mathsf{Code}_2 \\
\quad \_\oplus\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2 \\
\quad \_\otimes\_ & : & \mathsf{Code}_2 \to \mathsf{Code}_2 \to \mathsf{Code}_2
\end{array}
$$

$$
[\![\_]\!] : \mathsf{Code}_2 \to \mathsf{Set} \to \mathsf{Set} \to \mathsf{Set}
$$

$$
\begin{array}{l}
\textbf{data}\ \mu\ (\mathsf{F} : \mathsf{Code}_2)\ (\mathsf{A} : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \langle\_\rangle : [\![\ \mathsf{F}\ ]\!]_2\ \mathsf{A}\ (\mu\ \mathsf{F}\ \mathsf{A}) \to \mu\ \mathsf{F}\ \mathsf{A}
\end{array}
$$

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?

$$
\begin{array}{lll}
\textbf{data}\ \mathsf{Code}\ (\mathsf{Ix}\ :\ \mathsf{Set})\ :\ \mathsf{Set}\ \textbf{where} & & \\
\quad \textbf{U} & : & \mathsf{Code\ Ix} \\
\quad \textbf{K} & : & (\mathsf{A}\ :\ \mathsf{Set}) \rightarrow \mathsf{Code\ Ix} \\
\quad \textbf{I} & : & \mathsf{Ix} \rightarrow \mathsf{Code\ Ix} \\
\quad \_ \oplus \_ & : & \mathsf{Code\ Ix} \rightarrow \mathsf{Code\ Ix} \rightarrow \mathsf{Code\ Ix} \\
\quad \_ \otimes \_ & : & \mathsf{Code\ Ix} \rightarrow \mathsf{Code\ Ix} \rightarrow \mathsf{Code\ Ix}
\end{array}
$$

Mutually recursive datatypes

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

```
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
```

Interpretation

$$
\begin{aligned}
&\mathsf{Indexed} \;:\; \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{Indexed}\;\mathsf{Ix} \;=\; \mathsf{Ix} \to \mathsf{Set}
\end{aligned}
$$

Interpretation

$$
\begin{aligned}
&\mathsf{Indexed} : \mathsf{Set} \to \mathsf{Set} \\
&\mathsf{Indexed}\ \mathsf{Ix} = \mathsf{Ix} \to \mathsf{Set}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\_]\!] : \{\mathsf{Ix} : \mathsf{Set}\} \to \mathsf{Code}\ \mathsf{Ix} \to \mathsf{Indexed}\ \mathsf{Ix} \to \mathsf{Indexed}\ \mathsf{Ix} \\
&[\![\ \mathbf{U}\ ]\!]\ \mathsf{X}\ \mathsf{i} = \top \\
&[\![\ \mathbf{K}\ \mathsf{A}\ ]\!]\ \mathsf{X}\ \mathsf{i} = \mathsf{A} \\
&[\![\ \mathbf{I}\ \mathsf{j}\ ]\!]\ \mathsf{X}\ \mathsf{i} = \mathsf{X}\ \mathsf{j} \\
&[\![\ \mathsf{F} \oplus \mathsf{G}\ ]\!]\ \mathsf{X}\ \mathsf{i} = [\![\ \mathsf{F}\ ]\!]\ \mathsf{X}\ \mathsf{i} \uplus [\![\ \mathsf{G}\ ]\!]\ \mathsf{X}\ \mathsf{i} \\
&[\![\ \mathsf{F} \otimes \mathsf{G}\ ]\!]\ \mathsf{X}\ \mathsf{i} = [\![\ \mathsf{F}\ ]\!]\ \mathsf{X}\ \mathsf{i} \times [\![\ \mathsf{G}\ ]\!]\ \mathsf{X}\ \mathsf{i} \\
&[\![\ !\ \mathsf{j}\ ]\!]\ \mathsf{X}\ \mathsf{i} = \mathsf{j} \equiv \mathsf{i}
\end{aligned}
$$

Example

```
_▷_ : {Ix : Set} → Code Ix → Ix → Code Ix
F ▷ i = F ⊗ ! i
```

Example

$$\_\triangleright\_ : \{\mathsf{Ix} : \mathsf{Set}\} \to \mathsf{Code}\ \mathsf{Ix} \to \mathsf{Ix} \to \mathsf{Code}\ \mathsf{Ix}$$
$$\mathsf{F} \triangleright \mathsf{i} = \mathsf{F} \otimes !\,\mathsf{i}$$

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:

$$\mathsf{ZeroOneC} : \mathsf{Code}\ (\mathsf{Fin}\ 2)$$
$$\begin{aligned}\mathsf{ZeroOneC} = {} & ((\mathbf{I}\ 0 \otimes \mathbf{I}\ 0) \oplus (\mathbf{I}\ 1 \otimes \mathbf{I}\ 1) \oplus \mathbf{I}\ 0) \triangleright 0 \\ \otimes {} & ((\mathbf{I}\ 0 \otimes \mathbf{I}\ 1) \oplus (\mathbf{I}\ 1 \otimes \mathbf{I}\ 0) \oplus \mathbf{I}\ 1) \triangleright 1\end{aligned}$$

Map

$$
\begin{array}{l}
\_ \rightrightarrows \_ \;:\; \{\mathsf{Ix} \;:\; \mathsf{Set}\} \to \mathsf{Indexed}\ \mathsf{Ix} \to \mathsf{Indexed}\ \mathsf{Ix} \to \mathsf{Set} \\
\mathsf{R} \;\rightrightarrows\; \mathsf{S} \;=\; (\mathsf{ix} \;:\; \_) \to \mathsf{R}\ \mathsf{ix} \to \mathsf{S}\ \mathsf{ix}
\end{array}
$$

Map

$$\_ \rightrightarrows \_ : \{\mathsf{Ix} : \mathsf{Set}\} \to \mathsf{Indexed}\ \mathsf{Ix} \to \mathsf{Indexed}\ \mathsf{Ix} \to \mathsf{Set}$$
$$\mathsf{R} \rightrightarrows \mathsf{S} = (\mathsf{ix} : \_) \to \mathsf{R}\ \mathsf{ix} \to \mathsf{S}\ \mathsf{ix}$$

$$\begin{array}{lll}
\multicolumn{3}{l}{\mathsf{map} : \{\mathsf{Ix} : \mathsf{Set}\}\ (\mathsf{F} : \mathsf{Code}\ \mathsf{Ix}) \to \{\mathsf{R}\ \mathsf{S} : \mathsf{Indexed}\ \mathsf{Ix}\} \to} \\
\multicolumn{3}{l}{\qquad (\mathsf{R} \rightrightarrows \mathsf{S}) \to [\![\, \mathsf{F} \,]\!]\ \mathsf{R} \rightrightarrows [\![\, \mathsf{F} \,]\!]\ \mathsf{S}} \\
\mathsf{map}\ \mathbf{U} & \mathsf{f}\ \mathsf{i}\ _ & = \mathsf{tt} \\
\mathsf{map}\ (\mathbf{K}\ \mathsf{X}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = \mathsf{x} \\
\mathsf{map}\ (\mathbf{I}\ \mathsf{j}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = \mathsf{f}\ \mathsf{j}\ \mathsf{x} \\
\mathsf{map}\ (\mathsf{F} \oplus \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ (\mathsf{inj}_1\ \mathsf{x}) & = \mathsf{inj}_1\ (\mathsf{map}\ \mathsf{F}\ \mathsf{f}\ \mathsf{i}\ \mathsf{x}) \\
\mathsf{map}\ (\mathsf{F} \oplus \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ (\mathsf{inj}_2\ \mathsf{y}) & = \mathsf{inj}_2\ (\mathsf{map}\ \mathsf{G}\ \mathsf{f}\ \mathsf{i}\ \mathsf{y}) \\
\mathsf{map}\ (\mathsf{F} \otimes \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ (\mathsf{x}, \mathsf{y}) & = \mathsf{map}\ \mathsf{F}\ \mathsf{f}\ \mathsf{i}\ \mathsf{x}, \mathsf{map}\ \mathsf{G}\ \mathsf{f}\ \mathsf{i}\ \mathsf{y} \\
\mathsf{map}\ (!\ \mathsf{j}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = \mathsf{x}
\end{array}$$

Fixed points

```
data μ {Ix : Set} (F : Code Ix) (ix : Ix) : Set where
  ⟨_⟩ : ⟦ F ⟧ (μ F) ix → μ F ix
```

```
cata : {Ix : Set} {F : Code Ix} {R : Indexed Ix} →
       (⟦ F ⟧ R ⇉ R) → (μ F ⇉ R)
cata {F = F} ϕ ix ⟨ x ⟩ = ϕ ix (map F (cata ϕ) ix x)
```

Status

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)

Status

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?

Status

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.

Codes

```
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.

$$
\begin{array}{lll}
[\![\_]\!] & : & \{Ix\ Ox\ :\ Set\} \to Code\ Ix\ Ox \to Indexed\ Ix \to Indexed\ Ox \\
[\![\ \mathbf{U}\ ]\!] & X\ i & =\ \top \\
[\![\ \mathbf{K}\ A\ ]\!] & X\ i & =\ A \\
[\![\ \mathbf{I}\ j\ ]\!] & X\ i & =\ X\ j \\
[\![\ F \oplus G\ ]\!] & X\ i & =\ [\![\ F\ ]\!]\ X\ i \uplus [\![\ G\ ]\!]\ X\ i \\
[\![\ F \otimes G\ ]\!] & X\ i & =\ [\![\ F\ ]\!]\ X\ i \times [\![\ G\ ]\!]\ X\ i \\
[\![\ F \circledcirc G\ ]\!] & X\ i & =\ [\![\ F\ ]\!]\ ([\![\ G\ ]\!]\ X)\ i \\
[\![\ !\ j\ ]\!] & X\ i & =\ j \equiv i
\end{array}
$$

Map

Again, only the type changes:

$$
\begin{array}{lllll}
\mathsf{map} : \{\mathsf{Ix}\ \mathsf{Ox} : \mathsf{Set}\}\ (\mathsf{F} : \mathsf{Code}\ \mathsf{Ix}\ \mathsf{Ox}) \to \\
\qquad \{\mathsf{R}\ \mathsf{S} : \mathsf{Indexed}\ \mathsf{Ix}\} \to (\mathsf{R} \rightrightarrows \mathsf{S}) \to [\![\, \mathsf{F} \,]\!]\ \mathsf{R} \rightrightarrows [\![\, \mathsf{F} \,]\!]\ \mathsf{S} \\
\mathsf{map}\ \mathbf{U} & \mathsf{f}\ \mathsf{i}\ \_ & = & \mathsf{tt} \\
\mathsf{map}\ (\mathbf{K}\ \mathsf{X}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = & \mathsf{x} \\
\mathsf{map}\ (\mathbf{I}\ \mathsf{j}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = & \mathsf{f}\ \mathsf{j}\ \mathsf{x} \\
\mathsf{map}\ (\mathsf{F} \oplus \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ (\mathsf{inj}_1\ \mathsf{x}) & = & \mathsf{inj}_1\ (\mathsf{map}\ \mathsf{F}\ \mathsf{f}\ \mathsf{i}\ \mathsf{x}) \\
\mathsf{map}\ (\mathsf{F} \oplus \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ (\mathsf{inj}_2\ \mathsf{y}) & = & \mathsf{inj}_2\ (\mathsf{map}\ \mathsf{G}\ \mathsf{f}\ \mathsf{i}\ \mathsf{y}) \\
\mathsf{map}\ (\mathsf{F} \otimes \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ (\mathsf{x}, \mathsf{y}) & = & \mathsf{map}\ \mathsf{F}\ \mathsf{f}\ \mathsf{i}\ \mathsf{x}, \mathsf{map}\ \mathsf{G}\ \mathsf{f}\ \mathsf{i}\ \mathsf{y} \\
\mathsf{map}\ (\mathsf{F} \circledcirc \mathsf{G}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = & \mathsf{map}\ \mathsf{F}\ (\mathsf{map}\ \mathsf{G}\ \mathsf{f})\ \mathsf{i}\ \mathsf{x} \\
\mathsf{map}\ (!\ \mathsf{j}) & \mathsf{f}\ \mathsf{i}\ \mathsf{x} & = & \mathsf{x}
\end{array}
$$

Indexed Bifunctors

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

$$\mathrm{Code}_2 \;:\; (\mathrm{Ix}\ \mathrm{Jx}\ \mathrm{Ox} \;:\; \mathrm{Set}) \rightarrow \mathrm{Set}$$
$$\mathrm{Code}_2\ \mathrm{Ix}\ \mathrm{Jx}\ \mathrm{Ox} \;=\; \mathrm{Code}\ (\mathrm{Ix} \uplus \mathrm{Jx})\ \mathrm{Ox}$$

Indexed Bifunctors

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

$$
\begin{aligned}
&\mathsf{Code}_2 : (\mathsf{Ix}\ \mathsf{Jx}\ \mathsf{Ox} : \mathsf{Set}) \to \mathsf{Set} \\
&\mathsf{Code}_2\ \mathsf{Ix}\ \mathsf{Jx}\ \mathsf{Ox} = \mathsf{Code}\ (\mathsf{Ix} \uplus \mathsf{Jx})\ \mathsf{Ox}
\end{aligned}
$$

$$
\begin{aligned}
&[\![\_]\!]_2 : \{\mathsf{Ix}\ \mathsf{Jx}\ \mathsf{Ox} : \mathsf{Set}\} \to \mathsf{Code}_2\ \mathsf{Ix}\ \mathsf{Jx}\ \mathsf{Ox} \to \\
&\qquad \mathsf{Indexed}\ \mathsf{Ix} \to \mathsf{Indexed}\ \mathsf{Jx} \to \mathsf{Indexed}\ \mathsf{Ox} \\
&[\![\ \mathsf{F}\ ]\!]_2\ \mathsf{R}\ \mathsf{S} = [\![\ \mathsf{F}\ ]\!]\ [\mathsf{R}, \mathsf{S}]
\end{aligned}
$$

Fixed points

$$
\begin{array}{l}
\textbf{data}\ \mu\ \{\mathsf{Ix}\ \mathsf{Ox} : \mathsf{Set}\}\ (\mathsf{F} : \mathsf{Code}_2\ \mathsf{Ix}\ \mathsf{Ox}\ \mathsf{Ox}) \\
\quad (\mathsf{R} : \mathsf{Indexed}\ \mathsf{Ix}) : \mathsf{Indexed}\ \mathsf{Ox}\ \textbf{where} \\
\quad \langle\_\rangle : [\![\ \mathsf{F}\ ]\!]_2\ \mathsf{R}\ (\mu\ \mathsf{F}\ \mathsf{R}) \rightrightarrows \mu\ \mathsf{F}\ \mathsf{R}
\end{array}
$$

Compare with the PolyP version:

$$
\begin{array}{l}
\textbf{data}\ \mu\ (\mathsf{F} : \mathsf{Code}_2)\ (\mathsf{A} : \mathsf{Set}) : \mathsf{Set}\ \textbf{where} \\
\quad \langle\_\rangle : [\![\ \mathsf{F}\ ]\!]_2\ \mathsf{A}\ (\mu\ \mathsf{F}\ \mathsf{A}) \rightarrow \mu\ \mathsf{F}\ \mathsf{A}
\end{array}
$$

Fixed points in universe

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

```
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
```

Catamorphism

$$
\begin{aligned}
&\mathsf{cata} : \{\mathsf{Ix}\ \mathsf{Ox} : \mathsf{Set}\}\ \{\mathsf{F} : \mathsf{Code}_2\ \mathsf{Ix}\ \mathsf{Ox}\ \mathsf{Ox}\}\\
&\qquad\quad \{\mathsf{A} : \mathsf{Indexed}\ \mathsf{Ix}\}\ \{\mathsf{R} : \mathsf{Indexed}\ \mathsf{Ox}\} \rightarrow\\
&\qquad\quad (\llbracket\ \mathsf{F}\ \rrbracket_2\ \mathsf{A}\ \mathsf{R}\ \rightrightarrows\ \mathsf{R}) \rightarrow (\mu\ \mathsf{F}\ \mathsf{A}\ \rightrightarrows\ \mathsf{R})\\
&\mathsf{cata}\ \{\mathsf{F} = \mathsf{F}\}\ \phi\ \mathsf{i}\ \langle\ \mathsf{x}\ \rangle = \phi\ \mathsf{i}\ (\mathsf{bimap}\ \mathsf{F}\ \mathsf{id}_{\rightrightarrows}\ (\mathsf{cata}\ \phi)\ \mathsf{i}\ \mathsf{x})
\end{aligned}
$$

Special cases

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

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

$$\text{Multirec Ix} = \text{Code}_2\ (\text{Fin } 0)\ \text{Ix}\ \text{Ix}$$

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.

Advertisement

- 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.