Mathematical Structures for Data Types with Restricted Parametricity

Trends in Functional Programming, 12-14th June, St. Andrews

Dominic Orchard  
Alan Mycroft
Mathematically structured programming

- e.g. monoids, groups, functors, monads, comonads, monoidal functors (idioms), etc.

- Design pattern
  - Abstraction (detail hiding)
  - Generalisation (e.g. for all functors...)
  - Eases writing, reading, and reasoning
Parametric data types in Haskell as **functors**

```haskell
class Functor f where
    fmap :: (a → b) → f a → f b

instance [] where
    fmap = map
```

- *fmap* is polymorphic in *a, b* i.e. ∀ *a, b*
- *f* is a *parametric data type*, meaning
  
  ∀ *a* . *a* ∈ Type ⇒ *f a* ∈ Type
... but some data types are restricted in their parametricity

- e.g. sets implemented by balanced binary trees

\[
\forall a . \ a \in \text{Type} \land a \text{ is ordered} \\Rightarrow \text{Set } a \in \text{Type}
\]

- e.g. unboxed arrays of primitive (fixed size) types

\[
\forall a . \ a \in \text{Type} \land a \text{ is primitive} \\Rightarrow \text{UArray } a \in \text{Type}
\]

- Efficient data types often restricted
**Is Set a functor?**

\[ Set\text{.}map :: (Ord\ a,\ Ord\ b) \Rightarrow (a \rightarrow b) \rightarrow Set\ a \rightarrow Set\ b \]

\[ (\text{class } Ord\ a\ \text{where } \ldots ) \]

**class** \( \text{Functor } f \) **where**

\[ fmap :: (a \rightarrow b) \rightarrow f\ a \rightarrow f\ b \]

**instance** \( \text{Functor } Set \) **where**

\[ fmap = Set\text{.}map \]

*No instances for (Ord b, Ord a)*

*arising from a use of `Set.map'*

*In the expression: Set.map*

*In an equation for `fmap': fmap = Set.map*

*In the instance declaration for `Functor Set'*

\( fmap \) has no constraints i.e. polymorphic;

\( Set\text{.}map \) is constrained i.e. restricted polymorphic
This paper...

• Data types with restricted parametricity:
  ‣ usually real-world, efficient data types, **but**
  ‣ do not fit the language abstractions used;
  ‣ do not fit the mathematics used.

• This paper, for **functors**, **monads**, and **comonads**:
  ‣ fixes the mathematics;
  ‣ shows a language approach.

• Essential insight:

  *restricted parametricity* = *subcategories*
Parametric polymorphism

- Universal quantification \( \forall \alpha . \alpha \)

\[
fmap :: \forall a, b . (a \to b) \to (f\ a \to f\ b)
\]

- Uniform behaviour at any parameter type
**Restricted parametric polymorphism**

- Quantification over a subset of types
  \[ \forall \alpha . \alpha \in A \Rightarrow \alpha \]
- Described by *ad-hoc polymorphism*
  - Behaviour may vary at parameter type
- In Haskell, *type classes and class constraints*

\[
Set.map :: \forall a, b . (\text{Ord } a, \text{Ord } b) \Rightarrow (a \rightarrow b) \rightarrow (f \ a \rightarrow f \ b)
\]
What is the categorical interpretation of restricted parametricity?
Categorical interpretation of programs

- Programs make definitions in Hask
  - objects = types
  - morphisms (maps between objects) = functions

```
\[
\begin{align*}
id_{\text{Int}} & \quad \circ \quad f & \quad \circ \quad id_{\text{String}} \\
\text{Int} & \quad \leftarrow & \quad \text{String} \\
\downarrow g & \quad \quad & \quad \downarrow (g \circ f) \\
\text{Bool} & \quad \leftarrow & \quad \quad \text{....} \\
\end{align*}
\]
```
Functors

• Map between objects & morphisms of categories
Categorical interpretation: *Functor* class

- **Data type:**
  
  \[
  \text{data } F \ a = \ldots
  \]

  \(\forall a . \ a\)

  \(\forall a, b . \ a \to b\)

  \(\text{object-mapping} \text{ on Hask i.e. } F : |\text{Hask}| \to |\text{Hask}|\)

- **Instance of *Functor***

  \[
  \text{class } \text{Functor } f \text{ where }
  \]

  \[
  \text{fmap} :: (a \to b) \to f a \to f b
  \]

  \(\forall a \cdot a\)

  \(\forall a, b \cdot a \to b\)

  \(\text{Hask objects}\)

  \(\text{Hask morphisms}\)

  \(\text{morphism-mapping} \text{ on Hask via } f\text{map}\)

  \(\therefore \text{Functor describes } \text{endofunctors} \text{ on Hask}\)

  \(\uparrow\)

  (from a category back to the same category)
Categorical interpretation: *Functor* class

\[ \text{map} \ (g \circ f) \]

**Diagram:**
- **Hask**
  - **A**
  - **B**
  - **C**
  - **f**
  - **g**
  - **g \circ f**

**Hask**
- **[A]**
- **[B]**
- **[C]**
- **map f**
- **map g**

**Map:**
- **map**
- **(g \circ f)**
Categorical interpretation of class constraints

• Instances of a type class define a subset of all types

    class Eq a where
        (==) :: a → a → Bool
    instance Eq Int where ...
    instance Eq Bool where ...

    i.e. Eq ⊆ |Hask|

• A type class $C$ defines a subcategory of Hask
  - objects: $\forall a . C a \Rightarrow a$
  - morphisms: $\forall a, b . (C a, C b) \Rightarrow a \rightarrow b$
Set is a functor

\[ \text{Set.map} :: (\text{Ord } a, \text{Ord } b) \Rightarrow (a \rightarrow b) \rightarrow \text{Set } a \rightarrow \text{Set } b \]

maps morphisms of \textit{Ord}-subcategory

\[ \therefore \text{Set is a functor from Ord-subcategory to Hask} \]

\[ \text{Set} : \text{Ord} \rightarrow \text{Hask} \]

• but \textit{Functor} describes \textit{endofunctors} on Hask
  
  i.e. \[ F : \text{Hask} \rightarrow \text{Hask} \]

• Type error manifests the mathematical mismatch

• How then to describe this in Haskell?
Language solution: use constraint families*

```haskell
class ExoFunctor f where
  constraint SubCat f a
  fmap :: (SubCat f a, SubCat f b) => (a -> b) -> f a -> f b

instance ExoFunctor Set where
  constraint SubCat Set a = Ord a
  fmap = Set.map
```

* Allows constraint per instance of the Functor class

*[Orchard, Schrijvers, 2010]*
class ExoFunctor f where
   type SubCat f a :: Constraint
    fmap :: (SubCat f a, SubCat f b) ⇒
       (a → b) → f a → f b

*Allows constraint per instance of the Functor class*

instance ExoFunctor Set where
   type SubCat Set a = Ord a
    fmap = Set.map

*[implemented by Bolingbroke, 2011, http://blog.omega-prime.co.uk/?p=127]*
class ExoFunctor f where
  type SubCat f a b :: Constraint
  fmap :: (SubCat f a b) ⇒ (a → b) → f a → f b

- Allows more interesting structures
  e.g. ∀a,b . (Show a) ⇒ a → b

- Pre-proceeding: (non-full) subcategory of Hask with all Hask objects but only morphisms from Show objects

- Actually slightly more subtle (see post-proceeding)
Conclusion

- Slogan: restricted parametricity = subcategories
- Extended to relative monads (Altenkirch et. al) and relative comonads (with underlying non-endofunctor)
- All details in paper
- Provides elegant mathematical structuring to real-world, efficient data types

Thank you.

http://dorchard.co.uk
Additional slides
Ad-hoc polymorphism

- Restricted quantification \( \forall \alpha . \alpha \in A \Rightarrow \alpha \)
- Behaviour may vary at parameter type

\[
\text{eq} :: \text{Eq} \ t \Rightarrow t \rightarrow t \rightarrow \text{Bool} \quad \text{Haskell}
\]

\[
\text{eq} : \text{``t} \rightarrow \text{``t} \rightarrow \text{bool} \quad \text{SML}
\]

\[
\langle T \text{ extends Comparable} \rangle \text{ boolean eq}(T \ x, \ T \ y) \quad \text{Java}
\]