Mathematical Structures for Data Types with Restricted Parametricity

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

Dominic Orchard
Alan Mycroft

UNIVERSITY OF CAMBRIDGE

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***

$$\textbf{class}\ \mathit{Functor}\ f\ \textbf{where}$$
$$\quad \mathit{fmap} :: (a \to b) \to f\ a \to f\ b$$

$$\textbf{instance}\ [\,]\ \textbf{where}$$
$$\quad \mathit{fmap} = \mathrm{map}$$

- fmap is polymorphic in a, b i.e. $\forall a, b$
- f is a *parametric data type*, meaning

$$\forall a\ .\ a \in \mathit{Type} \Rightarrow f\ a \in \mathit{Type}$$

… but some data types are *restricted* in their parametricity

- e.g. sets implemented by balanced binary trees

i.e. $\forall a \;.\; a \in Type \wedge \underline{a \ is \ ordered}$

$\Rightarrow Set \ a \in Type$

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

i.e. $\forall a \;.\; a \in Type \wedge \underline{a \ is \ primitive}$

$\Rightarrow UArray \ a \in Type$

- Efficient data types often *restricted*

Is *Set* a functor?

$$Set.map :: (Ord\ a,\ Ord\ b) \Rightarrow (a \to b) \to Set\ a \to Set\ b$$

(**class** $Ord\ a$ **where** ...)

class $Functor\ f$ **where**
  $fmap :: (a \to b) \to f\ a \to f\ b$

instance $Functor\ Set$ **where**
  $fmap = Set.map$

$fmap$ has no constraints i.e. polymorphic; $Set.map$ is constrained i.e. restricted polymorphic

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'

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 `∀α . α`

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

- Uniform behaviour at any parameter type

Restricted *parametric polymorphism*

- Quantification over a subset of types

$$\forall \alpha \ . \ \alpha \in \mathbf{A} \Rightarrow \alpha$$

- Described by *ad-hoc polymorphism*
  - Behaviour may vary at parameter type

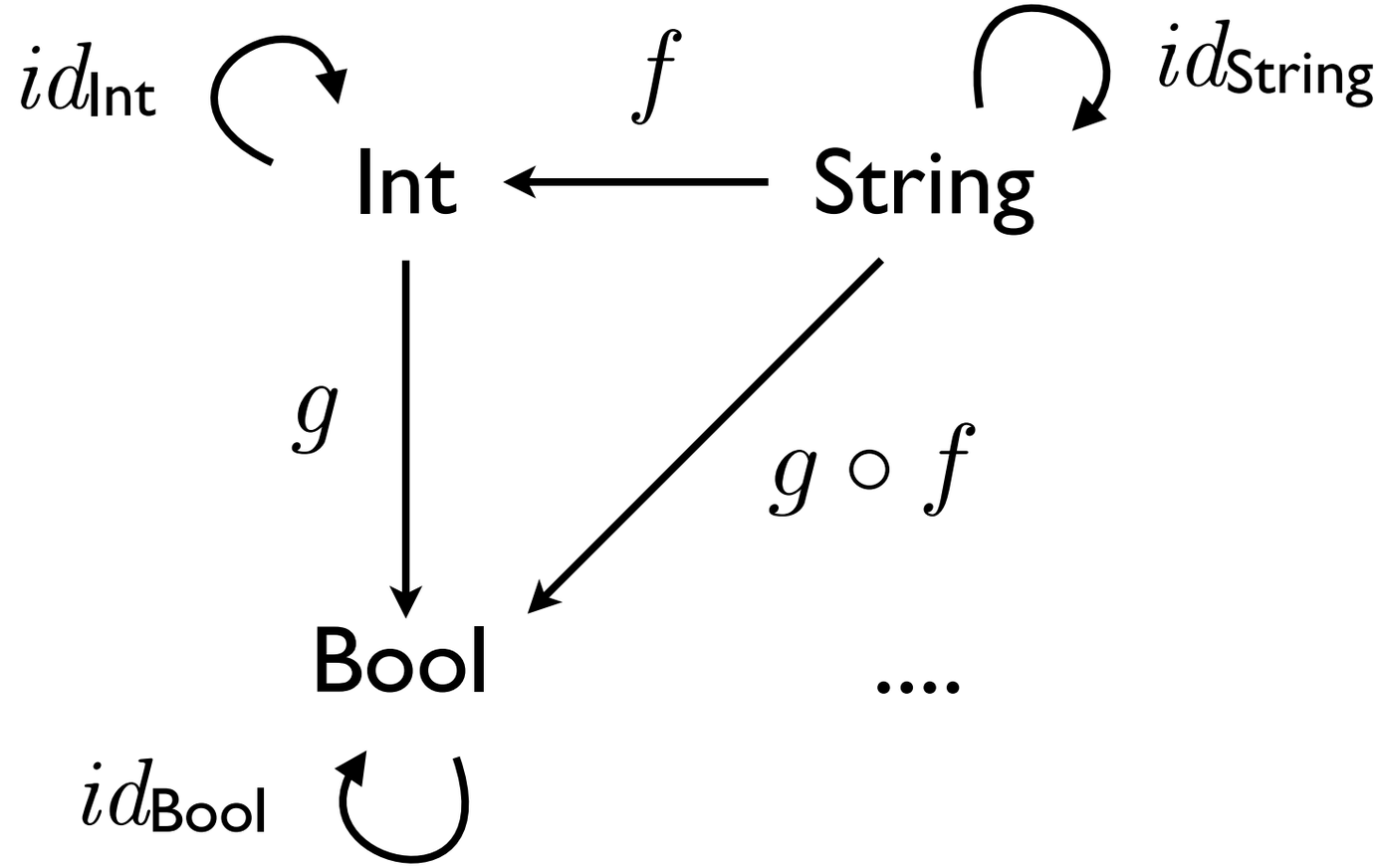
- In Haskell, *type classes* and *class constraints*

$$Set.map :: \forall a,b \ . \ (Ord \ a, \ Ord \ b) \Rightarrow (a \rightarrow b) \rightarrow (f \ a \rightarrow f \ b)$$

What is the categorical interpretation of restricted parametricity?

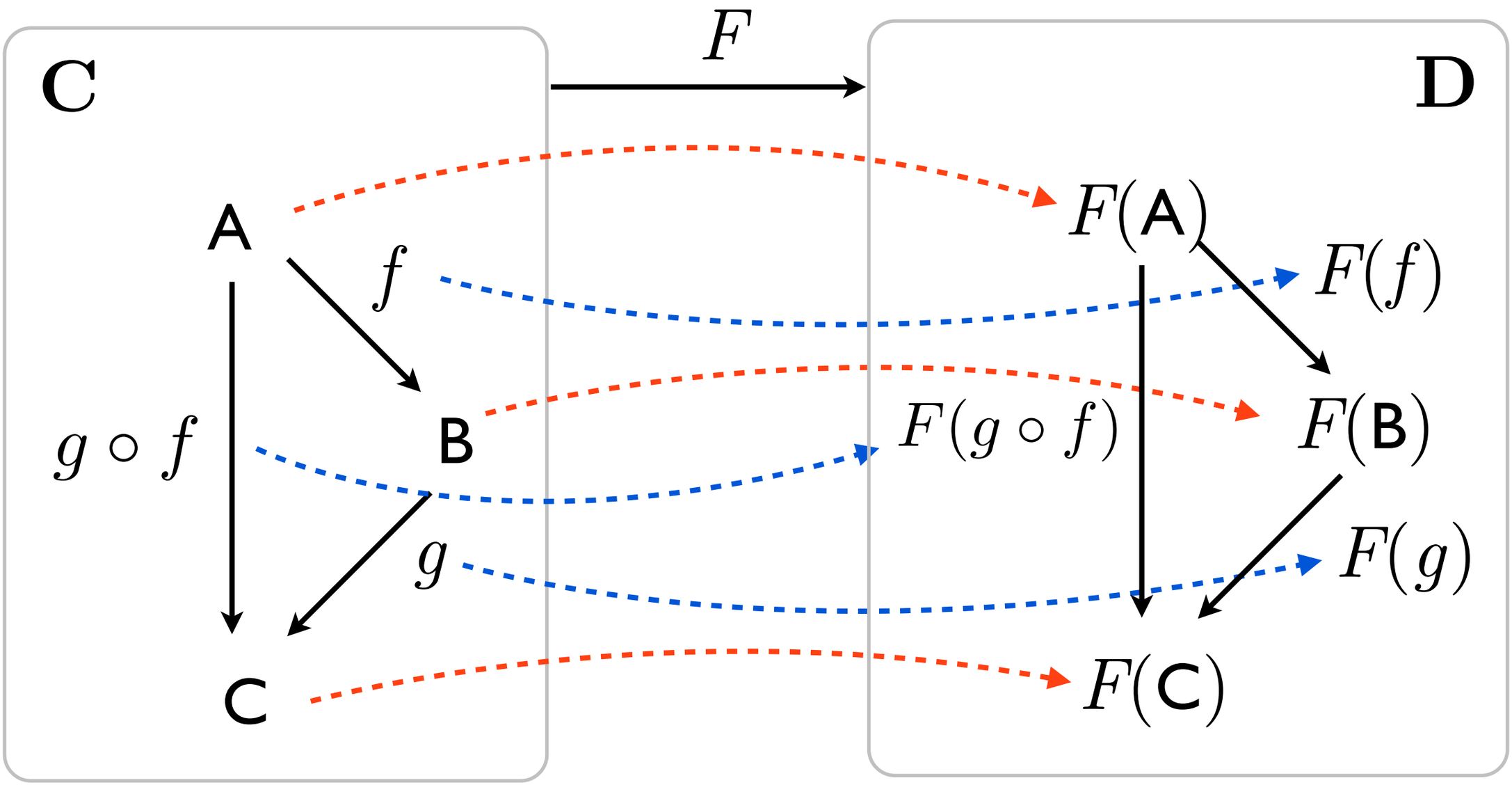
Categorical interpretation of programs

- Programs make definitions in $\mathbf{Hask}$
  - *objects* = types
  - *morphisms* (maps between objects) = functions



Functors

- Map between objects & morphisms of categories



Categorical interpretation: *Functor* class

- Data type: $\mathbf{data}\ F\ a = \ldots$
  - *object-mapping* on $\mathbf{Hask}$ i.e. $F : |\mathbf{Hask}| \to |\mathbf{Hask}|$
- Instance of *Functor*

$$\mathbf{class}\ Functor\ f\ \mathbf{where}$$
$$fmap :: (a \to b) \to f\ a \to f\ b$$

$\forall a\ .\ a$ — $\mathbf{Hask}$ objects

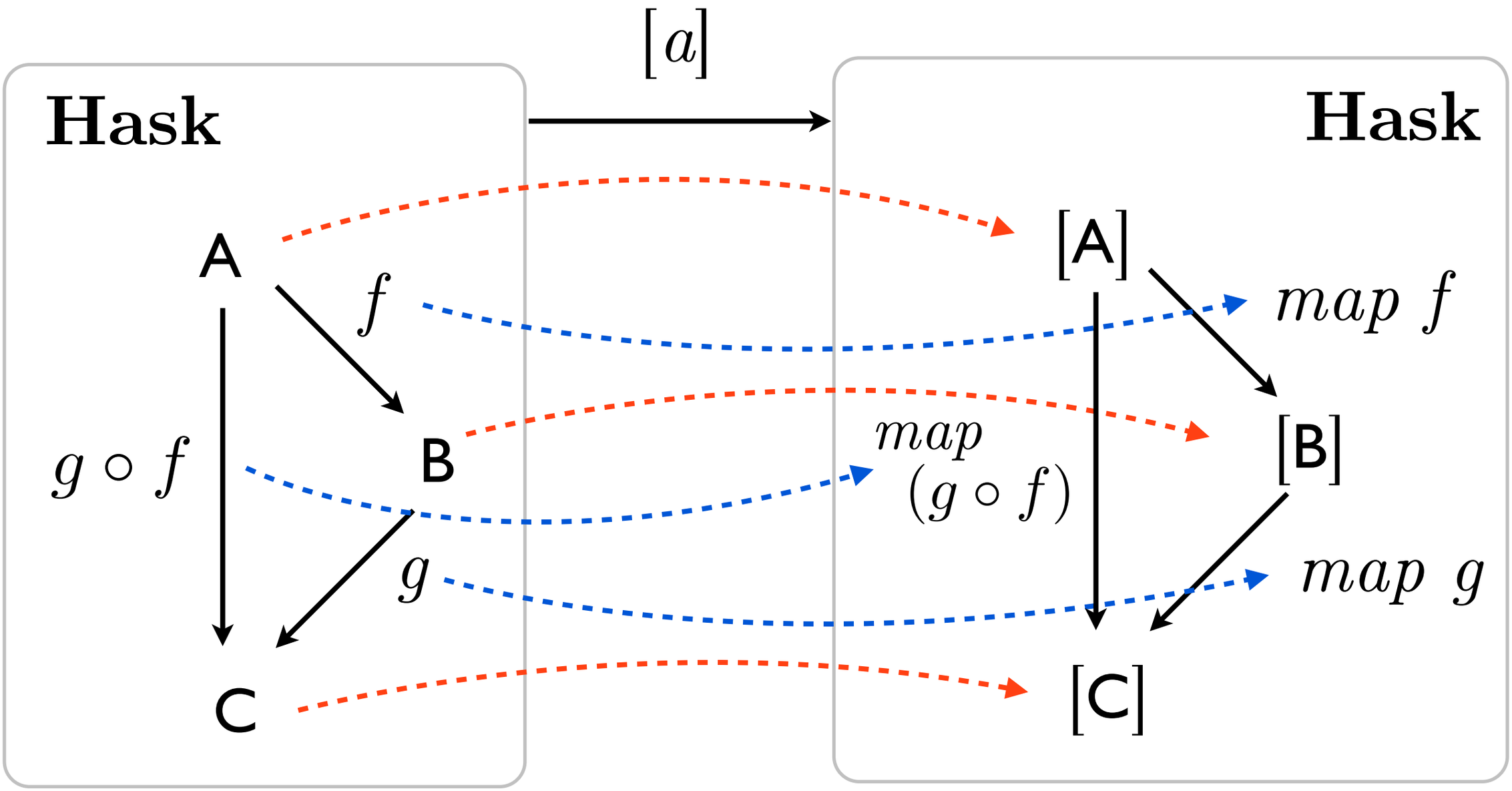
$\forall a,b\ .\ a \to b$ — $\mathbf{Hask}$ morphisms

  - *morphism-mapping* on $\mathbf{Hask}$ via $fmap$

$\therefore$ *Functor* describes *endofunctors* on $\mathbf{Hask}$

(from a category back to the same category)

Categorical interpretation: *Functor* class



Categorical interpretation of *class constraints*

- Instances of a type class define a *subset* of all types

$$\textbf{class } Eq\ a \textbf{ where}$$
$$\quad (==) :: a \to a \to Bool$$
$$\textbf{instance } Eq\ Int \textbf{ where } \ldots$$
$$\textbf{instance } Eq\ Bool \textbf{ where } \ldots$$

i.e. $Eq \subseteq |\mathbf{Hask}|$

- A type class C defines a *subcategory* of $\mathbf{Hask}$
  - objects: $\forall a\ .\ C\ a \Rightarrow a$
  - morphisms: $\forall a,b\ .\ (C\ a,\ C\ b) \Rightarrow a \to b$

Set **is** a functor

$$Set.map :: (Ord\ a,\ Ord\ b) \Rrightarrow (a \to b) \to Set\ a \to Set\ b$$

maps morphisms of Ord-subcategory

$\therefore$ Set is a functor from Ord-*subcategory* to $\mathbf{Hask}$

$$Set : Ord \to \mathbf{Hask}$$

- but $Functor$ describes *endofunctors* on $\mathbf{Hask}$

  i.e. $F : \mathbf{Hask} \to \mathbf{Hask}$

- Type error manifests the mathematical mismatch
- How then to describe this in Haskell?

Language solution: use *constraint families**

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

*[Orchard, Schrijvers, 2010]

Language solution (current): *constraint-kinded type families**

```
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]

Generalisation...

$$\textbf{class } \mathit{ExoFunctor}\ f \textbf{ where}$$
$$\quad \textbf{type } \mathit{SubCat}\ f\ a\ \mathrm{b} \textbf{ :: Constraint}$$
$$\quad \mathit{fmap} :: (\mathit{SubCat}\ f\ a\ b) \Rightarrow (a \to b) \to f\ a \to f\ b$$

- Allows more interesting structures

  e.g. $\forall a,b\ .\ (\mathit{Show}\ a) \Rightarrow a \to b$

- Pre-proceeding: (non-full) subcategory of $\mathbf{Hask}$ with all $\mathbf{Hask}$ 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 $\forall \alpha \;.\; \alpha \in \mathbf{A} \Rightarrow \alpha$
- Behaviour may vary at parameter type

$$eq :: Eq\ \mathrm{t} \Rightarrow t \to t \to Bool$$ Haskell

$$eq : ``t \to ``t \to bool$$ SML

$$\langle T\ \mathbf{extends}\ Comparable \rangle\ boolean\ eq(T\ x,\ T\ y)$$ Java