Haskell Type Constraints Unleashed (talk)

Dominic Orchard
University of Cambridge, UK
dominic.orchard@cl.cam.ac.uk

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

FLOPS 2010, *Sendai, Japan*
Wednesday April 21, 2010

Type terms in Haskell

$$C \Rightarrow \tau$$

type constraints   types

Framework of Current GHC/Haskell Type System Features

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where* ...
	synonym	Type synonyms **type** $T\ \bar{a} \ = \ \tau$	
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau} \ = \ \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau} \ = \ \tau$	

Haskell 98 Type System Features

$$C \Rightarrow \tau$$

Type classes

Data types
Type synonyms

Type classes

$$\texttt{class}\ \mathsf{Show}\ a\ \texttt{where}$$
$$show :: a \to \mathsf{String}$$

$$\texttt{instance}\ \mathsf{Show}\ \mathsf{Bool}\ \texttt{where}$$
$$show\ \mathsf{True} = \text{“True”}$$
$$show\ \mathsf{False} = \text{“False”}$$

$$show :: \mathsf{Show}\ a \Rightarrow a \to \mathsf{String}$$

Haskell 98 + GHC Type System Features

$$C \Rightarrow \tau$$

Type classes

Data types (GADTs)
Type synonyms
Type synonym families
Data type families

Type families

- Type-level function : $\tau \rightarrow \tau$, *or* type-indexed set of types
- Defined as rewrite rules from types to types

```
type family Collection e
type instance Collection Int = [Int]
type instance Collection Bool = Int
```

e.g. type `Collection Int` is a synonym for `[Int]`

Type families allow method signature flexibility

```
type family Collection e
type instance Collection Int = [Int]
type instance Collection Bool = Int

class CollectionElem e where
        insert :: e -> Collection e -> Collection e
        ...
```

e.g.

```
insert :: Int -> Collection Int -> Collection Int
```

Type families allow method signature flexibility

```
type family Collection e
type instance Collection Int = [Int]
type instance Collection Bool = Int

class CollectionElem e where
        insert :: e -> Collection e -> Collection e
        ...
```

e.g.

```
insert :: Int -> [Int] -> [Int]
```

Classes and families are *open*

- Declaration separate to definition e.g.

```
type family Collection e
class CollectionElem e where ...
```

- Definition with *instances*, possibly across files e.g.

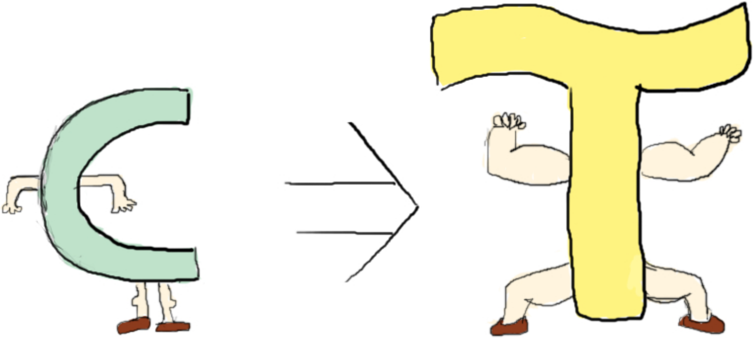
```
type instance Collection Int = [Int]
instance CollectionElem Int where ...
```

Under-development of Constraint Term Language

$$C \Rightarrow \tau$$

Type classes
(Equality constraints)

Data types
Type synonyms
Type synonym families
Data type families



Haskell Type Constraints ~~Unleashed~~ Beefed-Up!

Dominic Orchard
University of Cambridge, UK
dominic.orchard@cl.cam.ac.uk

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

FLOPS 2010, *Sendai, Japan*
Wednesday April 21, 2010

Example 1: Set functor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Set where
    fmap = Set.map
```

$\not\vdash$

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

- `fmap` is **fixed** with no constraints

Example 2: Polymorphic EDSL

```
class Expr sem where
    constant :: a -> sem a
    add      :: sem a -> sem a -> sem a
```

e.g. `(constant 1) ‘add‘ (constant 2)` could denote $1 + 2$.

```
data NumSemantics a = MkNum a
```

```
instance Expr NumSemantics where
    constant c = MkNum c
    add (MkNum e1) (MkNum e2)  = MkNum (e1 + e2)
```
$\not\vdash$

$(+) :: \text{Num } a \Rightarrow a \rightarrow a \rightarrow a$

Example 2: **Less**-Polymorphic EDSL

```
class Expr sem where
    constant :: a -> sem a
    add      :: Num a => sem a -> sem a -> sem a
```

e.g. `(constant 1) ‘add‘ (constant 2)` could denote $1 + 2$.

```
data NumSemantics a = MkNum a
```

```
instance Expr NumSemantics where
    constant c = MkNum c
    add (MkNum e1) (MkNum e2)  = MkNum (e1 + e2)
```

✔

$(+) :: \mathsf{Num}\ a \Rightarrow a \longrightarrow a \longrightarrow a$

Type families allow method signature flexibility

```
type family Collection e

class CollectionElem c where
    insert :: e -> Collection e -> Collection e
    ...
```

- Need constraint flexibility.
- Our solution: type indexed-families of constraints.

Our approach: extend constraint term features by analogy with type term features

Axis: Types and Constraints

types	*constraints*

Features: Datatypes and synonyms

types	*constraints*
Data types	
data $T\ \bar{a}$ *where* ...	
Type synonyms	
type $T\ \bar{a} = \tau$	

Axis: Generative and Synonyms

	types	*constraints*
generative	Data types **data** $T\ \bar{a}$ *where* ...	
	Type synonyms **type** $T\ \bar{a} = \tau$	

Axis: Generative and Synonyms

	types	*constraints*
generative	Data types **data** $T\ \bar{a}$ *where* ...	
synonym	Type synonyms **type** $T\ \bar{a} = \tau$	

Axis: Classes

	types	*constraints*
generative	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	

Axis: Constants and Families

		types	*constraints*
	generative	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	
families			

Axis: Constants and Families

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	
families			

Axis: Families

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a} = \tau$	
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau} = \ldots$	

Axis: Families

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau}\ =\ \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau}\ =\ \tau$	

Constraint Synonyms

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	Constraint synonyms
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau}\ =\ \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau}\ =\ \tau$	

Constraint Synonyms

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	Constraint synonyms **constraint** $K\ \bar{a}\ =\ C$
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau}\ =\ \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau}\ =\ \tau$	

Example (without constraint synonyms)

```
eval :: (Solver s, Queue q, Transformer t,
        Elem q ~ (Label s, Tree s a, TreeState t),
        ForSolver t ~ s)
        => ...

eval' :: (Solver s, Queue q, Transformer t,
         Elem q ~ (Label s, Tree s a, TreeState t),
         ForSolver t ~ s)
         => ...
```

Example (with constraint synonyms)

```
constraint Eval s q t a =
   (Solver s, Queue q, Transformer t,
          Elem q ~ (Label s, Tree s a, TreeState t),
          ForSolver t ~ s)

eval :: Eval s q t a => ...
eval' :: Eval s q t a => ...
```

Constraint Synonym Families

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a} = \tau$	Constraint synonyms **constraint** $K\ \bar{a} = C$
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau} = \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau} = \tau$	

Constraint Synonym Families

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a} = \tau$	Constraint synonyms **constraint** $K\ \bar{a} = C$
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau} = \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau} = \tau$	Constraint synonym families

Constraint Synonym Families

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where* ...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	Constraint synonyms **constraint** $K\ \bar{a}\ =\ C$
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau}\ =\ \ldots$	
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau}\ =\ \tau$	Constraint synonym families **constraint family** $K\ \bar{a}$ **constraint instance** $K\ \bar{\tau}\ =\ C$

Constraint Synonym Families

Type synonym family syntax:

$$\texttt{type family}\ T\ \bar{a}$$

$$\texttt{type instance}\ T\ \bar{\tau} = \tau$$

Analogous constraint synonym family syntax:

$$\texttt{constraint family}\ K\ \bar{a}$$

$$\texttt{constraint instance}\ K\ \bar{\tau} = C$$

Example 1: Set functor

```
class Functor f where
    fmap :: (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Set where
    fmap = Set.map
```

$\nvdash$

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

Example 1: Set functor - Solution 1

```
constraint family Inv f e
constraint instance Inv [] e = ()
constraint instance Inv Set e = Ord e

class Functor f where
    fmap :: (Inv f a, Inv f b) => (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map

instance Functor Set where
    fmap = Set.map
```

✔

Example 1: Set functor - Solution 2 (Associated)

```
class Functor f where
    constraint Inv f e
    fmap :: (Inv f a, Inv f b) => (a -> b) -> f a -> f b

instance Functor [] where
    constraint Inv [] e = ()
    fmap = map

instance Functor Set where
    constraint Inv Set e = Ord e
    fmap = Set.map
```

✔

Example 1: Set functor - Solution 3 (+ default)

```
class Functor f where
    constraint Inv f e = ()
    fmap :: (Inv f a, Inv f b) => (a -> b) -> f a -> f b

instance Functor [] where
    fmap = map                            <- note: unchanged!

instance Functor Set where
    constraint Inv Set e = Ord e
    fmap = Set.map
```

✔

Example 2: Polymorphic EDSL

```
class Expr sem where
    constant :: a -> sem a
    add      :: sem a -> sem a -> sem a

data NumSemantics a = MkNum a

instance Expr NumSemantics where
    constant c = MkNum c
    add (MkNum e1) (MkNum e2)  = MkNum (e1 + e2)
```

$\nvdash$

$$(+) :: \mathsf{Num}\ a \Rightarrow a \rightarrow a \rightarrow a$$

Example 2: Polymorphic EDSL

```
class Expr sem where
      constraint Pre sem a = ()
      constant :: Pre sem a => a -> sem a
      add      :: Pre sem a => sem a -> sem a -> sem a

data NumSemantics a = MkNum a

instance Expr NumSemantics where
      constraint Pre NumSemantics a = Num a
      constant c = MkNum c
      add (MkNum e1) (MkNum e2)  = MkNum (e1 + e2)
```

✔

Well-defined Families

- Confluence
  - No overlapping instances
  - No type-families application in instance heads
- Termination

Termination

(Based on type family termination [1])

`constraint instance K` $\bar{\tau}$ `= C`

$\forall$ constraint family applications `K'` $\bar{\tau}'$ $\in$ `C`:

1. $|\bar{\tau}| > |\bar{\tau}'|$ (strictly decreasing)
2. $\bar{\tau}'$ has no more occurences of any type variable than τ

   repetition of a type variable = possible term size increase (multiplication)
3. $\bar{\tau}'$ does not contain any type family applications

[1] Schrijvers, T., Jones, S.P., Chakravarty, M., Sulzmann, M.: Type checking with open type functions. SIGPLAN Not. 43(9) (2008) 5162

State of play

		types	*constraints*
constants	*generative*	Data types **data** $T\ \bar{a}$ *where* ...	Classes **class** $K\ \bar{a}$ *where*...
	synonym	Type synonyms **type** $T\ \bar{a}\ =\ \tau$	Constraint synonyms **constraint** $K\ \bar{a}\ =\ C$
families	*generative*	Data type families **data family** $T\ \bar{a}$ **data instance** $T\ \bar{\tau}\ =\ \ldots$	Class families?
	synonym	Type synonym families **type family** $T\ \bar{a}$ **type instance** $T\ \bar{\tau}\ =\ \tau$	Constraint synonym families **constraint family** $K\ \bar{a}$ **constraint instance** $K\ \bar{\tau}\ =\ C$

Paper contributions

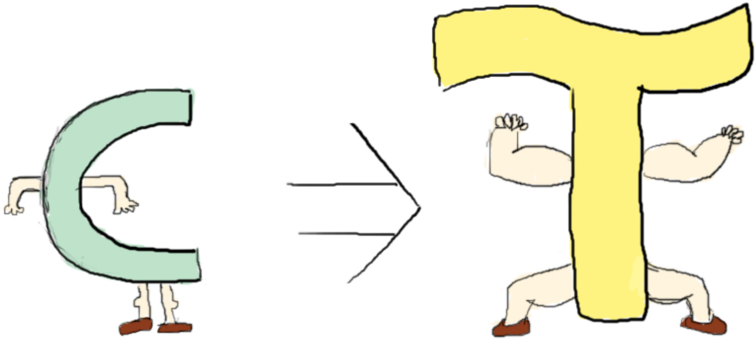
- Constraint synonyms and constraint synonym families
- Static semantics rules
- Termination conditions for constraint families (and interaction with classes)
- Provide encodings into GHC/Haskell and System F_C

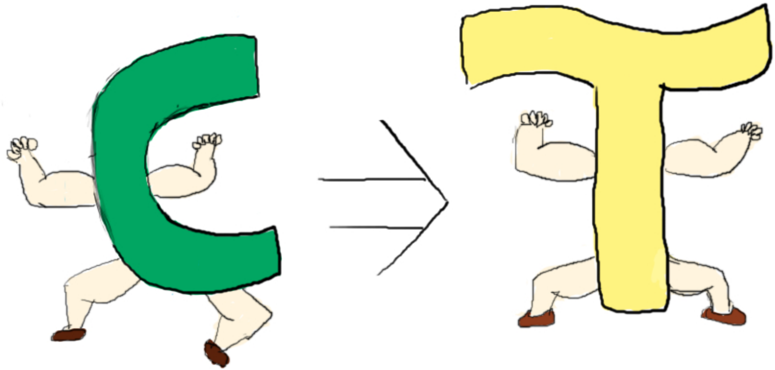
Further work

- Implement in GHC
- Class families?
- Improving refactoring with class synonym instances
- Open vs. closed as an axis

Conclusions

- Symmetrised the type system along the constraint-type divide
- Constraint synonyms: Easier to read/write code
- Constraint synonym families: Index constraints based on types
- Fixes many problems (functor problem)
- Polymorphic EDSLs can be polymorphic again, even with constraints!





Haskell Type Constraints Unleashed!

Back-up slides

Confluence

No type family application in instance heads

```
   constraint family K f a
① constraint instance K [] a = ()
② constraint instance K  (T a)  a = Ord a

   type instance T Char = BitSet        ✔

   type instance T Int = []             ✘
```

Given constraint $K\ [\,]\ \mathsf{Int}$:

$$
\begin{array}{lll}
① & K\ [\,]\ \mathsf{Int} & = () \\
② & K\ (T\ \mathsf{Int})\ \mathsf{Int} & \rightarrow\ K\ [\,]\ \mathsf{Int} \\
 & & =\ \mathsf{Ord\ Int} \\
 & & \neq ()
\end{array}
$$

No type family application in instance heads

```
   constraint family K f a
① constraint instance K [] a = ()
② constraint instance K (T a) a = Ord a

   type instance T Char = BitSet        ✓

   type instance T Int = []             ✗
```

Given constraint $K\ []\ \mathsf{Int}$:

$$\begin{array}{lll} ① & K\ []\ \mathsf{Int} & = () \\ ② & K\ (T\ \mathsf{Int})\ \mathsf{Int} & \rightarrow\ K\ []\ \mathsf{Int} \\ & & =\ \mathsf{Ord\ Int} \\ & & \neq () \end{array}$$

Termination (condition 2)

2. $\bar{\tau}'$ has no more occurences of any type variable than LHS

e.g.

```
constraint family K m a
constraint instance K [a] b = K b b
```

if `b = [a]`

`K [a] b` $\longrightarrow$ `K [a] [a]` $\longrightarrow$ `K [a] [a]` $\longrightarrow$ `...`

Termination (condition 3)

3. $\bar{\tau}'$ does not contain any type family applications

```
    constraint family K m a
①   constraint instance K (T (T m)) a = K (F m) a
```

$|$ `(T (T m)) a` $| = 4$ $|$ `(F m) a` $| = 3$

```
    type family F m
②   type instance F Int = T (T Int)
```

`K (T (T Int)) a` $\xrightarrow{①}$ `K (F Int) a` $\xrightarrow{②}$ `K (T (T Int)) a` $\xrightarrow{①}$ ...