Haskell Type Constraints Unleashed

Dominic Orchard
University of Cambridge

Tom Schrijvers
KU Leuven

Fun in the Afternoon, Standard Chartered.
February 17, 2010

$$C \Rightarrow \tau$$

Type classes

Data types
Type synonyms
Type synonym families

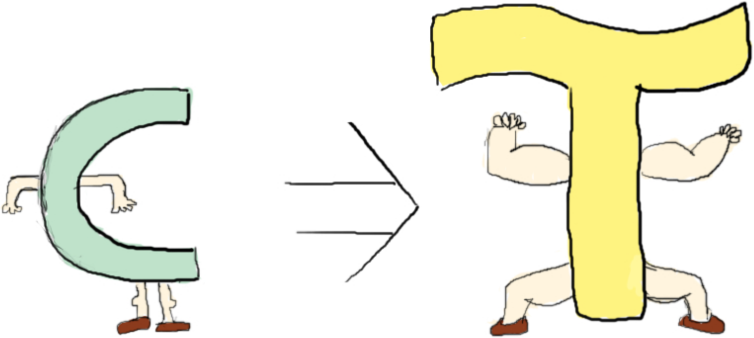
Reminder: Type families

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

e.g. `Collection Int` → `[Int]`

$$C \Rightarrow \tau$$

Type classes (Equality constraints)	Data types (GADTs) Type synonyms Type synonym families Data type families



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

Dominic Orchard
University of Cambridge

Tom Schrijvers
KU Leuven

Fun in the Afternoon, Standard Chartered.
February 17, 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
```

$\nvdash$

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

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 E a = MkE {eval :: a}
```

```
instance Expr E where
    constant c = MkE c
    add e1 e2  = MkE $ eval e1 + eval e2
```

$\nvdash$

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

Example 2: 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 E a = MkE {eval :: a}
```

```
instance Expr E where
    constant c = MkE c
    add e1 e2  = MkE $ eval e1 + eval e2
```

✔

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

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

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

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

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 E a = MkE {eval :: a}

instance Expr E where
    constant c = MkE c
    add e1 e2  = MkE $ eval e1 + eval 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 E a = MkE eval :: a

instance Expr E where
    constraint Pre E a = Num a
    constant c = MkE c
    add e1 e2  = MkE $ eval e1 + eval e2
```

✓

Well-defined Families

- Confluence
  - No overlapping instances
  - No type-families 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 $K \; [\,] \; \mathsf{Int}$:

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

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 LHS

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

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

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

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` $\xrightarrow{①}$ ...

Termination

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

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
- Improving refactoring with class synonym instances
- Open vs. closed as an axis

Improving refactoring with class synonym instances

```
constraint Num a
    = (Additive a, Multiplicative a, FromInteger a)
```

```
instance Num Int where ...
```

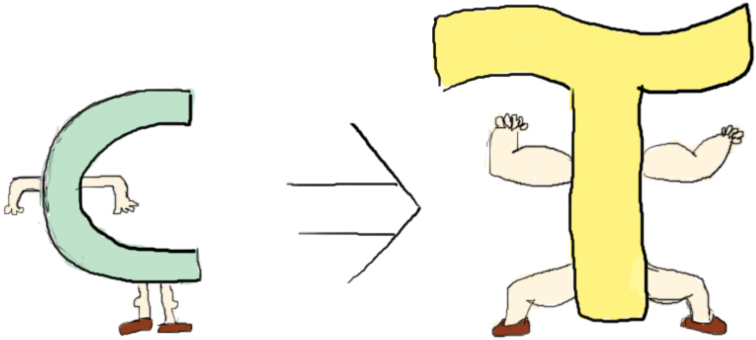
but

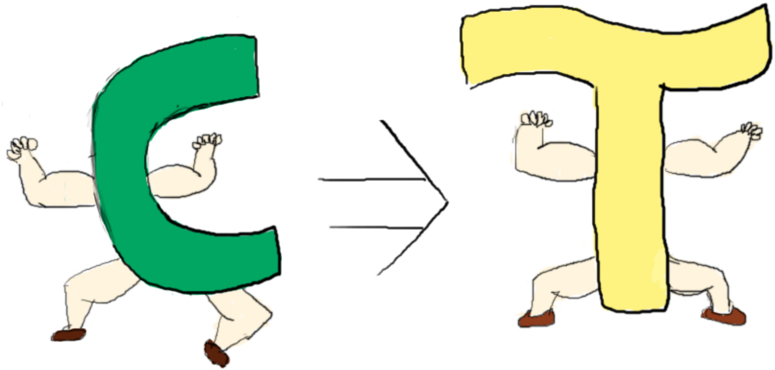
```
constraint Eq' a b = (Eq a, Eq b)
instance Eq' [Int] [Char] where
    x (==) y = ...
    x (==) y = ...
```

which (==) implementation is which?

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 Beefed-Up!