Type Families with Class, Type Classes with Family

Alejandro Serrano[1] Jurriaan Hage[1] Patrick Bahr[2]

[1]Utrecht University

[2]IT University of Copenhagen

Haskell Symposium 2015

Motivation

Type-level programming in Haskell/GHC

- functional dependencies
- type families
- data type promotion, kind polymorphism
- closed type families

Motivation

Type-level programming in Haskell/GHC

- functional dependencies
- type families
- data type promotion, kind polymorphism
- closed type families

Goal

- use type families to simulate type classes
- gain flexibility and expressiveness

Overview

- Simulate type classes using type families (without class methods)
- What can we do with this encoding?
- Proposal: type families with elaboration

Encoding type classes

Simplification (for now): type class = predicate on types

Encoding type classes

Simplification (for now): type class = predicate on types

class $C\ t_1 \ldots t_n \quad \rightsquigarrow \quad IsC :: \kappa_1 \to \ldots \to \kappa_n \to Bool$

Encoding type classes

Simplification (for now): type class = predicate on types

$$\textbf{class } C\ t_1 \ldots t_n \quad \rightsquigarrow \quad \mathit{IsC} :: \kappa_1 \to \ldots \to \kappa_n \to \mathit{Bool}$$

Instead of *Bool* we use:

$$\textbf{data } \mathit{Defined} = \mathit{Yes} \mid \mathit{No}$$

Encoding type classes

Simplification (for now): type class = predicate on types

class $C\ t_1 \ldots t_n \quad \rightsquigarrow \quad \mathit{IsC} :: \kappa_1 \to \ldots \to \kappa_n \to \mathit{Defined}$

Instead of *Bool* we use:

data $\mathit{Defined} = \mathit{Yes} \mid \mathit{No}$

Encoding type classes

Simplification (for now): type class = predicate on types

class $C\ t_1\ ...\ t_n \quad \rightsquigarrow \quad IsC :: \kappa_1 \rightarrow ... \rightarrow \kappa_n \rightarrow Defined$

Instead of *Bool* we use:

data $Defined = Yes \mid No$

Type family declaration

type family $IsC\ t_1\ ...\ t_n :: Defined$

Example: *Eq*

Type class declaration

class *Eq a* **where**
  $(\equiv) :: a \to a \to Bool$

Example: *Eq*

Type class declaration

class *Eq a* **where**

Example: *Eq*

Type class declaration

class *Eq a* **where**

Translation into type family declaration

type family *IsEq* $(a :: *) ::$ *Defined*

Example: *Eq*

Type class declaration

class *Eq a* **where**

Translation into type family declaration

type family *IsEq* $(a :: *) ::$ *Defined*

Usage

$f ::$ *Eq a* $\Rightarrow a \to a$

Example: *Eq*

Type class declaration

class *Eq a* **where**

Translation into type family declaration

type family *IsEq* $(a :: *) ::$ *Defined*

Usage

$f ::$ *IsEq a* $\sim$ *Yes* $\Rightarrow a \rightarrow a$

Example: *Eq*

Type class declaration

class *Eq a* **where**

Translation into type family declaration

type family *IsEq* $(a :: *) ::$ *Defined*

Usage

type *Eq a* $=$ *IsEq a* $\sim$ *Yes*

$f ::$ *IsEq a* $\sim$ *Yes* $\Rightarrow a \rightarrow a$

Example: *Eq*

Type class declaration

class *Eq a* **where**

Translation into type family declaration

type family *IsEq* $(a :: *) ::$ *Defined*

Usage

type *Eq a* $=$ *IsEq a* $\sim$ *Yes*

$f ::$ *Eq a* $\Rightarrow a \to a$

Example: *Eq*

Type class instance declarations

instance *Eq Int* **where** ...

instance $Eq\ a \Rightarrow Eq\ [a]$ **where** ...

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)$ **where** ...

Example: *Eq*

Type class instance declarations

instance *Eq Int* **where** ...

instance *Eq a* $\Rightarrow$ *Eq* $[a]$ **where** ...

instance $(Eq\ a, Eq\ b) \Rightarrow Eq\ (a, b)$ **where** ...

Type family instance declarations

type instance *IsEq Int* $=$ *Yes*

type instance *IsEq* $[a]$ $=$ *IsEq a*

type instance *IsEq* $(a, b) =$ *And* $(IsEq\ a)\ (IsEq\ b)$

Example: *Eq*

Type class instance declarations

instance *Eq Int* **where** ...

instance *Eq a* $\Rightarrow$ *Eq* [*a*] **where** ...

instance (*Eq a*, *Eq b*) $\Rightarrow$ *Eq* (*a*, *b*) **where** ...

Type fam

type

type instance *IsEq* [*a*] = *IsEq*

type instance *IsEq* (*a*, *b*) = *And* (*IsEq a*) (*IsEq b*)

type family *And* (*x* :: *Defined*) (*y* :: *Defined*) :: *Defined* **where**
  And Yes Yes = *Yes*
  And x y = *No*

What Can We Express

- super classes
- limited forms functional dependencies (requires injective type families)
- not supported: overlapping instances

What else can we do with this encoding?

What else can we do with this encoding?

- type class directives
- custom error messages
- instance chains

Non-membership

type instance $\mathit{IsShow}\ (a \to b) = \mathit{No}$

Non-membership

type instance $\mathit{IsShow}\ (a \to b) = \mathit{No}$

```
Couldn't match type 'No with 'Yes
Expected type: 'Yes
  Actual type: IsShow (t -> t)
```

Closed set of instances

```
type family IsIntegral t where
  IsIntegral Int     = Yes
  IsIntegral Integer = Yes
  IsIntegral t       = No
```

Disjointness

data *IntegralOrFractional* = *Integral* | *Fractional* | *None*

type family *IsIntegralOrFractional t* :: *IntegralOrFractional*

Disjointness

```
data IntegralOrFractional = Integral | Fractional | None

type family IsIntegralOrFractional t :: IntegralOrFractional

type instance IsIntegralOrFractional Int     = Integral
type instance IsIntegralOrFractional Integer = Integral
type instance IsIntegralOrFractional Double  = Fractional
```

Disjointness

```
data IntegralOrFractional = Integral | Fractional | None

type family IsIntegralOrFractional t :: IntegralOrFractional

type instance IsIntegralOrFractional Int     = Integral
type instance IsIntegralOrFractional Integer = Integral
type instance IsIntegralOrFractional Double  = Fractional

type IsIntegral t = IsIntegral' (IsIntegralOrFractional t)
type family IsIntegral' c :: Defined where
  IsIntegral' Integral = Yes
  IsIntegral' c        = No
```

Custom Error Messages

$$\textbf{data}\ \mathit{Defined} \quad = \mathit{Yes} \mid \mathit{No}$$

Custom Error Messages

$$\textbf{data}\ \mathit{Defined}\ e = \mathit{Yes} \mid \mathit{No}\ e$$

Custom Error Messages

```
data Defined e = Yes | No e
```

Example

```
type IsIntegral t = IsIntegral' (IsIntegralOrFractional t)

type family IsIntegral' c :: Defined Symbol where
  IsIntegral' Integral   = Yes
  IsIntegral' Fractional = No "is instance of Factional"
  IsIntegral' c          = No ""
```

Custom Error Messages

$$\textbf{data}\ \mathit{Defined}\ e = \mathit{Yes} \mid \mathit{No}\ e$$

Example

$$\textbf{type}\ \mathit{IsIntegral}\ t = \mathit{IsIntegral'}\ (\mathit{IsIntegralOrFractional}\ t)$$

$$\begin{aligned}
&\textbf{type family}\ \mathit{IsIntegral'}\ c :: \mathit{Defined}\ \mathit{Symbol}\ \textbf{where}\\
&\quad \mathit{IsIntegral'}\ \mathit{Integral} \quad = \mathit{Yes}\\
&\quad \mathit{IsIntegral'}\ \mathit{Fractional} = \mathit{No}\ \texttt{"is instance of Factional"}\\
&\quad \mathit{IsIntegral'}\ c \qquad\quad = \mathit{No}\ \texttt{""}
\end{aligned}$$

```
Couldn't match type 'No "is instance of Fractional"
    with 'Yes
Expected type: 'Yes
  Actual type: IsIntegral Double
```

Instance Chains [Morris & Jones]

- explicit failure
- provide alternatives (**else** clause)

Instance Chains [Morris & Jones]

- explicit failure
- provide alternatives (**else** clause)

Example

instance $Show\ (a \to b)$ **if** $(Enum\ a, Show\ a, Show\ b)$ **where**
  $show = ...$
else instance $Show\ (a \to b)$ **fails**

Instance Chains [Morris & Jones]

- explicit failure
- provide alternatives (**else** clause)

Example

instance $Show\ (a \to b)$ **if** $(Enum\ a, Show\ a, Show\ b)$ **where**
  $show = ...$
else instance $Show\ (a \to b)$ **fails**

As type family

type instance $IsShow\ (a \to b) = IsShowFn\ a\ b$
type family $IsShowFn\ a\ b$
  $= And_3\ (IsEnum\ a)\ (IsShow\ a)\ (IsShow\ b)$

What about class methods?

What about class methods?

Two possibilities:

- Use type classes
- Extend type families with elaboration

What about class methods?

data *Defined e* $=$ *Yes* $\mid$ *No e*

Two possibilities:

- Use type classes
- Extend type families with elaboration

What about class methods?

data *Defined e p* = *Yes p* | *No e*

Two possibilities:

- Use type classes
- Extend type families with elaboration

What about class methods?

Two possibilities:

- Use type classes
- Extend type families with elaboration

What about class methods?

Two possibilities:

- Use type classes
- Extend type families with elaboration

Elaboration at Rewriting

assign a dictionary to type families

Elaboration at Rewriting

assign a dictionary to type families

Example

$$\textbf{type family } \mathit{IsEq}\ (t :: *) :: \mathit{Defined}$$
$$\quad \textbf{dictionary } \mathit{eq} :: t \to t \to \mathit{Bool}$$

Elaboration at Rewriting

assign a dictionary to type families

Example

type family $IsEq\ (t :: *) :: Defined$
  dictionary $eq :: t \to t \to Bool$

type instance $IsEq\ Int = Yes$
  dictionary $eq = primEqInt$ -- the primitive Int comparison

Elaboration at Rewriting

assign a dictionary to type families

Example

$$\textbf{type family } \mathit{IsEq}\ (t :: *) :: \mathit{Defined}$$
$$\quad \textbf{dictionary } \mathit{eq} :: t \to t \to \mathit{Bool}$$

$$\textbf{type instance } \mathit{IsEq}\ \mathit{Int} = \mathit{Yes}$$
$$\quad \textbf{dictionary } \mathit{eq} = \mathit{primEqInt} \quad \text{-- the primitive Int comparison}$$

$$\begin{aligned}
&\textbf{type instance } \mathit{IsEq}\ [a] = e@(\mathit{IsEq}\ a)\\
&\quad \textbf{dictionary } \mathit{eq}\ [\,]\ \ [\,]\ \ = \mathit{True}\\
&\qquad\qquad \mathit{eq}\ (x : xs)\ (y : ys) = e@\mathit{eq}\ x\ y \wedge \mathit{eq}\ xs\ ys\\
&\qquad\qquad \mathit{eq}\ _\ \ _\ \ = \mathit{False}
\end{aligned}$$

Example: Data Types à la Carte

```
type family f :<: g :: Defined
dictionary  inj :: f a → g a
where       e :<: e          = Yes dictionary inj = id
            f :<: (x :+: y)  = d@(Choose f x y l@(f :<: x) r@(f :<: y))
                               dictionary inj = d@choice l@inj r@inj
            f :<: g          = No
```

```
type family Choose f x y fx fy :: Defined
dictionary  choice :: (f a → x a) → (f a → y a) → f a → (x :+: y) a
where       Choose f x y Yes fy   = Yes
                                    dictionary choice x y = Inl ∘ x
            Choose f x y fx  Yes  = Yes
                                    dictionary choice x y = Inr ∘ y
            Choose f x y fx  fy   = No
```

Conclusions

Future work

- study elaboration further
- use encoding as implementation of type classes?

Conclusions

Future work

- study elaboration further
- use encoding as implementation of type classes?

In the paper

- encoding of functional dependencies
- encoding of superclasses
- more examples
- proof of soundness & completeness (using OutsideIn(X))

Type Families with Class, Type Classes with Family

Alejandro Serrano[1] Jurriaan Hage[1] Patrick Bahr[2]

[1]Utrecht University

[2]IT University of Copenhagen

Haskell Symposium 2015