Closed Type Families with Overlapping Equations

Richard A. Eisenberg
U. of Pennsylvania

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research

Stephanie Weirich
U. of Pennsylvania

Friday, January 24, 2014
POPL
San Diego, CA, USA
Setting the Scene...

Goal:
Dependent types in Haskell

Why?
EDSLs, generic programming, greater compile-time confidence, ...
Type Families

A type family is a function on types.

“pattern”  Example: \texttt{Elt}

\[
\text{Elt } [a] = a
\]

\[
\text{Elt } \text{ByteString} = \text{Word8}
\]

\texttt{singleton} :: \texttt{Container }b \Rightarrow \texttt{Elt }b \to b

“application”

\texttt{Elt} is naturally open.
Type Families

A type family is a function on types.

Example: Not

Not True = False
Not False = True

Not is naturally closed.
Overlapping Equations

Example of overlapping equations: Contains

\texttt{Contains x []} = \texttt{False}
\texttt{Contains x (x : xs)} = \texttt{True}
\texttt{Contains x (y : ys)} = \texttt{Contains x ys}

The last two equations \texttt{overlap}.

We need closed type families to allow overlap.
What’s the Big Deal?

Choosing when and how to simplify uses of closed type families is non-trivial.
Attempt #0

Strategy: Try equations in order.

Example:

```haskell
type family F a where
  F Int   = Bool
  F a     = Char
```

Target: `F Double`  
Result: `Char`

Target: `F b`  
Result: `? Char ?`
Attempt #0

type family F a where
  F Int = Bool
  F a    = Char

foo :: b → F b
foo _ = ‘x’

bar :: Int → F Int
bar n = foo n

baz :: Bool
baz = bar 5

OK, because F b reduces to Char
OK, because Int → F Int
... but bar 5 is an instance of b → F b
‘x’! Yikes!
Attempt #0

Strategy: Try equations in order.

Example:

```haskell
type family F a where
  F Int = Bool
  F a  = Char
```

Target: `F Double`  
Result: `Char`

Target: `F b`  
Result: `Char`

Disaster!
Apartness

Strategy: Try equations in order, requiring all previous patterns to be apart from the target.

Requirement: \( b \) is not apart from \( \text{Int} \).

Property of apartness: If \( \text{apart}(\rho, \tau) \), then no instantiation of \( \tau \) matches \( \rho \).
Attempt #1

Strategy: Try equations in order, requiring all previous patterns to be *apart* from the target.

Example:

```haskell
type family F a where
  F Int  =  Bool
  F a     =  Char
```

Target: `F b` Result: `F b`

*Phew!* $b$ is not apart from `Int`. 
Attempt #1

Strategy: Try equations in order, with apartness. Two types are apart if they fail to unify.

Example:

```
type family F a where
  F Int = Bool
  F a   = Char

type family G c
```

Target: `F (G d)` Result: `? Char`?

Disaster! What if `G d` becomes `Int`?
Apartness, revisited

Strategy: Try equations in order, with apartness.

Requirement: \((G \ d)\) is not apart from \(\text{Int}\).

Property of apartness: If \(\text{apart}(\rho, \tau_1)\), then no \(\tau_2\), such that \(\tau_1 \sim^* \tau_2\), matches \(\rho\).
Implementing Apartness

If \( \text{apart}(\rho, \tau) \), then instances of \( \tau \) do not match \( \rho \).

If \( \text{apart}(\rho, \tau_1) \), then no \( \tau_2 \) (with \( \tau_1 \sim^* \tau_2 \)) matches \( \rho \).

Does \( \text{apart} \) have an implementation?

- Let \( \text{flatten}(\tau) \) be \( \tau \) with all type family applications replaced by fresh variables.
- Then: Yes! Let \( \text{apart}(\rho, \tau) := \neg \text{unify}(\rho, \text{flatten}(\tau)) \)
- We have proved the properties above from this definition.
Attempt #2

Strategy: Try equations in order, with apartness.

\[ \text{apart}(\rho, \tau) := \neg \text{unify}(\rho, \text{flatten}(\tau)) \]

Example:

```markdown
type family F a where
  F Int = Bool
  F a = Char

type family G c

Target: F (G d)  
Result: F (G d) 

Phew!
```

\[ \text{flatten}(G\ d) \text{ is } e, \text{ which is not apart from } \text{Int.} \]
Attempt #2

Strategy: Try equations in order, with apartness.

\[ \text{apart}(\rho, \tau) := \neg \text{unify}(\rho, \text{flatten}(\tau)) \]

Example:

\[
\text{type family } \text{And } a \ b \ \text{where}
\]
\[
\begin{align*}
\text{And } \text{False } a & = \text{False} \\
\text{And } b & \quad \text{False} = \text{False}
\end{align*}
\]

Target: \(\text{And } x \ \text{False} \)

Result: \(\text{And } x \ \text{False} \)

And \(x\) \ False is \textit{not} apart from \(\text{And } \text{False} \ a\)

What a shame! Can we do better?
Compatibility

Some overlap is patently benign.

Example: And

```
type family And a b where
  And False a = False
  And b False = False
```

Definition: Two equations are compatible iff, whenever the LHSs unify, the unifier also unifies the RHSs.
Strategy: Try equations in order, requiring all previous incompatible equations to be apart from the target.

Example:

```haskell
type family And a b where
  And False a     = False
  And b           False = False
```

Target: `And x False`  
Result: `False`

Yay!
• Proved type soundness with closed type families

• Implemented closed type families in GHC 7.8
Closed type families allow pattern-matching over types that classify terms.

Example: `CountArgs`

CountArgs (Int → Bool → Char) → 2
CountArgs [Double] → 0

type family CountArgs f where
  CountArgs (x → r) = 1 + CountArgs r
  CountArgs result = 0
Expressivity

Type families allow non-linear patterns.

Example:

\[
\text{type family } \text{Equal} \ a \ b \ \text{where} \\
\text{Equal} \ a \ a = \text{True} \\
\text{Equal} \ a \ b = \text{False}
\]

Target: \text{Equal Int Bool} \ \text{Result: False}

Target: \text{Equal Int } b \ \text{Result: Equal Int } b

Target: \text{Equal } c \ c \ \text{Result: True}

\text{Equal is manifestly reflexive.}
Expressivity

- **Elt** operates on types (an open kind)
  - **open** type family

- **Contains** operates on lists (a closed kind)
  - **closed** type family

- Closed type families on open kinds are particularly interesting

- Why? We can’t unravel any overlap
Caveat: Termination

• Proof of type soundness depends on termination of \( \sim \)

• GHC checks for termination of type family instances by default

• Proof without termination an open problem
Conclusions

Closed type families ...

• ... are useful
• ... are surprisingly subtle
• ... are expressive
• ... help bridge the gap between types and terms, leading toward dependent types
Closed Type Families with Overlapping Equations

Richard A. Eisenberg
U. of Pennsylvania

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research

Stephanie Weirich
U. of Pennsylvania