# 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: `Elt`

```
Elt [a]          = a
Elt ByteString = Word8
```

```
singleton :: Container b ⇒ Elt b → b
```

"application"

`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`

```
Contains x []         = False
Contains x (x : xs) = True
Contains x (y : ys) = Contains x ys
```

The last two equations 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:

```
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 is an instance of *b* → F *b*

… but bar 5 evaluates to ‘x’! Yikes!

# Attempt #0

Strategy: Try equations in order.

Example:

```
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 `Int`.

Property of apartness: If *apart*(ρ, τ), then no instantiation of τ matches ρ.

type pattern → ρ; type → τ; LHS of equation → ρ; target → τ

# Attempt #1

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

Example:

```
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
```

(`G d`) is apart from `Int`.

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 `Int`.

Property of apartness: If *apart*($\rho$, $\tau_1$), then no $\tau_2$, such that $\tau_1 \rightsquigarrow^* \tau_2$, matches $\rho$.

type family reduction relation

# Implementing Apartness

If *apart*(ρ, τ), then instances of τ do not match ρ.

If *apart*($\rho$, $\tau_1$), then no $\tau_2$ (with $\tau_1 \rightsquigarrow^* \tau_2$) matches ρ.

Does *apart* have an implementation?

- Let *flatten*(τ) be τ with all type family applications replaced by fresh variables.
- Then: Yes! Let *apart*(ρ, τ) := ¬*unify*(ρ, *flatten*(τ))
- We have proved the properties above from this definition.

# Attempt #2

Strategy: Try equations in order, with apartness.

$apart(\rho, \tau) := \neg unify(\rho, flatten(\tau))$

Example:

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

Target: `F (G d)`

*flatten*(`G d`) is *e*, which is not apart from `Int`.

Result: `F (G d)`

Phew!

# Attempt #2

Strategy: Try equations in order, with apartness.

*apart*(ρ, τ) := ¬*unify*(ρ, *flatten*(τ))

Example:

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

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

`And x False` is *not* apart from `And 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.

# ~~Attempt #3~~ Final Rule

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

Example:

```
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

# Expressivity

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:

```
type family Equal a b where
  Equal a a = True
  Equal a b = False
```

Target: `Equal Int Bool` Result: `False`

Target: `Equal Int b` Result: `Equal Int b`

Target: `Equal c c` Result: `True`

`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 $\leadsto$
- 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