Haskell Symposium 2015

Vancouver, Canada

Injective Type Families for Haskell

Jan Stolarek
Politechnika Łódzka

Simon Peyton Jones
Microsoft Research Cambridge

Richard A. Eisenberg
University of Pennsylvania

```
type family Id a
type instance Id Int  = Int
type instance Id Bool = Bool

id :: Id t -> Id t
id x = x

foo = id True

Couldn't match expected type ‘Id t’
  with actual type ‘Bool’
The type variable ‘t’ is ambiguous
```

```
type family Id a
type instance Id Int  = Int
type instance Id Bool = Bool

id :: Id t -> Id t
id x = x

foo = id True
```

Problem: not possible to infer at call site what type variable `t` should be

Our solution: injective type families

```
Couldn't match type 'Id t' with 'Id t0'
NB: 'Id' is a type function, and may not be injective
The type variable 't0' is ambiguous
Expected type: Id t -> Id t
  Actual type: Id t0 -> Id t0
```

Our contributions

Backwards-compatible extension to type families, which allows users to annotate type families with injectivity information.

Algorithm for checking validity of user's injectivity annotation (+ proofs).

Type inference using injectivity information.

Comparison of injective type families with functional dependencies.

What is injectivity?

Definition (Injectivity)

A type family F is n-injective (i.e. injective in its n'th argument) iff $\forall\overline{\sigma},\overline{\tau}$: $F\ \overline{\sigma} \sim F\ \overline{\tau} \implies \sigma_n \sim \tau_n$

Intuition

If F is n-injective then result of type family reduction $F\ \overline{\tau}$ uniquely determines the arguments τ_n

Annotating type families with injectivity information

```
type family F a b c = r | r -> a c
type instance F Int  Bool   Char   = Int
type instance F Int  Double Char   = Int
type instance F Char Int    Double = Char
```

Annotating type families with injectivity information

```
type family F a b c = r | r -> a c
type instance F Int  Bool   Char   = Int
type instance F Int  Double Char   = Int
type instance F Char Int    Double = Char
type instance F Char Char   Char   = Int
```

```
Type family equations violate injectivity annotation:
  F Int Bool Char = Int
  F Char Char Char = Int
```

Verifying injectivity annotation: design challenges

```
type family F1 a = r | r -> a
type instance F1 [a] = a
```

F1 is not injective:

```
F1 [F1 Int]
```

Verifying injectivity annotation: design challenges

```
type family F1 a = r | r -> a
type instance F1 [a] = a
```

F1 is not injective:

```
F1 [F1 Int] ~ F1 Int
```

Verifying injectivity annotation: design challenges

```
type family F1 a = r | r -> a
type instance F1 [a] = a
```

F1 is not injective:

```
[F1 Int] ~    Int
```

Do not allow bare type variable to appear as the RHS.

Verifying injectivity annotation: design challenges

```
type family F2 a = r | r -> a
type instance F2 a = a
```

F2 is injective.

Allow bare type variable as RHS if all LHS patterns are bare variables.

Verifying injectivity annotation: design challenges

```
type family F3 a = r | r -> a
type instance F3 [a] = F3 a
```

F3 is not injective:

```
F3 [Int]
```

Verifying injectivity annotation: design challenges

```
type family F3 a = r | r -> a
type instance F3 [a] = F3 a
```

F3 is not injective:

```
F3 [Int] ~ F3 Int
```

Verifying injectivity annotation: design challenges

```
type family F3 a = r | r -> a
type instance F3 [a] = F3 a
```

F3 is not injective:

```
[Int] ~    Int
```

Verifying injectivity annotation: design challenges

```
type family F3 a = r | r -> a
type instance F3 [a] = F3 a
```

F3 is not injective:

```
[Int] ~    Int
```

Disallow calls to type families?

Verifying injectivity annotation: design challenges

```
type family F4 a = r | r -> a
type instance F4 [a]       = [G a]
type instance F4 (Maybe a) = H a -> Int
```

`F4` is injective if G and H are injective.

Do not allow calls to type families at the top level of RHS.

Verifying injectivity annotation: design challenges

```
type family F5 a = r | r -> a
type instance F5 [a]       = [G a]
type instance F5 (Maybe a) = [H a]
```

F5 is not injective.

Assume that a type family application unifies with any type.

Injectivity check

Definition (Injectivity check)

A type family `F` is *n*-injective iff:

1. For every equation $F\ \overline{\sigma} = \tau$:
   - τ is not a type family application, and
   - if $\tau = \alpha_i$ (for some type variable α_i), then $\overline{\sigma} = \overline{\alpha}$.
2. Every pair of equations $F\ \overline{\sigma}_i = \tau_i$ and $F\ \overline{\sigma}_j = \tau_j$ (including $i = j$) is *pairwise-n-injective*.

Pairwise-n-injectivity

Definition (Pairwise-n-injectivity)

A pair of equations $F\ \overline{\sigma}_i = \tau_i$ and $F\ \overline{\sigma}_j = \tau_j$ is pairwise-n-injective iff either:

1. τ_i and τ_j do not *unify*
2. τ_i and τ_j *unify* with substitution θ and $\theta(\overline{\sigma}_{in}) = \theta(\overline{\sigma}_{jn})$

Pre-unification of types

Pre-unification algorithm

We use a special variant of the unification algorithm that:

1. treats type family application as possibly unifying with any other type
2. looks under injective type family applications
3. does not find solutions involving infinite types

```haskell
type family Id a = r | r -> a
type instance Id Int  = Int
type instance Id Bool = Bool

id :: Id t -> Id t
id x = x

foo = id True
```

```haskell
type family Id a = r | r -> a
type instance Id Int  = Int
type instance Id Bool = Bool

id :: (Id a ~ Id b) => a -> b
id x = x

foo = id True
```

Injective type families vs. functional dependencies

There is a close similarity between injective type families and type classes with functional dependencies.

Our implementation of injective type families is not yet as expressive as functional dependencies.

```
data Nat = Zero | Succ a

class Add a b r | a b -> r, r a -> b
instance                    Add Zero     b b
instance (Add a b r) => Add (Succ a) b (Succ r)
```

```haskell
data Nat = Zero | Succ a

type family AddTF a b = r | r a -> b where
     AddTF Zero     b = b
     AddTF (Succ a) b = Succ (AddTF a b)
```

Summary and future work

- More in the paper:
  - real-life examples
  - detailed description of type inference using injectivity
  - soundness and completeness, with proofs
  - kind injectivity
- Current work:
  - extending Core
  - generalized injectivity
  - full proof of soundness

Haskell Symposium 2015

Vancouver, Canada

Injective Type Families for Haskell

Jan Stolarek
Politechnika Łódzka

Simon Peyton Jones
Microsoft Research Cambridge

Richard A. Eisenberg
University of Pennsylvania

Table: Popularity of selected type-level programming language extensions.

Language extension	no. of using packages
`TypeFamilies`	1092
`GADTs`	612
`FunctionalDependencies`	563
`DataKinds`	247
`PolyKinds`	109

$$
\begin{array}{rcll r}
U(\alpha,\tau)\,\theta & = & U(\theta(\alpha),\tau)\,\theta & \alpha \in \mathit{dom}(\theta) & (1)\\
U(\alpha,\tau)\,\theta & = & \textbf{Just}\ \theta & \alpha \in \mathit{ftv}(\theta(\tau)) & (2)\\
U(\alpha,\tau)\,\theta & = & \textbf{Just}\ ([\alpha \mapsto \theta(\tau)] \circ \theta) & \alpha \notin \mathit{ftv}(\theta(\tau)) & (3)\\
U(\tau,\alpha)\,\theta & = & U(\alpha,\tau)\,\theta & & (4)\\
U(\sigma_1\,\sigma_2,\tau_1\,\tau_2)\,\theta & = & U(\sigma_1,\tau_1)\,\theta \ggg U(\sigma_2,\tau_2) & & (5)\\
U(H,H)\,\theta & = & \textbf{Just}\ \theta & & (6)\\
U(F(\overline{\sigma}),F(\overline{\tau}))\,\theta & = & U(\sigma_i,\tau_i)\,\theta \ggg & F \text{ is } i\text{-injective} & (7)\\
 & & \ldots \qquad\qquad \ggg & \text{...etc...} & \\
 & & U(\sigma_j,\tau_j) & F \text{ is } j\text{-injective} & \\
U(F(\overline{\sigma}),\tau)\,\theta & = & \textbf{Just}\ \theta & & (8)\\
U(\tau,F(\overline{\sigma}))\,\theta & = & \textbf{Just}\ \theta & & (9)\\
U(\sigma,\tau)\,\theta & = & \textbf{Nothing} & & (10)
\end{array}
$$