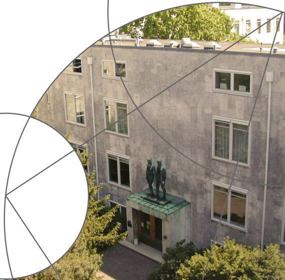
Composing and Decomposing Data Types

Data Types à la Carte with Closed Type Families

Patrick Bahr
University of Copenhagen,
Department of Computer Science
paba@di.ku.dk



20th June, 2014

Introduction

Goal

Improve Haskell implementation of Data Types à la Carte:

- More flexible
- Improved error reporting
- New use cases

Introduction

Goal

Improve Haskell implementation of Data Types à la Carte:

- More flexible
- Improved error reporting
- New use cases

How?

Using closed type families

- New feature in latest version of GHC
- Type-level functions
- Pattern matching similar(-ish) to term-level functions

Data Types à la Carte

Idea: Decompose data types into two-level types:

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

$$\textbf{data}\ \mathit{Exp} = \mathit{Val}\ \mathit{Int}$$
$$\quad\quad\quad\quad \mid \mathit{Add}\ \mathit{Exp}\ \mathit{Exp}$$

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

$$\textbf{data}\ \mathit{Exp} = \mathit{Val}\ \mathit{Int}$$
$$\mid\ \mathit{Add}\ \mathit{Exp}\ \mathit{Exp}$$

$\Longrightarrow$

Fixpoint of functor

$$\textbf{data}\ \mathit{Arith}\ a = \mathit{Val}\ \mathit{Int}$$
$$\mid\ \mathit{Add}\ a\ a$$
$$\textbf{type}\ \mathit{Exp} = \mathit{Fix}\ \mathit{Arith}$$

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

data *Exp* = *Val Int*
  | *Add Exp Exp*

Fixpoint of functor

= *Val Int*
  | *Add a a*

type *Exp* = *Fix Arith*

data *Fix f* = *In* (*f* (*Fix f*))

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

data *Exp* = *Val Int*
  | *Add Exp Exp*

⟹

Fixpoint of functor

data *Arith a* = *Val Int*
  | *Add a a*

type *Exp* = *Fix Arith*

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

$$\textbf{data}\ \mathit{Exp} = \mathit{Val}\ \mathit{Int}\ \mid\ \mathit{Add}\ \mathit{Exp}\ \mathit{Exp}$$

$\Longrightarrow$

Fixpoint of functor

$$\textbf{data}\ \mathit{Arith}\ a = \mathit{Val}\ \mathit{Int}\ \mid\ \mathit{Add}\ a\ a$$

$$\textbf{type}\ \mathit{Exp} = \mathit{Fix}\ \mathit{Arith}$$

Functors can be combined by coproduct construction $:+:$

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

data $Exp = Val\ Int$
$\quad\quad\mid Add\ Exp\ Exp$

$\Longrightarrow$

Fixpoint of functor

data $Arith\ a = Val\ Int$
$\quad\quad\mid Add\ a\ a$

type $Exp = Fix\ Arith$

Functors can be combined by coproduct construction $:+:$

data $Mul\ a = Mul\ a\ a$

type $Exp' = Fix\ (Arith :+: Mul)$

Data Types à la Carte

Idea: Decompose data types into two-level types:

Recursive data type

data *Exp* = *Val Int*
  | *Add Exp E*

$\Longrightarrow$

Fixpoint of functor

data *Arith a* = *Val Int*
  | *Add a a*
Fix Arith

data $(f :+: g)\ a = Inl\ (f\ a)$
  $\mid\ Inr\ (g\ a)$

Functors can be combined by coproduct construction :+:

data *Mul a* = *Mul a a*

type $Exp' = Fix\ (Arith :+: Mul)$

Data Types à la Carte (cont.)

Subtyping constraint :≺:

class *f* :≺: *g* **where**
  inj :: *f a* → *g a*
  prj :: *g a* → *Maybe* (*f a*)

Data Types à la Carte (cont.)

Subtyping constraint :≺:

class f :≺: g **where**
  $inj :: f\ a \rightarrow g\ a$
  $prj :: g\ a \rightarrow Maybe\ (f\ a)$

Example: smart constructors

$add :: (Arith :\prec: f) \Rightarrow Fix\ f \rightarrow Fix\ f \rightarrow Fix\ f$
$add\ x\ y = In\ (inj\ (Add\ x\ y))$

Data Types à la Carte (cont.)

Subtyping constraint :≺:

$$\textbf{class}\ f :\prec: g\ \textbf{where}$$
$$inj :: f\ a \to g\ a$$
$$prj :: g\ a \to Maybe\ (f\ a)$$

Example: smart constructors

$$add :: (Arith :\prec: f) \Rightarrow Fix\ f \to Fix\ f \to Fix\ f$$
$$add\ x\ y = In\ (inj\ (Add\ x\ y))$$

$$exp :: Fix\ (Arith :+: Mul)$$
$$exp = val\ 1\ \text{`}add\text{`}\ (val\ 2\ \text{`}mul\text{`}\ val\ 3)$$

Limitations of $:\prec:$

Definition of $:\prec:$

instance $f :\prec: f$ **where**

...

instance $(f :\prec: f_1) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**

...

instance $(f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**

...

Limitations of $:\prec:$

Definition of $:\prec:$

instance $f :\prec: f$ **where**

$\quad\ldots$

instance $(f :\prec: f_1) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**

$\quad\ldots$

instance $(f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**

$\quad\ldots$

- No backtracking!

Limitations of $:\prec:$

Definition of $:\prec:$

instance $f :\prec: f$ **where**

...

instance $f :\prec: (f \; :+: f_2)$ **where**

...

instance $(f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**

...

- No backtracking!

Limitations of $:\prec:$

Definition of $:\prec:$

instance $f :\prec: f$ **where**

$\dots$

instance $f :\prec: (f\ :+: f_2)$ **where**

$\dots$

instance $(f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**

$\dots$

- No backtracking!
- Asymmetric treatment of $:+:$
- Left-hand side is not inspected

Limitations of $:\prec:$

Definition of $:\prec:$

instance $f :\prec: f$ **where**
  ...
instance $f :\prec: (f :+: f_2)$ **where**
  ...
instance $(f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**
  ...

- No backtracking!
- Asymmetric treatment of $:+:$ $A :\prec: A :+: (B :+: C)$
- Left-hand side is not inspected

Limitations of $:\prec:$

Definition of $:\prec:$

instance $f :\prec: f$ **where**
    ...
instance $f :\prec: (f :+: f_2)$ **where**
    ...
instance $(f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2)$ **where**
    ...

- No backtracking!
- Asymmetric treatment of $:+:$ $A :\not\prec: (A :+: B) :+: C$
- Left-hand side is not inspected

Limitations of $:\prec:$

Definition of $:\prec:$

$$
\begin{array}{lll}
\textbf{instance} & f :\prec: f & \textbf{where} \\
\quad \ldots & & \\
\textbf{instance} & f :\prec: (f \;:+:\; f_2) & \textbf{where} \\
\quad \ldots & & \\
\textbf{instance}\ (f :\prec: f_2) \Rightarrow & f :\prec: (f_1 :+: f_2) & \textbf{where} \\
\quad \ldots & &
\end{array}
$$

- No backtracking!
- Asymmetric treatment of $:+:$ $\quad A :\not\prec: (A :+: B) :+: C$
- Left-hand side is not inspected $\quad A :+: B :\prec: (A :+: B) :+: C$

Limitations of $:\prec:$

Definition of $:\prec:$

$$
\begin{array}{llll}
\textbf{instance} & & f :\prec: f & \textbf{where} \\
\quad \ldots & & & \\
\textbf{instance} & & f :\prec: (f \ :+: f_2) & \textbf{where} \\
\quad \ldots & & & \\
\textbf{instance} & (f :\prec: f_2) \Rightarrow & f :\prec: (f_1 :+: f_2) & \textbf{where} \\
\quad \ldots & & &
\end{array}
$$

- No backtracking!
- Asymmetric treatment of $:+:$ $\quad A :\not\prec: (A :+: B) :+: C$
- Left-hand side is not inspected $\quad A :+: B :\not\prec: A :+: (B :+: C)$

Contributions

We re-implemented $:\prec:$ such that:

- Subtyping behaves as intuitively expected
- Ambiguous subtyping are avoided
- We can express isomorphism $:\simeq:$

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \Longleftrightarrow \quad$ "set of signatures in f" $\subseteq$ "set of signatures in g"

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \Longleftrightarrow \quad$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \iff$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Avoid ambiguous subtyping

Multiple occurrences of signatures are rejected:

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \Longleftrightarrow \quad$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Avoid ambiguous subtyping

Multiple occurrences of signatures are rejected:

$$A :\prec: A :+: A :+: C$$

$$A :+: A :\prec: A :+: B$$

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \Longleftrightarrow \quad$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Avoid ambiguous subtyping

Multiple occurrences of signatu

injection not unique!

$$A :\prec: A :+: A :+: C$$

$$A :+: A :\prec: A :+: B$$

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \iff$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Avoid ambiguous subtyping

Multiple occurrences of signatur

injection not unique!

$$A :\not\prec: A :+: A :+: C$$

$$A :+: A :\prec: A :+: B$$

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \iff$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Avoid ambiguous subtyping

Multiple occurrences of signatur

injection not unique!

$$A \;\color{red}{:\not\prec:}\; A :+: A :+: C$$

$$A :+: A :\prec: A :+: B$$

"injection" not injective!

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \iff$ "set of signatures in f" $\subseteq$ "set of signatures in g"

$$C :+: A \;:\prec:\; A :+: B :+: C$$

Avoid ambiguous subtyping

Multiple occurrences of signatures

injection not unique!

$$A \;:\not\prec:\; A :+: A :+: C$$

$$A :+: A \;:\not\prec:\; A :+: B$$

"injection" not injective!

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$$f :\simeq: g \iff \text{“set of signatures in } f\text{”} = \text{“set of signatures in } g\text{”}$$

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \iff$ "set of signatures in f" $=$ "set of signatures in g"

Easy to implement: $f :\simeq: g = (f :\prec: g, g :\prec: f)$

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \iff$ "set of signatures in f" $=$ "set of signatures in g"

Easy to implement: $f :\simeq: g = (f :\prec: g, g :\prec: f)$

Use case: improved projection function

The type of the projection function is unsatisfying:

$$prj :: (f :\prec: g) \Rightarrow g\ a \rightarrow Maybe\ (f\ a)$$

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \quad \Longleftrightarrow \quad$ "set of signatures in f" $=$ "set of signatures in g"

Easy to implement: $\quad f :\simeq: g = (f :\prec: g, g :\prec: f)$

Use case: improved projection function

The type of the projection function is unsatisfying:

$$prj :: (f :\prec: g) \Rightarrow g\ a \rightarrow Maybe\ (f\ a)$$

With $:\simeq:$ we can do better:

$$split :: (g :\simeq: f :+: r) \Rightarrow g\ a \rightarrow Either\ (f\ a)\ (r\ a)$$

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \iff$ "set of signatures in f" $=$ "set of signatures in g"

Easy to implement: $f :\simeq: g = (f :\prec: g, g :\prec: f)$

Use case: improved projection function

The type of the projection function is unsatisfying:

$$prj :: (f :\prec: g) \Rightarrow g\ a \to Maybe\ (f\ a)$$

With $:\simeq:$ we can do better:

$$split :: (g :\simeq: f :+: r) \Rightarrow g\ a \to (f\ a \to b) \to (r\ a \to b) \to b$$

Example: Desugaring

$\textbf{data}\ \mathit{Dbl}\ a = \mathit{Double}\ a$

Example: Desugaring

data *Dbl a* = *Double a*

class *Desug f g* **where**
  desugAlg :: *f* (*Fix g*) → *Fix g*

Example: Desugaring

$$\textbf{data}\ \mathit{Dbl}\ a = \mathit{Double}\ a$$

$$\begin{array}{l}\textbf{class}\ \mathit{Desug}\ f\ g\ \textbf{where}\\ \quad \mathit{desugAlg} :: f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Desug}\ f_1\ g, \mathit{Desug}\ f_2\ g) \Rightarrow \mathit{Desug}\ (f_1 :+: f_2)\ g\ \textbf{where}\\ \quad \mathit{desugAlg}\ (\mathit{Inl}\ x) = \mathit{desugAlg}\ x\\ \quad \mathit{desugAlg}\ (\mathit{Inr}\ x) = \mathit{desugAlg}\ x\\ \textbf{instance}\ (\mathit{Arith} :\prec: g) \Rightarrow \mathit{Desug}\ \mathit{Dbl}\ g\ \textbf{where}\\ \quad \mathit{desugAlg}\ (\mathit{Double}\ x) = \mathit{add}\ x\ x\\ \textbf{instance}\ (f :\prec: g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where}\\ \quad \mathit{desugAlg} = \mathit{In}\ .\ \mathit{inj}\end{array}$$

Example: Desugaring

$$\textbf{data}\ \mathit{Dbl}\ a = \mathit{Double}\ a$$

$$\begin{array}{l}
\textbf{class}\ \mathit{Desug}\ f\ g\ \textbf{where} \\
\quad \mathit{desugAlg} :: f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g
\end{array}$$

$$\begin{array}{l}
\textbf{instance}\ (\mathit{Desug}\ f_1\ g, \mathit{Desug}\ f_2\ g) \Rightarrow \mathit{Desug}\ (f_1 :+: f_2)\ g\ \textbf{where} \\
\quad \mathit{desugAlg}\ (\mathit{Inl}\ x) = \mathit{desugAlg}\ x \\
\quad \mathit{desugAlg}\ (\mathit{Inr}\ x) = \mathit{desugAlg}\ x \\
\textbf{instance}\ (\mathit{Arith} :\prec: g) \Rightarrow \mathit{Desug}\ \mathit{Dbl}\ g\ \textbf{where} \\
\quad \mathit{desugAlg}\ (\mathit{Double}\ x) = \mathit{add}\ x\ x \\
\textbf{instance}\ (f :\prec: g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where} \\
\quad \mathit{desugAlg} = \mathit{In}\ .\ \mathit{inj}
\end{array}$$

$$\begin{array}{l}
\mathit{desugar} :: (\mathit{Desug}\ f\ g, \mathit{Functor}\ f) \Rightarrow \mathit{Fix}\ f \to \mathit{Fix}\ g \\
\mathit{desugar} = \mathit{fold}\ \mathit{desugAlg}
\end{array}$$

Example: Desugaring

$$\textbf{data}\ \mathit{Dbl}\ a = \mathit{Double}\ a$$

$$\begin{array}{l}\textbf{class}\ \mathit{Desug}\ f\ g\ \textbf{where}\\ \quad \mathit{desugAlg} :: f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g\end{array}$$

$$\begin{array}{l}\textbf{instance}\ (\mathit{Desug}\ f_1\ g, \mathit{Desug}\ f_2\ g) \Rightarrow \mathit{Desug}\ (f_1 :+: f_2)\ g\ \textbf{where}\\ \quad \mathit{desugAlg}\ (\mathit{Inl}\ x) = \mathit{desugAlg}\ x\\ \quad \mathit{desugAlg}\ (\mathit{Inr}\ x) = \mathit{desugAlg}\ x\\ \textbf{instance}\ (\mathit{Arith} :\prec: g) \Rightarrow \mathit{Desug}\ \mathit{Dbl}\ g\ \textbf{where}\\ \quad \mathit{desugAlg}\ (\mathit{Double}\ x) = \mathit{add}\ x\ x\\ \textbf{instance}\ (f :\prec: g) \Rightarrow \mathit{Desug}\ f\ g\ \textbf{where}\\ \quad \mathit{desugAlg} = \mathit{In}\ .\ \mathit{inj}\end{array}$$

$$\begin{array}{l}\mathit{desugar} :: \mathit{Fix}\ (\mathit{Dbl} :+: \mathit{Arith} :+: \mathit{Mul}) \to \mathit{Fix}\ (\mathit{Arith} :+: \mathit{Mul})\\ \mathit{desugar} = \mathit{fold}\ \mathit{desugAlg}\end{array}$$

Example: Desugaring (cont.)

$$
\begin{aligned}
&\mathit{desugar} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Arith} :\prec: g, \mathit{Functor}\ f) \Rightarrow \\
&\qquad\qquad \mathit{Fix}\ f \to \mathit{Fix}\ g \\
&\mathit{desugar} = \mathit{fold}\ \mathit{desugAlg}
\end{aligned}
$$

Example: Desugaring (cont.)

$$\begin{aligned}
&\mathit{desugar} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Arith} :\prec: g, \mathit{Functor}\ f) \Rightarrow \\
&\qquad\quad \mathit{Fix}\ f \to \mathit{Fix}\ g \\
&\mathit{desugar} = \mathit{fold}\ \mathit{desugAlg}
\end{aligned}$$

$$\begin{aligned}
&\mathit{desugAlg} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Arith} :\prec: g) \Rightarrow f\ (\mathit{Fix}\ g) \to \mathit{Fix}\ g \\
&\mathit{desugAlg}\ e = \mathit{split}\ e\ (\lambda x \qquad\qquad\quad \to \mathit{In}\ x) \\
&\qquad\qquad\qquad\qquad\quad (\lambda(\mathit{Double}\ x) \to \mathit{add}\ x\ x)
\end{aligned}$$

Example: Desugaring (cont.)

$$\begin{aligned}
&\mathit{desugar} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Arith} :\prec: g, \mathit{Functor}\ f) \Rightarrow \\
&\qquad \mathit{Fix}\ f \rightarrow \mathit{Fix}\ g \\
&\mathit{desugar} = \mathit{fold}\ \mathit{desugAlg}
\end{aligned}$$

$$\begin{aligned}
&\mathit{desugAlg} :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Arith} :\prec: g) \Rightarrow f\ (\mathit{Fix}\ g) \rightarrow \mathit{Fix}\ g \\
&\mathit{desugAlg}\ e = \mathit{split}\ e\ (\lambda x \rightarrow \mathit{In}\ x) \\
&\qquad\qquad (\lambda(\mathit{Double}\ x) \rightarrow \mathit{add}\ x\ x)
\end{aligned}$$

$$\begin{aligned}
&\mathit{desugAlg}' :: (f :\simeq: g :+: \mathit{Dbl}, \mathit{Arith} :\prec: g, \mathit{Mul} :\prec: g) \Rightarrow \\
&\qquad f\ (\mathit{Fix}\ g) \rightarrow \mathit{Fix}\ g \\
&\mathit{desugAlg}'\ e = \mathit{split}\ e\ (\lambda x \rightarrow \mathit{In}\ x) \\
&\qquad\qquad (\lambda(\mathit{Double}\ x) \rightarrow \mathit{mul}\ (\mathit{val}\ 2)\ x)
\end{aligned}$$

Implementation of $:\prec:$

Idea

Type-level function *Embed*:

- take two signatures f, g as arguments
- check whether $f :\prec: g$

Idea

Type-level function *Embed*:

- take two signatures f, g as arguments
- check whether $f :\prec: g$

Derive implementation of *inj* and *prj*: ???

Idea

Type-level function *Embed*:

- take two signatures f, g as arguments
- check whether $f :\prec: g$
- if check is successful: produce proof object for $f :\prec: g$

Derive implementation of *inj* and *prj*:

Idea

Type-level function *Embed*:

- take two signatures f, g as arguments
- check whether $f :\prec: g$
- if check is successful: produce proof object for $f :\prec: g$

Derive implementation of *inj* and *prj*:

- also use a type class
- But: use proof object as oracle in instance declarations

Idea

Type-level function *Embed*:

- take two signatures f, g as arguments
- check whether $f :\prec: g$
- if check is successful: produce proof object for $f :\prec: g$

Derive implementation of *inj* and *prj*:

- also use a type class
- But: use proof object as oracle in instance declarations

No singleton types. This all happens at compile time!

Proof Objects

Definition

$$\textbf{data}\ \mathit{Pos} = \mathit{Here} \mid \mathit{Left}\ \mathit{Pos} \mid \mathit{Right}\ \mathit{Pos} \mid \mathit{Sum}\ \mathit{Pos}\ \mathit{Pos}$$

Proof Objects

Definition

$$\textbf{data}\ Pos = Here \mid Left\ Pos \mid Right\ Pos \mid Sum\ Pos\ Pos$$

$$\frac{}{Here : f :\prec: f}$$

Proof Objects

Definition

$$\textbf{data}\ \mathit{Pos} = \mathit{Here} \mid \mathit{Left}\ \mathit{Pos} \mid \mathit{Right}\ \mathit{Pos} \mid \mathit{Sum}\ \mathit{Pos}\ \mathit{Pos}$$

$$\frac{}{\mathit{Here} : f :\prec: f}$$

$$\frac{p : f :\prec: g_1}{\mathit{Left}\ p : f :\prec: g_1 :+: g_2} \qquad \frac{p : f :\prec: g_2}{\mathit{Right}\ p : f :\prec: g_1 :+: g_2}$$

Proof Objects

Definition

data $Pos = Here \mid Left\ Pos \mid Right\ Pos \mid Sum\ Pos\ Pos$

$$\frac{}{Here : f :\prec: f}$$

$$\frac{p : f :\prec: g_1}{Left\ p : f :\prec: g_1 :+: g_2} \qquad \frac{p : f :\prec: g_2}{Right\ p : f :\prec: g_1 :+: g_2}$$

$$\frac{p_1 : f_1 :\prec: g \quad p_2 : f_2 :\prec: g}{Sum\ p_1\ p_2 : f_1 :+: f_2 :\prec: g}$$

Proof Objects

Defin kind

$$\textbf{data}\ \mathit{Pos} = \mathit{Here} \mid \mathit{Left}\ \mathit{Pos} \mid \mathit{Right}\ \mathit{Pos} \mid \mathit{Sum}\ \mathit{Pos}\ \mathit{Pos}$$

$$\frac{}{\mathit{Here} : f :\prec: f}$$

$$\frac{p : f :\prec: g_1}{\mathit{Left}\ p : f :\prec: g_1 :+: g_2} \qquad \frac{p : f :\prec: g_2}{\mathit{Right}\ p : f :\prec: g_1 :+: g_2}$$

$$\frac{p_1 : f_1 :\prec: g \quad p_2 : f_2 :\prec: g}{\mathit{Sum}\ p_1\ p_2 : f_1 :+: f_2 :\prec: g}$$

Proof Objects

Defin kind type

$$\textbf{data}\ \mathit{Pos} = \mathit{Here} \mid \mathit{Left}\ \mathit{Pos} \mid \mathit{Right}\ \mathit{Pos} \mid \mathit{Sum}\ \mathit{Pos}\ \mathit{Pos}$$

$$\frac{}{\mathit{Here} : f :\prec: f}$$

$$\frac{p : f :\prec: g_1}{\mathit{Left}\ p : f :\prec: g_1 :+: g_2} \qquad \frac{p : f :\prec: g_2}{\mathit{Right}\ p : f :\prec: g_1 :+: g_2}$$

$$\frac{p_1 : f_1 :\prec: g \quad p_2 : f_2 :\prec: g}{\mathit{Sum}\ p_1\ p_2 : f_1 :+: f_2 :\prec: g}$$

Proof Objects

Defin kind type

data $Pos = Here \mid Left\ Pos \mid Right\ Pos \mid Sum\ Pos\ Pos$

type constructor

$$\frac{}{Here : f :\prec: f}$$

$$\frac{p : f :\prec: g_1}{Left\ p : f :\prec: g_1 :+: g_2} \qquad \frac{p : f :\prec: g_2}{Right\ p : f :\prec: g_1 :+: g_2}$$

$$\frac{p_1 : f_1 :\prec: g \quad p_2 : f_2 :\prec: g}{Sum\ p_1\ p_2 : f_1 :+: f_2 :\prec: g}$$

Construct Proof Objects

data $Emb = Found\ Pos \mid NotFound \mid Ambiguous$

Construct Proof Objects

data $\mathit{Emb} = \mathit{Found}\ \mathit{Pos} \mid \mathit{NotFound} \mid \mathit{Ambiguous}$

type family $\mathit{Embed}\ (f :: * \to *)\ (g :: * \to *) :: \mathit{Emb}$ **where**

Construct Proof Objects

data *Emb* = *Found Pos* | *NotFound* | *Ambiguous*

type family *Embed* $(f :: * \to *)$ $(g :: * \to *)$ :: *Emb* **where**

$$
\begin{aligned}
&\mathit{Embed}\ f\ f &&= \mathit{Found}\ \mathit{Here}\\
&\mathit{Embed}\ (f_1 :\!+\!: f_2)\ g &&= \mathit{Sum}'\ (\mathit{Embed}\ f_1\ g)\ (\mathit{Embed}\ f_2\ g)\\
&\mathit{Embed}\ f\ (g_1 :\!+\!: g_2) &&= \mathit{Choose}\ (\mathit{Embed}\ f\ g_1)\ (\mathit{Embed}\ f\ g_2)\\
&\mathit{Embed}\ f\ g &&= \mathit{NotFound}
\end{aligned}
$$

Construct Proof Objects

$$\textbf{data}\ \mathit{Emb} = \mathit{Found}\ \mathit{Pos} \mid \mathit{NotFound} \mid \mathit{Ambiguous}$$

$$
\begin{array}{l}
\textbf{type family}\ \mathit{Embed}\ (f :: * \to *)\ (g :: * \to *) :: \mathit{Emb}\ \textbf{where} \\
\quad \begin{array}{lcl}
\mathit{Embed}\ f\ f & = & \mathit{Found}\ \mathit{Here} \\
\mathit{Embed}\ (f_1 :+: f_2)\ g & = & \mathit{Sum}'\ (\mathit{Embed}\ f_1\ g)\ (\mathit{Embed}\ f_2\ g) \\
\mathit{Embed}\ f\ (g_1 :+: g_2) & = & \mathit{Choose}\ (\mathit{Embed}\ f\ g_1)\ (\mathit{Embed}\ f\ g_2) \\
\mathit{Embed}\ f\ g & = & \mathit{NotFound}
\end{array}
\end{array}
$$

$$
\begin{array}{l}
\textbf{type family}\ \mathit{Choose}\ (e_1 :: \mathit{Emb})\ (e_2 :: \mathit{Emb}) :: \mathit{Emb}\ \textbf{where} \\
\quad \begin{array}{llcl}
\mathit{Choose}\ (\mathit{Found}\ p_1) & (\mathit{Found}\ p_1) & = & \mathit{Ambiguous} \\
\mathit{Choose}\ \mathit{Ambiguous} & e_2 & = & \mathit{Ambiguous} \\
\mathit{Choose}\ e_1 & \mathit{Ambiguous} & = & \mathit{Ambiguous} \\
\mathit{Choose}\ (\mathit{Found}\ p_1) & e_2 & = & \mathit{Found}\ (\mathit{Left}\ p_1) \\
\mathit{Choose}\ e_1 & (\mathit{Found}\ p_2) & = & \mathit{Found}\ (\mathit{Right}\ p_2) \\
\mathit{Choose}\ \mathit{NotFound} & \mathit{NotFound} & = & \mathit{NotFound}
\end{array}
\end{array}
$$

Post-Processing

This is almost what we want.

Post-Processing

This is almost what we want.

- We avoid ambiguity on the right-hand side:

$$A \not\prec A :+: A :+: C$$

Post-Processing

This is almost what we want.

- We avoid ambiguity on the right-hand side:

$$A \mathrel{:\not\prec:} A \mathrel{:+:} A \mathrel{:+:} C$$

- We still have ambiguity on the left-hand side:

$$A \mathrel{:+:} A \mathrel{:\prec:} A \mathrel{:+:} B$$

Post-Processing

This is almost what we want.

- We avoid ambiguity on the right-hand side:

$$A \mathrel{:\not\prec:} A \mathrel{:+:} A \mathrel{:+:} C$$

- We still have ambiguity on the left-hand side:

$$A \mathrel{:+:} A \mathrel{:\prec:} A \mathrel{:+:} B$$

Solution: check for duplicates in *Pos*

type family *Dupl* (*p* :: *Pos*) :: *Bool* **where**
    ...

Post-Processing

This is almost what we want.

- We avoid ambiguity on the right-hand side:

$$A \;{:}\!\not\prec\!{:}\; A :+: A :+: C$$

- We still have ambiguity on the left-hand side:

Sum (*Left Here*) (*Left Here*)

$$A :+: A \;{:}\!\prec\!{:}\; A :+: B$$

Solution: check for duplicates in *Pos*

type family *Dupl* (*p* :: *Pos*) :: *Bool* **where**

. . .

Are we there yet?

- Check whether $f :\prec: g$
- Construct proof for $f :\prec: g$
- Derive *inj* and *prj*

Are we there yet?

- Check whether $f :\prec: g$ ✓
- Construct proof for $f :\prec: g$
- Derive *inj* and *prj*

Are we there yet?

- Check whether $f :\prec: g$ ✓
- Construct proof for $f :\prec: g$ ✓
- Derive *inj* and *prj*

Are we there yet?

- Check whether $f :\prec: g$ ✓
- Construct proof for $f :\prec: g$ ✓
- Derive *inj* and *prj*

Derive *inj* and *prj*

class $f :\prec: g$ **where**
  $inj :: f\ a \rightarrow g\ a$
  $prj :: g\ a \rightarrow Maybe\ (f\ a)$

instance $f :\prec: f$ **where** ...

instance $f :\prec: (f :+: g_2)$ **where** ...

instance $f :\prec: g_2$
  $\Rightarrow$ $f :\prec: (g_1 :+: g_2)$ **where** ...

Derive *inj* and *prj*

$$\textbf{class } Sub \quad f \quad g \textbf{ where}$$
$$\quad inj :: f\ a \to g\ a$$
$$\quad prj :: g\ a \to Maybe\ (f\ a)$$

$$\textbf{instance } Sub \quad f \quad f \quad \textbf{where} \ldots$$

$$\textbf{instance } Sub \quad f \quad (f \quad :+: g_2) \textbf{ where} \ldots$$

$$\textbf{instance } Sub \quad f \quad g_2$$
$$\Rightarrow \quad Sub \quad f \quad (g_1 \quad :+: g_2) \textbf{ where} \ldots$$

Derive *inj* and *prj*

$$
\begin{array}{l}
\textbf{class } \mathit{Sub}\ f\ g\ \textbf{where} \\
\quad \mathit{inj} :: f\ a \rightarrow g\ a \\
\quad \mathit{prj} :: g\ a \rightarrow \mathit{Maybe}\ (f\ a)
\end{array}
$$

$$\textbf{instance } \mathit{Sub}\ f\ f\ \textbf{where} \ldots$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Sub}\ f\ g_1 \\
\quad \Rightarrow \mathit{Sub}\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance } \mathit{Sub}\ f\ g_2 \\
\quad \Rightarrow \mathit{Sub}\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots
\end{array}
$$

Derive *inj* and *prj*

$$\textbf{class}\ Sub\ f\ g\ \textbf{where}$$
$$\quad inj :: f\ a \to g\ a$$
$$\quad prj :: g\ a \to Maybe\ (f\ a)$$

$$\textbf{instance}\ Sub\ f\ f\ \textbf{where}\ldots$$

$$\textbf{instance}\ Sub\ f\ g_1 \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where}\ldots$$

$$\textbf{instance}\ Sub\ f\ g_2 \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where}\ldots$$

$$\textbf{instance}\ (Sub\ f_1\ g, Sub\ f_2\ g) \Rightarrow Sub\ (f_1 :+: f_2)\ g\ \textbf{where}\ldots$$

Derive *inj* and *prj*

$$
\begin{array}{l}
\textbf{class}\ Sub\ (e :: Emb)\ f\ \ g\ \textbf{where} \\
\quad inj :: f\ a \rightarrow g\ a \\
\quad prj :: g\ a \rightarrow Maybe\ (f\ a)
\end{array}
$$

$$\textbf{instance}\ Sub\ f\ f\ \textbf{where}\ldots$$

$$
\begin{array}{l}
\textbf{instance}\ Sub\ f\ g_1 \\
\quad \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where}\ldots
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ Sub\ f\ g_2 \\
\quad \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where}\ldots
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (Sub\ f_1\ g, Sub\ f_2\ g) \\
\quad \Rightarrow Sub\ (f_1 :+: f_2)\ g\ \textbf{where}\ldots
\end{array}
$$

Derive *inj* and *prj*

$$\overline{Here : f :\prec: f}$$

$$\textbf{class}\ Sub\ (e :: Emb)\ f\ \ g\ \textbf{where}$$
$$inj :: f\ a \to g\ a$$
$$prj :: g\ a \to Maybe\ (f\ a)$$

$$\textbf{instance}\ Sub\ (Found\ Here)\ f\ f\ \textbf{where} \ldots$$

$$\textbf{instance}\ Sub\ f\ g_1 \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots$$

$$\textbf{instance}\ Sub\ f\ g_2 \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots$$

$$\textbf{instance}\ (Sub\ f_1\ g, Sub\ f_2\ g) \Rightarrow Sub\ (f_1 :+: f_2)\ g\ \textbf{where} \ldots$$

Derive *inj* and *prj*

$$\frac{p : f :\prec: g_1}{Left\ p : f :\prec: g_1 :+: g_2}$$

$$
\begin{aligned}
&\textbf{class } Sub\ (e :: Emb)\ f\ g\ \textbf{where} \\
&\quad inj :: f\ a \to g\ a \\
&\quad prj :: g\ a \to Maybe\ (f\ a)
\end{aligned}
$$

$$\textbf{instance } Sub\ (Found\ Here)\ f\ f\ \textbf{where} \ldots$$

$$
\begin{aligned}
&\textbf{instance } Sub\ (Found\ p)\ f\ g_1 \\
&\quad \Rightarrow Sub\ (Found\ (Left\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } Sub\ f\ g_2 \\
&\quad \Rightarrow Sub\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots
\end{aligned}
$$

$$
\begin{aligned}
&\textbf{instance } (Sub\ f_1\ g, Sub\ f_2\ g) \\
&\quad \Rightarrow Sub\ (f_1 :+: f_2)\ g\ \textbf{where} \ldots
\end{aligned}
$$

Derive *inj* and *prj*

$$\frac{p : f :\prec: g_2}{\mathit{Right}\ p : f :\prec: g_1 :+: g_2}$$

$$
\begin{array}{l}
\textbf{class}\ \mathit{Sub}\ (e :: \mathit{Emb})\ f\ g\ \textbf{where} \\
\quad \mathit{inj} :: f\ a \to g\ a \\
\quad \mathit{prj} :: g\ a \to \mathit{Maybe}\ (f\ a)
\end{array}
$$

$$\textbf{instance}\ \mathit{Sub}\ (\mathit{Found}\ \mathit{Here})\ f\ f\ \textbf{where} \ldots$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Sub}\ (\mathit{Found}\ p)\ f\ g_1 \\
\quad \Rightarrow \mathit{Sub}\ (\mathit{Found}\ (\mathit{Left}\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ \mathit{Sub}\ (\mathit{Found}\ p)\ f\ g_2 \\
\quad \Rightarrow \mathit{Sub}\ (\mathit{Found}\ (\mathit{Right}\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots
\end{array}
$$

$$
\begin{array}{l}
\textbf{instance}\ (\mathit{Sub}\ f_1\ g, \mathit{Sub}\ f_2\ g) \\
\quad \Rightarrow \mathit{Sub}\ (f_1 :+: f_2)\ g\ \textbf{where} \ldots
\end{array}
$$

Derive *inj* and *prj*

$$\frac{p_1 : f_1 :\prec: g \quad p_2 : f_2 :\prec: g}{Sum\ p_1\ p_2 : f_1 :+: f_2 :\prec: g}$$

$$\begin{array}{l} \textbf{class}\ Sub\ (e :: Emb)\ f\ \ g\ \textbf{where} \\ \quad inj :: f\ a \to g\ a \\ \quad prj :: g\ a \to Maybe\ (f\ a) \end{array}$$

$$\textbf{instance}\ Sub\ (Found\ Here)\ f\ f\ \textbf{where} \ldots$$

$$\begin{array}{l} \textbf{instance}\ Sub\ (Found\ p)\ f\ g_1 \\ \quad \Rightarrow Sub\ (Found\ (Left\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots \end{array}$$

$$\begin{array}{l} \textbf{instance}\ Sub\ (Found\ p)\ f\ g_2 \\ \quad \Rightarrow Sub\ (Found\ (Right\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots \end{array}$$

$$\begin{array}{l} \textbf{instance}\ (Sub\ (Found\ p_1)\ f_1\ g, Sub\ (Found\ p_2)\ f_2\ g) \\ \quad \Rightarrow Sub\ (Found\ (Sum\ p_1\ p_2))\ (f_1 :+: f_2)\ g\ \textbf{where} \ldots \end{array}$$

Derive *inj* and *prj*

$$\textbf{class}\ Sub\ (e :: Emb)\ f\ \ g\ \textbf{where}$$
$$\quad inj :: Proxy\ e \to f\ a \to g\ a$$
$$\quad prj :: Proxy\ e \to g\ a \to Maybe\ (f\ a)$$

$$\textbf{instance}\ Sub\ (Found\ Here)\ f\ f\ \textbf{where} \ldots$$

$$\textbf{instance}\ Sub\ (Found\ p)\ f\ g_1 \Rightarrow Sub\ (Found\ (Left\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots$$

$$\textbf{instance}\ Sub\ (Found\ p)\ f\ g_2 \Rightarrow Sub\ (Found\ (Right\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots$$

$$\textbf{instance}\ (Sub\ (Found\ p_1)\ f_1\ g, Sub\ (Found\ p_2)\ f_2\ g) \Rightarrow Sub\ (Found\ (Sum\ p_1\ p_2))\ (f_1 :+: f_2)\ g\ \textbf{where} \ldots$$

Are we there yet?

- Check whether $f :\prec: g$ ✓
- Construct proof for $f :\prec: g$ ✓
- Derive *inj* and *prj*

Are we there yet?

- Check whether $f :\prec: g$ ✓
- Construct proof for $f :\prec: g$ ✓
- Derive *inj* and *prj* ✓

Are we there yet?

- Check whether $f :\prec: g$ ✓
- Construct proof for $f :\prec: g$ ✓
- Derive *inj* and *prj* ✓ (sort of)

Final Implementation of :≺:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Sub}\ (e :: \mathit{Emb})\ f\ g\ \textbf{where} \\
\quad \mathit{inj}\ :: \mathit{Proxy}\ e \rightarrow f\ a \rightarrow g\ a \\
\quad \mathit{prj}\ :: \mathit{Proxy}\ e \rightarrow g\ a \rightarrow \mathit{Maybe}\ (f\ a)
\end{array}
$$

Final Implementation of $:\prec:$

class *Sub* (*e* :: *Emb*) *f* *g* **where**
  inj :: *Proxy* *e* → *f* *a* → *g* *a*
  prj :: *Proxy* *e* → *g* *a* → *Maybe* (*f* *a*)

type *f* $:\prec:$ *g* = *Sub* (*Post* (*Embed* *f* *g*)) *f* *g*

Final Implementation of :≺:

$$
\begin{array}{l}
\textbf{class}\ \mathit{Sub}\ (e :: \mathit{Emb})\ f\ g\ \textbf{where} \\
\quad \mathit{inj}' :: \mathit{Proxy}\ e \rightarrow f\ a \rightarrow g\ a \\
\quad \mathit{prj}' :: \mathit{Proxy}\ e \rightarrow g\ a \rightarrow \mathit{Maybe}\ (f\ a)
\end{array}
$$

$$
\textbf{type}\ f :\prec: g = \mathit{Sub}\ (\mathit{Post}\ (\mathit{Embed}\ f\ g))\ f\ g
$$

Final Implementation of $:\prec:$

$$
\begin{aligned}
&\textbf{class}\ Sub\ (e :: Emb)\ f\ g\ \textbf{where}\\
&\quad inj' :: Proxy\ e \to f\ a \to g\ a\\
&\quad prj' :: Proxy\ e \to g\ a \to Maybe\ (f\ a)
\end{aligned}
$$

$$\textbf{type}\ f :\prec: g = Sub\ (Post\ (Embed\ f\ g))\ f\ g$$

$$
\begin{aligned}
&inj :: (f :\prec: g) \Rightarrow f\ a \to g\ a\\
&inj = inj'\ (P :: Proxy\ (Post\ (Embed\ f\ g)))\\
&prj :: (f :\prec: g) \Rightarrow g\ a \to Maybe\ (f\ a)\\
&prj = prj'\ (P :: Proxy\ (Post\ (Embed\ f\ g)))
\end{aligned}
$$

Type-Level Programming in Haskell

Type-Level Programming in Haskell

- Now $:\prec:$ has the properties we want / expect
- Avoid "ambiguous" subtyping
- New isomorphism constraint $:\simeq:$

Type-Level Programming in Haskell

- Now $:\prec:$ has the properties we want / expect
- Avoid "ambiguous" subtyping
- New isomorphism constraint $:\simeq:$
- You can try it:
  `> cabal install compdata`

Type-Level Programming in Haskell

Compile Time Performance

- If done “wrong”, this implementation can be very slow!

Compile Time Performance

- If done “wrong”, this implementation can be very slow!
- Implementation presented here: $\mathcal{O}(n^2)$

Compile Time Performance

- If done “wrong”, this implementation can be very slow!
- Implementation presented here: $\mathcal{O}(n^2)$
- Slightly different implementation: $\mathcal{O}(2^n)$
  (but essentially the same)

Compile Time Performance

- If done "wrong", this implementation can be very slow!
- Implementation presented here: $\mathcal{O}(n^2)$
- Slightly different implementation: $\mathcal{O}(2^n)$
  (but essentially the same)
- micro benchmark:
  - derive $F :\prec: G$
  - 9 summands in F and G

Compile Time Performance

- If done "wrong", this implementation can be very slow!
- Implementation presented here: $\mathcal{O}(n^2)$
- Slightly different implementation: $\mathcal{O}(2^n)$
  (but essentially the same)
- micro benchmark:
  - derive $F :\prec: G$
  - 9 summands in F and G
  - Implementation presented here: 0.5s

Compile Time Performance

- If done "wrong", this implementation can be very slow!
- Implementation presented here: $\mathcal{O}(n^2)$
- Slightly different implementation: $\mathcal{O}(2^n)$
  (but essentially the same)
- micro benchmark:
  - derive $F :\prec: G$
  - 9 summands in F and G
  - Implementation presented here: 0.5s
  - Naive implementation: 45s

Compile Time Performance

- If done "wrong", this implementation can be very slow!
- Implementation presented here: $\mathcal{O}(n^2)$
- Slightly different implementation: $\mathcal{O}(2^n)$
  (but essentially the same)
- micro benchmark:
  - derive $F :\prec: G$
  - 9 summands in F and G
  - Implementation presented here: 0.5s
  - Naive implementation: 45s
- Type families on kind $*$ are expensive!

Type-Level Programming in Haskell

Type-Level Programming in Haskell

Error Messages

- $A :\prec: B :+: C$?

Error Messages

- $A :\prec: B :+: C$?

```
No instance for
   (Sub NotFound A (B :+: C))
```

Error Messages

- $A :\prec: B :+: C$?

```
No instance for
   (Sub NotFound A (B :+: C))
```

The original implementation would give:

```
No instance for (A :<: C)
```

Error Messages

- $A :\prec: B :+: C$?

```
No instance for
   (Sub NotFound A (B :+: C))
```

- $A :+: A :\prec: A :+: B$?

```
No instance for
   (Sub Ambiguous (A :+: A) (A :+: B))
```

Error Messages

- $A :\prec: B :+: C$?

```
No instance for
   (Sub NotFound A (B :+: C))
```

- $A :+: A :\prec: A :+: B$?

```
No instance for
   (Sub Ambiguous (A :+: A) (A :+: B))
```

- $A :\prec: A :+: B$?

Error Messages

- $A :\prec: B :+: C$?

```
No instance for
   (Sub NotFound A (B :+: C))
```

- $A :+: A :\prec: A :+: B$?

```
No instance for
   (Sub Ambiguous (A :+: A) (A :+: B))
```

- $a :\prec: a :+: B$?

Error Messages

- $A :\prec: B :+: C$?

```
No instance for
   (Sub NotFound A (B :+: C))
```

- $A :+: A :\prec: A :+: B$?

```
No instance for
   (Sub Ambiguous (A :+: A) (A :+: B))
```

- $a :\prec: a :+: B$?

```
No instance for
   (Sub (Post (Embed a (a :+: B))) a (a :+: B))
```

Conclusion

- We can do cool stuff with closed type families.

Conclusion

- We can do cool stuff with closed type families.
- But:
  - Compile time performance unpredictable.
  - We need a way to customise error messages.