Composing and Decomposing Data Types
A Closed Type Families Implementation of Data Types à la Carte

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

Experimenting with Closed Type Families

- What can we do with them?
- How do they compare to type classes?
- How do they interact with type classes?
Introduction

Experimenting with Closed Type Families

- What can we do with them?
- How do they compare to type classes?
- How do they interact with type classes?

Application: Data Types à la Carte

Specifically: the subtyping constraint $\prec$: 
Introduction

Experimenting with Closed Type Families

- What can we do with them?
- How do they compare to type classes?
- How do they interact with type classes?

Application: Data Types à la Carte

Specifically: the subtyping constraint $\ll$

- Can we get rid of some of the restrictions?
- Can we improve error messages?
- What price do we have to pay?
Data Types à la Carte [Swierstra 2008]

Idea: Decompose data types into two-level types:
Data Types à la Carte [Swierstra 2008]

Idea: Decompose data types into two-level types:

Recursive data type

```haskell
data Exp = Val Int
  | Add Exp Exp
```

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

```haskell
data Mul a = Mul a a
type Exp' = Fix (Arith :+: Mul)
```

Patrick Bahr — Composing and Decomposing Data Types — WGP ’14, 31st August, 2014
Slide 3
Data Types à la Carte [Swierstra 2008]

Idea: Decompose data types into two-level types:

**Recursive data type**

\[
\textbf{data} \quad \textit{Exp} = \textit{Val \ Int} \\
\mid \textit{Add \ Exp \ Exp}
\]

**Fixpoint of functor**

\[
\textbf{data} \quad \textit{Arith \ a} = \textit{Val \ Int} \\
\mid \textit{Add \ a \ a}\\
\textbf{type} \quad \textit{Exp} = \textit{Fix \ Arith}
\]
Data Types à la Carte [Swierstra 2008]

Idea: Decompose data types into two-level types:

**Recursive data type**

```
data Exp = Val Int  
  | Add Exp Exp
```

**Fixpoint of functor**

```
data Fix f = In (f (Fix f))

= Val Int  
  | Add a a

type Exp = Fix Arith
```
Data Types à la Carte [Swierstra 2008]

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
```
Idea: Decompose data types into two-level types:

**Recursive data type**

\[
\text{data } \ Exp = \ \text{Val } \text{Int} \\
| \ Add \ Exp \ Exp
\]

**Fixpoint of functor**

\[
\text{data } \ Arith \ a = \ \text{Val } \text{Int} \\
| \ Add \ a \ a
\]

\[
\text{type } \ Exp = \ Fix \ Arith
\]

Functors can be combined by coproduct construction \(:+:\)
Data Types à la Carte [Swierstra 2008]

Idea: Decompose data types into two-level types:

**Recursive data type**

\[
\textbf{data} \ Exp = \text{Val} \ Int \\
\quad | \quad \text{Add} \ Exp \ Exp 
\]

**Fixpoint of functor**

\[
\textbf{data} \ Arith a = \text{Val} \ Int \\
\quad | \quad \text{Add} \ a \ a \\
\textbf{type} \ Exp = \text{Fix} \ Arith
\]

Functors can be combined by coproduct construction :++:

\[
\textbf{data} \ Mul a = Mul a a \\
\textbf{type} \ Exp' = \text{Fix} \ (Arith :++: Mul)
\]
Data Types à la Carte [Swierstra 2008]

Idea: Decompose data types into two-level types:

**Recursive data type**

```haskell
data Exp = Val Int | Add Exp Exp
```

**Fixpoint of functor**

```haskell
data Arith a = Val Int | Add a a
```

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

```haskell
data (f :+: g) a = Inl (f a) | Inr (g a)
```

```haskell
data Mul a = Mul a a
```

```haskell
type Exp' = Fix (Arith :+: Mul)
```
Subtyping constraint `≺`:

```
class f ≺ g where
  inj :: f a → g a
  prj :: g a → Maybe (f a)
```
Subtyping constraint $\ll$:

\[
\text{class } f \ll g \text{ where}\\
\text{inj} :: f \; a \rightarrow g \; a \\
\text{prj} :: g \; a \rightarrow \text{Maybe} \; (f \; a)
\]

e.g. Mul $\ll$ Arith $+: Mul$
Subtyping constraint :<:

\[
\text{class } f \preceq g \text{ where }
\]

\[
\begin{align*}
\text{inj} & : f \ a \to g \ a \\
\text{prj} & : g \ a \to \text{Maybe } (f \ a)
\end{align*}
\]

Example: smart constructors

\[
\begin{align*}
\text{add} & : (\text{Arith} \preceq f) \Rightarrow \text{Fix } f \to \text{Fix } f \to \text{Fix } f \\
\text{add } x \ y & = \text{In } (\text{inj } (\text{Add } x \ y))
\end{align*}
\]
Subtyping constraint $\ll$:

\[
\text{class } f \ll g \text{ where} \\
\text{inj} :: f \ a \rightarrow g \ a \\
\text{prj} :: g \ a \rightarrow \text{Maybe} (f \ a)
\]

e.g. \( \text{Mul} \ll \text{Arith} :+: \text{Mul} \)

Example: smart constructors

\[
\text{add} :: (\text{Arith} \ll f) \Rightarrow \text{Fix} f \rightarrow \text{Fix} f \rightarrow \text{Fix} f \\
\text{add} \ x \ y = \text{In} (\text{inj} (\text{Add} \ x \ y))
\]

\[
\text{exp} :: \text{Fix} (\text{Arith} :+: \text{Mul}) \\
\text{exp} = \text{val} 1 \ 	ext{"add"} (\text{val} 2 \ 	ext{"mul"} \ 	ext{val} 3)
\]
Limitations of $\preceq$:

**Definition of $\preceq$:**

- \[\text{instance } f \preceq: f \quad \text{where} \]
  \[
  \ldots
  \]
- \[\text{instance } (f \preceq: f_1) \Rightarrow f \preceq: (f_1 :+: f_2) \quad \text{where} \]
  \[
  \ldots
  \]
- \[\text{instance } (f \preceq: f_2) \Rightarrow f \preceq: (f_1 :+: f_2) \quad \text{where} \]
  \[
  \ldots
  \]

- Asymmetric treatment of $:+$
- Left-hand side is not inspected
- Ambiguity
Limitations of $\preceq$:

**Definition of $\preceq$:**

\[
\begin{align*}
\text{instance} & \quad f \preceq: f \quad \text{where} \\
\quad & \quad \ldots \\
\text{instance} & \quad f \preceq: (f :+: f_2) \quad \text{where} \\
\quad & \quad \ldots \\
\text{instance} (f \preceq: f_2) & \Rightarrow f \preceq: (f_1 :+: f_2) \quad \text{where} \\
\quad & \quad \ldots
\end{align*}
\]
Limitations of \( \triangleleft \):

Definition of \( \triangleleft \):

\[
\text{instance } f \triangleleft f \text{ where } \\
\ldots \\
\text{instance } f \triangleleft (f :+: f_2) \text{ where } \\
\ldots \\
\text{instance } (f \triangleleft f_2) \Rightarrow f \triangleleft (f_1 :+: f_2) \text{ where } \\
\ldots 
\]

- Asymmetric treatment of \( :+: \):
- Left-hand side is not inspected
- Ambiguity
Limitations of $\preceq$:

**Definition of $\preceq$:**

- $\text{instance } f \preceq: f \quad \text{where}$
  
  $\ldots$

- $\text{instance } f \preceq: (f :+: f_2) \quad \text{where}$
  
  $\ldots$

- $\text{instance } (f \preceq: f_2) \Rightarrow f \preceq: (f_1 :+: f_2) \quad \text{where}$
  
  $\ldots$

- Asymmetric treatment of $:+: A \preceq: A :+:(B :+: C)$
- Left-hand side is not inspected
- Ambiguity
Limitations of $≺$:

### Definition of $≺$:

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

- Asymmetric treatment of $:+$:
  
  $A ≺: (A :+ B) :+ C$

- Left-hand side is not inspected

- Ambiguity
Limitations of $\preceq$:

**Definition of $\preceq$:**

\[
\begin{align*}
\text{instance} & \quad f \preceq: f \quad \text{where} \\
& \quad \ldots \\
\text{instance} & \quad f \preceq: (f :+ f_2) \quad \text{where} \\
& \quad \ldots \\
\text{instance} & \quad (f \preceq: f_2) \Rightarrow f \preceq: (f_1 :+ f_2) \quad \text{where} \\
& \quad \ldots
\end{align*}
\]

- Asymmetric treatment of $:+$: $A \not\preceq: (A :+ B) :+ C$
- Left-hand side is not inspected $A :+ B \preceq: (A :+ B) :+ C$
- Ambiguity
Limitations of \( \ll\) :

<table>
<thead>
<tr>
<th>Definition of ( \ll)</th>
</tr>
</thead>
<tbody>
<tr>
<td>instance f ( \ll) f where</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>instance f ( \ll) (f ( \mathbin{:+}) f2) where</td>
</tr>
<tr>
<td>...</td>
</tr>
<tr>
<td>instance (f ( \ll) f2) ( \Rightarrow) f ( \ll) (f1 ( \mathbin{:+}) f2) where</td>
</tr>
<tr>
<td>...</td>
</tr>
</tbody>
</table>

- Asymmetric treatment of \( \mathbin{:+}\):
  \[ A \not\ll (A \mathbin{:+} B) \mathbin{:+} C \]
- Left-hand side is not inspected
  \[ A \mathbin{:+} B \not\ll A \mathbin{:+} (B \mathbin{:+} C) \]
- Ambiguity
Limitations of $\triangleleft$:

**Definition of $\triangleleft$:**

```
instance f $\triangleleft$ f where
...
```

```
instance f $\triangleleft$ (f :+: f2) where
...
```

```
instance (f $\triangleleft$ f2) $\Rightarrow$ f $\triangleleft$ (f1 :+: f2) where
...
```

- Asymmetric treatment of :+:  
  $$A \not\triangleleft (A :+: B) :+: C$$
- Left-hand side is not inspected  
  $$A :+: B \not\triangleleft A :+: (B :+: C)$$
- Ambiguity  
  $$A \triangleleft A :+: (A :+: B)$$
Contributions

We re-implemented \( \prec \): such that:

- Subtyping behaves as intuitively expected*
- Ambiguous subtyping is avoided
- We can express isomorphism \( \simeq \):

*terms and conditions may apply
Improved subtyping constraint :≺:

Subtyping :≺: behaves as intuitively expected
Improved subtyping constraint \( \prec \):

\[
\text{Subtyping} \: \prec \: \text{ behaves as intuitively expected}
\]

\[
f \prec g \iff \exists \text{ unique injection from } f \text{ to } g
\]
Improved subtyping constraint $\triangleleft$:

**Subtyping $\triangleleft$:** behaves as intuitively expected

\[ f \triangleleft g \iff \exists \text{ unique injection from } f \text{ to } g \]

\[ C :+: A \triangleleft A :+: B :+: C \]
Improved subtyping constraint $\prec$:

**Subtyping $\prec$:** behaves as intuitively expected

\[ f \prec g \iff \exists \text{ unique injection from } f \text{ to } g \]

\[ C :+: A \prec A :+: B :+: C \]

**Avoid ambiguous subtyping**

Multiple occurrences of signatures are rejected:
Improved subtyping constraint $\ll$: 

**Subtyping $\ll$: behaves as intuitively expected**

\[ f :\ll: g \iff \exists \text{ unique injection from } f \text{ to } g \]

\[ C : ++: A \ll: A : ++: B : ++: C \]

**Avoid ambiguous subtyping**

Multiple occurrences of signatures are rejected:

\[ A \ll: A : ++: A : ++: C \]

\[ A : ++: A \ll: A : ++: B \]
**Improved subtyping constraint $\ll$:**

**Subtyping $\ll$: behaves as intuitively expected**

\[
f : \ll : g \iff \exists \text{ unique injection from } f \text{ to } g
\]

\[C : A : A : B : C\]

**Avoid ambiguous subtyping**

Multiple occurrences of signatures are rejected:

\[A : A : A : C\]

\[A : A : A : B\]
Improved subtyping constraint $\ll$: 

**Subtyping $\ll$: behaves as intuitively expected**

\[ f \ll g \iff \exists \text{ unique injection from } f \text{ to } g \]

\[ C :+; A \ll A :+; B :+; C \]

**Avoid ambiguous subtyping**

Multiple occurrences of signatures:

\[ A :\not\ll A :+; A :+; C \]

\[ A :+; A \ll A :+; B \]

injection not unique!
Improved subtyping constraint $\triangleleft$:

**Subtyping $\triangleleft$:** behaves as intuitively expected

$$f \triangleleft g \iff \exists \text{ unique injection from } f \text{ to } g$$

$$C : A \triangleleft A : B \triangleleft C$$

**Avoid ambiguous subtyping**

Multiple occurrences of signatures are rejected:

$$A \triangleleft A : A \triangleleft C$$

$$A : A \triangleleft A : B$$

injection not unique!

“injection” not injective!
Improved subtyping constraint \( \prec \):

**Subtyping \( \prec \): behaves as intuitively expected**

\[
f \prec g \iff \exists \text{ unique injection from } f \text{ to } g
\]

\[C :+ : A \prec A :+ : B :+ : C\]

**Avoid ambiguous subtyping**

Multiple occurrences of signatures:

\[A : \not\prec A :+ : A :+ : C\]

\[A :+ : A : \not\prec A :+ : B\]

"injection" not injective!
Type isomorphism constraint :≈:

We can express isomorphism :≈:

\[ f :\approx: g \iff \exists \text{ unique bijection from } f \text{ to } g \]
**Type isomorphism constraint \(\simeq:\)**

We can express isomorphism \(\simeq:\):

\[
f : \simeq : g \iff \exists \text{ unique bijection from } f \text{ to } g
\]

Easy to implement: 

\[
f : \simeq : g = (f : \prec : g, g : \prec : f)
\]
Type isomorphism constraint \( \simeq \):

We can express isomorphism \( \simeq \):

\[
f : \simeq : g \iff \exists \text{ unique bijection from } f \text{ to } g
\]

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:

\[
\text{prj} :: (f : \prec : g) \Rightarrow g \ a \rightarrow \text{Maybe} \ (f \ a)
\]
Type isomorphism constraint $\simeq$:

We can express isomorphism $\simeq$:

$$f \simeq g \iff \exists \text{ unique bijection from } f \text{ to } g$$

Easy to implement:

$$f \simeq g = (f \lll g, g \lll f)$$

Use case: improved projection function

The type of the projection function is unsatisfying:

$$\text{prj} :: (f \lll g) \Rightarrow g \ a \rightarrow \text{Maybe} \ (f \ a)$$

With $\simeq$: we can do better:

$$\text{split} :: (g \simeq f :+ r) \Rightarrow g \ a \rightarrow \text{Either} \ (f \ a) \ (r \ a)$$
Type isomorphism constraint \(\simeq\):

We can express isomorphism \(\simeq\):

\[
f : \simeq g \iff \exists \text{ unique bijection from } f \text{ to } g
\]

Easy to implement:

\[f \simeq g = (f \preceq g, g \preceq f)\]

Use case: improved projection function

The type of the projection function is unsatisfying:

\[
prj :: (f \preceq\preceq g) \Rightarrow g \ a \rightarrow \text{Maybe } (f \ a)
\]

With \(\simeq\): we can do better:

\[
split :: (g \simeq f \triangleright r) \Rightarrow g \ a \rightarrow (f \ a \rightarrow b) \rightarrow (r \ a \rightarrow b) \rightarrow b
\]
Implementation of ≺:
Idea

Type-level function *Embed*:  
- take two signatures $f$, $g$ as arguments  
- check whether $f \preceq g$

No singleton types. This all happens at compile time!
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  
- produce proof object \( p \) for \( f :\prec: g \)

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

Type-level function $Embed$:

- take two signatures $f$, $g$ as arguments
- produce proof object $p$ 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
- produce proof object \( p \) for \( f :<: g \)

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!
Idea

Type-level function \textit{Embed}:

\begin{itemize}
  \item take two signatures \(f, g\) as arguments
  \item produce \textbf{proof object} \(p\) for \(f :≼:\leq: g\)
\end{itemize}

Derive implementation of \textit{inj} and \textit{prj}:

\begin{itemize}
  \item also use a type class
  \item But: use proof object as \textit{oracle} in instance declarations
\end{itemize}

No singleton types. This all happens at compile time!
Idea

Type-level function *Embed*:
- take two signatures \( f, g \) as arguments
- produce *proof object* \( p \) for \( f : \leftarrow : g \)
- check whether \( p \) also proves \( f : \leftarrow : g \)

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**

\[
\begin{align*}
data \ Prf &= \text{Refl} \mid \text{Left Prf} \mid \text{Right Prf} \mid \text{Sum Prf Prf}
\end{align*}
\]
Proof Objects

Definition

\[
data \ Prf = \text{Refl} \mid \text{Left } Prf \mid \text{Right } Prf \mid \text{Sum } Prf \ Prf
\]

\[
\text{Refl} : f \dashv \vdash f
\]
Proof Objects

Definition

\[
data \ Prf = \text{Refl} \mid \text{Left Prf} \mid \text{Right Prf} \mid \text{Sum Prf Prf}
\]

\[
\begin{align*}
\text{Refl} : f & : \leftrightarrow : f \\
\text{Left} p : f & : \leftrightarrow : g_1 \quad \text{Left} p : f & : \leftrightarrow : g_1 : \vdash : g_2 \\
\text{Right} p : f & : \leftrightarrow : g_2 \quad \text{Right} p : f & : \leftrightarrow : g_1 : \vdash : g_2
\end{align*}
\]
Proof Objects

Definition

\[ \text{data } \Prf = \text{Refl} \mid \text{Left } \Prf \mid \text{Right } \Prf \mid \text{Sum } \Prf \Prf \]

- \( \text{Refl} : f \leftrightarrow f \)
- \( p : f \leftrightarrow g_1 \quad \text{Left } p : f \leftrightarrow g_1 :+: g_2 \)
- \( p : f \leftrightarrow g_2 \quad \text{Right } p : f \leftrightarrow g_1 :+: g_2 \)
- \( p_1 : f_1 \leftrightarrow g \quad p_2 : f_2 \leftrightarrow g \)
- \( \text{Sum } p_1 \ p_2 : f_1 :+: f_2 \leftrightarrow g \)
Proof Objects

\[
\textbf{data} \Prf = \text{Refl} \mid \text{Left } \Prf \mid \text{Right } \Prf \mid \text{Sum } \Prf \Prf
\]

\[
\text{Refl} : f \leftrightarrow f
\]

\[
p : f \leftrightarrow g_1 \quad \text{Left } p : f \leftrightarrow g_1 : +: g_2
\]

\[
p : f \leftrightarrow g_2 \quad \text{Right } p : f \leftrightarrow g_1 : +: g_2
\]

\[
p_1 : f_1 \leftrightarrow g \quad p_2 : f_2 \leftrightarrow g
\]

\[
\text{Sum } p_1 \ p_2 : f_1 : +: f_2 \leftrightarrow g
\]
Proof Objects

**Definition**

\[
data \ Prf = \text{Refl} \mid \text{Left } Prf \mid \text{Right } Prf \mid \text{Sum } Prf \ Prf
\]

- \(\text{Refl} : f :<\!\!: f\)
- \(\text{Left } p : f :<\!\!: g_1 +: g_2\)
- \(\text{Right } p : f :<\!\!: g_1 +: g_2\)
- \(\text{Sum } p_1, p_2 : f_1 +: f_2 :<\!\!: g\)
Proof Objects

**Definition**

\[
data \text{Prf} = \text{Refl} \mid \text{Left Prf} \mid \text{Right Prf} \mid \text{Sum Prf Prf}
\]

- **Refl**: \( f : \langle\langle f \rangle\rangle \)
- **Left**: \( p : f :\langle\langle g \rangle\rangle \) \( \Rightarrow \) \( \text{Left } p : f :\langle\langle g_1 \rangle\rangle :+: g_2 \)
- **Right**: \( p : f :\langle\langle g \rangle\rangle \) \( \Rightarrow \) \( \text{Right } p : f :\langle\langle g_1 \rangle\rangle :+: g_2 \)
- **Sum**: \( p_1 : f_1 :\langle\langle g \rangle\rangle \), \( p_2 : f_2 :\langle\langle g \rangle\rangle \) \( \Rightarrow \) \( \text{Sum } p_1 \ p_2 : f_1 :+: f_2 :\langle\langle g \rangle\rangle \)
Construct Proof Objects

\textbf{data} \textit{Emb} = \textit{Found Prf} \mid \textit{NotFound} \mid \textit{Ambiguous}
Construct Proof Objects

data \texttt{Emb} = \texttt{Found Prf} | \texttt{NotFound} | \texttt{Ambiguous}

type family \texttt{Embed} (f :: \ast \rightarrow \ast) (g :: \ast \rightarrow \ast) :: \texttt{Emb} where
Construct Proof Objects

**data** $Emb = Found \ Prf \mid NotFound \mid Ambiguous$

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

- $Embed \ f \ f = Found \ \text{Refl}$
- $Embed \ (f_1 :+ : f_2) \ g = \text{Sum}' \ (Embed \ f_1 \ g) \ (Embed \ f_2 \ g)$
- $Embed \ f \ (g_1 :+ : g_2) = \text{Choose} \ (Embed \ f \ g_1) \ (Embed \ f \ g_2)$
- $Embed \ f \ g = NotFound$
Construct Proof Objects

**data** \( Emb = \text{Found Prf} \mid \text{NotFound} \mid \text{Ambiguous} \)

**type family** \( \text{Embed} (f :: \ast \rightarrow \ast) (g :: \ast \rightarrow \ast) :: Emb \)** where

- \( \text{Embed } f \ f = \text{Found Refl} \)
- \( \text{Embed } (f_1 :+: f_2) \ g = \text{Sum'} (\text{Embed } f_1 \ g) (\text{Embed } f_2 \ g) \)
- \( \text{Embed } f \ (g_1 :+: g_2) = \text{Choose } (\text{Embed } f \ g_1) (\text{Embed } f \ g_2) \)
- \( \text{Embed } f \ g = \text{NotFound} \)

**type family** \( \text{Choose } (e_1 :: Emb) (e_2 :: Emb) :: Emb \)** where

- \( \text{Choose } (\text{Found } p_1) (\text{Found } p_2) = \text{Ambiguous} \)
- \( \text{Choose } \text{Ambiguous} \ e_2 = \text{Ambiguous} \)
- \( \text{Choose } e_1 \text{ Ambiguous} = \text{Ambiguous} \)
- \( \text{Choose } (\text{Found } p_1) \ e_2 = \text{Found } (\text{Left } p_1) \)
- \( \text{Choose } e_1 \ (\text{Found } p_2) = \text{Found } (\text{Right } p_2) \)
- \( \text{Choose } \text{NotFound} \ \text{NotFound} = \text{NotFound} \)
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:

  $$A \not\preceq A :+ A :+ C$$

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

  $$A :+ A \prec A :+ B$$
Post-Processing

This is almost what we want.

- We avoid ambiguity on the right-hand side:
  
  \[ A \not\approx A :+ A :+ C \]

- We still have ambiguity on the left-hand side:
  
  \[ A :+ A \not\approx A :+ B \]

Solution: check for duplicates in \( \text{Prf} \)

\textbf{type family} \( \text{Dupl} \) \( (p :: \text{Prf}) :: \text{Bool} \) \textbf{where}

\[ \ldots \]
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:

  \[ A :+ A \prec A :+ B \]

Solution: check for duplicates in \( Prf \)

\textbf{type family} \( \text{Dupl} \ (p :: Prf) :: \text{Bool} \) \textbf{where}

\[ \ldots \]
Are we there yet?

- Construct proof $p$ for $f : \precsim \precsim : g$
- Check whether $p$ proves $f : \precsim : g$
- Derive $inj$ and $prj$
Are we there yet?

- Construct proof $p$ for $f : \prec \prec : g$
- Check whether $p$ proves $f : \prec : g$
- Derive $inj$ and $prj$
Are we there yet?

- Construct proof $p$ for $f :\sim:\sim g$ ✓
- Check whether $p$ proves $f :\sim: g$ ✓
- Derive $inj$ and $prj$
Are we there yet?

- Construct proof $p$ for $f \xleftrightarrow{} g$ ✓
- Check whether $p$ proves $f \xleftrightarrow{} g$ ✓
- Derive $inj$ and $prj$
Derive \( inj \) and \( prj \)

\[
\text{class } f \preceq g \text{ where}
\]

\[
i \text{inj} :: f \ a \to g \ a
\]

\[
prj :: g \ a \to \text{Maybe} \ (f \ a)
\]

\[
\text{instance } f \preceq f \text{ where} \ldots
\]

\[
\text{instance } f \preceq (f :+: g_2) \text{ where} \ldots
\]

\[
\text{instance } f \preceq g_2 \Rightarrow
f \preceq (g_1 :+ g_2) \text{ where} \ldots
\]
Derive \textit{inj} and \textit{prj}

\begin{verbatim}
class f <: g where
    inj :: f a \to g a
    prj :: g a \to Maybe (f a)
\end{verbatim}

\begin{verbatim}
instance
    f <: f where \ldots

instance
    \Rightarrow
    f <: g_1
    f <: (g_1 :+: g_2) where \ldots

instance
    \Rightarrow
    f <: g_2
    f <: (g_1 :+: g_2) where \ldots
\end{verbatim}
Derive \( \text{inj} \) and \( \text{prj} \)

```haskell
class \( f \preceq g \) where
    inj :: \( f \ a \rightarrow g \ a \)
    prj :: \( g \ a \rightarrow \text{Maybe} (f \ a) \)
```

```haskell
instance \( f \preceq f \) where
    ...

instance \( f \preceq g_1 \) where
    ....
    \( f \preceq (g_1 :+: g_2) \) where ...

instance \( f \preceq g_2 \) where
    ....
    \( f \preceq (g_1 :+: g_2) \) where ...

instance \( (f_1 \preceq g, f_2 \preceq g) \) where
    ....
    \( (f_1 :+: f_2) \preceq g \) where ...
```
Derive \( inj \) and \( prj \)

```haskell
class Sub f g where
  inj :: f a → g a
  prj :: g a → Maybe (f a)
```

```haskell
instance Sub f f where...
instance Sub f g₁ ⇒ Sub f (g₁ :+: g₂) where...
instance Sub f g₂ ⇒ Sub f (g₁ :+: g₂) where...
instance (Sub f₁ g₁, Sub f₂ g) ⇒ Sub (f₁ :+: f₂) g where...
```
Derive \textit{inj} and \textit{prj}

\textbf{class} \textit{Sub} (\textit{e :: Emb}) \textit{f g where}
\begin{itemize}
  \item \textit{inj} :: \textit{f a} \rightarrow \textit{g a}
  \item \textit{prj} :: \textit{g a} \rightarrow \text{Maybe} (\textit{f a})
\end{itemize}

\textbf{instance} \textit{Sub} \textit{f f} \textit{where . . .}

\textbf{instance} \textit{Sub} \textit{f g_1} \textit{Sub} \textit{f (g_1 :+: g_2)} \textit{where . . .}

\textbf{instance} \textit{Sub} \textit{f g_2} \textit{Sub} \textit{f (g_1 :+: g_2)} \textit{where . . .}

\textbf{instance} \textit{(Sub f_1 g, Sub f_2 g)} \textit{Sub} \textit{(f_1 :+: f_2)} \textit{g where . . .}
Derive \( \text{inj} \) and \( \text{prj} \)

```haskell
class Sub (e :: Emb) f g where
    inj :: f a \rightarrow g a
    prj :: g a \rightarrow \text{Maybe} (f a)
```

```haskell
instance Sub (Found Refl) f f where . . .

instance Sub (Found p) f g_1
  \Rightarrow Sub (Found (Left p)) f (g_1 :+: g_2) where . . .

instance Sub (Found p) f g_2
  \Rightarrow Sub (Found (Right p)) f (g_1 :+: g_2) where . . .

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 where . . .
```
Derive $\text{inj}$ and $\text{prj}$

\[
\text{class } \text{Sub} \ (e :: \text{Emb}) \ f \ g \ \text{where}
\]
\[
\begin{align*}
\text{inj} & :: f \ a \rightarrow g \ a \\
\text{prj} & :: g \ a \rightarrow \text{Maybe} \ (f \ a)
\end{align*}
\]

\[
\text{type } f \prec g = \text{Sub} \ (\text{Embed} \ f \ g) \ f \ g
\]

\[
\text{instance } \text{Sub} \ (\text{Found} \ \text{Refl}) \ f \ f \ \text{where} \ldots
\]

\[
\text{instance } \text{Sub} \ (\text{Found} \ p) \ f \ g_1 \\
\Rightarrow \text{Sub} \ (\text{Found} \ (\text{Left} \ p)) \ f \ (g_1 :+ g_2) \ \text{where} \ldots
\]

\[
\text{instance } \text{Sub} \ (\text{Found} \ p) \ f \ g_2 \\
\Rightarrow \text{Sub} \ (\text{Found} \ (\text{Right} \ p)) \ f \ (g_1 :+ g_2) \ \text{where} \ldots
\]

\[
\text{instance } (\text{Sub} \ (\text{Found} \ p_1) \ f_1 \ g, \text{Sub} \ (\text{Found} \ p_2) \ f_2 \ g) \\
\Rightarrow \text{Sub} \ (\text{Found} \ (\text{Sum} \ p_1 p_2)) \ (f_1 :+ f_2) \ g \ \text{where} \ldots
\]
Conclusion

- This approach generalises to similar applications
- Improves type class-based implementation in many aspects

But:
- We need a way to customise error messages.
- Compile time performance unpredictable.

Implemented in the `compdata` package

> cabal install compdata
Conclusion

- This approach generalises to similar applications
- Improves type class-based implementation in many aspects
- But:
  - We need a way to customise error messages.
  - Compile time performance unpredictable.
Conclusion

• This approach generalises to similar applications

• Improves type class-based implementation in many aspects

• But:
  • We need a way to customise error messages.
  • Compile time performance unpredictable.

• Implemented in the compdata package

> cabal install compdata
Discussion
Error Messages

- \( A :\ll B :\| C \) ?
Error Messages

- $A :<: B :+: C$?

  No instance for
  $(\text{Sub} \ \text{NotFound} \ A \ (B :+: C))$
Error Messages

• $A \ll B :+ C$?

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

The original implementation would give:
No instance for (A :< C)
Error Messages

- \( A :\ll : B :\ll + : C ? \)
  
  No instance for
  
  (Sub NotFound A (B :\ll + : C))

- \( A :\ll + : A :\ll : A :\ll + : B ? \)
  
  No instance for
  
  (Sub Ambiguous (A :\ll + : A) (A :\ll + : B))
Error Messages

- $A :\ll : B :+: C$ ?
  
  No instance for
  
  (Sub NotFound $A$ ($B :+: C$))

- $A :+: A :\ll : A :+: B$ ?
  
  No instance for
  
  (Sub Ambiguous ($A :+: A$) ($A :+: B$))

- $A :\ll : A :+: B$ ?
Error Messages

• \( A \prec B :+ : C \) ?

  No instance for
  \( (\text{Sub NotFound } A (B :+ : C)) \)

• \( A :+ : A \prec A :+ : B \) ?

  No instance for
  \( (\text{Sub Ambiguous } (A :+ : A) (A :+ : B)) \)

• \( a \prec a :+ : B \) ?
Error Messages

- $A : \triangleleft: B :+: C$?
  
  No instance for
  (Sub NotFound A (B :+: C))

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

- $a : \triangleleft: a :+: B$?
  
  No instance for
  (Sub (Post (Embed a (a :+: B))) a (a :+: B))
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: $O(n^2)$
- Slightly different implementation: $O(2^n)$
  (but essentially the same)
Compile Time Performance

- If done “wrong”, this implementation can be very slow!
- Implementation presented here: $O(n^2)$
- Slightly different implementation: $O(2^n)$ (but essentially the same)
- Micro benchmark:
  - derive $F \preceq: G$
  - 9 summands in $F$ and $G$
Compile Time Performance

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