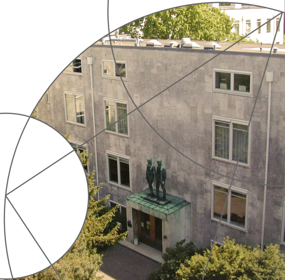
Faculty of Science

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 $:\prec:$

- 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

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

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
```

$\Longrightarrow$

Fixpoint of functor

```
data Arith a = 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 $Exp = Val\ Int$
$\quad |\ Add\ Exp\ Exp$

$\longrightarrow$

Fixpoint of functor

data $Fix\ f = In\ (f\ (Fix\ f))$

$= Val\ Int$
$\quad |\ Add\ a\ a$

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*

Data Types à la Carte [Swierstra 2008]

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 [Swierstra 2008]

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 :+:

$$\textbf{data}\ \mathit{Mul}\ a = \mathit{Mul}\ a\ a$$

$$\textbf{type}\ \mathit{Exp}' = \mathit{Fix}\ (\mathit{Arith} :+: \mathit{Mul})$$

Data Types à la Carte [Swierstra 2008]

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)
  | *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 $:\prec:$

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

Data Types à la Carte (cont.)

Subtyping constraint :≺:

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

e.g. Mul :≺: $Arith$:+: Mul

Data Types à la Carte (cont.)

Subtyping constraint $:\prec:$

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

e.g. $Mul :\prec: Arith :+: Mul$

Example: smart constructors

$$
\begin{array}{l}
add :: (Arith :\prec: f) \Rightarrow Fix\ f \rightarrow Fix\ f \rightarrow Fix\ f \\
add\ x\ y = In\ (inj\ (Add\ x\ y))
\end{array}
$$

Data Types à la Carte (cont.)

Subtyping constraint $:\prec:$

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

e.g. $Mul :\prec: Arith :+: Mul$

Example: smart constructors

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

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

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 :≺:

Definition of :≺:

$$\textbf{instance} \quad f :\prec: f \quad \textbf{where}$$
$$\ldots$$
$$\textbf{instance} \quad f :\prec: (f :+: f_2) \ \textbf{where}$$
$$\ldots$$
$$\textbf{instance} \ (f :\prec: f_2) \Rightarrow f :\prec: (f_1 :+: f_2) \ \textbf{where}$$
$$\ldots$$

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**

...

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

Limitations of $:\prec:$

Definition of $:\prec:$

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

$\ldots$

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

$\ldots$

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

$\ldots$

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

Limitations of $:\prec:$

Definition of $:\prec:$

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

$\ldots$

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

$\ldots$

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

$\ldots$

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

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**

…

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

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**
…

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

Limitations of :≺:

Definition of :≺:

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**
  ...

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

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 $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \iff \quad \exists$ unique injection from f to g

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \Longleftrightarrow \quad \exists$ unique injection from f to g

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

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \Longleftrightarrow \quad \exists$ unique injection from f to 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

$f :\prec: g \quad \Longleftrightarrow \quad \exists$ unique injection from f to 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

$f :\prec: g \iff \exists$ unique injection from f to 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

$f :\prec: g \iff \exists$ unique injection from f to 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

$f :\prec: g \quad \iff \quad \exists$ unique injection from f to g

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

Avoid ambiguous subtyping

Multiple occurrences of signatur[injection not unique!]

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

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

"injection" not injective!

Improved subtyping constraint $:\prec:$

Subtyping $:\prec:$ behaves as intuitively expected

$f :\prec: g \quad \iff \quad \exists$ unique injection from f to g

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

Avoid ambiguous subtyping

Multiple occurrences of signatur

injection not unique!

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

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

"injection" not injective!

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \quad \iff \quad \exists$ unique bijection from f to g

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \quad \iff \quad \exists$ unique bijection from f to g

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

Type isomorphism constraint :≃:

We can express isomorphism :≃:

$f :\simeq: g \quad \Longleftrightarrow \quad \exists$ unique bijection from f to 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:

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

Type isomorphism constraint $:\simeq:$

We can express isomorphism $:\simeq:$

$f :\simeq: g \quad \iff \quad \exists$ unique bijection from f to 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 :≃:

We can express isomorphism :≃:

$f :\simeq: g \iff \exists$ unique bijection from f 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:

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

With :≃: we can do better:

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

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
- 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 $f :\prec: g$

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!

Idea

Type-level function *Embed*:

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

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!

Idea

Type-level function *Embed*:

- take two signatures f, g as arguments
- produce proof object p for $f :\prec\!\!\prec: g$
- check whether p also proves $f :\prec: g$

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

$$\textbf{data}\ \mathit{Prf} = \mathit{Refl} \mid \mathit{Left}\ \mathit{Prf} \mid \mathit{Right}\ \mathit{Prf} \mid \mathit{Sum}\ \mathit{Prf}\ \mathit{Prf}$$

Proof Objects

Definition

$$\textbf{data}\ \mathit{Prf} = \mathit{Refl} \mid \mathit{Left}\ \mathit{Prf} \mid \mathit{Right}\ \mathit{Prf} \mid \mathit{Sum}\ \mathit{Prf}\ \mathit{Prf}$$

$$\frac{}{\mathit{Refl} : f :\prec\!\!\prec: f}$$

Proof Objects

Definition

$$\textbf{data}\ \mathit{Prf} = \mathit{Refl} \mid \mathit{Left}\ \mathit{Prf} \mid \mathit{Right}\ \mathit{Prf} \mid \mathit{Sum}\ \mathit{Prf}\ \mathit{Prf}$$

$$\frac{}{\mathit{Refl} : f :\prec\!\!\prec: f}$$

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

Proof Objects

Definition

data *Prf* = *Refl* | *Left Prf* | *Right Prf* | *Sum Prf Prf*

$$\frac{}{\mathit{Refl} : f :\prec\!\!\prec: f}$$

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

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

Proof Objects

Defin kind

data *Prf* = *Refl* | *Left Prf* | *Right Prf* | *Sum Prf Prf*

$$\frac{}{\mathit{Refl} : f \mathrel{:\prec\!\!\prec:} f}$$

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

$$\frac{p_1 : f_1 \mathrel{:\prec\!\!\prec:} g \qquad p_2 : f_2 \mathrel{:\prec\!\!\prec:} g}{\mathit{Sum}\ p_1\ p_2 : f_1 \mathrel{:+:} f_2 \mathrel{:\prec\!\!\prec:} g}$$

Proof Objects

Defin kind type

$$\textbf{data}\ \mathit{Prf} = \mathit{Refl} \mid \mathit{Left}\ \mathit{Prf} \mid \mathit{Right}\ \mathit{Prf} \mid \mathit{Sum}\ \mathit{Prf}\ \mathit{Prf}$$

$$\frac{}{\mathit{Refl} : f :\prec\!\!\prec: f}$$

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

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

Proof Objects

Defin kind type

data *Prf* = *Refl* | *Left Prf* | *Right Prf* | *Sum Prf Prf*

type constructor

$$\frac{}{\mathit{Refl} : f :\prec\!\!\prec: f}$$

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

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

Construct Proof Objects

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

Construct Proof Objects

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

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

Construct Proof Objects

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

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

$$
\begin{array}{ll}
Embed\ f\ f & = Found\ Refl \\
Embed\ (f_1 :+: f_2)\ g & = Sum'\ (Embed\ f_1\ g)\ (Embed\ f_2\ g) \\
Embed\ f\ (g_1 :+: g_2) & = Choose\ (Embed\ f\ g_1)\ (Embed\ f\ g_2) \\
Embed\ f\ g & = NotFound
\end{array}
$$

Construct Proof Objects

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

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

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

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

$$
\begin{aligned}
&\mathit{Choose}\ (\mathit{Found}\ p_1)\ (\mathit{Found}\ p_2) &&= \mathit{Ambiguous} \\
&\mathit{Choose}\ \mathit{Ambiguous}\ e_2 &&= \mathit{Ambiguous} \\
&\mathit{Choose}\ e_1\ \mathit{Ambiguous} &&= \mathit{Ambiguous} \\
&\mathit{Choose}\ (\mathit{Found}\ p_1)\ e_2 &&= \mathit{Found}\ (\mathit{Left}\ p_1) \\
&\mathit{Choose}\ e_1\ (\mathit{Found}\ p_2) &&= \mathit{Found}\ (\mathit{Right}\ p_2) \\
&\mathit{Choose}\ \mathit{NotFound}\ \mathit{NotFound} &&= \mathit{NotFound}
\end{aligned}
$$

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:

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

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

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

Post-Processing

This is almost what we want.

- We avoid ambiguity on the right-hand side:

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

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

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

Solution: check for duplicates in *Prf*

type family *Dupl* (*p* :: *Prf*) :: *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 *Sum* (*Left Refl*) (*Left Refl*)

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

Solution: check for duplicates in *Prf*

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

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 $f :\prec\!\!\prec: g$ ✓
- Check whether p proves $f :\prec: g$
- 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 $f :\prec\!\!\prec: g$ ✓
- Check whether p proves $f :\prec: g$ ✓
- Derive *inj* and *prj*

Derive *inj* and *prj*

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

$$\textbf{instance}\quad f :\prec: f \quad \textbf{where} \ldots$$

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

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

Derive *inj* and *prj*

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

$$\textbf{instance}\ f :\prec: f\ \textbf{where} \ldots$$

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

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

Derive *inj* and *prj*

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

$$
\begin{array}{ll}
\textbf{instance} & f \mathrel{:\prec:} f \quad \textbf{where} \ldots \\
\\
\textbf{instance} & f \mathrel{:\prec:} g_1 \\
\quad \Rightarrow & f \mathrel{:\prec:} (g_1 \mathrel{:+:} g_2) \quad \textbf{where} \ldots \\
\\
\textbf{instance} & f \mathrel{:\prec:} g_2 \\
\quad \Rightarrow & f \mathrel{:\prec:} (g_1 \mathrel{:+:} g_2) \quad \textbf{where} \ldots \\
\\
\textbf{instance}\ (& f_1 \mathrel{:\prec:} g, \quad f_2 \mathrel{:\prec:} g) \\
\quad \Rightarrow & (f_1 \mathrel{:+:} f_2) \mathrel{:\prec:} g\ \textbf{where} \ldots
\end{array}
$$

Derive *inj* and *prj*

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

```
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 g, Sub f2 g)
  ⇒ Sub (f1 :+: f2) g where ...
```

Derive *inj* and *prj*

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

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

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

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

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

Derive *inj* and *prj*

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

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

instance Sub (Found p)           f  g1
  ⇒     Sub (Found (Left p))    f  (g1 :+: g2)    where ...

instance Sub (Found p)           f  g2
  ⇒     Sub (Found (Right p))   f  (g1 :+: g2)    where ...

instance (Sub (Found p1) f1    g, Sub (Found p2) f2    g)
  ⇒     Sub (Found (Sum p1 p2))  (f1 :+: f2)    g where ...
```

Derive *inj* and *prj*

$$
\begin{array}{l}
\textbf{class}\ Sub\ (e :: Emb)\ f\ \ g\ \textbf{where} \\
\quad inj :: f\ a \to g\ a \\
\quad prj :: g\ a \to Maybe\ (f\ a) \\
\\
\textbf{type}\ f :\prec: g = Sub\ (Embed\ f\ g)\ f\ g \\
\\
\textbf{instance}\ Sub\ (Found\ Refl)\ f\ f\ \textbf{where} \ldots \\
\\
\textbf{instance}\ Sub\ (Found\ p)\ f\ g_1 \\
\quad \Rightarrow Sub\ (Found\ (Left\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots \\
\\
\textbf{instance}\ Sub\ (Found\ p)\ f\ g_2 \\
\quad \Rightarrow Sub\ (Found\ (Right\ p))\ f\ (g_1 :+: g_2)\ \textbf{where} \ldots \\
\\
\textbf{instance}\ (Sub\ (Found\ p_1)\ f_1\ g, Sub\ (Found\ p_2)\ f_2\ g) \\
\quad \Rightarrow Sub\ (Found\ (Sum\ p_1\ p_2))\ (f_1 :+: f_2)\ g\ \textbf{where} \ldots
\end{array}
$$

Conclusion

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

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

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

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

- $A \mathrel{:\prec:} B \mathrel{:+:} C$?

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

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

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

- $a \mathrel{:\prec:} a \mathrel{:+:} B$?

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))
```

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!