Recovering explicit recursion in Generic Haskell

Andres Löh

16. Oktober 2002

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
equal\langle t :: * \rangle & :: t \to t \to Bool \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle a :+: b \rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a \rangle\ a_1\ a_2 \\
equal\langle a :+: b \rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b \rangle\ b_1\ b_2 \\
equal\langle a :+: b \rangle\ \_\ \_ & = False \\
equal\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = equal\langle a \rangle\ a_1\ a_2 \wedge equal\langle b \rangle\ b_1\ b_2
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{type}\ Equal\langle\!\langle * \rangle\!\rangle\ t & = t \to t \to Bool \\
\mathbf{type}\ Equal\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u. Equal\langle\!\langle \kappa \rangle\!\rangle\ u \to Equal\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
equal\langle t :: \kappa \rangle & :: Equal\langle\!\langle \kappa \rangle\!\rangle\ t \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle :+: \rangle\ eq_a\ eq_b\ (Inl\ a_1)\ (Inl\ a_2) & = eq_a\ a_1\ a_2 \\
equal\langle :+: \rangle\ eq_a\ eq_b\ (Inr\ b_1)\ (Inr\ b_2) & = eq_b\ b_1\ b_2 \\
equal\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = False \\
equal\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\mathit{equal}\langle t :: * \rangle & :: t \to t \to \mathit{Bool} \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{equal}\langle b \rangle\ b_1\ b_2 \\
\mathit{equal}\langle a :+: b \rangle\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \wedge \mathit{equal}\langle b \rangle\ b_1\ b_2
\end{array}
$$

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\mathit{equal}\langle t :: * \rangle & :: t \to t \to \mathit{Bool} \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{equal}\langle b \rangle\ b_1\ b_2 \\
\mathit{equal}\langle a :+: b \rangle\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \wedge \mathit{equal}\langle b \rangle\ b_1\ b_2
\end{array}
$$

➜ POPL-style

Two styles

Ralf Hinze introduced two styles of generic functions:

$$\begin{array}{ll}
\mathit{equal}\langle t :: * \rangle & :: t \to t \to \mathit{Bool} \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{equal}\langle b \rangle\ b_1\ b_2 \\
\mathit{equal}\langle a :+: b \rangle\ _\ _ & = \mathit{False} \\
\mathit{equal}\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \wedge \mathit{equal}\langle b \rangle\ b_1\ b_2
\end{array}$$

- ➜ POPL-style
- ✓ explicit recursion

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
equal\langle t :: * \rangle & :: t \to t \to Bool \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle a :+: b \rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a \rangle\ a_1\ a_2 \\
equal\langle a :+: b \rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b \rangle\ b_1\ b_2 \\
equal\langle a :+: b \rangle\ \_\ \_ & = False \\
equal\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = equal\langle a \rangle\ a_1\ a_2 \wedge equal\langle b \rangle\ b_1\ b_2
\end{array}
$$

- ➜ POPL-style
- ✓ explicit recursion
- ✓ simple type signature

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
equal\langle t :: * \rangle & :: t \to t \to Bool \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle a :+: b \rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a \rangle\ a_1\ a_2 \\
equal\langle a :+: b \rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b \rangle\ b_1\ b_2 \\
equal\langle a :+: b \rangle\ \_\ \_ & = False \\
equal\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = equal\langle a \rangle\ a_1\ a_2 \wedge equal\langle b \rangle\ b_1\ b_2
\end{array}
$$

- ➜ POPL-style
- ✓ explicit recursion
- ✓ simple type signature
- ✗ works only for kind $*$ types

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
equal\langle t :: * \rangle & :: t \to t \to Bool \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle a :+: b \rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a \rangle\ a_1\ a_2 \\
equal\langle a :+: b \rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b \rangle\ b_1\ b_2 \\
equal\langle a :+: b \rangle\ \_\ \_ & = False \\
equal\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = equal\langle a \rangle\ a_1\ a_2 \wedge equal\langle b \rangle\ b_1\ b_2
\end{array}
$$

- ➜ POPL-style
- ✓ explicit recursion
- ✓ simple type signature
- ✗ works only for kind $*$ types
- ✗ not implemented in Generic Haskell

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Equal}\langle\!\langle *\rangle\!\rangle\ t & = t \to t \to \mathit{Bool} \\
\textbf{type}\ \mathit{Equal}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Equal}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{equal}\langle t :: \kappa \rangle & :: \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = eq_a\ a_1\ a_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = eq_b\ b_1\ b_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\textbf{type } \mathit{Equal}\langle\!\langle * \rangle\!\rangle\ t & = t \to t \to \mathit{Bool} \\
\textbf{type } \mathit{Equal}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u. \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Equal}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{equal}\langle t :: \kappa \rangle & :: \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = eq_a\ a_1\ a_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = eq_b\ b_1\ b_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

- ➜ MPC-style

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Equal}\langle\!\langle * \rangle\!\rangle\ t & = t \to t \to \mathit{Bool} \\
\textbf{type}\ \mathit{Equal}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u. \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Equal}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{equal}\langle t :: \kappa \rangle & :: \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = eq_a\ a_1\ a_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = eq_b\ b_1\ b_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

- ➔ MPC-style
- ✘ implicit recursion

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Equal}\langle\!\langle * \rangle\!\rangle\ t & = t \to t \to \mathit{Bool} \\
\textbf{type}\ \mathit{Equal}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u. \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Equal}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{equal}\langle t :: \kappa \rangle & :: \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = eq_a\ a_1\ a_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = eq_b\ b_1\ b_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

- ➔ MPC-style
- ✕ implicit recursion
- ✕ complicated type signature

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Equal}\langle\!\langle * \rangle\!\rangle\ t & = t \to t \to \mathit{Bool} \\
\textbf{type}\ \mathit{Equal}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u. \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Equal}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{equal}\langle t :: \kappa \rangle & :: \mathit{Equal}\langle\!\langle \kappa \rangle\!\rangle\ t \\
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = eq_a\ a_1\ a_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = eq_b\ b_1\ b_2 \\
\mathit{equal}\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

- ➔ MPC-style
- ✗ implicit recursion
- ✗ complicated type signature
- ✓ works for all types of all kinds!

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{aligned}
&\textbf{type } Equal\langle\!\langle * \rangle\!\rangle\ t &&= t \to t \to Bool \\
&\textbf{type } Equal\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t &&= \forall u. Equal\langle\!\langle \kappa \rangle\!\rangle\ u \to Equal\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
&equal\langle t :: \kappa \rangle &&:: Equal\langle\!\langle \kappa \rangle\!\rangle\ t \\
&equal\langle Unit \rangle\ Unit\ Unit &&= True \\
&equal\langle :+: \rangle\ eq_a\ eq_b\ (Inl\ a_1)\ (Inl\ a_2) &&= eq_a\ a_1\ a_2 \\
&equal\langle :+: \rangle\ eq_a\ eq_b\ (Inr\ b_1)\ (Inr\ b_2) &&= eq_b\ b_1\ b_2 \\
&equal\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ &&= False \\
&equal\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) &&= eq_a\ a_1\ a_2 \land eq_b\ b_1\ b_2
\end{aligned}
$$

- ➔ MPC-style
- ✗ implicit recursion
- ✗ complicated type signature
- ✓ works for all types of all kinds!
- ✓ implemented in Generic Haskell

Two styles

Ralf Hinze introduced two styles of generic functions:

$$
\begin{array}{ll}
equal\langle t :: * \rangle & :: t \to t \to Bool \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle a :+: b \rangle\ (Inl\ a_1)\ (Inl\ a_2) & = equal\langle a \rangle\ a_1\ a_2 \\
equal\langle a :+: b \rangle\ (Inr\ b_1)\ (Inr\ b_2) & = equal\langle b \rangle\ b_1\ b_2 \\
equal\langle a :+: b \rangle\ \_\ \_ & = False \\
equal\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = equal\langle a \rangle\ a_1\ a_2 \wedge equal\langle b \rangle\ b_1\ b_2
\end{array}
$$

$$
\begin{array}{ll}
\mathbf{type}\ Equal\langle\langle * \rangle\rangle\ t & = t \to t \to Bool \\
\mathbf{type}\ Equal\langle\langle \kappa \to \kappa' \rangle\rangle\ t & = \forall u . Equal\langle\langle \kappa \rangle\rangle\ u \to Equal\langle\langle \kappa' \rangle\rangle\ (t\ u) \\
equal\langle t :: \kappa \rangle & :: Equal\langle\langle \kappa \rangle\rangle\ t \\
equal\langle Unit \rangle\ Unit\ Unit & = True \\
equal\langle :+: \rangle\ eq_a\ eq_b\ (Inl\ a_1)\ (Inl\ a_2) & = eq_a\ a_1\ a_2 \\
equal\langle :+: \rangle\ eq_a\ eq_b\ (Inr\ b_1)\ (Inr\ b_2) & = eq_b\ b_1\ b_2 \\
equal\langle :+: \rangle\ eq_a\ eq_b\ \_\ \_ & = False \\
equal\langle :\times: \rangle\ eq_a\ eq_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = eq_a\ a_1\ a_2 \wedge eq_b\ b_1\ b_2
\end{array}
$$

Comparison

Explicit recursion (POPL) is more intuitive …

Comparison

Explicit recursion (POPL) is more intuitive …

… but Generic Haskell implements only implicit recursion (MPC).

Comparison

Explicit recursion (POPL) is more intuitive …
…but Generic Haskell implements only implicit recursion (MPC).

This talk

Step by step towards explicit recursion in Generic Haskell – without losing the advantages of the current implementation.

Why implicit recursion?

Why implicit recursion?

Generic functions are specialised

Why implicit recursion?

Generic functions are specialised

- If called with a specific type argument, a specialised instance of the generic function is generated.

Why implicit recursion?

Generic functions are specialised

- If called with a specific type argument, a specialised instance of the generic function is generated.
- In implicitly recursive style, generic functions *poly* fulfill the property

  $$poly\langle f\ a\rangle = poly\langle f\rangle\ (poly\langle a\rangle)$$

  That makes specialisation compositional and guaranteed to terminate.

Why implicit recursion?

Generic functions are specialised

- If called with a specific type argument, a specialised instance of the generic function is generated.
- In implicitly recursive style, generic functions *poly* fulfill the property

  $$poly\langle f\ a\rangle = poly\langle f\rangle\ (poly\langle a\rangle)$$

  That makes specialisation compositional and guaranteed to terminate.
- Explictly recursive functions as introduced by Ralf Hinze have a few significant limitations (next to the restriction to type arguments of one fixed kind) and are hard to implement directly.

A closer look

- In implicit recursive style, a generic function consists of multiple *cases* for different type arguments.

A closer look

- In implicit recursive style, a generic function consists of multiple *cases* for different type arguments.
- Each type argument is the name of a type

A closer look

- In implicit recursive style, a generic function consists of multiple *cases* for different type arguments.
- Each type argument is the name of a type
  - defined by a **data** statement
  - defined by a **newtype** statement
  - defined by a **type** statement
  - builtin – such as $(\rightarrow)$, $[\,]$, or $(,)$

A closer look

- In implicit recursive style, a generic function consists of multiple *cases* for different type arguments.
- Each type argument is the name of a type
  - defined by a **data** statement
  - defined by a **newtype** statement
  - defined by a **type** statement
  - builtin – such as $(\to)$, $[\,]$, or $(,)$
- The *kind* of the type argument determines the *type* of the case.

A closer look

- In implicit recursive style, a generic function consists of multiple *cases* for different type arguments.
- Each type argument is the name of a type
  - defined by a **data** statement
  - defined by a **newtype** statement
  - defined by a **type** statement
  - builtin – such as $(\rightarrow)$, $[\,]$, or $(,)$
- The *kind* of the type argument determines the *type* of the case.
- Yes, there are also special cases for constructors and labels and even for specific constructors, but we'll ignore them for now.

Reintroducing explicit recursion

Just a syntactic variation …

$$
\begin{aligned}
&\mathit{equal}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ \mathit{Unit} && = \mathit{True}\\
&\mathit{equal}\langle :+: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) && = \mathit{eq}_a\ a_1\ a_2\\
&\mathit{equal}\langle :+: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) && = \mathit{eq}_b\ b_1\ b_2\\
&\mathit{equal}\langle :+: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ \_\ \_ && = \mathit{False}\\
&\mathit{equal}\langle :\times: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) && = \mathit{eq}_a\ a_1\ a_2 \wedge \mathit{eq}_b\ b_1\ b_2
\end{aligned}
$$

- → This is the original (implicit) definition of *equal*.

Reintroducing explicit recursion

Just a syntactic variation ...

$$
\begin{array}{ll}
\mathit{equal}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle :+: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{eq}_a\ a_1\ a_2 \\
\mathit{equal}\langle :+: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{eq}_b\ b_1\ b_2 \\
\mathit{equal}\langle :+: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle :\times: \rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = \mathit{eq}_a\ a_1\ a_2 \wedge \mathit{eq}_b\ b_1\ b_2
\end{array}
$$

➜ We add type variables to the type arguments in the cases.

Reintroducing explicit recursion

Just a syntactic variation …

$$
\begin{array}{ll}
\mathit{equal}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle a :+: b\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{eq}_a\ a_1\ a_2 \\
\mathit{equal}\langle a :+: b\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{eq}_b\ b_1\ b_2 \\
\mathit{equal}\langle a :+: b\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle a :\times: b\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = \mathit{eq}_a\ a_1\ a_2 \wedge \mathit{eq}_b\ b_1\ b_2
\end{array}
$$

- We add type variables to the type arguments in the cases.
- We ignore the additional arguments (eq_a and eq_b), and instead refer to them making use of the newly introduced type variables.

Reintroducing explicit recursion

Just a syntactic variation ...

$$
\begin{aligned}
&equal\langle Unit\rangle\ Unit\ Unit && = True \\
&equal\langle a :+: b\rangle\ (Inl\ a_1)\ (Inl\ a_2) && = equal\langle a\rangle\ a_1\ a_2 \\
&equal\langle a :+: b\rangle\ (Inr\ b_1)\ (Inr\ b_2) && = equal\langle b\rangle\ b_1\ b_2 \\
&equal\langle a :+: b\rangle\ \_\ \_ && = False \\
&equal\langle a :\times: b\rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) && = equal\langle a\rangle\ a_1\ a_2 \land equal\langle b\rangle\ b_1\ b_2
\end{aligned}
$$

- We add type variables to the type arguments in the cases.
- We ignore the additional arguments (eq_a and eq_b), and instead refer to them making use of the newly introduced type variables.
- Note that there is no difference at all for the *Unit* case (kind $*$ type cases).

Another example using type-indexed types

$$
\begin{aligned}
&\textbf{data}\ \mathit{Pair}\ a\ b && = \mathit{Null} \mid \mathit{Pair}\ a\ b\\
&\textbf{type}\ \mathit{FMap}\langle \mathit{Unit}\rangle\ v && = \mathit{FMapUnit}\ (\mathit{Maybe}\ v)\\
&\textbf{type}\ \mathit{FMap}\langle a :+: b\rangle\ v && = \mathit{FMapSum}\ (\mathit{Pair}\ (\mathit{FMap}\langle a\rangle\ v)\ (\mathit{FMap}\langle b\rangle\ v))\\
&\textbf{type}\ \mathit{FMap}\langle a :\times: b\rangle\ v && = \mathit{FMapProd}\ (\mathit{FMap}\langle a\rangle\ (\mathit{FMap}\langle b\rangle\ v))\\
&\mathit{empty}\langle \mathit{Unit}\rangle && = \mathit{FMapUnit}\ \mathit{Nothing}\\
&\mathit{empty}\langle a :+: b\rangle && = \mathit{FMapSum}\ \mathit{Null}\\
&\mathit{empty}\langle a :\times: b\rangle && = \mathit{FMapProd}\ \mathit{empty}\langle a\rangle
\end{aligned}
$$

- Type-indexed finite maps (so-called *tries*) store sum types in a pair of maps and product types in a nested map.
- The function *empty* constructs a finite map with no elements.
- The translation works exactly the same way as on the previous slide, but we present it in the other direction.

Another example using type-indexed types

$$
\begin{aligned}
&\textbf{data}\ \mathit{Pair}\ a\ b && = \mathit{Null} \mid \mathit{Pair}\ a\ b \\
&\textbf{type}\ \mathit{FMap}\langle \mathit{Unit}\rangle\ v && = \mathit{FMapUnit}\ (\mathit{Maybe}\ v) \\
&\textbf{type}\ \mathit{FMap}\langle a:+:b\rangle\ \mathit{fm}_a\ \mathit{fm}_b\ v && = \mathit{FMapSum}\ (\mathit{Pair}\ (\mathit{fm}_a\ v)\ (\mathit{fm}_b\ v)) \\
&\textbf{type}\ \mathit{FMap}\langle a:\times:b\rangle\ \mathit{fm}_a\ \mathit{fm}_b\ v && = \mathit{FMapProd}\ (\mathit{fm}_a\ (\mathit{fm}_b\ v)) \\
&\mathit{empty}\langle \mathit{Unit}\rangle && = \mathit{FMapUnit}\ \mathit{Nothing} \\
&\mathit{empty}\langle a:+:b\rangle\ \mathit{empty}_a\ \mathit{empty}_b && = \mathit{FMapSum}\ \mathit{Null} \\
&\mathit{empty}\langle a:\times:b\rangle\ \mathit{empty}_a\ \mathit{empty}_b && = \mathit{FMapProd}\ \mathit{empty}_a
\end{aligned}
$$

- Type-indexed finite maps (so-called *tries*) store sum types in a pair of maps and product types in a nested map.
- The function *empty* constructs a finite map with no elements.
- The translation works exactly the same way as on the previous slide, but we present it in the other direction.

Another example using type-indexed types

$$
\begin{array}{ll}
\textbf{data}\ \mathit{Pair}\ a\ b & = \mathit{Null} \mid \mathit{Pair}\ a\ b \\
\textbf{type}\ \mathit{FMap}\langle \mathit{Unit}\rangle\ v & = \mathit{FMapUnit}\ (\mathit{Maybe}\ v) \\
\textbf{type}\ \mathit{FMap}\langle{:}{+}{:}\rangle\ \mathit{fm}_a\ \mathit{fm}_b\ v & = \mathit{FMapSum}\ (\mathit{Pair}\ (\mathit{fm}_a\ v)\ (\mathit{fm}_b\ v)) \\
\textbf{type}\ \mathit{FMap}\langle{:}{\times}{:}\rangle\ \mathit{fm}_a\ \mathit{fm}_b\ v & = \mathit{FMapProd}\ (\mathit{fm}_a\ (\mathit{fm}_b\ v)) \\
\mathit{empty}\langle \mathit{Unit}\rangle & = \mathit{FMapUnit}\ \mathit{Nothing} \\
\mathit{empty}\langle{:}{+}{:}\rangle\ \mathit{empty}_a\ \mathit{empty}_b & = \mathit{FMapSum}\ \mathit{Null} \\
\mathit{empty}\langle{:}{\times}{:}\rangle\ \mathit{empty}_a\ \mathit{empty}_b & = \mathit{FMapProd}\ \mathit{empty}_a
\end{array}
$$

- Type-indexed finite maps (so-called *tries*) store sum types in a pair of maps and product types in a nested map.
- The function *empty* constructs a finite map with no elements.
- The translation works exactly the same way as on the previous slide, but we present it in the other direction.

Increased expressive power

$$
\begin{aligned}
&single\langle Unit\rangle\ (Unit, v) && = FMapUnit\ (Just\ v)\\
&single\langle a :+: b\rangle\ (Inl\ a, v) && = FMapSum\ (Pair\ (single\langle a\rangle\ (a, v))\ empty\langle b\rangle)\\
&single\langle a :+: b\rangle\ (Inr\ b, v) && = FMapSum\ (Pair\ empty\langle a\rangle\ (single\langle b\rangle\ (b, v)))\\
&single\langle a :\times: b\rangle\ (a :\times: b, v) && = FMapProd\ (single\langle a\rangle\ (a, single\langle b\rangle\ (b, v)))
\end{aligned}
$$

- Yet another example. The function *single* takes a key-value pair and constructs a map which contains just that single association.
- The difference is that the function refers to both *single* and *empty* on the right hand side.

Increased expressive power

$$
\begin{aligned}
&single\langle Unit\rangle\ (Unit, v) && = FMapUnit\ (Just\ v)\\
&single\langle a :+: b\rangle\ si_a\ si_b\ (Inl\ a, v) && = FMapSum\ (Pair\ (si_a\ (a, v))\ em_b)\\
&single\langle a :+: b\rangle\ si_a\ si_b\ (Inr\ b, v) && = FMapSum\ (Pair\ em_a\ (si_b\ (b, v)))\\
&single\langle a :\times: b\rangle\ si_a\ si_b\ (a :\times: b, v) && = FMapProd\ (si_a\ (a, si_b\ (b, v)))
\end{aligned}
$$

- Yet another example. The function *single* takes a key-value pair and constructs a map which contains just that single association.
- The difference is that the function refers to both *single* and *empty* on the right hand side.

Increased expressive power

$$
\begin{aligned}
&single\langle Unit\rangle\ (Unit, v) &&= FMapUnit\ (Just\ v)\\
&single\langle :+: \rangle\ si_a\ si_b\ (Inl\ a, v) &&= FMapSum\ (Pair\ (si_a\ (a, v))\ em_b)\\
&single\langle :+: \rangle\ si_a\ si_b\ (Inr\ b, v) &&= FMapSum\ (Pair\ em_a\ (si_b\ (b, v)))\\
&single\langle :\times: \rangle\ si_a\ si_b\ (a :\times: b, v) &&= FMapProd\ (si_a\ (a, si_b\ (b, v)))
\end{aligned}
$$

- Yet another example. The function *single* takes a key-value pair and constructs a map which contains just that single association.
- The difference is that the function refers to both *single* and *empty* on the right hand side.
- After the reverse-translation, it becomes clear that there is no way to refer to the *empty* function for a child type.

Increased expressive power

$$
\begin{array}{ll}
single\langle Unit\rangle\ (Unit, v) & = FMapUnit\ (Just\ v) \\
single\langle :+: \rangle\ si_a\ si_b\ (Inl\ a, v) & = FMapSum\ (Pair\ (si_a\ (a, v))\ em_b) \\
single\langle :+: \rangle\ si_a\ si_b\ (Inr\ b, v) & = FMapSum\ (Pair\ em_a\ (si_b\ (b, v))) \\
single\langle :\times: \rangle\ si_a\ si_b\ (a :\times: b, v) & = FMapProd\ (si_a\ (a, si_b\ (b, v)))
\end{array}
$$

- Yet another example. The function *single* takes a key-value pair and constructs a map which contains just that single association.
- The difference is that the function refers to both *single* and *empty* on the right hand side.
- After the reverse-translation, it becomes clear that there is no way to refer to the *empty* function for a child type.
- Thus, the explicitly recursive syntax seems to give us more power than the implicit one.

Increased expressive power

$$
\begin{aligned}
&single\langle Unit\rangle\ (Unit, v) &&= FMapUnit\ (Just\ v)\\
&single\langle{:+:}\rangle\ si_a\ si_b\ (Inl\ a, v) &&= FMapSum\ (Pair\ (si_a\ (a, v))\ em_b)\\
&single\langle{:+:}\rangle\ si_a\ si_b\ (Inr\ b, v) &&= FMapSum\ (Pair\ em_a\ (si_b\ (b, v)))\\
&single\langle{:\times:}\rangle\ si_a\ si_b\ (a{:\times:}b, v) &&= FMapProd\ (si_a\ (a, si_b\ (b, v)))
\end{aligned}
$$

- Yet another example. The function *single* takes a key-value pair and constructs a map which contains just that single association.
- The difference is that the function refers to both *single* and *empty* on the right hand side.
- After the reverse-translation, it becomes clear that there is no way to refer to the *empty* function for a child type.
- Thus, the explicitly recursive syntax seems to give us more power than the implicit one.
- Let us investigate how we can express *single* in implicitly recursive syntax.

Translation A: Tupling

Since we need both *single* and *empty* in the definition of *single*, we could define both functions at the same time, as a pair.

$$singlee\langle Unit\rangle = \lambda(Unit, v) \rightarrow FMapUnit\ (Just\ v)$$

$$\begin{aligned} singlee\langle{:}{+}{:}\rangle\ si_a\ si_b \\ = \lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\ (Inl\ a, v) \rightarrow FMapSum\ (Pair\ (si_a\ (a, v))\ em_b) \\ (Inr\ b, v) \rightarrow FMapSum\ (Pair\ em_a\ (si_b\ (b, v))) \end{aligned}$$

$$\begin{aligned} singlee\langle{:}{\times}{:}\rangle\ si_a\ si_b \\ = \lambda(a{:}{\times}{:}b, v) \rightarrow FMapProd\ (si_a\ (a, si_b\ (b, v))) \end{aligned}$$

Translation A: Tupling

Since we need both *single* and *empty* in the definition of *single*, we could define both functions at the same time, as a pair.

$$
\begin{aligned}
&\mathit{singlee}\langle \mathit{Unit}\rangle = (\lambda(\mathit{Unit}, v) \to \mathit{FMapUnit}\ (\mathit{Just}\ v) \\
&\qquad\qquad\qquad\ ,\mathit{empty}\langle \mathit{Unit}\rangle) \\
&\mathit{singlee}\langle{:+:}\rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad = (\lambda x \to \mathbf{case}\ x\ \mathbf{of} \\
&\qquad\qquad\qquad\qquad (\mathit{Inl}\ a, v) \to \mathit{FMapSum}\ (\mathit{Pair}\ (si_a\ (a, v))\ em_b) \\
&\qquad\qquad\qquad\qquad (\mathit{Inr}\ b, v) \to \mathit{FMapSum}\ (\mathit{Pair}\ em_a\ (si_b\ (b, v))) \\
&\qquad\qquad\quad ,\mathit{empty}\langle{:+:}\rangle\ em_a\ em_b) \\
&\mathit{singlee}\langle{:\times:}\rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad = (\lambda(a :\times: b, v) \to \mathit{FMapProd}\ (si_a\ (a, si_b\ (b, v))) \\
&\qquad\qquad\quad ,\mathit{empty}\langle{:\times:}\rangle\ em_a\ em_b) \\
&\mathit{single}\langle t :: *\rangle \ \ = \mathit{fst}\ (\mathit{singlee}\langle t :: *\rangle)
\end{aligned}
$$

Translation A: Tupling

Since we need both *single* and *empty* in the definition of *single*, we could define both functions at the same time, as a pair.

$$
\begin{aligned}
&\mathit{singlee}\langle \mathit{Unit}\rangle = (\lambda(\mathit{Unit}, v) \rightarrow \mathit{FMapUnit}\ (\mathit{Just}\ v) \\
&\qquad\qquad\qquad\quad ,\mathit{empty}\langle \mathit{Unit}\rangle) \\
&\mathit{singlee}\langle :+: \rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad\qquad = (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Inl}\ a, v) \rightarrow \mathit{FMapSum}\ (\mathit{Pair}\ (si_a\ (a, v))\ em_b) \\
&\qquad\qquad\qquad\qquad\qquad (\mathit{Inr}\ b, v) \rightarrow \mathit{FMapSum}\ (\mathit{Pair}\ em_a\ (si_b\ (b, v))) \\
&\qquad\qquad\qquad\quad ,\mathit{empty}\langle :+: \rangle\ em_a\ em_b) \\
&\mathit{singlee}\langle :\times: \rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad\qquad = (\lambda(a :\times: b, v) \rightarrow \mathit{FMapProd}\ (si_a\ (a, si_b\ (b, v))) \\
&\qquad\qquad\qquad\quad ,\mathit{empty}\langle :\times: \rangle\ em_a\ em_b) \\
&\mathit{single}\langle t :: * \rangle \quad = \mathit{fst}\ (\mathit{singlee}\langle t :: * \rangle)
\end{aligned}
$$

✓ Tupling works without modification of the compiler.

Translation A: Tupling

Since we need both *single* and *empty* in the definition of *single*, we could define both functions at the same time, as a pair.

$$
\begin{aligned}
&singlee\langle Unit\rangle = (\lambda(Unit, v) \rightarrow FMapUnit\ (Just\ v) \\
&\qquad\qquad\qquad\quad , empty\langle Unit\rangle) \\
&singlee\langle{:+:}\rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad = (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad\qquad\qquad (Inl\ a, v) \rightarrow FMapSum\ (Pair\ (si_a\ (a, v))\ em_b) \\
&\qquad\qquad\qquad\qquad (Inr\ b, v) \rightarrow FMapSum\ (Pair\ em_a\ (si_b\ (b, v))) \\
&\qquad\qquad\quad , empty\langle{:+:}\rangle\ em_a\ em_b) \\
&singlee\langle{:\times:}\rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad = (\lambda(a{:\times:}b, v) \rightarrow FMapProd\ (si_a\ (a, si_b\ (b, v))) \\
&\qquad\qquad\quad , empty\langle{:\times:}\rangle\ em_a\ em_b) \\
&single\langle t :: *\rangle \;\; = fst\ (singlee\langle t :: *\rangle)
\end{aligned}
$$

- ✓ Tupling works without modification of the compiler.
- ✗ It is extremely verbose and looks complicated.

Translation A: Tupling

Since we need both *single* and *empty* in the definition of *single*, we could define both functions at the same time, as a pair.

$$
\begin{aligned}
&\mathit{singlee}\langle \mathit{Unit}\rangle = (\lambda(\mathit{Unit}, v) \rightarrow \mathit{FMapUnit}\ (\mathit{Just}\ v) \\
&\qquad\qquad\qquad , \mathit{empty}\langle \mathit{Unit}\rangle) \\
&\mathit{singlee}\langle :+: \rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad = (\lambda x \rightarrow \textbf{case}\ x\ \textbf{of} \\
&\qquad\qquad\qquad\qquad (\mathit{Inl}\ a, v) \rightarrow \mathit{FMapSum}\ (\mathit{Pair}\ (si_a\ (a, v))\ em_b) \\
&\qquad\qquad\qquad\qquad (\mathit{Inr}\ b, v) \rightarrow \mathit{FMapSum}\ (\mathit{Pair}\ em_a\ (si_b\ (b, v))) \\
&\qquad\qquad\quad , \mathit{empty}\langle :+: \rangle\ em_a\ em_b) \\
&\mathit{singlee}\langle :\times: \rangle\ (si_a, em_a)\ (si_b, em_b) \\
&\qquad\qquad = (\lambda(a :\times: b, v) \rightarrow \mathit{FMapProd}\ (si_a\ (a, si_b\ (b, v))) \\
&\qquad\qquad\quad , \mathit{empty}\langle :\times: \rangle\ em_a\ em_b) \\
&\mathit{single}\langle t :: * \rangle \quad = \mathit{fst}\ (\mathit{singlee}\langle t :: * \rangle)
\end{aligned}
$$

- ✓ Tupling works without modification of the compiler.
- ✗ It is extremely verbose and looks complicated.
- ✗ The approach is not possible on the type-level.

Translation B: Dependencies

Since recently, the Generic Haskell compiler supports *dependencies* between generic functions (and type-indexed types):

$$
\begin{aligned}
&single\langle Unit\rangle\ (Unit, v) && = FMapUnit\ (Just\ v)\\
&single\langle{:+:}\rangle\ si_a\quad si_b\quad (Inl\ a, v) && = FMapSum\ (Pair\ (si_a\ (a, v))\ em_b)\\
&single\langle{:+:}\rangle\ si_a\quad si_b\quad (Inr\ b, v) && = FMapSum\ (Pair\ em_a\ (si_b\ (b, v)))\\
&single\langle{:\times:}\rangle\ si_a\quad si_b\quad (a :\times: b, v) && = FMapProd\ (si_a\ (a, si_b\ (b, v)))
\end{aligned}
$$

Translation B: Dependencies

Since recently, the Generic Haskell compiler supports *dependencies* between generic functions (and type-indexed types):

$$\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty}$$

$$\begin{aligned}
&\mathit{single}\langle \mathit{Unit}\rangle\ (\mathit{Unit}, v) &&= \mathit{FMapUnit}\ (\mathit{Just}\ v)\\
&\mathit{single}\langle :+: \rangle\ \mathit{si}_a\ \mathit{em}_a\ \mathit{si}_b\ \mathit{em}_b\ (\mathit{Inl}\ a, v) &&= \mathit{FMapSum}\ (\mathit{Pair}\ (\mathit{si}_a\ (a, v))\ \mathit{em}_b)\\
&\mathit{single}\langle :+: \rangle\ \mathit{si}_a\ \mathit{em}_a\ \mathit{si}_b\ \mathit{em}_b\ (\mathit{Inr}\ b, v) &&= \mathit{FMapSum}\ (\mathit{Pair}\ \mathit{em}_a\ (\mathit{si}_b\ (b, v)))\\
&\mathit{single}\langle :\times: \rangle\ \mathit{si}_a\ \mathit{em}_a\ \mathit{si}_b\ \mathit{em}_b\ (a :\times: b, v) &&= \mathit{FMapProd}\ (\mathit{si}_a\ (a, \mathit{si}_b\ (b, v)))
\end{aligned}$$

- The **dependency** line specifies that *single*, in the :+: and :×: cases, gets extra arguments for both *single* and *empty* (in that order).

Translation B: Dependencies

Since recently, the Generic Haskell compiler supports *dependencies* between generic functions (and type-indexed types):

$$\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty}$$

$$\begin{aligned}
&\mathit{single}\langle \mathit{Unit}\rangle\ (\mathit{Unit}, v) &&= \mathit{FMapUnit}\ (\mathit{Just}\ v)\\
&\mathit{single}\langle {:}{+}{:}\rangle\ \mathit{si}_a\ \mathit{em}_a\ \mathit{si}_b\ \mathit{em}_b\ (\mathit{Inl}\ a, v) &&= \mathit{FMapSum}\ (\mathit{Pair}\ (\mathit{si}_a\ (a, v))\ \mathit{em}_b)\\
&\mathit{single}\langle {:}{+}{:}\rangle\ \mathit{si}_a\ \mathit{em}_a\ \mathit{si}_b\ \mathit{em}_b\ (\mathit{Inr}\ b, v) &&= \mathit{FMapSum}\ (\mathit{Pair}\ \mathit{em}_a\ (\mathit{si}_b\ (b, v)))\\
&\mathit{single}\langle {:}{\times}{:}\rangle\ \mathit{si}_a\ \mathit{em}_a\ \mathit{si}_b\ \mathit{em}_b\ (a\ {:}{\times}{:}\ b, v) &&= \mathit{FMapProd}\ (\mathit{si}_a\ (a, \mathit{si}_b\ (b, v)))
\end{aligned}$$

- The **dependency** line specifies that *single*, in the :+: and :×: cases, gets extra arguments for both *single* and *empty* (in that order).
- A call such as $\mathit{single}\langle \mathit{List}\ \mathit{Int}\rangle$ will no longer be translated into $\mathit{single}\langle \mathit{List}\rangle\ (\mathit{single}\langle \mathit{Int}\rangle)$, but into $\mathit{single}\langle \mathit{List}\rangle\ (\mathit{single}\langle \mathit{Int}\rangle)\ (\mathit{empty}\langle \mathit{Int}\rangle)$, reflecting the dependency.

Translation B: Dependencies

Since recently, the Generic Haskell compiler supports *dependencies* between generic functions (and type-indexed types):

$$
\begin{array}{ll}
\textbf{dependency}\ single \leftarrow single\ empty & \\
single\langle Unit\rangle\ (Unit, v) & = FMapUnit\ (Just\ v) \\
single\langle :+: \rangle\ si_a\ em_a\ si_b\ em_b\ (Inl\ a, v) & = FMapSum\ (Pair\ (si_a\ (a, v))\ em_b) \\
single\langle :+: \rangle\ si_a\ em_a\ si_b\ em_b\ (Inr\ b, v) & = FMapSum\ (Pair\ em_a\ (si_b\ (b, v))) \\
single\langle :\times: \rangle\ si_a\ em_a\ si_b\ em_b\ (a :\times: b, v) & = FMapProd\ (si_a\ (a, si_b\ (b, v)))
\end{array}
$$

- The **dependency** line specifies that *single*, in the $:+:$ and $:\times:$ cases, gets extra arguments for both *single* and *empty* (in that order).
- A call such as $single\langle List\ Int\rangle$ will no longer be translated into $single\langle List\rangle\ (single\langle Int\rangle)$, but into $single\langle List\rangle\ (single\langle Int\rangle)\ (empty\langle Int\rangle)$, reflecting the dependency.
- A **dependency** line could easily be inferred automatically if explicit recursion is used.

Where are we now?

- With explicit recursion, a generic function still consists of multiple cases for different type arguments.
- The type arguments are no longer just names of known types or type constructors (such as $\langle :+: \rangle$ or $\langle [\,] \rangle$).
- They are names of types saturated with type variables (such as $\langle a :+: b \rangle$ or $\langle [a] \rangle$.
- One can think about the type variables as implicitly abstracted (i. e. $\langle \Lambda a\ b.a :+: b \rangle$ or $\Lambda a.[a]$), so the type arguments still have different kinds.
- We can refer to multiple generic functions on the right hand side, not only recursively to the one we are defining.
- The compiler will translate the function into an implicitly recursive function with dependencies.

Where are we now?

- With explicit recursion, a generic function still consists of multiple cases for different type arguments.
- The type arguments are no longer just names of known types or type constructors (such as $\langle :+: \rangle$ or $\langle [\,] \rangle$).
- They are names of types saturated with type variables (such as $\langle a :+: b \rangle$ or $\langle [a] \rangle$.
- One can think about the type variables as implicitly abstracted (i. e. $\langle \Lambda a\ b. a :+: b \rangle$ or $\Lambda a.[a]$), so the type arguments still have different kinds.
- We can refer to multiple generic functions on the right hand side, not only recursively to the one we are defining.
- The compiler will translate the function into an implicitly recursive function with dependencies.
- But what about the types of the generic functions?

What about the types?

What about the types?

- We have ignored the types of the generic functions so far.

What about the types?

- We have ignored the types of the generic functions so far.
- But **dependency** lines affect the types of generic functions.

What about the types?

- We have ignored the types of the generic functions so far.
- But **dependency** lines affect the types of generic functions.

Let's look at our example *single*:

$$\textbf{type}\ Single\langle\!\langle * \rangle\!\rangle\ t = \forall v.(t, v) \rightarrow FMap\langle t \rangle\ v$$
$$\textbf{type}\ Single\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\ t = \forall u.Single\langle\!\langle \kappa \rangle\!\rangle\ u \rightarrow Empty\langle\!\langle \kappa \rangle\!\rangle\ u \rightarrow Single\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u)$$

What about the types?

- We have ignored the types of the generic functions so far.
- But **dependency** lines affect the types of generic functions.

Let's look at our example *single*:

$$\textbf{type } \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t = \forall v.(t, v) \to \mathit{FMap}\langle t \rangle\ v$$
$$\textbf{type } \mathit{Single}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \to \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u)$$

What about the types?

- We have ignored the types of the generic functions so far.
- But **dependency** lines affect the types of generic functions.

Let's look at our example *single*:

$$\textbf{type}\ Single\langle\!\langle * \rangle\!\rangle\ t = \forall v.(t, v) \to FMap\langle t \rangle\ v$$
$$\textbf{type}\ Single\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t = \forall u.Single\langle\!\langle \kappa \rangle\!\rangle\ u \to Empty\langle\!\langle \kappa \rangle\!\rangle\ u \to Single\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u)$$

- The extra arguments are expected in the order that the **dependency** line specifies.

What about the types?

- We have ignored the types of the generic functions so far.
- But **dependency** lines affect the types of generic functions.

Let's look at our example *single*:

$$\textbf{type}\ \mathit{Single}\langle\!\langle *\rangle\!\rangle\ t = \forall v.(t,v) \to \mathit{FMap}\langle t\rangle\ v$$
$$\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \to \kappa'\rangle\!\rangle\ t = \forall u.\mathit{Single}\langle\!\langle \kappa\rangle\!\rangle\ u \to \mathit{Empty}\langle\!\langle \kappa\rangle\!\rangle\ u \to \mathit{Single}\langle\!\langle \kappa'\rangle\!\rangle\ (t\ u)$$

- The extra arguments are expected in the order that the **dependency** line specifies.
- If we let the compiler infer dependencies automatically, we cannot know that order.

What about the types?

- We have ignored the types of the generic functions so far.
- But **dependency** lines affect the types of generic functions.

Let's look at our example *single*:

$$\textbf{type}\ Single\langle\!\langle * \rangle\!\rangle\ t = \forall v.(t, v) \to FMap\langle t \rangle\ v$$
$$\textbf{type}\ Single\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t = \forall u.Single\langle\!\langle \kappa \rangle\!\rangle\ u \to Empty\langle\!\langle \kappa \rangle\!\rangle\ u \to Single\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u)$$

- The extra arguments are expected in the order that the **dependency** line specifies.
- If we let the compiler infer dependencies automatically, we cannot know that order.
- We need (at the user level) a type system that can deal with multiple dependencies without expressing them by function types in a specific order.

A type system with named arguments

$$\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t\rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \rightarrow \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa\rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}$$

- We will transform this type stepwise.

A type system with named arguments

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \rightarrow \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}
$$

➔ We will transform this type stepwise.

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \rightarrow \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}
$$

A type system with named arguments

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \to \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \to \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \to \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}
$$

- We add an extra type argument to the kind indexed type.

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \to \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \to \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \qquad \to \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}
$$

A type system with named arguments

$$
\begin{aligned}
&\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty}\\
&\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t && = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t\rangle\ v\\
&\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\ t && = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u\\
& && \qquad \rightarrow \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u\\
& && \qquad \rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u)\\
&\mathit{single}\langle t :: \kappa\rangle && :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{aligned}
$$

- We replace the dependency by a *dependency constraint*.

$$
\begin{aligned}
&\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty}\\
&\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\langle s\rangle\ t && = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t\rangle\ v\\
&\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\langle \Lambda a.s\rangle\ t && = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\langle a\rangle\ u\\
& && \qquad \rightarrow \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\langle a\rangle\ u\\
& && \qquad \rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\langle s\rangle\ (t\ u)\\
&\mathit{single}\langle t :: \kappa\rangle && :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\langle t\rangle\ t
\end{aligned}
$$

A type system with named arguments

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \quad\quad \rightarrow \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
 & \quad\quad \rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}
$$

- The dependency line is now superfluous.

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle \langle s \rangle\ t & = \forall v.(t, v) \rightarrow \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \rightarrow \kappa' \rangle\!\rangle \langle \Lambda a.s \rangle\ t & = \forall u.(\mathit{single}\langle a \rangle :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
 & \quad\quad , \mathit{empty}\langle a \rangle :: \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
 & \quad\quad) \Rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle \langle s \rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle \langle t \rangle\ t
\end{array}
$$

A type system with named arguments

$$
\begin{array}{ll}
\textbf{dependency}\ \mathit{single} \leftarrow \mathit{single}\ \mathit{empty} & \\
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle\ t & = \forall v.(t, v) \to \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\ t & = \forall u.\mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ u \\
& \qquad \to \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle\ u \\
& \qquad \to \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle\ t
\end{array}
$$

➜ The dependency line is now superfluous.

$$
\begin{array}{ll}
\textbf{type}\ \mathit{Single}\langle\!\langle * \rangle\!\rangle \langle s \rangle\ t & = \forall v.(t, v) \to \mathit{FMap}\langle t \rangle\ v \\
\textbf{type}\ \mathit{Single}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle \langle \Lambda a.s \rangle\ t & = \forall u.(\mathit{single}\langle a \rangle :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
& \qquad , \mathit{empty}\langle a \rangle :: \mathit{Empty}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
& \qquad) \Rightarrow \mathit{Single}\langle\!\langle \kappa' \rangle\!\rangle \langle s \rangle\ (t\ u) \\
\mathit{single}\langle t :: \kappa \rangle & :: \mathit{Single}\langle\!\langle \kappa \rangle\!\rangle \langle t \rangle\ t
\end{array}
$$

Dependency constraints by example

$$
\begin{aligned}
&\textbf{type } \mathit{SShallow}\langle\!\langle * \rangle\!\rangle \langle s \rangle\ t &&= t \to \mathit{String} \\
&\textbf{type } \mathit{SShallow}\langle\!\langle \kappa \to \kappa' \rangle\!\rangle \langle \Lambda a.s \rangle\ t &&= \forall u.(\mathit{showEllipsis}\langle a \rangle :: \mathit{SEllipsis}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
& && \qquad ,\mathit{showRecord}\langle a \rangle :: \mathit{SRecord}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
& && \qquad ,\mathit{showShallow}\langle a \rangle :: \mathit{SShallow}\langle\!\langle \kappa \rangle\!\rangle \langle a \rangle\ u \\
& && \qquad) \Rightarrow \mathit{SShallow}\langle\!\langle \kappa' \rangle\!\rangle \langle s \rangle\ (t\ u) \\
&\mathit{showShallow}\langle t :: \kappa \rangle &&:: \mathit{SShallow}\langle\!\langle \kappa \rangle\!\rangle \langle t \rangle\ t
\end{aligned}
$$

Dependency constraints by example

$$\begin{aligned}
&\textbf{type } SShallow\langle\!\langle * \rangle\!\rangle\langle s\rangle\ t &&= t \to String\\
&\textbf{type } SShallow\langle\!\langle \kappa \to \kappa' \rangle\!\rangle\langle \Lambda a.s\rangle\ t &&= \forall u.(showEllipsis\langle a\rangle :: SEllipsis\langle\!\langle \kappa \rangle\!\rangle\langle a\rangle\ u\\
& && \quad ,showRecord\langle a\rangle :: SRecord\langle\!\langle \kappa \rangle\!\rangle\langle a\rangle\ u\\
& && \quad ,showShallow\langle a\rangle :: SShallow\langle\!\langle \kappa \rangle\!\rangle\langle a\rangle\ u\\
& && \quad) \Rightarrow SShallow\langle\!\langle \kappa' \rangle\!\rangle\langle s\rangle\ (t\ u)\\
&showShallow\langle t :: \kappa\rangle &&:: SShallow\langle\!\langle \kappa \rangle\!\rangle\langle t\rangle\ t
\end{aligned}$$

$$\begin{aligned}
&showShallow\langle (Int, Fix\ [\,])\rangle &&:: (Int, Fix\ [\,]) \to String\\
&showShallow\langle Either\ a\ [(a,b)]\rangle &&:: \forall u\ v.\\
& && \quad (showEllipsis\langle a\rangle :: SEllipsis\langle\!\langle * \rangle\!\rangle\langle a\rangle\ u\\
& && \quad ,showRecord\langle a\rangle :: SRecord\langle\!\langle * \rangle\!\rangle\langle a\rangle\ u\\
& && \quad ,showShallow\langle a\rangle :: SShallow\langle\!\langle * \rangle\!\rangle\langle a\rangle\ u\\
& && \quad ,showEllipsis\langle b\rangle :: SEllipsis\langle\!\langle * \rangle\!\rangle\langle a\rangle\ v\\
& && \quad ,showRecord\langle b\rangle :: SRecord\langle\!\langle * \rangle\!\rangle\langle a\rangle\ v\\
& && \quad ,showShallow\langle b\rangle :: SShallow\langle\!\langle * \rangle\!\rangle\langle a\rangle\ v\\
& && \quad) \Rightarrow SShallow\langle\!\langle * \rangle\!\rangle\langle Either\ a\ [b]\rangle\ (Either\ a\ [b])
\end{aligned}$$

Satisfying dependency constraints

- Dependency constraints can be satisfied in any order in **let** bindings:

$$\begin{aligned}
&\textbf{let}\ \mathit{showRecord}\langle a\rangle\ \mathit{ns} = \mathit{show}\ \mathit{ns}\\
&\quad\ \mathit{showShallow}\langle a\rangle\ \mathit{ns} = \mathit{showShallow}\langle[\mathit{Int}]\rangle\ \mathit{ns}\\
&\quad\ \mathit{showEllipsis}\langle a\rangle\ \mathit{ns} = \texttt{"<sum:\textvisiblespace"} +\!\!+\ \mathit{sum}\ \mathit{ns} +\!\!+ \texttt{">"}\\
&\textbf{in}\ \mathit{showShallow}\langle \mathit{MyRatherComplexTree}\ a\rangle\ \mathit{tree}\\
&\qquad\qquad :: [\mathit{Int}] \to \mathit{String}
\end{aligned}$$

(Skip to Summary) (Skip to Conclusions)

Satisfying dependency constraints

- Dependency constraints can be satisfied in any order in **let** bindings:

$$
\begin{array}{l}
\textbf{let}\ \mathit{showRecord}\langle a\rangle\ \mathit{ns}\ \ = \mathit{show}\ \mathit{ns} \\
\quad \mathit{showShallow}\langle a\rangle\ \mathit{ns} = \mathit{showShallow}\langle [\mathit{Int}]\rangle\ \mathit{ns} \\
\quad \mathit{showEllipsis}\langle a\rangle\ \mathit{ns}\ = \texttt{"<sum: "} +\!\!+\ \mathit{sum}\ \mathit{ns} +\!\!+ \texttt{">"} \\
\textbf{in}\ \mathit{showShallow}\langle \mathit{MyRatherComplexTree}\ a\rangle\ \mathit{tree} \\
\qquad\qquad\qquad\qquad :: [\mathit{Int}] \rightarrow \mathit{String}
\end{array}
$$

- Dependency constraints are propagated and thus need not be satisfied immediately:

$$
\begin{array}{l}
\textbf{let}\ \mathit{showEllipsis}\langle a\rangle\ x = \texttt{"..."} \\
\textbf{in}\ \mathit{compare}\ (\mathit{showShallow}\langle \mathit{TreeA}\ a\rangle\ \mathit{tree\_a})\ (\mathit{showShallow}\langle \mathit{TreeB}\ a\rangle\ \mathit{tree\_b}) \\
\qquad\qquad\qquad :: \forall u.(\mathit{showRecord}\langle a\rangle :: u \rightarrow \mathit{String} \\
\qquad\qquad\qquad\qquad ,\mathit{showShallow}\langle a\rangle :: u \rightarrow \mathit{String} \\
\qquad\qquad\qquad\qquad) \Rightarrow \mathit{Ordering}
\end{array}
$$

- This behaviour is reminiscent of implicit parameters.

(Skip to Summary) (Skip to Conclusions)

Dependency constraints, slightly formalised

We extend the type language by types qualified with dependency constraints.

Types	$t ::= \ldots$
	$\mid \quad (D) \Rightarrow t$
Dependency constraint set	$D ::= \varepsilon$
	$\mid \quad n\langle a\rangle :: t, D$

Dependency constraints can be reordered.
Dependency constraints can be satisfied in a **let** binding:

$$\frac{e :: (n\langle a\rangle :: t, D) \Rightarrow t' \qquad e' :: t}{\textbf{let}\ n\langle a\rangle = e'\ \textbf{in}\ e :: (D) \Rightarrow t'}$$

They are propagated elsewhere:

$$\frac{e :: (D) \Rightarrow t' \to t \qquad e' :: (D') \Rightarrow t}{(e\ e') :: (\textbf{merge}\ (D, D')) \Rightarrow t}$$

Summary

- We can write generic functions using explicit recursion.
- Internally, it is translated into an equivalent implicitly recursive function.
- The types of generic functions remain kind-indexed.
- Being able to recurse on other generic functions by name in the right hand side makes a large class of generic functions much easier to write.
- Having a type system with dependency constraints opens up the possibility to infer the $\langle\!\langle \kappa \to \kappa' \rangle\!\rangle$-line in kind-indexed types. Only the simple (kind $*$) type needs to be written.

(Skip to Conclusions)

Summary

- We can write generic functions using explicit recursion.
- Internally, it is translated into an equivalent implicitly recursive function.
- The types of generic functions remain kind-indexed.
- Being able to recurse on other generic functions by name in the right hand side makes a large class of generic functions much easier to write.
- Having a type system with dependency constraints opens up the possibility to infer the $\langle\!\langle \kappa \to \kappa' \rangle\!\rangle$-line in kind-indexed types. Only the simple (kind $*$) type needs to be written.

Differences to Ralf Hinze's POPL-style

- Generic definitions always work on the same type language.
- There are less restrictions.

(Skip to Conclusions)

Explicit recursion and default cases

Default cases (formerly called emphcopy lines) can be used to extend or modify existing generic function by providing new cases.

$$\begin{aligned}
&\mathit{equal}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ \mathit{Unit} &&= \mathit{True}\\
&\mathit{equal}\langle{:+:}\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) &&= \mathit{eq}_a\ a_1\ a_2\\
&\mathit{equal}\langle{:+:}\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) &&= \mathit{eq}_b\ b_1\ b_2\\
&\mathit{equal}\langle{:+:}\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ _\ _ &&= \mathit{False}\\
&\mathit{equal}\langle{:\times:}\rangle\ \mathit{eq}_a\ \mathit{eq}_b\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) &&= \mathit{eq}_a\ a_1\ a_2 \wedge \mathit{eq}_b\ b_1\ b_2\\
&\mathit{rEqual}\langle \mathit{Range}\rangle\ _\ _ &&= \mathit{True}\\
&\mathit{rEqual}\langle t\rangle &&= \mathit{equal}\langle t\rangle
\end{aligned}$$

➔ With *rEqual*, ranges are ignored for comparision *everywhere*.

(Skip to Conclusions)

Explicit recursion and default cases

Default cases (formerly called emphcopy lines) can be used to extend or modify existing generic function by providing new cases.

$$
\begin{array}{ll}
\mathit{equal}\langle \mathit{Unit} \rangle\ \mathit{Unit}\ \mathit{Unit} & = \mathit{True} \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \\
\mathit{equal}\langle a :+: b \rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) & = \mathit{equal}\langle b \rangle\ b_1\ b_2 \\
\mathit{equal}\langle a :+: b \rangle\ \_\ \_ & = \mathit{False} \\
\mathit{equal}\langle a :\times: b \rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) & = \mathit{equal}\langle a \rangle\ a_1\ a_2 \wedge \mathit{equal}\langle b \rangle\ b_1\ b_2 \\
\mathit{rEqual}\langle \mathit{Range} \rangle\ \_\ \_ & = \mathit{True} \\
\mathit{rEqual}\langle t \rangle & = \mathit{equal}\langle t \rangle
\end{array}
$$

- With *rEqual*, ranges are ignored for comparision *everywhere*.
- With explicit recursion, this is no longer obvious.

(Skip to Conclusions)

Explicit recursion and default cases

Default cases (formerly called emphcopy lines) can be used to extend or modify existing generic function by providing new cases.

$$
\begin{aligned}
&\mathit{equal}\langle \mathit{Unit}\rangle\ \mathit{Unit}\ \mathit{Unit} &&= \mathit{True}\\
&\mathit{equal}\langle a :+: b\rangle\ (\mathit{Inl}\ a_1)\ (\mathit{Inl}\ a_2) &&= \mathbf{this}\langle a\rangle\ a_1\ a_2\\
&\mathit{equal}\langle a :+: b\rangle\ (\mathit{Inr}\ b_1)\ (\mathit{Inr}\ b_2) &&= \mathbf{this}\langle b\rangle\ b_1\ b_2\\
&\mathit{equal}\langle a :+: b\rangle\ \_\ \_ &&= \mathit{False}\\
&\mathit{equal}\langle a :\times: b\rangle\ (a_1 :\times: b_1)\ (a_2 :\times: b_2) &&= \mathbf{this}\langle a\rangle\ a_1\ a_2 \wedge \mathbf{this}\langle b\rangle\ b_1\ b_2\\
&\mathit{rEqual}\langle \mathit{Range}\rangle\ \_\ \_ &&= \mathit{True}\\
&\mathit{rEqual}\langle t\rangle &&= \mathit{equal}\langle t\rangle
\end{aligned}
$$

- With *rEqual*, ranges are ignored for comparision *everywhere*.
- With explicit recursion, this is no longer obvious, but different possibilities exist.

(Skip to Conclusions)

Explicit recursion and generic abstraction

Generic abstractions are Generic Haskell's way of defining one-line generic functions in terms of others, thereby restricting the kind.

$$mapTwice\langle t :: * \to * \rangle\ f = gmap\langle t \rangle\ (f \circ f)$$

Explicit recursion and generic abstraction

Generic abstractions are Generic Haskell's way of defining one-line generic functions in terms of others, thereby restricting the kind.

$$mapTwice\langle t :: * \to * \rangle\; f = gmap\langle t \rangle\; (f \circ f)$$

- Currently, generic abstractions are essentially inlined by the compiler, leading to a non-compositional specialisation for generic abstractions.

Explicit recursion and generic abstraction

Generic abstractions are Generic Haskell's way of defining one-line generic functions in terms of others, thereby restricting the kind.

$mapTwice\langle t :: * \rightarrow * \rangle\ f = gmap\langle t \rangle\ (f \circ f)$

- Currently, generic abstractions are essentially inlined by the compiler, leading to a non-compositional specialisation for generic abstractions.
- With the dependency constraint type system, the function can be seen as depending on the definition of *gmap*, thereby becoming an ordinary polymorphic function in the translation.

Conclusions

- The proposed syntax makes it possible to write generic functions in Generic Haskell in a more intuitive way.
- The new model works fine in conjunction with other features of Generic Haskell, such as default cases or generic abstraction, even simplifying the latter.

Conclusions

- The proposed syntax makes it possible to write generic functions in Generic Haskell in a more intuitive way.
- The new model works fine in conjunction with other features of Generic Haskell, such as default cases or generic abstraction, even simplifying the latter.

Implementation

- The **dependency** feature is implemented, but the rest is not. The type inferencer is still a challenge.

Conclusions

- The proposed syntax makes it possible to write generic functions in Generic Haskell in a more intuitive way.
- The new model works fine in conjunction with other features of Generic Haskell, such as default cases or generic abstraction, even simplifying the latter.

Implementation

- The **dependency** feature is implemented, but the rest is not. The type inferencer is still a challenge.

Future work

- Simplifications: can default cases and generic abstractions be unified?
- Possible extensions such as complex type patterns allow even more expressivity.