Implementing Dependent Types in Haskell

Andres Löh[1]

joint work with Conor McBride[2] and Wouter Swierstra[2]

[1]Universiteit Utrecht

[2]University of Nottingham

September 12, 2007

Motivation

- Present a type checker for dependent types, implemented in Haskell.
- Only a core language as a basis for experimentation:
  - much like F_ω is for Haskell/GHC.
- There are many design choices:
  - Keep it simple ...
  - ... yet powerful enough to demonstrate some of the advantages gained by dependent types.
- For programmers interested in type systems, not type theorists interested in programming.

Why dependent types?

- Lots of type-level programming in Haskell: more static guarantees, but
  - duplication of concepts on different layers
  - more and more type system extensions
  - some of them with restrictions and metatheory that is difficult to understand

Why dependent types?

- Lots of type-level programming in Haskell: more static guarantees, but
  - duplication of concepts on different layers
  - more and more type system extensions
  - some of them with restrictions and metatheory that is difficult to understand
- Dependent types offer:
  - type-level programming becomes term-level programming
  - programs, properties, and proofs within a single formalism
  - a comparatively clean theory on the surface
  - of course, there's another set of problems, but . . .

Simply-typed Lambda Calculus $\lambda_{\rightarrow}$

$$\mathsf{t} ::= \mathsf{a} \mid \mathsf{t} \rightarrow \mathsf{t}'$$

Simply-typed Lambda Calculus $\lambda_{\rightarrow}$

$$t ::= a \mid t \rightarrow t'$$
$$e ::= e :: t \mid x \mid e_1\ e_2 \mid \lambda x \rightarrow e$$

Simply-typed Lambda Calculus $\lambda_{\rightarrow}$

$$
\begin{aligned}
t &::= a \mid t \rightarrow t' \\
e &::= e :: t \mid x \mid e_1\ e_2 \mid \lambda x \rightarrow e \\
\Gamma &::= \varepsilon \mid \Gamma, a :: * \mid \Gamma, x :: t
\end{aligned}
$$

Simply-typed Lambda Calculus $\lambda_{\rightarrow}$

$$
\begin{aligned}
\mathsf{t} &::= \mathsf{a} \mid \mathsf{t} \rightarrow \mathsf{t}' \\
\mathsf{e} &::= \mathsf{e} :: \mathsf{t} \mid \mathsf{x} \mid \mathsf{e}_1\ \mathsf{e}_2 \mid \lambda \mathsf{x} \rightarrow \mathsf{e} \\
\Gamma &::= \varepsilon \mid \Gamma, \mathsf{a} :: * \mid \Gamma, \mathsf{x} :: \mathsf{t}
\end{aligned}
$$

$$
\frac{}{\mathsf{valid}(\varepsilon)} \qquad \frac{\mathsf{valid}(\Gamma)}{\mathsf{valid}(\Gamma, \mathsf{a} :: *)} \qquad \frac{\mathsf{valid}(\Gamma) \quad \Gamma \vdash \mathsf{t} :: *}{\mathsf{valid}(\Gamma, \mathsf{x} :: \mathsf{t})}
$$

Simply-typed Lambda Calculus $\lambda_{\rightarrow}$

$$
\begin{aligned}
t &::= a \mid t \rightarrow t' \\
e &::= e :: t \mid x \mid e_1\ e_2 \mid \lambda x \rightarrow e \\
\Gamma &::= \varepsilon \mid \Gamma, a :: * \mid \Gamma, x :: t
\end{aligned}
$$

$$
\frac{}{\text{valid}(\varepsilon)} \qquad \frac{\text{valid}(\Gamma)}{\text{valid}(\Gamma, a :: *)} \qquad \frac{\text{valid}(\Gamma) \quad \Gamma \vdash t :: *}{\text{valid}(\Gamma, x :: t)}
$$

$$
\frac{\Gamma(a) = *}{\Gamma \vdash a :: *} \qquad \frac{\Gamma \vdash t :: * \quad \Gamma \vdash t' :: *}{\Gamma \vdash t \rightarrow t' :: *}
$$

Simply-typed Lambda Calculus $\lambda_{\rightarrow}$

$$t ::= a \mid t \rightarrow t'$$
$$e ::= e :: t \mid x \mid e_1\ e_2 \mid \lambda x \rightarrow e$$
$$\Gamma ::= \varepsilon \mid \Gamma, a :: * \mid \Gamma, x :: t$$

$$\frac{}{\text{valid}(\varepsilon)} \qquad \frac{\text{valid}(\Gamma)}{\text{valid}(\Gamma, a :: *)} \qquad \frac{\text{valid}(\Gamma) \quad \Gamma \vdash t :: *}{\text{valid}(\Gamma, x :: t)}$$

$$\frac{\Gamma(a) = *}{\Gamma \vdash a :: *} \qquad \frac{\Gamma \vdash t :: * \quad \Gamma \vdash t' :: *}{\Gamma \vdash t \rightarrow t' :: *}$$

$$\frac{\Gamma \vdash t :: * \quad \Gamma \vdash e ::_{\downarrow} t}{\Gamma \vdash (e :: t) ::_{\uparrow} t} \qquad \frac{\Gamma(x) = t}{\Gamma \vdash x ::_{\uparrow} t} \qquad \frac{\Gamma \vdash e_1 ::_{\uparrow} t \rightarrow t' \quad \Gamma \vdash e_2 ::_{\downarrow} t}{\Gamma \vdash e_1\ e_2 ::_{\uparrow} t'}$$

$$\frac{\Gamma \vdash e ::_{\uparrow} t}{\Gamma \vdash e ::_{\downarrow} t} \qquad \frac{\Gamma, x :: t \vdash e ::_{\downarrow} t'}{\Gamma \vdash \lambda x \rightarrow e ::_{\downarrow} t \rightarrow t'}$$

Evaluation in $\lambda_{\rightarrow}$

$$\mathsf{v} ::= \mathsf{n} \mid \lambda \mathsf{x} \rightarrow \mathsf{v}$$

$$\mathsf{n} ::= \mathsf{x} \mid \mathsf{n}\ \mathsf{v}$$

Evaluation in $\lambda_{\rightarrow}$

$$v ::= n \mid \lambda x \rightarrow v$$

$$n ::= x \mid n\ v$$

$$\frac{e \Downarrow v}{e :: t \Downarrow v} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e_1 \Downarrow \lambda x \rightarrow v_1 \qquad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow v_1[x \mapsto v_2]} \qquad \frac{e_1 \Downarrow n_1 \qquad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow n_1\ v_2} \qquad \frac{e \Downarrow v}{\lambda x \rightarrow e \Downarrow \lambda x \rightarrow v}$$

Moving to dependent types: dependent functions

The construct

$$\forall x :: t.t' \text{ (often also written } \Pi\, x :: t.t')$$

generalizes and thereby replaces the function arrow

$$t \to t'$$

Difference: x may occur in t'. If it doesn't, we just write $t \to t'$ as syntactic sugar.

Note: This also generalizes (Haskell's) parametric polymorphism, but we do not enforce parametricity.

Moving to dependent types: everything is a term

We collapse the multi-level structure (terms, types, [kinds]) – everything is a term. The $\forall$ moves to the term-level, in turn lambda abstraction and application become available to (former) types.

- The symbol

  $$::$$

  becomes a relation between two terms.
- Computation arrives in the world of the types.
- Also automatically introduces "kinds".

Example

We can state a large class of properties as types:

$$\forall (a :: *)\ (xs :: \text{List}\ a).\text{reverse}\ (\text{reverse}\ xs) == xs$$

$$\forall (x :: \text{Nat})\ (xs :: \text{List Nat}).\quad \text{Holds}\ (\text{sorted}\ xs) \rightarrow \text{Holds}\ (\text{sorted}\ (\text{insert}\ x\ xs))$$

- An inhabitant of such types is a proof (Curry-Howard).
- Consistency of the type system is an advantage.

Conversion rule

Computation on types also introduces a problem: When are two types equal?

$$\begin{aligned} \mathsf{Vec}\ (2+2)\ \mathsf{Nat} &= \mathsf{Vec}\ 4\ \mathsf{Nat} \\ \mathsf{Vec}\ (x+0)\ \mathsf{Nat} &= \mathsf{Vec}\ x\ \mathsf{Nat} && \text{(assuming } x :: \mathsf{Nat} \text{ in the context)} \\ \mathsf{Vec}\ (x+y)\ \mathsf{Nat} &= \mathsf{Vec}\ (y+x)\ \mathsf{Nat} && \text{(assuming } x, y :: \mathsf{Nat} \text{ in the context)} \end{aligned}$$

Conversion rule:

$$\frac{\Gamma \vdash e :: t' \quad t = t'}{\Gamma \vdash e :: t}$$

In our case: evaluate both terms to normal form, then compare for (alpha-)equality.

- We try to keep the calculus strongly normalizing.

Dependently-typed Lambda Calculus λ_Π

$$e, t ::= e :: t \mid * \mid \forall x :: t.t' \mid x \mid e_1\ e_2 \mid \lambda x \rightarrow e$$

Dependently-typed Lambda Calculus λ_Π

$$
\begin{aligned}
e, t &::= e :: t \mid * \mid \forall x :: t.t' \mid x \mid e_1\ e_2 \mid \lambda x \to e \\
\Gamma &::= \varepsilon \mid \Gamma, x :: t
\end{aligned}
$$

Dependently-typed Lambda Calculus λ_Π

$$
\begin{aligned}
e, t &::= e :: t \mid * \mid \forall x :: t.t' \mid x \mid e_1\ e_2 \mid \lambda x \to e \\
\Gamma &::= \varepsilon \mid \Gamma, x :: t
\end{aligned}
$$

$$
\frac{}{\text{valid}(\varepsilon)} \qquad \frac{\text{valid}(\Gamma) \qquad \Gamma \vdash t ::_{\downarrow} *}{\text{valid}(\Gamma, x :: t)}
$$

Dependently-typed Lambda Calculus λ_Π

$$e, t ::= e :: t \mid * \mid \forall x :: t.t' \mid x \mid e_1\ e_2 \mid \lambda x \rightarrow e$$
$$\Gamma \quad ::= \varepsilon \mid \Gamma, x :: t$$

$$\frac{}{\text{valid}(\varepsilon)} \qquad \frac{\text{valid}(\Gamma) \quad \Gamma \vdash t ::_{\downarrow} *}{\text{valid}(\Gamma, x :: t)}$$

$$\frac{\Gamma(a) = *}{\Gamma \vdash a :: *} \qquad \frac{\Gamma \vdash t :: * \quad \Gamma \vdash t' :: *}{\Gamma \vdash t \rightarrow t' :: *}$$

Dependently-typed Lambda Calculus λ_Π

$$
\begin{array}{lcl}
e, t & ::= & e :: t \mid * \mid \forall x :: t.t' \mid x \mid e_1\ e_2 \mid \lambda x \to e \\
\Gamma & ::= & \varepsilon \mid \Gamma, x :: t
\end{array}
$$

$$
\frac{}{\text{valid}(\varepsilon)} \qquad \frac{\text{valid}(\Gamma) \quad \Gamma \vdash t ::_\downarrow *}{\text{valid}(\Gamma, x :: t)}
$$

$$
\frac{\Gamma(a) = *}{\Gamma \vdash a :: *} \qquad \frac{\Gamma \vdash t :: * \quad \Gamma \vdash t' :: *}{\Gamma \vdash t \to t' :: *}
$$

$$
\frac{\Gamma \vdash t ::_\downarrow * \quad \Gamma \vdash e ::_\downarrow t}{\Gamma \vdash (e :: t) ::_\uparrow t} \qquad \frac{}{\Gamma \vdash * ::_\uparrow *} \qquad \frac{\Gamma \vdash t ::_\downarrow * \quad \Gamma, x :: t \vdash t' ::_\downarrow *}{\Gamma \vdash \forall x :: t.t' ::_\uparrow *}
$$

$$
\frac{\Gamma(x) = t}{\Gamma \vdash x ::_\uparrow t} \qquad \frac{\Gamma \vdash e_1 ::_\uparrow \forall x :: t.t' \quad \Gamma \vdash e_2 ::_\downarrow t}{\Gamma \vdash e_1\ e_2 ::_\uparrow t'[x \mapsto e_2]}
$$

$$
\frac{\Gamma \vdash e ::_\uparrow t' \quad t \Downarrow v \quad t' \Downarrow v}{\Gamma \vdash e ::_\downarrow t} \qquad \frac{\Gamma, x :: t \vdash e ::_\downarrow t'}{\Gamma \vdash \lambda x \to e ::_\downarrow \forall x :: t.t'}
$$

Evaluation in λ_Π

$$v ::= x\ \bar{v} \mid * \mid \forall x :: v.v' \mid \lambda x \to v$$

Evaluation in λ_Π

$$v ::= x\ \overline{v} \mid * \mid \forall x :: v.v' \mid \lambda x \rightarrow v$$

$$\frac{e \Downarrow v}{e :: t \Downarrow v} \qquad \frac{}{* \Downarrow *} \qquad \frac{t \Downarrow v \quad t' \Downarrow v'}{\forall x :: t.t' \Downarrow \forall x :: v.v'} \qquad \frac{}{x \Downarrow x}$$

Evaluation in λ_Π

$$v ::= x\ \bar{v} \mid * \mid \forall x :: v.v' \mid \lambda x \to v$$

$$\frac{e \Downarrow v}{e :: t \Downarrow v} \qquad \frac{}{* \Downarrow *} \qquad \frac{t \Downarrow v \quad t' \Downarrow v'}{\forall x :: t.t' \Downarrow \forall x :: v.v'} \qquad \frac{}{x \Downarrow x}$$

$$\frac{e_1 \Downarrow \lambda x \to v_1 \quad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow v_1[x \mapsto v_2]} \qquad \frac{e_1 \Downarrow n_1 \quad e_2 \Downarrow v_2}{e_1\ e_2 \Downarrow n_1\ v_2} \qquad \frac{e \Downarrow v}{\lambda x \to e \Downarrow \lambda x \to v}$$

Implementation in Haskell

- Abstract Syntax
- Evaluation
- Substitution
- Typechecking
- Quotation

Abstract Syntax

$$
\begin{aligned}
&\textbf{data}\ \mathsf{Term}_{\uparrow} \\
&\quad = \mathsf{Ann}\ \mathsf{Term}_{\downarrow}\ \mathsf{Term}_{\downarrow} \\
&\quad \mid\ \mathsf{Star} \\
&\quad \mid\ \mathsf{Pi}\ \mathsf{Term}_{\downarrow}\ \mathsf{Term}_{\downarrow} \\
&\quad \mid\ \mathsf{Var}\ \mathsf{Int} \\
&\quad \mid\ \mathsf{Par}\ \mathsf{Name} \\
&\quad \mid\ \mathsf{Term}_{\uparrow}\ \texttt{:@:}\ \mathsf{Term}_{\downarrow} \\
&\quad \textbf{deriving}\ (\mathsf{Show}, \mathsf{Eq}) \\
&\textbf{data}\ \mathsf{Term}_{\downarrow} \\
&\quad = \mathsf{Inf}\quad \mathsf{Term}_{\uparrow} \\
&\quad \mid\ \mathsf{Lam}\ \mathsf{Term}_{\downarrow} \\
&\quad \textbf{deriving}\ (\mathsf{Show}, \mathsf{Eq})
\end{aligned}
$$

Contexts, Values

```
type Type    = Value
type Context = [(Name, Type)]
data Value
  = VLam (Value → Value)
  | VStar
  | VPi Value (Value → Value)
  | VNeutral Neutral
data Neutral
  = NPar  Name
  | NApp Neutral Value
```

Evaluation

$$\mathsf{eval}_{\downarrow} :: \mathsf{Term}_{\downarrow} \rightarrow \mathsf{Env} \rightarrow \mathsf{Value}$$
$$\mathsf{eval}_{\uparrow} :: \mathsf{Term}_{\uparrow} \rightarrow \mathsf{Env} \rightarrow \mathsf{Value}$$

Typechecking

$$\text{type}_{\uparrow} :: \text{Int} \rightarrow \text{Context} \rightarrow \text{Term}_{\uparrow} \rightarrow \text{Result Type}$$
$$\text{type}_{\downarrow} :: \text{Int} \rightarrow \text{Context} \rightarrow \text{Term}_{\downarrow} \rightarrow \text{Type} \rightarrow \text{Result ()}$$

```
type↑ i Γ (Ann e t)
    = do type↓ i Γ t VStar
         let v = eval↓ t []
         type↓ i Γ e v
         return v
type↑ i Γ Star
    = return VStar
```

Typechecking, continued

$\mathsf{type}_{\uparrow} :: \mathsf{Int} \to \mathsf{Context} \to \mathsf{Term}_{\uparrow} \to \mathsf{Result\ Type}$
$\mathsf{type}_{\downarrow} :: \mathsf{Int} \to \mathsf{Context} \to \mathsf{Term}_{\downarrow} \to \mathsf{Type} \to \mathsf{Result}\ ()$

```
type↑ i Γ (Pi t t′)
    = do type↓ i Γ t VStar
         let v = eval↓ t []
         type↓ (i + 1) ((Bound i, v) : Γ)
               (subst↓ 0 (Par (Bound i)) t′) VStar
         return VStar
```

$\mathsf{subst}_{\uparrow} :: \mathsf{Int} \to \mathsf{Term}_{\uparrow} \to \mathsf{Term}_{\uparrow} \to \mathsf{Term}_{\uparrow}$
$\mathsf{subst}_{\downarrow} :: \mathsf{Int} \to \mathsf{Term}_{\uparrow} \to \mathsf{Term}_{\downarrow} \to \mathsf{Term}_{\downarrow}$

Typechecking, continued

$$\begin{aligned}&\mathsf{type}_{\downarrow}\ \mathsf{i}\ \Gamma\ (\mathsf{Inf}\ \mathsf{e})\ \mathsf{v}\\&\quad = \textbf{do}\ \mathsf{v}' \leftarrow \mathsf{type}_{\uparrow}\ \mathsf{i}\ \Gamma\ \mathsf{e}\\&\qquad\quad \mathsf{unless}\ (\mathsf{quote}_0\ \mathsf{v} == \mathsf{quote}_0\ \mathsf{v}')\ (\mathsf{throwError}\ \texttt{"type mismatch"})\end{aligned}$$

Typechecking, continued

$\mathsf{type}_{\downarrow}\ \mathsf{i}\ \Gamma\ (\mathsf{Inf}\ \mathsf{e})\ \mathsf{v}$
$\quad = \textbf{do}\ \mathsf{v}' \leftarrow \mathsf{type}_{\uparrow}\ \mathsf{i}\ \Gamma\ \mathsf{e}$
$\qquad\qquad \mathsf{unless}\ (\mathsf{quote}_0\ \mathsf{v} == \mathsf{quote}_0\ \mathsf{v}')\ (\mathsf{throwError}\ \texttt{"type mismatch"})$

$\mathsf{quote}_0 :: \mathsf{Value} \rightarrow \mathsf{Term}_{\downarrow}$
$\mathsf{quote}_0 = \mathsf{quote}\ 0$

$\mathsf{quote} :: \mathsf{Int} \rightarrow \mathsf{Value} \rightarrow \mathsf{Term}_{\downarrow}$

Typechecking, continued

$\text{type}_\downarrow\ \text{i}\ \Gamma\ (\text{Inf e})\ \text{v}$
$\quad = \textbf{do}\ \text{v}' \leftarrow \text{type}_\uparrow\ \text{i}\ \Gamma\ \text{e}$
$\qquad\quad \text{unless}\ (\text{quote}_0\ \text{v} == \text{quote}_0\ \text{v}')$ (throwError `"type mismatch"`)

$\text{quote}_0 :: \text{Value} \rightarrow \text{Term}_\downarrow$
$\text{quote}_0 = \text{quote}\ 0$

$\text{quote} :: \text{Int} \rightarrow \text{Value} \rightarrow \text{Term}_\downarrow$

Example:

$\quad \text{quote}\ 0\ (\text{VLam}\ (\lambda x \rightarrow \text{VLam}\ (\lambda y \rightarrow x)))$
$= \text{Lam}\ (\text{quote}\ 1\ (\text{VLam}\ (\lambda y \rightarrow \text{vpar}\ (\text{Unquoted}\ 0))))$
$= \text{Lam}\ (\text{Lam}\ (\text{quote}\ 2\ (\text{vpar}\ (\text{Unquoted}\ 0))))$
$= \text{Lam}\ (\text{Lam}\ (\text{neutralQuote}\ 2\ (\text{NPar}\ (\text{Unquoted}\ 0))))$
$= \text{Lam}\ (\text{Lam}\ (\text{Var}\ 1))$

Where are the dependent types?

- Total implementation is about 100 lines of Haskell code.
- Easy to see that we have gained advantages compared to $\lambda_{\rightarrow}$.
- Hard to actually use the power of dependent types without adding datatypes to the language.

Adding datatypes

- Add the type.
- Add the constructors (introduction forms).
  - Types
  - Add constructors to values.
- Add an eliminator (eliminator forms).
  - Type
  - Add evaluation rules for eliminator.

Natural numbers

$$
\begin{aligned}
\mathsf{e} &::= \cdots \mid \mathsf{Nat} \mid \mathsf{Zero} \mid \mathsf{Succ\ e} \mid \mathsf{natElim\ e\ e\ e\ e} \\
\mathsf{v} &::= \cdots \mid \mathsf{Nat} \mid \mathsf{Zero} \mid \mathsf{Succ\ v} \\
\mathsf{n} &::= \cdots \mid \mathsf{natElim\ v\ v\ n}
\end{aligned}
$$

Evaluation

$$\frac{}{\mathsf{Nat} \Downarrow \mathsf{Nat}} \qquad \frac{}{\mathsf{Zero} \Downarrow \mathsf{Zero}} \qquad \frac{\mathsf{k} \Downarrow \mathsf{l}}{\mathsf{Succ}\ \mathsf{k} \Downarrow \mathsf{Succ}\ \mathsf{l}}$$

Evaluation

$$\frac{}{\mathsf{Nat} \Downarrow \mathsf{Nat}} \qquad \frac{}{\mathsf{Zero} \Downarrow \mathsf{Zero}} \qquad \frac{\mathsf{k} \Downarrow \mathsf{l}}{\mathsf{Succ\ k} \Downarrow \mathsf{Succ\ l}}$$

$$\frac{\mathsf{mz} \Downarrow \mathsf{v}}{\mathsf{natElim\ m\ mz\ ms\ Zero} \Downarrow \mathsf{v}} \qquad \frac{\mathsf{ms\ k\ (natElim\ m\ mz\ ms\ k)} \Downarrow \mathsf{v}}{\mathsf{natElim\ m\ mz\ ms\ (Succ\ k)} \Downarrow \mathsf{v}}$$

Typing

$$\frac{}{\Gamma \vdash \mathsf{Nat} :: *} \quad \frac{}{\Gamma \vdash \mathsf{Zero} :: \mathsf{Nat}} \quad \frac{\Gamma \vdash k :: \mathsf{Nat}}{\Gamma \vdash \mathsf{Succ}\ k :: \mathsf{Nat}}$$

Typing

$$\frac{}{\Gamma \vdash \mathsf{Nat} :: *} \quad \frac{}{\Gamma \vdash \mathsf{Zero} :: \mathsf{Nat}} \quad \frac{\Gamma \vdash \mathsf{k} :: \mathsf{Nat}}{\Gamma \vdash \mathsf{Succ}\ \mathsf{k} :: \mathsf{Nat}}$$

$$\frac{\begin{array}{c}\Gamma \vdash \mathsf{m} :: \mathsf{Nat} \to * \\ \Gamma, \mathsf{m} :: \mathsf{Nat} \to * \vdash \mathsf{mz} :: \mathsf{m}\ \mathsf{Zero} \\ \Gamma, \mathsf{m} :: \mathsf{Nat} \to * \vdash \mathsf{ms} :: \forall \mathsf{k} :: \mathsf{Nat}.\mathsf{m}\ \mathsf{k} \to \mathsf{m}\ (\mathsf{Succ}\ \mathsf{k}) \\ \Gamma \vdash \mathsf{n} :: \mathsf{Nat}\end{array}}{\Gamma \vdash \mathsf{natElim}\ \mathsf{m}\ \mathsf{mz}\ \mathsf{ms}\ \mathsf{n} :: \mathsf{m}\ \mathsf{n}}$$

Eliminator vs. fold

$$
\begin{aligned}
\text{natFold} :: \forall m :: * \quad\quad\quad . \quad & m \\
\rightarrow\ & (m \rightarrow m) \\
\rightarrow\ & \text{Nat} \rightarrow a \\
\text{natElim} :: \forall m :: \text{Nat} \rightarrow *. \quad & m\ \text{Zero} \\
\rightarrow\ & (\forall k :: \text{Nat}.m\ k \rightarrow m\ (\text{Succ}\ k)) \\
\rightarrow\ & \forall n :: \text{Nat}.m\ n
\end{aligned}
$$

$$\text{natFold}\ r\ \text{zero}\ \text{succ} = \text{natElim}\ (\lambda\_ \rightarrow r)\ \text{zero}\ (\lambda\_\ \text{rec} \rightarrow \text{succ}\ \text{rec})$$

Addition

$$\begin{aligned}\mathsf{plus} = \mathsf{natElim}\ &(\lambda\_ \rightarrow \mathsf{Nat} \rightarrow \mathsf{Nat})\\ &(\lambda \mathsf{n} \rightarrow \mathsf{n})\\ &(\lambda\_\ \mathsf{rec}\ \mathsf{n} \rightarrow \mathsf{Succ}\ (\mathsf{rec}\ \mathsf{n}))\end{aligned}$$

Implementation in Haskell

Systematically extend all the functions . . .

Vectors

Vectors are lists that keep track of their length.

$\mathsf{Vec} :: \forall (\mathsf{a} :: *)\ (\mathsf{n} :: \mathsf{Nat}).\ *$

Vectors

Vectors are lists that keep track of their length.

$\mathsf{Vec} :: \forall (\mathsf{a} :: *)\ (\mathsf{n} :: \mathsf{Nat}).\ *$

$\mathsf{Nil} \quad :: \forall \mathsf{a} :: *. \mathsf{Vec\ a\ Zero}$

$\mathsf{Cons} :: \forall \mathsf{a} :: *. \forall \mathsf{n} :: \mathsf{Nat}. \mathsf{a} \rightarrow \mathsf{Vec\ a\ n} \rightarrow \mathsf{Vec\ a\ (Succ\ n)}$

Vectors

Vectors are lists that keep track of their length.

$\mathsf{Vec} :: \forall (\mathsf{a} :: *)\ (\mathsf{n} :: \mathsf{Nat}).\ *$

$\mathsf{Nil} \quad :: \forall \mathsf{a} :: *.\mathsf{Vec\ a\ Zero}$

$\mathsf{Cons} :: \forall \mathsf{a} :: *.\forall \mathsf{n} :: \mathsf{Nat}.\mathsf{a} \to \mathsf{Vec\ a\ n} \to \mathsf{Vec\ a\ (Succ\ n)}$

$$
\begin{aligned}
\mathsf{vecElim} :: \forall \mathsf{a} :: *.\forall \mathsf{m} :: (\forall \mathsf{n} :: \mathsf{Nat}.\mathsf{Vec\ a\ n} \to *).\\
&\ \ \mathsf{m\ Zero\ (Nil\ a)}\\
&\to (\forall \mathsf{n} :: \mathsf{Nat}.\forall \mathsf{x} :: \mathsf{a}.\forall \mathsf{xs} :: \mathsf{Vec\ a\ n}.\\
&\qquad \mathsf{m\ n\ xs} \to \mathsf{m\ (Succ\ n)\ (Cons\ a\ n\ x\ xs)})\\
&\to \forall \mathsf{n} :: \mathsf{Nat}.\forall \mathsf{xs} :: \mathsf{Vec\ a\ n}.\mathsf{m\ n\ xs}
\end{aligned}
$$

Vector append

```
append =
  (λa → vecElim a
          (λm _ → ∀(n :: Nat).Vec a n → Vec a (plus m n))
          (λ_ v → v)
          (λm v vs rec n w → Cons a (plus m n) v (rec n w)))
   :: ∀(a :: ∗) (m :: Nat) (v :: Vec a m) (n :: Nat) (w :: Vec a n).
     Vec a (plus m n)
```

Other interesting types

- Zero
- One (or Unit)
- Two (or Bool)
- Fin
- Eq
- Holds
- dependent pairs
- ...

Lots of missing features

- implicit arguments
- proper case analysis
- user feedback (error messages)
- ...