Why Dependent Types Matter

Thorsten Altenkirch

University of Nottingham

based on joint work with Conor McBride

and cartoons by

The established social order

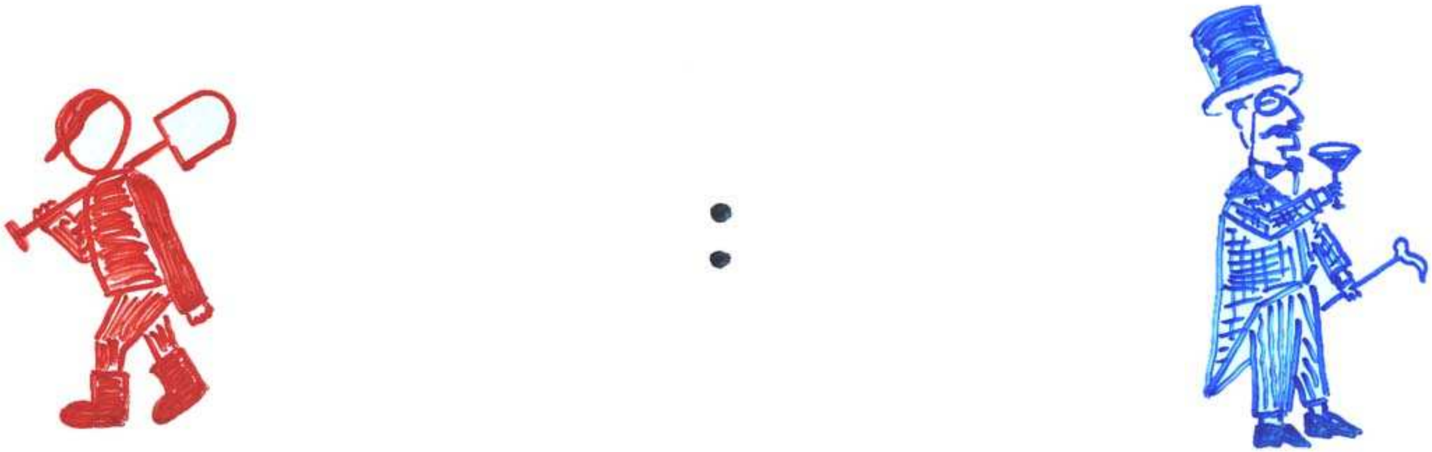
-
-

The established social order

:

terms

The established social order



terms types

The established social order

:

terms

types

do all the work

The established social order

:

terms

do all the work

types

are never around when

there is work to be done

The established social order

:

terms

do all the work

engage in criminal activity

types

are never around when

there is work to be done

The established social order

:

terms

do all the work

engage in criminal activity

types

are never around when

there is work to be done

commit no crime

The established social order

:

terms

do all the work

engage in criminal activity

can be stopped and searched

types

are never around when

there is work to be done

commit no crime

The established social order

:

terms

do all the work

engage in criminal activity

can be stopped and searched

types

are never around when there is work to be done

commit no crime

cannot be investigated.

The established social order

:

terms

do all the work

engage in criminal activity

can be stopped and searched

belong to and are

hold in check by types

types

are never around when

there is work to be done

commit no crime

cannot be investigated.

The established social order

terms

do all the work

engage in criminal activity

can be stopped and searched

belong to and are
hold in check by types

types

are never around when
there is work to be done

commit no crime

cannot be investigated.

The established social order

terms

do all the work

engage in criminal activity

can be stopped and searched

belong to and are

hold in check by types

types

are never around when

there is work to be done

commit no crime

cannot be investigated.

Ersatz dependent types

Ersatz dependent types

- In modern type systems types have to do some work

Ersatz dependent types

- In modern type systems types have to do some work
- Polymorphic types can be use to represent square matrices

$$
\begin{array}{lcl}
\mathrm{Matrix}\ a & = & \mathrm{Matrix}'\ \mathrm{Nil}\ a \\
\mathrm{Matrix}'\ t\ a & = & \mathrm{Zero}\ (t\ (t\ a)) \mid \mathrm{Succ}\ (\mathrm{Cons}\ t)\ a \\
\mathrm{Nil}\ a & = & \mathrm{Nil} \\
\mathrm{Cons}\ t\ a & = & \mathrm{Cons}\ a\ (t\ a)
\end{array}
$$

Ersatz dependent types

- In modern type systems types have to do some work
- Polymorphic types can be use to represent square matrices

$$
\begin{array}{lcl}
\text{Matrix}\ a & = & \text{Matrix}'\ \text{Nil}\ a \\
\text{Matrix}'\ t\ a & = & \text{Zero}\ (t\ (t\ a)) \mid \text{Succ}\ (\text{Cons}\ t)\ a \\
\text{Nil}\ a & = & \text{Nil} \\
\text{Cons}\ t\ a & = & \text{Cons}\ a\ (t\ a)
\end{array}
$$

- Conor showed in *Faking it* how to use the logic programming of Haskell's class system to simulate some usages of dependent types.

Ersatz dependent types

- In modern type systems types have to do some work
- Polymorphic types can be use to represent square matrices

$$
\begin{array}{lcl}
\mathrm{Matrix}\ a & = & \mathrm{Matrix}'\ \mathrm{Nil}\ a \\
\mathrm{Matrix}'\ t\ a & = & \mathrm{Zero}\ (t\ (t\ a)) \mid \mathrm{Succ}\ (\mathrm{Cons}\ t)\ a \\
\mathrm{Nil}\ a & = & \mathrm{Nil} \\
\mathrm{Cons}\ t\ a & = & \mathrm{Cons}\ a\ (t\ a)
\end{array}
$$

- Conor showed in *Faking it* how to use the logic programming of Haskell's class system to simulate some usages of dependent types.

Breaking the old social order

Breaking the old social order

- Data is validated wrt other data

Breaking the old social order

- Data is validated wrt other data
- If types are to capture the validity of data, we must let them depend on terms.

Breaking the old social order

- Data is validated wrt other data
- If types are to capture the validity of data, we must let them depend on terms.

Breaking the old social order

- Data is validated wrt other data
- If types are to capture the validity of data, we must let them depend on terms.

Breaking the old social order

- Data is validated wrt other data
- If types are to capture the validity of data, we must let them depend on terms.

Overview

Overview

- Examples for programming with dependent types:

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

verify nth,eval

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

  verify nth,eval

  reflect eq

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

  verify nth,eval

  reflect eq
- Emphasis on safe and efficient execution

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

  verify nth,eval

  reflect eq
- Emphasis on safe and efficient execution
- Using **epigram** currently developed by Conor, using ideas from

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

  verify nth,eval

  reflect eq
- Emphasis on safe and efficient execution
- Using **epigram** currently developed by Conor, using ideas from
  - LEGO / OLEG

Overview

- Examples for programming with dependent types:
  - nth – safe access to lists/vectors
  - eval – safe eval
  - eq – generic equality
- Illustrating patterns in DTP

  verify nth,eval

  reflect eq
- Emphasis on safe and efficient execution
- Using **epigram** currently developed by Conor, using ideas from
  - LEGO / OLEG
  - ALF

nth — no dependent types

$$\text{let } \frac{\mathrm{A} \in * \quad l \in \mathrm{List}\ \mathrm{A} \quad n \in \mathrm{Nat}}{\mathrm{nth}\ \mathrm{A}\ l\ n \in \mathrm{A}}$$

$$\mathrm{nth}\ \mathrm{A}\ l\ n \mapsto\ ?$$

nth — no dependent types

$$\text{let}\ \frac{l \in \text{List A} \quad n \in \text{Nat}}{\text{nth} \quad l\ n \in \text{A}}$$

$$\text{nth} \quad l\ n \mapsto ?$$

- Hindley-Milner: Type quantification and application can be made implicit.

nth — no dependent types

$$\text{let}\ \frac{l \in \text{List A} \quad n \in \text{Nat}}{\text{nth} \quad l\ n \in \text{A}}$$

$$\begin{aligned} \text{nth nil } n &\mapsto ? \\ \text{nth } (\text{cons } a\ as)\ n &\mapsto ? \end{aligned}$$

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information

nth — no dependent types

$$\text{let } \frac{l \in \mathrm{List}\ \mathrm{A} \qquad n \in \mathrm{Nat}}{\mathrm{nth} \quad l\ n \in \mathrm{A}}$$

$$\begin{array}{ll} \mathrm{nth}\ \mathrm{nil}\ n & \mapsto\ ? \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ 0 & \mapsto\ ? \\ \mathrm{nth}\ (\mathrm{cons}\ a\ as)\ (\mathrm{s}\ n) & \mapsto\ ? \end{array}$$

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information

nth — no dependent types

$$\text{let}\ \frac{l \in \text{List A} \quad n \in \text{Nat}}{\text{nth} \quad l\ n \in \text{A}}$$

$$\begin{array}{ll} \text{nth nil}\ n & \mapsto ? \\ \text{nth (cons}\ a\ as)\ 0 & \mapsto a \\ \text{nth (cons}\ a\ as)\ (\text{s}\ n) & \mapsto ? \end{array}$$

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information

nth — no dependent types

$$\text{let}\ \frac{l \in \mathsf{List}\ \mathsf{A} \qquad n \in \mathsf{Nat}}{\mathsf{nth}\quad l\ n \in \mathsf{A}}$$

$$\begin{aligned}
\mathsf{nth}\ \mathsf{nil}\ n &\mapsto ?\\
\mathsf{nth}\ (\mathsf{cons}\ a\ as)\ 0 &\mapsto a\\
\mathsf{nth}\ (\mathsf{cons}\ a\ as)\ (\mathsf{s}\ n) &\mapsto \mathsf{nth}\ as\ n
\end{aligned}$$

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information

nth is not good

$$\text{let } \frac{l \in \mathrm{List\ A} \qquad n \in \mathrm{Nat}}{\mathrm{nth}\ l\ n \in \mathrm{A}}$$

$$\begin{aligned}
&\mathrm{nth}\ \mathrm{nil}\ n && \mapsto ? \\
&\mathrm{nth}\ (\mathrm{cons}\ a\ as)\ 0 && \mapsto a \\
&\mathrm{nth}\ (\mathrm{cons}\ a\ as)\ (\mathrm{s}\ n) && \mapsto \mathrm{nth}\ as\ n
\end{aligned}$$

nth is not good

$$\text{let}\ \frac{l \in \text{List A} \quad n \in \text{Nat}}{\text{nth}\ l\ n \in \text{A}} \qquad \begin{array}{ll} \text{nth nil}\ n & \mapsto\ ? \\ \text{nth}\ (\text{cons}\ a\ as)\ 0 & \mapsto a \\ \text{nth}\ (\text{cons}\ a\ as)\ (\text{s}\ n) & \mapsto \text{nth}\ as\ n \end{array}$$

- The function nth is partial

$$\text{nth}\ 3\ [1, 2]$$

leads to a runtime error.

nth is not good

let $\dfrac{l \in \text{List A} \quad n \in \text{Nat}}{\text{nth } l \; n \in \text{A}}$

$$
\begin{aligned}
&\text{nth nil } n && \mapsto ? \\
&\text{nth } (\text{cons } a \; as) \; 0 && \mapsto a \\
&\text{nth } (\text{cons } a \; as) \; (\text{s } n) && \mapsto \text{nth } as \; n
\end{aligned}
$$

- The function nth is partial

$$\text{nth } 3 \; [1, 2]$$

  leads to a runtime error.

- Reason: The type of nth is not informative enough.

data types

data $\dfrac{}{\mathrm{Nat} \in *}$ where $\quad \dfrac{}{0 \in \mathrm{Nat}} \quad \dfrac{n \in \mathrm{Nat}}{\mathrm{s}\ n \in \mathrm{Nat}}$

data $\dfrac{\mathrm{A} \in *}{\mathrm{List}\ \mathrm{A} \in *}$ where

$$\frac{\mathrm{A} \in *}{\mathrm{nil}_{\mathrm{A}} \in \mathrm{List}\ \mathrm{A}} \qquad \frac{\mathrm{A} \in * \qquad a \in \mathrm{A} \qquad as \in \mathrm{List}\ \mathrm{A}}{\mathrm{cons}_{\mathrm{A}}\ a\ as \in \mathrm{List}\ \mathrm{A}}$$

data types

data $\dfrac{}{\mathrm{Nat} \in *}$ where $\dfrac{}{0 \in \mathrm{Nat}}$ $\dfrac{n \in \mathrm{Nat}}{\mathrm{s}\ n \in \mathrm{Nat}}$

data $\dfrac{\mathrm{A} \in *}{\mathrm{List\ A} \in *}$ where

$$\dfrac{}{\mathrm{nil} \quad \in \mathrm{List\ A}} \qquad \dfrac{a \in \mathrm{A} \qquad as \in \mathrm{List\ A}}{\mathrm{cons} \quad a\ as \in \mathrm{List\ A}}$$

Better data types, better nth

Better data types, better nth

$$\text{data } \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *} \text{ where } \frac{n \in \mathrm{Nat}}{0'_n \in \mathrm{Fin}\ (\mathrm{s}\ n)} \quad \frac{n \in \mathrm{Nat} \qquad i \in \mathrm{Fin}\ n}{\mathrm{s}'_n\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

Better data types, better nth

$$\text{data } \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *} \text{ where } \frac{n \in \mathrm{Nat}}{0'_n \in \mathrm{Fin}\ (\mathrm{s}\ n)} \quad \frac{n \in \mathrm{Nat} \qquad i \in \mathrm{Fin}\ n}{\mathrm{s}'_n\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data } \frac{\mathrm{A} \in * \qquad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *} \text{ where } \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0} \quad \frac{n \in \mathrm{Nat} \qquad a \in \mathrm{A} \qquad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}_n\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)}\quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

Better data types, better nth

$$\text{data } \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *} \text{ where } \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)} \quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data } \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *} \text{ where } \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0} \quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let } \frac{n \in \mathrm{Nat} \quad l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}_n\ l\ i \in \mathrm{A}}$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)}\quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let}\ \frac{l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}\ l\ i \in \mathrm{A}}$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)}\quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let}\ \frac{l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}\ l\ i \in \mathrm{A}}$$

$$\mathrm{nth}\ l\ n \mapsto ?$$

Better data types, better nth

$$\text{data}\ \frac{n \in \text{Nat}}{\text{Fin}\ n \in *}\ \text{where}\ \frac{n \in \text{Nat}}{0' \in \text{Fin}\ (\text{s}\ n)}\quad \frac{i \in \text{Fin}\ n}{\text{s}'\ i \in \text{Fin}\ (\text{s}\ n)}$$

$$\text{data}\ \frac{\text{A} \in * \quad n \in \text{Nat}}{\text{Vec A}\ n \in *}\ \text{where}\ \frac{}{\text{vnil} \in \text{Vec A}\ 0}\quad \frac{a \in \text{A} \quad as \in \text{Vec A}\ n}{\text{vcons}\ a\ as \in \text{Vec A}\ (\text{s}\ n)}$$

$$\text{let}\ \frac{l \in \text{Vec A}\ n \quad i \in \text{Fin}\ n}{\text{nth}\ l\ i \in \text{A}}$$

$$\text{nth}\ l\ 0' \mapsto ?$$
$$\text{nth}\ l\ (\text{s}'\ i) \mapsto ?$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)}\quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let}\ \frac{l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}\ l\ i \in \mathrm{A}}$$

$$\mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ 0' \mapsto ?$$

$$\mathrm{nth}\ l\ (\mathrm{s}'\ i) \mapsto ?$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)}\quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let}\ \frac{l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}\ l\ i \in \mathrm{A}}$$

$$\mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ 0' \mapsto ?$$

$$\mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ (\mathrm{s}'\ i) \mapsto ?$$

Better data types, better nth

$$\text{data}\ \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *}\ \text{where}\ \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)}\quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data}\ \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *}\ \text{where}\ \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0}\quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let}\ \frac{l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}\ l\ i \in \mathrm{A}}$$

$$\begin{aligned}
\mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ 0' &\mapsto a \\
\mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ (\mathrm{s}'\ i) &\mapsto \mathrm{nth}\ as\ i
\end{aligned}$$

- nth is a total function.

Better data types, better nth

$$\text{data } \frac{n \in \mathrm{Nat}}{\mathrm{Fin}\ n \in *} \text{ where } \frac{n \in \mathrm{Nat}}{0' \in \mathrm{Fin}\ (\mathrm{s}\ n)} \quad \frac{i \in \mathrm{Fin}\ n}{\mathrm{s}'\ i \in \mathrm{Fin}\ (\mathrm{s}\ n)}$$

$$\text{data } \frac{\mathrm{A} \in * \quad n \in \mathrm{Nat}}{\mathrm{Vec}\ \mathrm{A}\ n \in *} \text{ where } \frac{}{\mathrm{vnil} \in \mathrm{Vec}\ \mathrm{A}\ 0} \quad \frac{a \in \mathrm{A} \quad as \in \mathrm{Vec}\ \mathrm{A}\ n}{\mathrm{vcons}\ a\ as \in \mathrm{Vec}\ \mathrm{A}\ (\mathrm{s}\ n)}$$

$$\text{let } \frac{l \in \mathrm{Vec}\ \mathrm{A}\ n \quad i \in \mathrm{Fin}\ n}{\mathrm{nth}\ l\ i \in \mathrm{A}}$$

$$\begin{aligned} \mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ 0' &\mapsto a \\ \mathrm{nth}\ (\mathrm{vcons}\ a\ as)\ (\mathrm{s}'\ i) &\mapsto \mathrm{nth}\ as\ i \end{aligned}$$

- nth is a total function.
- $\mathrm{nth}\ 3\ [1, 2]$ is not well-typed.

Verify

Verify

- How can we use nth on lists of unknown length user input,...?

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

Verify

- How can we use nth on lists of unknown length user input,. . . ?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\mathrm{verify}\ n\ i \quad \mapsto \quad ?$$

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let } \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & ? \\ \mathrm{verify}\ (\mathrm{s}\ n)\ i & \mapsto & ? \end{array}$$

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let } \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\ \mathrm{verify}\ (\mathrm{s}\ n)\ i & \mapsto & ? \end{array}$$

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let } \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\ \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & ? \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \mapsto & ? \end{array}$$

Verify

- How can we use nth on lists of unknown length user input,. . . ?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl}
\mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\
\mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{just}\ 0' \\
\mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \mapsto & ?
\end{array}$$

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\ \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{just}\ 0' \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \| & \mathrm{verify}\ n\ i \quad \mapsto \quad ? \end{array}$$

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let } \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\ \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{just}\ 0' \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \| & \mathrm{verify}\ n\ i \\ & | & \mathrm{nothing} \quad \mapsto \quad ? \\ & | & \mathrm{just}\ i \quad \mapsto \quad ? \end{array}$$

Verify

- How can we use nth on lists of unknown length user input,...?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl}
\mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\
\mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{just}\ 0' \\
\mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \| & \mathrm{verify}\ n\ i \\
 & | & \mathrm{nothing} \quad \mapsto \quad \mathrm{nothing} \\
 & | & \mathrm{just}\ i \quad \mapsto \quad ?
\end{array}$$

Verify

- How can we use nth on lists of unknown length user input,. . . ?

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Maybe}\ (\mathrm{Fin}\ n)}$$

$$\begin{array}{lcl}
\mathrm{verify}\ 0\ i & \mapsto & \mathrm{nothing} \\
\mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{just}\ 0' \\
\mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \| & \mathrm{verify}\ n\ i \\
 & | & \mathrm{nothing} \quad \mapsto \quad \mathrm{nothing} \\
 & | & \mathrm{just}\ i \quad \mapsto \quad \mathrm{just}\ (\mathrm{s}'\ i)
\end{array}$$

Going further

Going further

- The type of verify is not informative enough for some of its potential applications.

Going further

- The type of verify is not informative enough for some of its potential applications.
- How is $\mathrm{just}\ j \equiv \mathrm{verify}\ n\ i$ related to j?

Going further

- The type of verify is not informative enough for some of its potential applications.
- How is $\mathrm{just}\ j \equiv \mathrm{verify}\ n\ i$ related to j?
- When does verify return $\mathrm{nothing}$?

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ \ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lll} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \quad \begin{array}{lll} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \quad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\mathrm{verify}\ n\ i \quad \mapsto \quad ?$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n+i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & ? \\ \mathrm{verify}\ (\mathrm{s}\ n)\ i & \mapsto & ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ \ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)}\ \ \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{tooBig}\ 0\ i \\ \mathrm{verify}\ (\mathrm{s}\ n)\ i & \mapsto & ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \quad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\begin{array}{lcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{tooBig}\ 0\ i \\ \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & ? \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \mapsto & ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \text{Fin}\ n}{\text{val}\ i \in \text{Nat}} \qquad \begin{array}{lcl} \text{val}\ 0' & \mapsto & 0 \\ \text{val}\ (\text{s}'\ i) & \mapsto & \text{s}\ (\text{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \text{Nat}}{\text{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \text{Fin}\ n}{\text{bound}\ n\ i \in \text{Bound}\ n\ (\text{val}\ i)} \quad \frac{i, n \in \text{Nat}}{\text{tooBig}\ n\ i \in \text{Bound}\ n\ (n+i)}$$

$$\text{let}\ \frac{n, i \in \text{Nat}}{\text{verify}\ n\ i \in \text{Bound}\ n\ i}$$

$$\begin{array}{lcl} \text{verify}\ 0\ i & \mapsto & \text{tooBig}\ 0\ i \\ \text{verify}\ (\text{s}\ n)\ 0 & \mapsto & \text{bound}\ n\ 0' \\ \text{verify}\ (\text{s}\ n)\ (\text{s}\ i) & \mapsto & ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \text{Fin}\ n}{\text{val}\ i \in \text{Nat}} \quad \begin{array}{lcl} \text{val}\ 0' & \mapsto & 0 \\ \text{val}\ (\text{s}'\ i) & \mapsto & \text{s}\ (\text{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \text{Nat}}{\text{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \text{Fin}\ n}{\text{bound}\ n\ i \in \text{Bound}\ n\ (\text{val}\ i)} \quad \frac{i, n \in \text{Nat}}{\text{tooBig}\ n\ i \in \text{Bound}\ n\ (n+i)}$$

$$\text{let}\ \frac{n, i \in \text{Nat}}{\text{verify}\ n\ i \in \text{Bound}\ n\ i}$$

$$\begin{array}{lcl} \text{verify}\ 0\ i & \mapsto & \text{tooBig}\ 0\ i \\ \text{verify}\ (\text{s}\ n)\ 0 & \mapsto & \text{bound}\ n\ 0' \\ \text{verify}\ (\text{s}\ n)\ (\text{s}\ i) & \| & \text{verify}\ n\ i \quad \mapsto \quad ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\begin{array}{lcllcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{tooBig}\ 0\ i & & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{bound}\ n\ 0' & & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \| & \mathrm{verify}\ n\ i & & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ (n{+}i)) & | & \mathrm{tooBig}\ n\ i & \mapsto & ? \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{val}\ i) & | & \mathrm{bound}\ n\ i & \mapsto & ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{val}\ i \in \mathrm{Nat}} \qquad \begin{array}{lcl} \mathrm{val}\ 0' & \mapsto & 0 \\ \mathrm{val}\ (\mathrm{s}'\ i) & \mapsto & \mathrm{s}\ (\mathrm{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathrm{Fin}\ n}{\mathrm{bound}\ n\ i \in \mathrm{Bound}\ n\ (\mathrm{val}\ i)} \quad \frac{i, n \in \mathrm{Nat}}{\mathrm{tooBig}\ n\ i \in \mathrm{Bound}\ n\ (n+i)}$$

$$\text{let}\ \frac{n, i \in \mathrm{Nat}}{\mathrm{verify}\ n\ i \in \mathrm{Bound}\ n\ i}$$

$$\begin{array}{lclcl} \mathrm{verify}\ 0\ i & \mapsto & \mathrm{tooBig}\ 0\ i & & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ 0 & \mapsto & \mathrm{bound}\ n\ 0' & & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ i) & \| & \mathrm{verify}\ n\ i & & \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{s}\ (n+i)) & | & \mathrm{tooBig}\ n\ i & \mapsto & \mathrm{tooBig}\ (\mathrm{s}\ n)\ i \\ \mathrm{verify}\ (\mathrm{s}\ n)\ (\mathrm{val}\ i) & | & \mathrm{bound}\ n\ i & \mapsto & ? \end{array}$$

verify improved.

$$\text{let}\ \frac{i \in \mathsf{Fin}\ n}{\mathsf{val}\ i \in \mathsf{Nat}} \qquad \begin{array}{lcl} \mathsf{val}\ 0' & \mapsto & 0 \\ \mathsf{val}\ (\mathsf{s}'\ i) & \mapsto & \mathsf{s}\ (\mathsf{val}\ i) \end{array}$$

$$\text{data}\ \frac{n, i \in \mathsf{Nat}}{\mathsf{Bound}\ n\ i \in *}\ \text{where}\ \frac{i \in \mathsf{Fin}\ n}{\mathsf{bound}\ n\ i \in \mathsf{Bound}\ n\ (\mathsf{val}\ i)} \quad \frac{i, n \in \mathsf{Nat}}{\mathsf{tooBig}\ n\ i \in \mathsf{Bound}\ n\ (n{+}i)}$$

$$\text{let}\ \frac{n, i \in \mathsf{Nat}}{\mathsf{verify}\ n\ i \in \mathsf{Bound}\ n\ i}$$

$$\begin{array}{lclcl} \mathsf{verify}\ 0\ i & \mapsto & \mathsf{tooBig}\ 0\ i & & \\ \mathsf{verify}\ (\mathsf{s}\ n)\ 0 & \mapsto & \mathsf{bound}\ n\ 0' & & \\ \mathsf{verify}\ (\mathsf{s}\ n)\ (\mathsf{s}\ i) & \| & \mathsf{verify}\ n\ i & & \\ \mathsf{verify}\ (\mathsf{s}\ n)\ (\mathsf{s}\ (n{+}i)) & | & \mathsf{tooBig}\ n\ i & \mapsto & \mathsf{tooBig}\ (\mathsf{s}\ n)\ i \\ \mathsf{verify}\ (\mathsf{s}\ n)\ (\mathsf{val}\ i) & | & \mathsf{bound}\ n\ i & \mapsto & \mathsf{bound}\ (\mathsf{s}\ n)\ (\mathsf{s}'\ i) \end{array}$$

Definitional equality

Definitional equality

- The typing of verify depends on the equations:

$$
\begin{aligned}
0+n &\equiv n \\
(\mathrm{s}\ m)+n &\equiv \mathrm{s}\ (m+n)
\end{aligned}
$$

Definitional equality

- The typing of verify depends on the equations:

$$\begin{aligned} 0+n &\equiv n \\ (\mathrm{s}\ m)+n &\equiv \mathrm{s}\ (m+n) \end{aligned}$$

- This equations need to be true *definitionally*.

Definitional equality

- The typing of verify depends on the equations:

$$\begin{aligned} 0+n &\equiv n \\ (\mathrm{s}\ m)+n &\equiv \mathrm{s}\ (m+n) \end{aligned}$$

- This equations need to be true *definitionally*.
- If we need $n+0 = n$ we have to use propositional equality.

Propositional equality

Propositional equality

$$\text{data } \frac{\mathrm{A} \in * \quad a, b \in \mathrm{A}}{a{=}b \in *} \text{ where } \frac{a \in \mathrm{A}}{\mathrm{refl}\ a \in a{=}a}$$

Propositional equality

$$\text{data}\ \frac{\mathrm{A} \in * \quad a, b \in \mathrm{A}}{a{=}b \in *}\ \text{where}\ \frac{a \in \mathrm{A}}{\mathrm{refl}\ a \in a{=}a}$$

$$\text{let}\ \frac{n \in \mathrm{Nat}}{\mathrm{add}_0\ n \in n{+}0{=}n}$$

$$\begin{array}{llllll}
\mathrm{add}_0\ 0 & \mapsto & \mathrm{refl}\ 0 & & & \\
\mathrm{add}_0\ (\mathrm{s}\ n) & \| & n + 0 & & & \\
 & | & m & \| & \mathrm{add}_0\ n & \\
 & | & n & | & \mathrm{refl} & \mapsto \ \mathrm{refl}
\end{array}$$

Propositional equality

$$\text{data } \frac{\mathrm{A} \in * \quad a, b \in \mathrm{A}}{a{=}b \in *} \text{ where } \frac{a \in \mathrm{A}}{\mathrm{refl}\ a \in a{=}a}$$

$$\text{let } \frac{n \in \mathrm{Nat}}{\mathrm{add}_0\ n \in n{+}0{=}n}$$

$$\begin{array}{llllll}
\mathrm{add}_0\ 0 & \mapsto & \mathrm{refl}\ 0 & & & \\
\mathrm{add}_0\ (\mathrm{s}\ n) & \| & n + 0 & & & \\
 & | & m & \| & \mathrm{add}_0\ n & \\
 & | & n & | & \mathrm{refl} & \mapsto \ \mathrm{refl}
\end{array}$$

$$\text{let } \frac{q \in a{=}b \quad \mathrm{P} \in \mathrm{A} \to * \quad x \in \mathrm{P}\ a}{\mathrm{subst}\ q\ \mathrm{P}\ x \in \mathrm{P}\ b} \quad \mathrm{subst}\ (\mathrm{refl}\ a)\ \mathrm{P}\ x \mapsto x$$

Problems with $=$

Problems with $=$

- Programs cluttered with coercions.

Problems with $=$

- Programs cluttered with coercions.
- Programming requires theorem proving.

Problems with $=$

- Programs cluttered with coercions.
- Programming requires theorem proving.
- Equality on functions is not extensional, i.e.

$$\text{let}\ \frac{f, g \in \mathrm{A} \to \mathrm{B} \quad p \in \forall x \in \mathrm{A}. f\ x = g\ x}{\mathrm{ext}\ p \in f = g}$$

cannot be derived.

Solutions ?

Solutions ?

- In many cases the need for propositional equality can be avoided.

Solutions ?

- In many cases the need for propositional equality can be avoided.
- DML shows that many equalities needed in programming can be proven automatically.
  Proposal: Integrate an extensible constraint prover into the elaboration process.

Solutions ?

- In many cases the need for propositional equality can be avoided.
- DML shows that many equalities needed in programming can be proven automatically.
  Proposal: Integrate an extensible constraint prover into the elaboration process.
- The problem with extensional equality can be overcome using a different approach to equality.

Eval

Eval

- Implement an evaluator for
  a simply typed object language.

Eval

- Implement an evaluator for a simply typed object language.
- We use type-checking to avoid run-time errors.

Eval

- Implement an evaluator for
  a simply typed object language.
- We use type-checking to avoid run-time errors.
- First we implement a simply typed version.

Eval

- Implement an evaluator for
  a simply typed object language.
- We use type-checking to avoid run-time errors.
- First we implement a simply typed version.
- Then a dependently typed version,
  exploiting the *verify pattern*.

The object language

data $\frac{}{\mathrm{Val} \in *}$ where $\quad \frac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{Val}} \quad \frac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{Val}}$

The object language

data $\dfrac{}{\mathrm{Val} \in *}$ where $\qquad \dfrac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{Val}} \qquad \dfrac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{Val}}$

data $\dfrac{}{\mathrm{Tm} \in *}$ where

$$\dfrac{v \in \mathrm{Val}}{\mathrm{tval}\ v \in \mathrm{Tm}} \qquad \dfrac{t, u_0, u_1 \in \mathrm{Tm}}{\mathrm{tif}\ t\ u_0\ u_1 \in \mathrm{Tm}} \qquad \dfrac{t, u, \in \mathrm{Tm}}{\mathrm{tadd}\ t\ u \in \mathrm{Tm}}$$

Object types

data $\overline{} \atop \text{Ty} \in *$ where $\quad \dfrac{}{\text{nat} \in \text{Ty}} \quad \dfrac{}{\text{bool} \in \text{Ty}}$

Object types

data $\dfrac{}{\mathrm{Ty} \in *}$ where $\dfrac{}{\mathrm{nat} \in \mathrm{Ty}}$ $\dfrac{}{\mathrm{bool} \in \mathrm{Ty}}$

let $\dfrac{t \in \mathrm{Tm}}{\mathrm{verify}\ t \in \mathrm{Maybe}\ \ \mathrm{Ty}}$ $\ldots$

Eval — simply typed

$$\text{let}\ \frac{t \in \mathrm{Tm}}{\mathrm{eval}\ t \in \mathrm{Val}}$$

Eval — simply typed

$$\text{let}\ \frac{t \in \mathrm{Tm}}{\mathrm{eval}\ t \in \mathrm{Val}}$$

$$
\begin{array}{llllllll}
\mathrm{eval}\ (\mathrm{tval}\ v) & \mapsto & v & & & & & \\
\mathrm{eval}\ (\mathrm{tif}\ t\ u_0\ u_1) & \Vert & \mathrm{eval}\ t & & & & & \\
 & \mid & \mathrm{vnat}\ n & \mapsto & ? & & & \\
 & \mid & \mathrm{vbool\ true} & \mapsto & \mathrm{eval}\ u_0 & & & \\
 & \mid & \mathrm{vbool\ false} & \mapsto & \mathrm{eval}\ u_1 & & & \\
\mathrm{eval}\ (\mathrm{tadd}\ t\ u) & \Vert & \mathrm{eval}\ t & & & & & \\
 & \mid & \mathrm{vnat}\ m & \Vert & \mathrm{eval}\ u & & & \\
 & & & \mid & \mathrm{vnat}\ n & \mapsto & \mathrm{vnat}(m+n) & \\
 & & & \mid & \mathrm{vbool}\ b & \mapsto & ? & \\
 & \mid & \mathrm{vbool}\ b & \mapsto & ? & & &
\end{array}
$$

Safe eval ?

let $$\frac{t \in \mathrm{Tm}}{\mathrm{seval}\ t \in \mathrm{Maybe}\ \ \mathrm{Val}}$$

Safe eval ?

$$\text{let } \frac{t \in \mathrm{Tm}}{\mathrm{seval}\ t \in \mathrm{Maybe}\ \ \mathrm{Val}}$$

$$\begin{array}{llll}
\mathrm{seval}\ t & \| & \mathrm{verify}\ t & \\
 & | & \mathrm{just}\ u & \mapsto \ \mathrm{just}\ (\mathrm{eval}\ u) \\
 & | & \mathrm{nothing} & \mapsto \ \mathrm{nothing}
\end{array}$$

Safe eval ?

Safe eval ?

- We know that seval will never crash . . .

Safe eval ?

- We know that seval will never crash . . .
- . . . but the compiler doesn't!

Safe eval ?

- We know that seval will never crash . . .
- . . . but the compiler doesn't!
- seval is inefficient:
  - Values carry tags at runtime

Safe eval ?

- We know that seval will never crash . . .
- . . . but the compiler doesn't!
- seval is inefficient:
  - Values carry tags at runtime
  - eval checks the tags

Object language using dependent types

Object language using dependent types

$$\text{data}\ \frac{a \in \mathrm{Ty}}{\mathrm{TVal}\ a \in *}\ \text{where}\qquad \frac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{TVal}\ \mathrm{nat}}\qquad \frac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{TVal}\ \mathrm{bool}}$$

Object language using dependent types

$$\text{data}\ \frac{a \in \mathrm{Ty}}{\mathrm{TVal}\ a \in *}\ \text{where}\quad \frac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{TVal}\ \mathrm{nat}}\quad \frac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{TVal}\ \mathrm{bool}}$$

$$\text{data}\ \frac{a \in *}{\mathrm{TTm}\ a \in *}\ \text{where}$$

Object language using dependent types

$$\text{data}\ \frac{a \in \mathrm{Ty}}{\mathrm{TVal}\ a \in *}\ \text{where} \qquad \frac{n \in \mathrm{Nat}}{\mathrm{vnat}\ n \in \mathrm{TVal}\ \mathrm{nat}} \qquad \frac{b \in \mathrm{Bool}}{\mathrm{vbool}\ b \in \mathrm{TVal}\ \mathrm{bool}}$$

$$\text{data}\ \frac{a \in *}{\mathrm{TTm}\ a \in *}\ \text{where}$$

$$\frac{v \in \mathrm{TVal}\ a}{\mathrm{tval}\ v \in \mathrm{TTm}\ a} \qquad \frac{t \in \mathrm{TTm}\ \mathrm{bool} \quad u_0, u_1 \in \mathrm{TTm}\ a}{\mathrm{tif}\ t\ u_0\ u_1 \in \mathrm{TTm}\ a} \qquad \frac{t, u, \in \mathrm{TTm}\ \mathrm{nat}}{\mathrm{tadd}\ t\ u \in \mathrm{TTm}\ \mathrm{nat}}$$

Eval — dependently

$$\text{let}\ \frac{t \in \mathrm{TTm}\ a}{\mathrm{eval}\ t \in \mathrm{Val}\ a}$$

Eval — dependently

$$\text{let}\ \frac{t \in \mathrm{TTm}\ a}{\mathrm{eval}\ t \in \mathrm{Val}\ a}$$

$$
\begin{array}{llllll}
\mathrm{eval}\ (\mathrm{tval}\ v) & \mapsto & v & & & \\
\mathrm{eval}\ (\mathrm{tif}\ t\ u_0\ u_1) & \| & \mathrm{eval}\ t & & & \\
 & | & \mathrm{vbool\ true} & \mapsto & \mathrm{eval}\ u_0 & \\
 & | & \mathrm{vbool\ false} & \mapsto & \mathrm{eval}\ u_1 & \\
\mathrm{eval}\ (\mathrm{tadd}\ t\ u) & \| & \mathrm{eval}\ t & & & \\
 & | & \mathrm{vnat}\ m & \| & \mathrm{eval}\ u & \\
 & & & | & \mathrm{vnat}\ n & \mapsto\ \mathrm{vnat}(m+n)
\end{array}
$$

Safe eval !

Safe eval !

$$\text{let } \frac{t \in \mathrm{TTm}\ a}{\mathrm{strip}\ t \in \mathrm{Tm}} \qquad \ldots$$

Safe eval !

$$\text{let}\ \frac{t \in \mathrm{TTm}\ a}{\mathrm{strip}\ t \in \mathrm{Tm}} \qquad \ldots$$

$$\text{data}\ \frac{t \in \mathrm{Tm}}{\mathrm{Verify}\ t \in \mathrm{Ty}}\ \text{where} \quad \frac{}{\mathrm{error} \in \mathrm{Verify}\ t} \quad \frac{t \in \mathrm{TTm}\ a}{\mathrm{ok}\ t \in \mathrm{Verify}\ (\mathrm{strip}\ t)}$$

$$\text{let}\ \frac{t \in \mathrm{Tm}}{\mathrm{verify}\ t \in \mathrm{Verify}\ t} \qquad \ldots$$

$$\text{let}\ \frac{t \in \mathrm{Tm}}{\mathrm{seval}\ t \in \mathrm{Maybe}\ (\Sigma_{a \in \mathrm{Ty}} \mathrm{Val}\ a)}$$

$$\begin{array}{llll} \mathrm{seval}\ t & \| & \mathrm{verify}\ t & \\ & | & \mathrm{just}\ u & \mapsto\ \mathrm{just}\ (\mathrm{eval}\ u) \\ & | & \mathrm{error} & \mapsto\ \mathrm{nothing} \end{array}$$

Safe eval !

Safe eval !

- The compiler knows that seval will never crash.

Safe eval !

- The compiler knows that seval will never crash.
- We can generate a certificate (proof carrying code).

Safe eval !

- The compiler knows that seval will never crash.
- We can generate a certificate (proof carrying code).
- seval is efficient:

Safe eval !

- The compiler knows that seval will never crash.
- We can generate a certificate (proof carrying code).
- seval is efficient:
  - No tags at runtime.

Safe eval !

- The compiler knows that seval will never crash.
- We can generate a certificate (proof carrying code).
- seval is efficient:
  - No tags at runtime.
  - No checking.

Generic equality

Generic equality

- Implement a generic equality function for non nested, concrete data types

Generic equality

- Implement a generic equality function for non nested, concrete data types
- Usually requires a language extension (Generic Haskell)

Generic equality

- Implement a generic equality function for non nested, concrete data types
- Usually requires a language extension (Generic Haskell)
- Topic developed further in our (Conor and me) paper:
  Generic programming within dependently typed programming
  Working Conference on Generic Programming 2002

Codes and data

Codes and data

$$\text{data} \; \frac{}{\mathrm{Ty} \in *} \; \text{where}$$

Codes and data

$$\text{data } \frac{}{\mathrm{Ty} \in *} \text{ where}$$

$$\frac{}{\mathrm{unit} \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{prod}\ a\ b \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{sum}\ a\ b \in \mathrm{Ty}} \quad \frac{}{\mathrm{rec} \in \mathrm{Ty}}$$

Codes and data

data $\dfrac{}{\mathrm{Ty} \in *}$ where

$$\frac{}{\mathrm{unit} \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{prod}\ a\ b \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{sum}\ a\ b \in \mathrm{Ty}} \quad \frac{}{\mathrm{rec} \in \mathrm{Ty}}$$

data $\dfrac{r, a \in \mathrm{Ty}}{\mathrm{Val}\ r\ a \in *}$ where

Codes and data

data $\dfrac{}{\mathrm{Ty} \in *}$ where

$$\dfrac{}{\mathrm{unit} \in \mathrm{Ty}} \qquad \dfrac{a, b \in \mathrm{Ty}}{\mathrm{prod}\ a\ b \in \mathrm{Ty}} \qquad \dfrac{a, b \in \mathrm{Ty}}{\mathrm{sum}\ a\ b \in \mathrm{Ty}} \qquad \dfrac{}{\mathrm{rec} \in \mathrm{Ty}}$$

data $\dfrac{r, a \in \mathrm{Ty}}{\mathrm{Val}\ r\ a \in *}$ where

$$\dfrac{}{\mathrm{void} \in \mathrm{Val}\ r\ \mathrm{unit}} \qquad \dfrac{x \in \mathrm{Val}\ r\ a \quad y \in \mathrm{Val}\ r\ b}{\mathrm{pair}\ x\ y \in \mathrm{Val}\ r\ (\mathrm{prod}\ a\ b)}$$

Codes and data

$$\text{data} \; \frac{}{\mathrm{Ty} \in *} \; \text{where}$$

$$\frac{}{\mathrm{unit} \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{prod}\ a\ b \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{sum}\ a\ b \in \mathrm{Ty}} \quad \frac{}{\mathrm{rec} \in \mathrm{Ty}}$$

$$\text{data} \; \frac{r, a \in \mathrm{Ty}}{\mathrm{Val}\ r\ a \in *} \; \text{where}$$

$$\frac{}{\mathrm{void} \in \mathrm{Val}\ r\ \mathrm{unit}} \quad \frac{x \in \mathrm{Val}\ r\ a \quad y \in \mathrm{Val}\ r\ b}{\mathrm{pair}\ x\ y \in \mathrm{Val}\ r\ (\mathrm{prod}\ a\ b)}$$

$$\frac{x \in \mathrm{Val}\ a}{\mathrm{inl}\ x \in \mathrm{Val}\ (\mathrm{sum}\ a\ b)} \quad \frac{y \in \mathrm{Val}\ b}{\mathrm{inr}\ y \in \mathrm{Val}\ (\mathrm{sum}\ a\ b)}$$

Codes and data

$$\text{data}\ \frac{}{\mathrm{Ty} \in *}\ \text{where}$$

$$\frac{}{\mathrm{unit} \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{prod}\ a\ b \in \mathrm{Ty}} \quad \frac{a, b \in \mathrm{Ty}}{\mathrm{sum}\ a\ b \in \mathrm{Ty}} \quad \frac{}{\mathrm{rec} \in \mathrm{Ty}}$$

$$\text{data}\ \frac{r, a \in \mathrm{Ty}}{\mathrm{Val}\ r\ a \in *}\ \text{where}$$

$$\frac{}{\mathrm{void} \in \mathrm{Val}\ r\ \mathrm{unit}} \quad \frac{x \in \mathrm{Val}\ r\ a \quad y \in \mathrm{Val}\ r\ b}{\mathrm{pair}\ x\ y \in \mathrm{Val}\ r\ (\mathrm{prod}\ a\ b)}$$

$$\frac{x \in \mathrm{Val}\ a}{\mathrm{inl}\ x \in \mathrm{Val}\ (\mathrm{sum}\ a\ b)} \quad \frac{y \in \mathrm{Val}\ b}{\mathrm{inr}\ y \in \mathrm{Val}\ (\mathrm{sum}\ a\ b)}$$

$$\frac{x \in \mathrm{Val}\ r\ r}{\mathrm{in}\ x \in \mathrm{Val}\ r\ r}$$

Example

Example

$$\text{let } \frac{a \in \mathrm{Ty}}{\mathrm{Data}\ a \in \mathrm{Ty}} \quad \mathrm{Data}\ a \mapsto \mathrm{Val}\ a\ a$$

Example

$$\text{let } \frac{a \in \mathrm{Ty}}{\mathrm{Data}\ a \in \mathrm{Ty}} \quad \mathrm{Data}\ a \mapsto \mathrm{Val}\ a\ a$$

$$\text{let } \frac{}{\mathrm{nat} \in \mathrm{Ty}} \quad \mathrm{nat} \mapsto \mathrm{sum\ unit\ rec}$$

Example

$$\text{let}\ \frac{a \in \mathrm{Ty}}{\mathrm{Data}\ a \in \mathrm{Ty}} \quad \mathrm{Data}\ a \mapsto \mathrm{Val}\ a\ a$$

$$\text{let}\ \frac{}{\mathrm{nat} \in \mathrm{Ty}} \quad \mathrm{nat} \mapsto \mathrm{sum}\ \mathrm{unit}\ \mathrm{rec}$$

$$\text{let}\ \frac{}{\mathrm{zero} \in \mathrm{Data}\ \mathrm{nat}} \quad \mathrm{zero} \mapsto \mathrm{inl}\ \mathrm{void}$$

Example

$$\text{let } \frac{a \in \mathrm{Ty}}{\mathrm{Data}\ a \in \mathrm{Ty}} \quad \mathrm{Data}\ a \mapsto \mathrm{Val}\ a\ a$$

$$\text{let } \frac{}{\mathrm{nat} \in \mathrm{Ty}} \quad \mathrm{nat} \mapsto \mathrm{sum\ unit\ rec}$$

$$\text{let } \frac{}{\mathrm{zero} \in \mathrm{Data\ nat}} \quad \mathrm{zero} \mapsto \mathrm{inl\ void}$$

$$\text{let } \frac{n \in \mathrm{Data\ nat}}{\mathrm{succ}\ n \in \mathrm{Data\ nat}} \quad \mathrm{succ}\ n \mapsto \mathrm{inr}\ (\mathrm{in}\ n)$$

Generic equality

Generic equality

$$\text{let } \frac{x, y \in \mathrm{Val}\ r\ t}{\mathrm{eq}\ x\ y \in \mathrm{Bool}}$$

Generic equality

$$\text{let } \frac{x, y \in \mathrm{Val}\ r\ t}{\mathrm{eq}\ x\ y \in \mathrm{Bool}}$$

$$\begin{array}{lllll}
\mathrm{eq} & \mathrm{void} & \mathrm{void} & \mapsto & \mathrm{true} \\
\mathrm{eq} & (\mathrm{pair}\ x\ y) & (\mathrm{pair}\ x'\ y') & \mapsto & (\mathrm{eq}\ x\ x')\ \&\&\ (\mathrm{eq}\ y\ y') \\
\mathrm{eq} & (\mathrm{inl}\ x) & (\mathrm{inl}\ x') & \mapsto & \mathrm{eq}\ x\ x' \\
\mathrm{eq} & (\mathrm{inl}\ x) & (\mathrm{inr}\ y) & \mapsto & \mathrm{false} \\
\mathrm{eq} & (\mathrm{inr}\ y) & (\mathrm{inl}\ x) & \mapsto & \mathrm{false} \\
\mathrm{eq} & (\mathrm{inr}\ y) & (\mathrm{inr}\ y') & \mapsto & \mathrm{eq}\ y\ y' \\
\mathrm{eq} & (\mathrm{in}\ x) & (\mathrm{in}\ x') & \mapsto & \mathrm{eq}\ x\ x'
\end{array}$$

Advantages of Dependently Typed Programming

Advantages of Dependently Typed Programming

- Avoidance of run-time errors

Advantages of Dependently Typed Programming

- Avoidance of run-time errors
- More efficient code (elimination of tags)

Advantages of Dependently Typed Programming

- Avoidance of run-time errors
- More efficient code (elimination of tags)
- Extensions of Type System as library

Advantages of Dependently Typed Programming

- Avoidance of run-time errors
- More efficient code (elimination of tags)
- Extensions of Type System as library
- Easier to reason about

Important issues

Important issues

- Definitional equality should be well behaved

Important issues

- Definitional equality should be well behaved
- Inductive families have to be supported

Important issues

- Definitional equality should be well behaved
- Inductive families have to be supported
- Type inference is generalized by elaboration
  Extensible elaboration ?

Important issues

- Definitional equality should be well behaved
- Inductive families have to be supported
- Type inference is generalized by elaboration
  Extensible elaboration ?
- Programs are constructed interactively,
  starting with the type as a partial specification