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
The established social order

terms

:::

types

Why Dependent Types Matter – p.2/32
The established social order

terms: do all the work

types:
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

types are never around when there is work to be done

generate in criminal activity
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

```
Matrix a = Matrix' Nil a
Matrix' t a = Zero (t (t a)) | Succ (Cons t) a
Nil a = Nil
Cons t a = Cons a (t a)
```
Ersatz dependent types

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

\[
\begin{align*}
\text{Matrix } a & = \text{Matrix'} \text{ Nil } a \\
\text{Matrix'} \ t \ a & = \text{Zero } (t \ (t \ a)) \mid \text{Succ } (\text{Cons } t) \ a \\
\text{Nil } a & = \text{Nil} \\
\text{Cons } t \ a & = \text{Cons } a \ (t \ a)
\end{align*}
\]

- 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 used to represent square matrices

\[
\begin{align*}
\text{Matrix } a &= \text{Matrix’ } \text{Nil } a \\
\text{Matrix’ } t \ a &= \text{Zero } (t (t \ a)) \mid \text{Succ } (\text{Cons } t) \ a \\
\text{Nil } a &= \text{Nil} \\
\text{Cons } t \ a &= \text{Cons } a (t \ a)
\end{align*}
\]

- 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

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

Illustrating patterns in DTP

verify, reflect

Emphasis on safe and efficient execution

Using epigram currently developed by Conor, using ideas from LEGO / OLEG
Overview

Examples for programming with dependent types:
Overview

Examples for programming with dependent types:

- \texttt{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
Examples for programming with dependent types:

- `nth` – safe access to lists/vectors
- `eval` – safe eval
- `eq` – generic equality

Illustrating patterns in DTP
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:
- \texttt{nth} – safe access to lists/vectors
- \texttt{eval} – safe eval
- \texttt{eq} – generic equality

Illustrating patterns in DTP
\texttt{verify \ nth,eval}
\texttt{reflect \ eq}

Emphasis on safe and efficient execution

Using \texttt{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:
  - \texttt{nth} – safe access to lists/vectors
  - \texttt{eval} – safe eval
  - \texttt{eq} – generic equality

- Illustrating patterns in DTP
  - \texttt{verify} \texttt{nth,eval}
  - \texttt{reflect} \texttt{eq}

- Emphasis on safe and efficient execution

- Using \texttt{epigram} currently developed by Conor, using ideas from
  - LEGO / OLEG
  - ALF
\[ \text{nth} \quad \text{— no dependent types} \]

\[
\begin{align*}
\text{let} & \quad A \in \ast \\
& \quad l \in \text{List } A \\
& \quad n \in \text{Nat} \\
\implies & \quad \text{nth } A \; l \; n \in A
\end{align*}
\]

\[ \text{nth } A \; l \; n \leftrightarrow ? \]
nth — no dependent types

\[
\begin{align*}
\text{let} & \quad l \in \text{List } A \quad n \in \text{Nat} \\
\text{nth} & \quad l \ n \in A
\end{align*}
\]

nth \quad l \ n \rightarrow ?

Hindley-Milner: Type quantification and application can be made implicit.
nth — no dependent types

\[
\begin{array}{c}
\text{let} \\
\quad l \in \text{List} \ A \quad n \in \text{Nat} \\
\quad \text{nth} \quad l \ n \in A
\end{array}
\]

\[
\begin{align*}
\text{nth \ nil \ } n & \quad \mapsto \quad ? \\
\text{nth \ (cons \ a \ as) \ } n & \quad \mapsto \quad ?
\end{align*}
\]

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information
\textbf{nth} — no dependent types

\[
\text{let } l \in \text{List } A \quad n \in \text{Nat} \\
\therefore \text{nth } l \ n \in A
\]

\begin{align*}
\text{nth \ nil \ n} & \mapsto \ ? \\
\text{nth \ (cons \ a \ as) \ 0} & \mapsto \ ? \\
\text{nth \ (cons \ a \ as) \ (s \ n)} & \mapsto \ ?
\end{align*}

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

Split left hand sides using type information
nth — no dependent types

let \( l \in \text{List } A \quad n \in \text{Nat} \)

\[
\begin{align*}
\text{nth} \quad l \cdot n & \in A \\
\text{nth} \quad \text{nil} \cdot n & \mapsto \text{?} \\
\text{nth} \quad (\text{cons } a \cdot a s) \cdot 0 & \mapsto a \\
\text{nth} \quad (\text{cons } a \cdot a s) \cdot (s \cdot n) & \mapsto \text{nth} \cdot a s \cdot n
\end{align*}
\]

- Hindley-Milner: Type quantification and application can be made implicit.
- Split left hand sides using type information
The function \( \text{nth} \) is not good

<table>
<thead>
<tr>
<th>( \text{let } l \in \text{List } A \quad n \in \text{Nat} )</th>
<th>( \text{nth } l \ n \in A )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \text{nth nil } n )</td>
<td>( \mapsto ? )</td>
</tr>
<tr>
<td>( \text{nth (cons } a \ a s) \ 0 )</td>
<td>( \mapsto a )</td>
</tr>
<tr>
<td>( \text{nth (cons } a \ a s) \ (s \ n) )</td>
<td>( \mapsto \text{nth } a s \ n )</td>
</tr>
</tbody>
</table>
The function \( \text{nth} \) is partial

\[
\text{nth } \emptyset \ n \ \cdot \ \cdot \\
\text{nth} \ (\text{cons} \ a \ as) \ 0 \ \rightarrow \ a \\
\text{nth} \ (\text{cons} \ a \ as) \ (\text{succ} \ n) \ \rightarrow \ \text{nth} \ as \ n
\]

leads to a runtime error.
**nth** is not good

\[
\begin{align*}
\text{let } & l \in \text{List A} \quad n \in \text{Nat} \\
\text{nth } & l \quad n \in \text{A} \\
\text{nth nil } & n \quad \mapsto ? \\
\text{nth (cons a as) } & 0 \quad \mapsto a \\
\text{nth (cons a as) } & (s \ n) \quad \mapsto \text{nth as n}
\end{align*}
\]

- 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 \text{Nat} \in \ast \quad \text{where} \quad \begin{align*} \text{0} \in \text{Nat} \\ \text{s} \, n \in \text{Nat} \end{align*}

data A \in \ast \quad \text{where} \quad \begin{align*} \text{List} \, A \in \ast \\ \text{nil}_A \in \text{List} \, A \\ \text{cons}_A \, a \, a \, s \in \text{List} \, A \end{align*}
data types

data \text{Nat} \in * \quad \text{where} \quad 0 \in \text{Nat} \quad n \in \text{Nat} \\
\quad s\ n \in \text{Nat}

data A \in * \quad \text{where} \\
\text{List A} \in * \\
nil \in \text{List A} \quad a \in A \quad a\ s \in \text{List A} \\
\text{cons} \ a\ a\ s \in \text{List A}
Better data types, better $n^{th}$
Better data types, better $nth$

\[
\begin{align*}
\text{data } & n \in \text{Nat} \\
\text{where } & \text{Fin } n \in * \\
\text{let } & 0' n \in \text{Fin } (s n) \\
& s' n i \in \text{Fin } (s n)
\end{align*}
\]
Better data types, better $nth$

\[
\begin{align*}
\text{data } & n \in \text{Nat} \quad \text{where} \quad n \in \text{Nat} \quad i \in \text{Fin} \ n \\
\text{Fin } n \in * & 0'_n \in \text{Fin} \ (s \ n) \quad s'_n \ i \in \text{Fin} \ (s \ n)
\end{align*}
\]

\[
\begin{align*}
\text{data } & A \in * \ n \in \text{Nat} \quad \text{where} \quad n \in \text{Nat} \\
\text{Vec } A \ n \in * & \text{vnil } \in \text{Vec } A \ 0 \\
& \text{vcons}_n \ a \ a s \in \text{Vec } A \ (s \ n)
\end{align*}
\]
Better data types, better \textit{nth}

\texttt{data} \quad n \in \text{Nat} \quad \text{where} \quad \texttt{Fin} \ n \in \ast
\quad \begin{array}{l}
  0' \in \text{Fin} \ (s \ n) \\
  s' \ i \in \text{Fin} \ (s \ n)
\end{array}

\texttt{data} \quad A \in \ast \quad n \in \text{Nat} \quad \text{where} \quad \texttt{Vec} \ A \ n \in \ast
\quad \begin{array}{l}
  \text{vnil} \in \text{Vec} \ A \ 0 \\
  \text{vcons} \ a \ as \in \text{Vec} \ A \ (s \ n)
\end{array}
Better data types, better \texttt{nth}

\[
\begin{align*}
data \ & \begin{array}{c} \text{where} \ \n \in \text{Nat} \\
\text{Fin} \ n \in * \\
\end{array} \\
& \begin{array}{c} 0' \in \text{Fin} \ (s \ n) \\
\end{array} \\
& \begin{array}{c} s' \ i \in \text{Fin} \ (s \ n) \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
data \ & \begin{array}{c} A \in * \ n \in \text{Nat} \\
\text{Vec} \ A \ n \in * \\
\end{array} \ \\
& \begin{array}{c} \text{where} \ \text{vnil} \in \text{Vec} \ A \ 0 \\
\end{array} \\
& \begin{array}{c} \text{vcons} \ a \ as \in \text{Vec} \ A \ (s \ n) \\
\end{array}
\end{align*}
\]

\[
\begin{align*}
\text{let} \ & \begin{array}{c} n \in \text{Nat} \ l \in \text{Vec} \ A \ n \\
\end{array} \ \\
\text{refl} \ & \begin{array}{c} i \in \text{Fin} \ n \\
\end{array} \\
\end{align*}
\]

\[
\begin{align*}
& \begin{array}{c} \text{nth} \ n \ l \ i \in A \\
\end{array}
\end{align*}
\]
Better data types, better \texttt{nth}

\[
\begin{align*}
\text{data } \text{Fin } n & \in * \quad \text{where} \quad 0' \in \text{Fin } (s\ n) \quad s' \ i \in \text{Fin } (s\ n) \\
\text{data } \text{Vec } A \ n & \in * \quad \text{where} \quad \text{vnil} \in \text{Vec } A \ 0 \quad \text{vcons} \ a \ as \in \text{Vec } A \ (s\ n) \\
\text{let } \text{nth} & \quad l \ i \in A
\end{align*}
\]
Better data types, better \texttt{nth}

\begin{align*}
data \ & \text{where} \quad n \in \text{Nat} & n \in \text{Nat} & i \in \text{Fin} \ n \\
\text{Fin} \ n \in * & 0' \in \text{Fin} \ (s \ n) & s' \ i \in \text{Fin} \ (s \ n) \\
A \in * & n \in \text{Nat} & a \in A \quad as \in \text{Vec} \ A \ n \\
data \ & \text{where} \quad \text{Vec} \ A \ n \in * & \text{vnil} \in \text{Vec} \ A \ 0 & \text{vcons} \ a \ as \in \text{Vec} \ A \ (s \ n) \\
& \quad l \in \text{Vec} \ A \ n \quad i \in \text{Fin} \ n \\
\text{let} \ & \quad \text{nth} \ l \ i \in A \\
\text{nth} \ l \ n & \mapsto \ ?
\end{align*}
Better data types, better \texttt{nth}

\[
\begin{align*}
\text{data} & \quad n \in \text{Nat} \\
\text{Fin} \ n & \in * \\
\text{where} & \quad 0' \in \text{Fin} \ (s \ n) \\
& \quad s' \ i \in \text{Fin} \ (s \ n) \\
\text{data} & \quad A \in * \\
\text{Vec} \ A \ n & \in * \\
\text{where} & \quad \text{vnil} \in \text{Vec} \ A \ 0 \\
& \quad \text{vcons} \ a \ as \in \text{Vec} \ A \ (s \ n) \\
\text{let} & \quad l \in \text{Vec} \ A \ n \\
& \quad i \in \text{Fin} \ n \\
\text{let} & \quad \text{nth} \ n i \in A \\
\text{nth} \ l \ 0' & \mapsto ? \\
\text{nth} \ l \ (s' \ i) & \mapsto ? 
\end{align*}
\]
Better data types, better \texttt{nth}

\[
data \quad \begin{align*}
\text{Fin } n & \in \ast \\
\text{let } & n \in \text{Nat} \\
\text{vec } & A \ n \in \ast \\
\text{where } & 0' \in \text{Fin } (s \ n) \\
\text{where } & s' i \in \text{Fin } (s \ n) \\
\text{vec } & A \ n \in \ast \\
\text{let } & l \in \text{Vec } A \ n \\
\text{where } & v\text{nil } \in \text{Vec } A \ 0 \\
\text{where } & v\text{cons } a \ as \in \text{Vec } A \ (s \ n) \\
\text{let } & i \in \text{Fin } n \\
\text{let } & n\text{th } l \ i \in A
\end{align*}
\]

\texttt{nth } (v\text{cons } a \ as) \ 0' \mapsto ?

\texttt{nth } l \ (s' i) \mapsto ?\]
Better data types, better \texttt{nth}

\[
\begin{align*}
\text{data } & \quad \text{where } \quad \text{data } & \quad \text{where } \\
\text{Fin } n \in * & \quad n \in \text{Nat} & \quad i \in \text{Fin } n \\
\text{where } & & \quad 0' \in \text{Fin } (s \ n) \\
\text{A } \in * & \quad n \in \text{Nat} & \quad s' \ i \in \text{Fin } (s \ n) \\
\text{Vec } A \ n \in * & \quad \text{where } & \quad a \in A & \quad \text{as } \in \text{Vec } A \ n \\
\text{let } & & \quad \text{vnil } \in \text{Vec } A \ 0 \\
\text{nth } l \ i \in A & \quad \text{vcons } a \ \text{as } \in \text{Vec } A \ (s \ n) \\
\text{nth } (\text{vcons } a \ \text{as}) \ 0' & \quad \leftrightarrow ? \\
\text{nth } (\text{vcons } a \ \text{as}) \ (s' \ i) & \quad \leftrightarrow ?
\end{align*}
\]
Better data types, better `nth`

\[
\begin{align*}
\text{data } \text{Fin } n & \in * \\
\text{where } & \\
\text{Fin } n & \in * \\
0' & \in \text{Fin } (s \ n) \\
s' i & \in \text{Fin } (s \ n)
\end{align*}
\]

\[
\begin{align*}
\text{data } \text{Vec } A n & \in * \\
\text{where } & \\
\text{Vec } A n & \in * \\
v\text{nil} & \in \text{Vec } A 0 \\
v\text{cons } a as & \in \text{Vec } A (s \ n)
\end{align*}
\]

\[
\begin{align*}
\text{let } & \\
\text{nth } l i & \in A \\
nth (v\text{cons } a as) 0' & \Rightarrow a \\
nth (v\text{cons } a as) (s' i) & \Rightarrow nth as i
\end{align*}
\]

\[\text{nth is a total function.}\]
Better data types, better \( \text{nth} \)

\[
\begin{align*}
\text{data} & \quad \text{where} \\
\text{Fin} \ n & \in * \\
\text{data} & \quad \text{where} \\
\text{Vec} \ A \ n & \in *
\end{align*}
\]

\[
\begin{align*}
\begin{array}{c}
\text{n} \in \text{Nat} \\
0' \in \text{Fin} \ (s \ n) \\
s' \ i \in \text{Fin} \ (s \ n)
\end{array} & \quad \begin{array}{c}
\text{a} \in \text{A} \\
\text{as} \in \text{Vec} \ A \ n
\end{array} & \quad \begin{array}{c}
\text{vnil} \in \text{Vec} \ A \ 0 \\
\text{vcons} \ a \ as \in \text{Vec} \ A \ (s \ n)
\end{array}
\end{align*}
\]

\[
\begin{align*}
l & \in \text{Vec} \ A \ n \\
i & \in \text{Fin} \ n
\end{align*}
\]

\[
\begin{align*}
\text{let} & \\
\text{nth} & \quad l \ i \in A
\end{align*}
\]

\[
\begin{align*}
\text{nth} \ (\text{vcons} \ a \ as) \ 0' \rightarrow a \\
\text{nth} \ (\text{vcons} \ a \ as) \ (s' \ i) \rightarrow \text{nth} \ as \ i
\end{align*}
\]

\( \text{nth} \) is a total function.

\( \text{nth} \ 3 \ [1, 2] \) is not well-typed.
Verify
How can we use \texttt{nth} on lists of unknown length user input,…?
How can we use `nth` on lists of unknown length user input, ...?

\[
\text{let } \frac{n, i \in \text{Nat}}{\text{verify } n \ i \in \text{Maybe} \ (\text{Fin} \ n)}
\]
How can we use \texttt{nth} on lists of unknown length user input, \ldots?

\[
\text{let } n, i \in \text{Nat} \quad \Rightarrow \quad \text{verify } n \ i \in \text{Maybe} (\text{Fin } n)
\]

\[
\text{verify } n \ i \quad \mapsto \quad ?
\]
How can we use \texttt{nth} on lists of unknown length user input, \ldots? 

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

\[
\text{verify } 0 \ i \quad \mapsto \quad ?
\]

\[
\text{verify } (s \ n) \ i \quad \mapsto \quad ?
\]
How can we use \texttt{nth} on lists of unknown length user input, \ldots?

\[
\begin{align*}
\text{let } & n, i \in \text{Nat} \\
\text{verify } & n \ i \in \text{Maybe (Fin } n) \\
\text{verify } 0 \ i & \mapsto \text{nothing} \\
\text{verify } (s \ n) \ i & \mapsto \ ?
\end{align*}
\]
How can we use \texttt{nth} on lists of unknown length user input, \ldots?

\[
\text{let } n, i \in \text{Nat} \\
\text{verify } n \cdot i \in \text{Maybe} (\text{Fin } n)
\]

\[
\begin{align*}
\text{verify } 0 \cdot i & \mapsto \text{nothing} \\
\text{verify } (s \ n) \ 0 & \mapsto ? \\
\text{verify } (s \ n) \ (s \ i) & \mapsto ?
\end{align*}
\]
How can we use `nth` on lists of unknown length user input, \ldots?

\[
\text{let } n, i \in \text{Nat} \\
\text{verify } n \; i \in \text{Maybe} \; (\text{Fin } n)
\]

\[
\begin{align*}
\text{verify } 0 \; i & \mapsto \text{nothing} \\
\text{verify } (s \; n) \; 0 & \mapsto \text{just } 0' \\
\text{verify } (s \; n) \; (s \; i) & \mapsto ?
\end{align*}
\]
How can we use \texttt{nth} on lists of unknown length user input,\ldots ?

\[
\begin{align*}
\text{let} & \quad n, i \in \text{Nat} \\
\text{verify} & \quad n \ i \in \text{Maybe} \ (\text{Fin} \ n)
\end{align*}
\]

\[
\begin{align*}
\text{verify} \ 0 \ i & \quad \mapsto \quad \text{nothing} \\
\text{verify} \ (\text{\texttt{s}} \ n) \ 0 & \quad \mapsto \quad \text{just} \ 0' \\
\text{verify} \ (\text{\texttt{s}} \ n) \ (\text{\texttt{s}} \ i) & \quad \parallel \quad \text{verify} \ n \ i \quad \mapsto \quad ?
\end{align*}
\]
How can we use \texttt{nth} on lists of unknown length user input, \ldots?

\[
\begin{align*}
\text{let} & \quad n, i \in \text{Nat} \\
\text{verify} & \quad n \ i \in \text{Maybe} \ (\text{Fin} \ n)
\end{align*}
\]

\[
\begin{align*}
\text{verify} \ 0 \ i & \quad \mapsto \quad \text{nothing} \\
\text{verify} \ (s \ n) \ 0 & \quad \mapsto \quad \text{just} \ 0' \\
\text{verify} \ (s \ n) \ (s \ i) & \quad \| \quad \text{verify} \ n \ i \\
& \quad | \quad \text{nothing} \quad \mapsto \quad ? \\
& \quad | \quad \text{just} \ i \quad \mapsto \quad ?
\end{align*}
\]
Verify

How can we use \texttt{nth} on lists of unknown length user input, \ldots?

\[
\text{let } \quad n, i \in \textbf{Nat} \\
\text{verify } n \ i \in \textbf{Maybe} \ (\textbf{Fin} \ n)
\]

\[
\begin{align*}
\text{verify } 0 \ i & \quad \mapsto \quad \text{nothing} \\
\text{verify } (s \ n) \ 0 & \quad \mapsto \quad \text{just } 0'
\end{align*}
\]

\[
\begin{align*}
\text{verify } (s \ n) \ (s \ i) \ & \quad || \quad \text{verify } n \ i \\
| \quad \text{nothing} & \quad \mapsto \quad \text{nothing} \\
| \quad \text{just } i & \quad \mapsto \quad ?
\end{align*}
\]
How can we use \texttt{nth} on lists of unknown length user input, \ldots?

\begin{align*}
\text{let} \quad & n, i \in \mathbb{Nat} \\
\text{verify} \quad & n \ i \in \text{Maybe} \ (\text{Fin} \ n)
\end{align*}

\begin{align*}
\text{verify} \ 0 \ i & \quad \mapsto \quad \text{nothing} \\
\text{verify} \ (s \ n) \ 0 & \quad \mapsto \quad \text{just} \ 0' \\
\text{verify} \ (s \ n) \ (s \ i) \ || \ & \text{verify} \ n \ i \\
& | \quad \text{nothing} \quad \mapsto \quad \text{nothing} \\
& | \quad \text{just} \ i \quad \mapsto \quad \text{just} \ (s' \ i)
\end{align*}
Going further
Going further

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

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

- The type of `verify` is not informative enough for some of its potential applications.
- How is `just j ≡ verify n i` related to `j`?
- When does `verify` return `nothing`?
prove improved.

\[
\begin{align*}
  \text{let } i \in \text{Fin } n & \quad \text{val } 0' \quad \leftrightarrow \quad 0 \\
  \text{val } i \in \text{Nat} & \quad \text{val } (s' \ i) \quad \leftrightarrow \quad s \ (\text{val } i)
\end{align*}
\]
\[
\begin{align*}
& i \in \text{Fin } n \quad \text{val } 0' \quad \mapsto \quad 0 \\
\text{let } & \quad \text{val } i \in \text{Nat} \quad \text{val } (s' \ i) \quad \mapsto \quad s \ (\text{val } i) \\
& n, i \in \text{Nat} \\
\text{data } & \quad \text{where } \\
\text{Bound } & \quad n \ i \in * \\
& i \in \text{Fin } n \quad \text{bound } n \ i \in \text{Bound } n \ (\text{val } i) \\
i, n \in \text{Nat} & \quad \text{tooBig } n \ i \in \text{Bound } n \ (n+i)
\end{align*}
\]

\textbf{verify improved.}
verify improved.

\[
\begin{align*}
\text{let } i \in \text{Fin } n & \quad \text{val } 0' \quad \mapsto \quad 0 \\
\text{val } i \in \text{Nat} & \quad \text{val } (s' \ i) \quad \mapsto \quad s \ (\text{val } i)
\end{align*}
\]

\[
\begin{align*}
\text{data } n, i \in \text{Nat} & \quad \text{where}
\\
\text{Bound } n \ i \in * & \quad \text{where}
\\
\text{bound } n \ i \in \text{Bound } n \ (\text{val } i) & \quad \text{tooBig } n \ i \in \text{Bound } n \ (n+i)
\end{align*}
\]

\[
\begin{align*}
\text{let } n, i \in \text{Nat} & \\
\text{verify } n \ i \in \text{Bound } n \ i
\end{align*}
\]
verify improved.

\[
\begin{align*}
&i \in \text{Fin } n \quad \text{val } 0' \quad \mapsto \quad 0 \\
&\text{let } \quad \text{val } i \in \text{Nat} \quad \text{val } (s' \ i) \quad \mapsto \quad s \ (\text{val } i)
\end{align*}
\]

\[
\begin{align*}
&n, i \in \text{Nat} \\
&\text{data} \quad \text{where} \\
&\text{Bound } n \ i \in \ast \\
&\text{let } \\
&\text{verify } n \ i \in \text{Bound } n \ i
\end{align*}
\]

\[
\begin{align*}
&i \in \text{Fin } n \\
&\text{bound } n \ i \in \text{Bound } n \ (\text{val } i) \\
&i, n \in \text{Nat} \\
&\text{tooBig } n \ i \in \text{Bound } n \ (n+i)
\end{align*}
\]

\[
\begin{align*}
&n, i \in \text{Nat} \\
&\text{let } \\
&\text{verify } n \ i \in \text{Bound } n \ i
\end{align*}
\]

\[
\begin{align*}
&\text{verify } n \ i \\
&\mapsto \quad ?
\end{align*}
\]
```
let
val i ∈ Nat
  val 0'  ↦  0
  val (s' i)  ↦  s (val i)

data
  Bound n i ∈ *
  where
    n, i ∈ Nat
    bound n i ∈ Bound n (val i)
    tooBig n i ∈ Bound n (n+i)

let
    n, i ∈ Nat
    verify n i ∈ Bound n i

verify 0 i  ↦  ?
verify (s n) i  ↦  ?
```
verify improved.

\[
\begin{align*}
    i & \in \text{Fin } n & \text{val } 0' & \mapsto 0 \\
    \text{let } & val i \in \text{Nat} & \text{val } (s' i) & \mapsto s (\text{val } i) \\
    n, i & \in \text{Nat} & \text{data } & \text{where } & \text{bound } n i & \in \text{Bound } n (\text{val } i) \\
    & & & & \text{tooBig } n i & \in \text{Bound } n (n+i) \\
    \text{let } & n, i \in \text{Nat} & & & & \text{verify } n i & \in \text{Bound } n i \\
    \text{verify } 0 & i & \mapsto & \text{tooBig } 0 & i \\
    \text{verify } (s & n) i & \mapsto & ? 
\end{align*}
\]
verify improved.

\[
\begin{aligned}
\text{let } & \quad i \in \text{ Fin } n \quad \text{val } 0' \quad \mapsto \quad 0 \\
\text{val } i \in \text{ Nat } \quad & \quad \text{val } (s' i) \quad \mapsto \quad s \text{ (val } i) \\
\end{aligned}
\]

\[
\begin{aligned}
\text{data } & \quad n, i \in \text{ Nat } \\
\text{Bound } n i \in \ast & \quad \text{where } \\
\text{where } \quad \text{bound } n i \in \text{ Bound } n \text{ (val } i) \\
\text{tooBig } n i \in \text{ Bound } n \text{ (n+i)} \\
& \quad \text{let } \\
& \quad \text{verify } n i \in \text{ Bound } n i \\
& \quad \text{verify } 0 i \quad \mapsto \quad \text{tooBig } 0 i \\
& \quad \text{verify } (s n) 0 \quad \mapsto \quad ? \\
& \quad \text{verify } (s n) (s i) \quad \mapsto \quad ? \\
\end{aligned}
\]
\texttt{verify} improved.

\begin{align*}
\text{let} & \quad i \in \text{Fin } n \\
\text{val } & \quad \text{0}' \quad \mapsto \quad 0 \\
\text{val } & \quad (s' \ i) \quad \mapsto \quad s \ (\text{val } i)
\end{align*}

\begin{align*}
data & \quad n, \ i \in \text{Nat} \\
\text{where} & \quad i \in \text{Fin } n \\
\text{bound } & \quad n \ i \in \text{Bound } n \ (\text{val } i) \\
\text{tooBig } & \quad n \ i \in \text{Bound } n \ (n+i)
\end{align*}

\begin{align*}
\text{let} & \quad n, \ i \in \text{Nat} \\
\text{verify } & \quad n \ i \in \text{Bound } n \ i \\
\text{verify } & \quad 0 \ i \quad \mapsto \quad \text{tooBig } 0 \ i \\
\text{verify } & \quad (s \ n) \ 0 \quad \mapsto \quad \text{bound } n \ 0' \\
\text{verify } & \quad (s \ n) \ (s \ i) \quad \mapsto \quad ?
\end{align*}
Verify improved.

\[
\begin{align*}
\text{let } & \quad i \in \text{Fin } n \quad \text{val } 0' \quad \rightarrow \quad 0 \\
\text{val } i \in \text{Nat} & \quad \text{val } (s' \ i) \quad \rightarrow \quad s \ (\text{val } i)
\end{align*}
\]

\[
\begin{align*}
\text{data } & \quad n, \ i \in \text{Nat} \quad \text{where} \\
\text{Bound } n \ i \in \ast & \quad \text{where} \\
\text{bound } n \ i \in \text{Bound } n \ (\text{val } i) & \quad \text{tooBig } n \ i \in \text{Bound } n \ (n+i)
\end{align*}
\]

\[
\begin{align*}
\text{let } & \quad n, \ i \in \text{Nat} \\
\text{verify } n \ i \in \text{Bound } n \ i & \\
\text{verify } 0 \ i & \quad \rightarrow \quad \text{tooBig } 0 \ i \\
\text{verify } (s \ n) \ 0 & \quad \rightarrow \quad \text{bound } n \ 0' \\
\text{verify } (s \ n) \ (s' \ i) & \quad \mid \quad \text{verify } n \ i \quad \rightarrow \quad ?
\end{align*}
\]
let \( i \in \text{Fin } n \)  
\[
\begin{align*}
\text{val } 0' & \iff 0 \\
\text{val } (s' i) & \iff s \text{ (val } i) \\
\end{align*}
\]

\[
\begin{aligned}
data n, i \in \text{Nat} & \quad \text{where } \\
\text{Bound } n i \in * & \\
\text{let } n, i \in \text{Nat} & \quad \text{where } \\
\text{bound } n i \in \text{Bound } n \text{ (val } i) & \\
\text{verify } n i \in \text{Bound } n i & \\
\end{aligned}
\]

\[
\begin{aligned}
\text{verify } 0 i & \iff \text{tooBig } 0 i \\
\text{verify } (s n) 0 & \iff \text{bound } n 0' \\
\text{verify } (s n) (s i) & \iff \text{verify } n i \\
\text{verify } (s n) (s (n+i)) & \iff \text{tooBig } n i \\
\text{verify } (s n) \text{ (val } i) & \iff \text{bound } n i \\
\end{aligned}
\]
\begin{align*}
\text{let } & \quad i \in \text{Fin } n \quad \text{val } 0' \quad \mapsto \quad 0 \\
\text{val } i \in \text{Nat} & \quad \text{val } (s' i) \quad \mapsto \quad s \times (\text{val } i) \\
n, \ i \in \text{Nat} & \quad \text{data } \quad \text{Bound } n \ i \in \ast \\
\text{let } & \quad \text{where } \quad \text{bound } n \ i \in \text{Bound } n \ (\text{val } i) \\
\text{tooBig } n \ i \in \text{Bound } n \ (n+i) \\
n, \ i \in \text{Nat} & \quad \text{verify } n \ i \in \text{Bound } n \ i \\
\text{verify } 0 \ i & \quad \mapsto \quad \text{tooBig } 0 \ i \\
\text{verify } (s \ n) \ 0 & \quad \mapsto \quad \text{bound } n \ 0' \\
\text{verify } (s \ n) \ (s \ i) & \quad \mapsto \quad \text{verify } n \ i \\
\text{verify } (s \ n) \ (s \ (n+i)) & \quad \mapsto \quad \text{tooBig } n \ i \\
\text{verify } (s \ n) \ (\text{val } i) & \quad \mapsto \quad ?
\end{align*}
verify improved.

\[
\begin{align*}
\text{let} & \quad i \in \text{Fin } n \\
\quad & \quad \text{val } 0' \quad \mapsto \quad 0 \\
\quad & \quad \text{val } (s' \ i) \quad \mapsto \quad s \ (\text{val } i)
\end{align*}
\]

\[
\begin{align*}
data & \quad n, i \in \text{Nat} \\
\quad & \quad \text{where} \\
\quad & \quad \text{Bound } n \ i \in \ast \\
\quad & \quad \text{bound } n \ i \in \text{Bound } n \ (\text{val } i) \\
\quad & \quad \text{tooBig } n \ i \in \text{Bound } n \ (n+i)
\end{align*}
\]

\[
\begin{align*}
\text{let} & \quad n, i \in \text{Nat} \\
\quad & \quad \text{verify } n \ i \in \text{Bound } n \ i \\
\quad & \quad \text{verify } 0 \ i \quad \mapsto \quad \text{tooBig } 0 \ i \\
\quad & \quad \text{verify } (s \ n) \ 0 \quad \mapsto \quad \text{bound } n \ 0' \\
\quad & \quad \text{verify } (s \ n) \ (s \ i) \quad \parallel \quad \text{verify } n \ i \\
\quad & \quad \text{verify } (s \ n) \ (s \ (n+i)) \quad \parallel \quad \text{tooBig } n \ i \quad \mapsto \quad \text{tooBig } (s \ n) \ i \\
\quad & \quad \text{verify } (s \ n) \ (\text{val } i) \quad \parallel \quad \text{bound } n \ i \quad \mapsto \quad \text{bound } (s \ n) \ (s' \ i)
\end{align*}
\]
Definitional equality

The typing of \( \text{defeq} \) depends on the equations:

This equations need to be true definitionally.

If we need \( \text{defeq} \) we have to use propositional equality.
Definitional equality

The typing of verify depends on the equations:

\[ 0 + n \equiv n \]
\[ (s \ m) + n \equiv s (m + n) \]
Definitional equality

- The typing of `verify` depends on the equations:

\[ 0 + n \equiv n \]
\[ (s\ m) + n \equiv s\ (m + n) \]

- This equations need to be true \textit{definitionally}.
Definitional equality

The typing of \textit{verify} depends on the equations:

\[
0 + n \equiv n
\]
\[
(s \ m) + n \equiv s \ (m + n)
\]

This equations need to be true \textit{definitionally}.

If we need \( n + 0 = n \) we have to use propositional equality.
Propositional equality
Propositional equality

\[
\text{data } \begin{array}{c}
A \in \ast \\
\text{where }
\end{array}
\begin{array}{c}
a, b \in A \\
a = b \in \ast \\
\text{refl } a \in a = a
\end{array}
\text{ } a \in A
\]
Propositional equality

\[
\text{data } \quad \frac{A \in \ast \quad a, b \in A}{a = b \in \ast} \quad \text{where} \quad \frac{a \in A}{\text{refl } a \in a = a}
\]

\[
\begin{align*}
\text{let } & \quad n \in \text{Nat} \\
\text{add} \_ \_ \_ &= \text{add} \_ \_ \_ (s \_ \_ n) \quad \mid \quad n + 0 \\
\text{add} \_ \_ \_ n & \in n + 0 = n \\
\end{align*}
\]

\[
\begin{align*}
\text{add}_0 \ 0 & \quad \rightarrow \quad \text{refl} \ 0 \\
\text{add}_0 \ (s \ n) & \quad \mid \quad n + 0 \\
\mid \quad m & \quad \mid \quad \text{add}_0 \ n \\
\mid \quad n & \quad \mid \quad \text{refl} \quad \rightarrow \quad \text{refl}
\end{align*}
\]
Propositional equality

\[
\text{data } \prod A \to \ast \quad a, b \in A \quad \text{where} \quad a = b \in \ast \quad \text{refl } a \in a = a
\]

\[
\text{let } n \in \mathbb{Nat} \quad \text{add}_0 (s \ n) \parallel n + 0 \quad \text{add}_0 n \quad \text{m} \parallel \text{add}_0 n
\]

\[
\text{let } q \in a = b \quad P \in A \to \ast \quad x \in P \ a \quad \text{subst } (\text{refl } a) P x \leftrightarrow x
\]
Problems with

- Programs cluttered with coercions.
- Programming requires theorem proving.
- Equality on functions is not extensional, i.e. `let` cannot be derived.
Problems with $\equiv$

- Programs cluttered with coercions.
Problems with $\equiv$

- Programs cluttered with coercions.
- Programming requires theorem proving.
Problems with $\equiv$

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

\[
\begin{align*}
\text{let } & f, g \in A \to B \quad p \in \forall x \in A. f \ x = g \ x \\
\text{ext } & p \in f \equiv g
\end{align*}
\]

cannot be derived.
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.
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.
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

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.
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*. 

Why Dependent Types Matter – p.17/32
The object language

\[
\text{data Val} \in * \quad \text{where}
\]
\[
\begin{align*}
\text{vnat } n & \in \text{Val} \\
\text{vbool } b & \in \text{Val}
\end{align*}
\]
The object language

\[
\text{data } \text{Val} \in * \quad \text{where}
\]

\[
\begin{align*}
\text{data } &\text{Val} \in * \\
\text{data } &\text{Tm} \in * \\
\text{data } &\text{Val} \in * \\
\text{data } &\text{Tm} \in * \\
\text{data } &\text{Val} \in * \\
\text{data } &\text{Tm} \in * \\
\text{data } &\text{Val} \in * \\
\text{data } &\text{Tm} \in * \\
\text{data } &\text{Val} \in * \\
\text{data } &\text{Tm} \in * \\
\text{data } &\text{Val} \in * \\
\text{data } &\text{Tm} \in * \\
\end{align*}
\]

\[
\begin{align*}
&n \in \text{Nat} \\
&b \in \text{Bool} \\
&v_{\text{nat}} \; n \in \text{Val} \\
&v_{\text{bool}} \; b \in \text{Val} \\
&v_{\text{val}} \; v \in \text{Tm} \\
&t, u_0, u_1 \in \text{Tm} \\
&t_{\text{if}} \; t \; u_0, u_1 \in \text{Tm} \\
&t_{\text{add}} \; t \; u \in \text{Tm} \\
\end{align*}
\]
Object types

```haskell
data Ty ∈ * where
  nat ∈ Ty  
  bool ∈ Ty
```
Object types

\[
\text{data } \text{Ty} \in \ast \quad \text{where} \quad \text{nat} \in \text{Ty} \quad \text{bool} \in \text{Ty} \\
\text{let } t \in \text{Tm} \\
\text{verify } t \in \text{Maybe Ty} \\
\ldots
\]
Eval — simply typed

\[
\begin{align*}
  & \text{let } t \in \text{Tm} \\
  & \text{eval } t \in \text{Val}
\end{align*}
\]
Eval — simply typed

\[
\begin{align*}
& t \in Tm \\
& \text{let} \quad \quad \quad \quad \\
& \text{eval } t \in \text{Val} \\
& \text{eval (tval } v) \quad \rightarrow \quad v \\
& \text{eval (tif } t u_0 u_1 \text{) } \quad \| \quad \text{eval } t \\
& \quad \quad \| \quad \text{vnat } n \quad \rightarrow \quad ? \\
& \quad \quad \| \quad \text{vbool true} \quad \rightarrow \quad \text{eval } u_0 \\
& \quad \quad \| \quad \text{vbool false} \quad \rightarrow \quad \text{eval } u_1 \\
& \text{eval (tadd } t u) \quad \| \quad \text{eval } t \\
& \quad \quad \| \quad \text{vnat } m \quad \| \quad \text{eval } u \\
& \quad \quad \quad \| \quad \text{vnat } n \quad \rightarrow \quad \text{vnat}(m+n) \\
& \quad \quad \quad \| \quad \text{vbool } b \quad \rightarrow \quad ? \\
& \quad \quad \| \quad \text{vbool } b \quad \rightarrow \quad ?
\end{align*}
\]
Safe eval?

\[
\begin{align*}
\text{let } & t \in \text{Tm} \\
& \text{seval } t \in \text{Maybe Val}
\end{align*}
\]
Safe eval ?

\[
\text{let } \quad t \in \text{Tm} \\
\text{seval } t \in \text{Maybe Val}
\]

\[
\text{seval } t \mid \mid \text{ verify } t \\
| \quad \text{just } u \quad \mapsto \quad \text{just (eval } u) \\
| \quad \text{nothing} \quad \mapsto \quad \text{nothing}
\]
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

\[
\begin{align*}
\text{data} & \quad a \in \text{Ty} \\
\text{TVal} & \quad a \in \ast \\
\text{where} & \\
\text{vnat} & \quad n \in \text{TVal} \ \text{nat} \\
\text{vbool} & \quad b \in \text{TVal} \ \text{bool}
\end{align*}
\]
Object language using dependent types

\[
\begin{align*}
\text{data } &\quad \begin{array}{c}
(\exists a: \text{Ty}) \quad a \in \text{Ty} \\
\text{TVal} &\quad a \in \star \\
\text{where} \quad &
\end{array} \\
\text{where} \quad &
\begin{array}{c}
(\exists n: \text{Nat}) \quad n \in \text{Nat} \\
\text{vnat} &\quad n \in \text{TVal} \text{ nat} \\
\text{vbool} &\quad b \in \text{TVal} \text{ bool} \\
\text{TTm} &\quad a \in \star \\
\text{data} \quad &
\end{array}
\end{align*}
\]
Object language using dependent types

\[
\begin{align*}
& a \in \text{Ty} \\
\text{data} & \quad \text{where} \\
& \text{TVal} a \in \ast \\
& v \text{nat} \ n \in \text{TVal} \ \text{nat} \\
& v \text{bool} \ b \in \text{TVal} \ \text{bool} \\
& \hline
& n \in \text{Nat} \\
& b \in \text{Bool} \\
& \hline
\end{align*}
\]

\[
\begin{align*}
& a \in \ast \\
\text{data} & \quad \text{where} \\
& \text{TTm} a \in \ast \\
& v \text{Val} \ v \in \text{TTm} a \\
& t \text{bool} \ t \in \text{TTm} \ \text{bool} \\
& u_0, u_1 \in \text{TTm} a \\
& \text{tif} \ t u_0 u_1 \in \text{TTm} a \\
& \text{tadd} \ t u \in \text{TTm} \ \text{nat} \\
& \hline
\end{align*}
\]
Eval — dependently

\[
\begin{align*}
t & \in \text{T}_{\text{TTm}} a \\
\text{let} & \quad \text{eval } t \in \text{Val } a
\end{align*}
\]
Eval — dependently

\[ t \in \text{TTm} \ a \]
\[ \text{let } \quad \quad \quad \quad \quad \]
\[ \text{eval } t \in \text{Val} \ a \]

\[
\begin{align*}
\text{eval (tval } v) & \quad \mapsto \quad v \\
\text{eval (tif } t \ u_0 \ u_1) & \quad \| \quad \text{eval } t \\
& \quad \| \quad \text{vbool } \text{true} \quad \mapsto \quad \text{eval } u_0 \\
& \quad \| \quad \text{vbool } \text{false} \quad \mapsto \quad \text{eval } u_1 \\
\text{eval (tadd } t \ u) & \quad \| \quad \text{eval } t \\
& \quad \| \quad \text{vnat } m \quad \| \quad \text{eval } u \\
& \quad \| \quad \text{vnat } n \quad \mapsto \quad \text{vnat}(m+n)
\end{align*}
\]
Safe eval!
Safe eval!

\[ t \in \text{TTm a} \]
\[
\begin{align*}
\text{let} & \quad \text{strip } t \in \text{Tm} \\
\end{align*}
\]

\[ \ldots \]
Safe eval!

\[
\begin{align*}
    t & \in \text{TTm } a \\
    \text{let} & \quad \text{strip } t \in \text{Tm} \\
    t & \in \text{Tm} \\
    \text{data} & \quad \text{where} \\
    \text{Verify } t & \in \text{Ty} \\
    \text{error} & \in \text{Verify } t \\
    \text{ok} & \in \text{Verify } \left(\text{strip } t\right) \\
    t & \in \text{Tm} \\
    \text{let} & \quad \text{where} \\
    \text{verify } t & \in \text{Verify } t \\
    t & \in \text{Tm} \\
    \text{let} & \quad \text{where} \\
    \text{seval } t & \in \text{Maybe } \left(\sum_{a \in \text{TyVal }} a\right) \\
\end{align*}
\]

\[
\begin{align*}
    \text{seval } t & \mid \text{verify } t \\
    | \quad \text{just } u & \mapsto \text{just } \left(\text{eval } u\right) \\
    | \quad \text{error} & \mapsto \text{nothing}
\end{align*}
\]
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 \texttt{seval} will never crash.
- We can generate a certificate (proof carrying code).
- \texttt{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

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.
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

```
data _______ where
  Ty ∈ *
```
Codes and data

\[
\begin{align*}
\text{data} & \quad \text{where} \\
& \quad \text{Ty} \in * \\
\text{unit} & \in \text{Ty} \\
\text{prod} a \ b & \in \text{Ty} \\
\text{sum} a \ b & \in \text{Ty} \\
\text{rec} & \in \text{Ty}
\end{align*}
\]
Codes and data

\[
\begin{align*}
data & \quad \text{where} \\
Ty & \in * \\
\hline
unit & \in Ty \\
\hline
prod & a, b \in Ty \\
\hline
sum & a, b \in Ty \\
\hline
rec & \in Ty \\
\hline
r, a & \in Ty \\
data & \quad \text{where} \\
Val & r, a \in * 
\end{align*}
\]
Codes and data

\[
\begin{align*}
\text{data} & \quad \text{where} \\
\text{Ty} & \in \ast \\
\text{unit} & \in \text{Ty} \\
\text{prod} & \begin{cases} a, b \in \text{Ty} \end{cases} \\
\text{sum} & \begin{cases} a, b \in \text{Ty} \end{cases} \\
\text{rec} & \in \text{Ty} \\
\text{Val} & \begin{cases} r, a \in \text{Ty} \end{cases} \\
\text{void} & \in \text{Val} \quad \text{unit} \\
\text{x} & \in \text{Val} \quad \text{y} \in \text{Val} \\
\text{pair} & \begin{cases} x, y \in \text{Val} \quad \text{(prod} \quad a, b) \end{cases}
\end{align*}
\]
Codes and data

```
data Ty ∈ *

unit ∈ Ty
  a, b ∈ Ty
  prod a b ∈ Ty
  sum a b ∈ Ty
  rec ∈ Ty

r, a ∈ Ty
  data Val r a ∈ *

x ∈ Val r a
  y ∈ Val r b
  pair x y ∈ Val r (prod a b)

x ∈ Val a
  y ∈ Val b
  inl x ∈ Val (sum a b)
  inr y ∈ Val (sum a b)
```
Codes and data

\[
\begin{align*}
\text{data} & \quad \text{where} \\
& \quad \text{Ty} \in \ast \\
\text{unit} \in \text{Ty} & \quad a, b \in \text{Ty} & \quad a, b \in \text{Ty} & \quad r, a \in \text{Ty} \\
\text{prod} a b \in \text{Ty} & \quad \text{sum} a b \in \text{Ty} & \quad \text{rec} \in \text{Ty} \\
\text{Val} r a \in \ast & \quad x \in \text{Val} r a & \quad y \in \text{Val} r b \\
\text{void} \in \text{Val} r \text{unit} & \quad \text{pair} x y \in \text{Val} r (\text{prod} a b) \\
\text{inl} x \in \text{Val} (\text{sum} a b) & \quad \text{inr} y \in \text{Val} (\text{sum} a b) \\
\end{align*}
\]
Example
Example

\[
\text{let } a \in \text{Ty} \\
\frac{}{\text{Data } a \in \text{Ty}} \quad \frac{}{\text{Data } a \mapsto \text{Val } a\ a}
\]
Example

\[
\text{let } \frac{a \in Ty}{\text{Data } a \in Ty} \quad \text{Data } a \leftrightarrow \text{Val } a \ a
\]

\[
\text{let } \frac{\text{nat} \in Ty}{\text{nat} \leftrightarrow \text{sum unit rec}}
\]
Example

\[
\begin{align*}
\text{let } & a \in \text{Ty} \quad \text{Data } a \mapsto \text{Val } a a \\
\text{let } & \text{nat} \in \text{Ty} \quad \text{nat} \mapsto \text{sum} \ \text{unit} \ \text{rec} \\
\text{let } & \text{zero} \in \text{Data nat} \quad \text{zero} \mapsto \text{inl} \ \text{void}
\end{align*}
\]
Example

\[
\begin{align*}
\text{let } & \quad a \in \text{Ty} \quad \Rightarrow \quad \text{Data } a \in \text{Ty} \\
\text{Data } a & \leftrightarrow \text{Val } a \ a
\end{align*}
\]

\[
\begin{align*}
\text{let } & \quad \text{nat } \in \text{Ty} \\
\text{nat} & \leftrightarrow \text{sum } \text{unit } \text{rec}
\end{align*}
\]

\[
\begin{align*}
\text{let } & \quad \text{zero } \in \text{Data } \text{nat} \\
\text{zero} & \leftrightarrow \text{inl } \text{void}
\end{align*}
\]

\[
\begin{align*}
\text{let } & \quad n \in \text{Data } \text{nat} \\
\text{succ } n & \in \text{Data } \text{nat} \\
\text{succ } n & \leftrightarrow \text{inr } (\text{in } n)
\end{align*}
\]
Generic equality
Generic equality

\[
\text{let } \frac{x, y \in \text{Val } r t}{\text{eq } x y \in \text{Bool}}
\]
Generic equality

\[
\begin{align*}
\text{let } & \quad x, y \in \text{Val} \ r \ t \\
\text{eq } & \quad x \ y \in \text{Bool} \\
\text{eq } & \quad \text{void } \text{void} \quad \mapsto \quad \text{true} \\
\text{eq } & \quad (\text{pair } x \ y) \ (\text{pair } x' \ y') \quad \mapsto \quad (\text{eq } x \ x') \ \&\& \ (\text{eq } y \ y') \\
\text{eq } & \quad (\text{inl } x) \ (\text{inl } x') \quad \mapsto \quad \text{eq } x \ x' \\
\text{eq } & \quad (\text{inl } x) \ (\text{inr } y) \quad \mapsto \quad \text{false} \\
\text{eq } & \quad (\text{inr } y) \ (\text{inl } x) \quad \mapsto \quad \text{false} \\
\text{eq } & \quad (\text{inr } y) \ (\text{inr } y') \quad \mapsto \quad \text{eq } y \ y' \\
\text{eq } & \quad (\text{in } x) \ (\text{in } x') \quad \mapsto \quad \text{eq } x \ x' 
\end{align*}
\]
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
- Already comes with a verification language (totality checker)
- Safety constraints can be checked statically (safe mobile code)
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 Dependent Typing Programming

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