Vectors are records, too!

Jesper Cockx[1] Gaëtan Gilbert[2]
Nicolas Tabareau[2] Matthieu Sozeau[2]

[1] Gothenburg University, Sweden

[2] INRIA, France

21 June 2018

TYPES' most popular example[1]

$$
\begin{array}{l}
\mathsf{data}\ \mathbb{V}\ (A : \mathsf{Set}) : (n : \mathbb{N}) \to \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{nil} \quad\ \ : \mathbb{V}\ A\ \mathsf{zero} \\
\quad \mathsf{cons} : (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m) \to \mathbb{V}\ A\ (\mathsf{suc}\ m)
\end{array}
$$

[1] Disclaimer: I did not actually count all examples since 1990.

TYPES' most popular example[1]

$$\begin{aligned}
\mathbb{V} &: (A : \mathsf{Set})(n : \mathbb{N}) \to \mathsf{Set} \\
\mathbb{V}\ A\ \mathsf{zero} &= \top \\
\mathbb{V}\ A\ (\mathsf{suc}\ n) &= A \times \mathbb{V}\ A\ n
\end{aligned}$$

[1] Disclaimer: I did not actually count all examples since 1990.

TYPES' most popular example[1]

$$
\begin{array}{l}
\mathsf{data}\ \mathbb{V}\ (A : \mathsf{Set}) : (n : \mathbb{N}) \to \mathsf{Set}\ \mathsf{where} \\
\quad \mathsf{nil} \quad\ : \mathbb{V}\ A\ \mathsf{zero} \\
\quad \mathsf{cons} : (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m) \to \mathbb{V}\ A\ (\mathsf{suc}\ m)
\end{array}
$$

vs.

$$
\begin{array}{lcl}
\mathbb{V} : (A : \mathsf{Set})(n : \mathbb{N}) \to \mathsf{Set} & & \\
\mathbb{V}\ A\ \mathsf{zero} & = & \top \\
\mathbb{V}\ A\ (\mathsf{suc}\ n) & = & A \times \mathbb{V}\ A\ n
\end{array}
$$

[1] Disclaimer: I did not actually count all examples since 1990.

Presenting. . .

A common representation of indexed datatypes and recursive types as **case-splitting datatypes**.

An elaboration algorithm to automatically transform an indexed datatype into a case-splitting datatype.

Inductive types vs. recursive types

Case-splitting datatypes

Elaborating indexed datatypes

Inductive types vs. recursive types

Case-splitting datatypes

Elaborating indexed datatypes

Inductive type

Recursive type

Inductive type | Recursive type

- Intuitive notation

$$
\begin{aligned}
&\text{data } \mathbb{V}\ (A : \mathsf{Set}) : (n : \mathbb{N}) \to \mathsf{Set} \text{ where} \\
&\quad \mathsf{nil} \quad\ : \mathbb{V}\ A\ \mathsf{zero} \\
&\quad \mathsf{cons} : (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m) \to \mathbb{V}\ A\ (\mathsf{suc}\ m)
\end{aligned}
$$

Inductive type | Recursive type |

- Intuitive notation
- Eta equality

$$
\begin{aligned}
x : \mathbb{V}\ A\ \mathsf{zero} &\ \vdash\ x \equiv \mathsf{tt} \\
x : \mathbb{V}\ A\ (\mathsf{suc}\ m) &\ \vdash\ x \equiv (x\ .\pi_1, x\ .\pi_2)
\end{aligned}
$$

Inductive type II

- Intuitive notation
- Pattern matching

Recursive type I

- Eta equality

$$\mathsf{tail} : (m : \mathbb{N})(xs : \mathbb{V}\ A\ (\mathsf{suc}\ m)) \to \mathbb{V}\ A\ m$$
$$\mathsf{tail}\ m\ (\mathsf{cons}\ \lfloor m \rfloor\ x\ xs) = xs$$

Inductive type II

- Intuitive notation
- Pattern matching

Recursive type II

- Eta equality
- Forcing & detagging for free

$$
\begin{aligned}
&\mathsf{cons} : (m : Nat)(x : A)(xs : \mathbb{V}\ A\ m) \\
&\qquad \to \mathbb{V}\ A\ (\mathsf{suc}\ m) \\
&\mathsf{cons}\ \ m\ x\ xs = (x, xs)
\end{aligned}
$$

Inductive type III

- Intuitive notation
- Pattern matching
- Structural recursion

Recursive type II

- Eta equality
- Forcing & detagging for free

Inductive type III

- Intuitive notation
- Pattern matching
- Structural recursion

Recursive type III

- Eta equality
- Forcing & detagging for free
- Large indices

$$
\begin{array}{llll}
\multicolumn{4}{l}{\_ \leq \_ : \mathbb{N} \to \mathbb{N} \to \mathsf{Prop}} \\
\mathsf{zero} & \leq\ n & = & \top \\
(\mathsf{suc}\ m) & \leq\ \mathsf{zero} & = & \bot \\
(\mathsf{suc}\ m) & \leq\ (\mathsf{suc}\ n) & = & m \leq n
\end{array}
$$

Inductive type IIII

- Intuitive notation
- Pattern matching
- Structural recursion
- Non-indexed / non-stratified types

Recursive type III

- Eta equality
- Forcing & detagging for free
- Large indices

$\mathbb{N} = \top \uplus \mathbb{N}$ is not a valid definition!

Inductive type IIII

- Intuitive notation
- Pattern matching
- Structural recursion
- Non-indexed / non-stratified types

Recursive type IIII

- Eta equality
- Forcing & detagging for free
- Large indices
- Non-positive types

data U : Set where
  $\_\Rightarrow\_ : U \to U \to U$

$\models\ : \mathsf{U} \to \mathsf{Set}$

$\models (t_1 \Rightarrow t_2) = \models t_1 \to \models t_2$

Inductive types vs. recursive types

Case-splitting datatypes

Elaborating indexed datatypes

$\boxed{\textbf{Indexed datatype}}$

$$\text{data } \mathbb{V}\ A : \mathbb{N} \to \mathsf{Set} \text{ where}$$
$$\ldots$$

$\boxed{\textbf{Recursive type}}$

$$\mathbb{V}\ A\ \mathsf{zero} \quad = \ldots$$
$$\mathbb{V}\ A\ (\mathsf{suc}\ m) = \ldots$$

Indexed datatype	$\textsf{data}\ \mathbb{V}\ A : \mathbb{N} \to \textsf{Set}\ \textsf{where}$ $\quad \ldots$
Case-splitting datatype	$\mathbb{V}\ A\ n = \textsf{case}_n \{\ldots\}$
Recursive type	$\mathbb{V}\ A\ \textsf{zero} = \ldots$ $\mathbb{V}\ A\ (\textsf{suc}\ m) = \ldots$

$\boxed{\textbf{Indexed datatype}}$

$\Uparrow$

$\boxed{\textbf{Case-splitting datatype}}$

$\Downarrow$

$\boxed{\textbf{Recursive type}}$

$$\mathsf{data}\ \mathbb{V}\ A : \mathbb{N} \to \mathsf{Set}\ \mathsf{where}$$
$$\ldots$$

$$\mathbb{V}\ A\ n = \mathsf{case}_n\,\{\ldots\}$$

$$\begin{aligned}\mathbb{V}\ A\ \mathsf{zero} &= \ldots\\ \mathbb{V}\ A\ (\mathsf{suc}\ m) &= \ldots\end{aligned}$$

Indexed datatype

$\Uparrow \quad \Downarrow$

Case-splitting datatype

$\Downarrow$

Recursive type

$$\mathsf{data}\ \mathbb{V}\ A : \mathbb{N} \to \mathsf{Set}\ \mathsf{where}$$
$$\ldots$$

$$\mathbb{V}\ A\ n = \mathsf{case}_n\,\{\ldots\}$$

$$\begin{aligned} \mathbb{V}\ A\ \mathsf{zero} &= \ldots \\ \mathbb{V}\ A\ (\mathsf{suc}\ m) &= \ldots \end{aligned}$$

General syntax for case-splitting datatypes

$$Q ::= \mathsf{c}_1\ \Delta_1 \mid \ldots \mid \mathsf{c}_\mathsf{k}\ \Delta_k$$

$$\mid \quad \mathsf{case}_x \left\{ \begin{array}{l} \mathsf{c}_1\ \hat{\Delta}_1 \mapsto^{\tau_1} Q_1 \\ \vdots \\ \mathsf{c}_\mathsf{n}\ \hat{\Delta}_n \mapsto^{\tau_n} Q_n \end{array} \right\}$$

Case tree for $\mathbb{V}\ A\ n$

$$\mathsf{case}_n \left\{ \begin{array}{lcl} \mathsf{zero} & \mapsto & \mathsf{nil} \\ \mathsf{suc}\ m & \mapsto & \mathsf{cons}\ (x : A)(xs : \mathbb{V}\ A\ m) \end{array} \right\}$$

Case tree for $m \leq n$

$$\mathsf{case}_m \left\{ \begin{array}{l} \mathsf{zero} \mapsto \mathsf{lz} \\ \mathsf{suc}\ m' \mapsto \mathsf{case}_n \\ \qquad \left\{ \begin{array}{ll} \mathsf{zero} & \mapsto \\ \mathsf{suc}\ n' & \mapsto \mathsf{ls}\ (p : m' \leq n') \end{array} \right\} \end{array} \right\}$$

From case tree to a datatype:
Ignore case splits;
gather all constructors in a flat list.

From case tree to a recursive definition:
Translate case splits with tools from
‘eliminating dependent pattern matching’.

Inductive types vs. recursive types

Case-splitting datatypes

Elaborating indexed datatypes

Problem:

We don't want to write case trees, we want to write datatypes!

Problem:

We don't want to write case trees, we want to write datatypes!

Solution:

Elaborate datatypes to case trees automatically.

State of elaborating a datatype

$$\Delta \vdash \left\{ \begin{array}{l} \mathsf{c}_1\ \Delta_1\ [\Phi_1] \\ \vdots \\ \mathsf{c}_\mathsf{k}\ \Delta_k\ [\Phi_k] \end{array} \right\}$$

- Δ is 'outer' telescope of datatype indices
- $\mathsf{c}_1, \ldots, \mathsf{c}_\mathsf{k}$ are the constructor names
- Δ_i is 'inner' telescope of arguments of c_i
- Φ_i is a set of constraints $\{v_j /^? p_j\}$

Initial elaboration state

$$(A : \mathsf{Set})(n : \mathbb{N}) \vdash$$

$$\left\{ \begin{array}{lll} \mathsf{nil} & & [\mathsf{zero} \mathrel{/^?} n] \\ \mathsf{cons} & (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m) & [\mathsf{suc}\ m \mathrel{/^?} n] \end{array} \right\}$$

Elaboration step: case split on index

$$(A : \mathsf{Set})(n : \mathbb{N}) \vdash$$

$$\left\{ \begin{array}{lll} \mathsf{nil} & & [\mathsf{zero} \;/^?\; n] \\ \mathsf{cons} & (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m) & [\mathsf{suc}\ m \;/^?\; n] \end{array} \right\}$$

$$\Downarrow$$

$$(A : \mathsf{Set}) \vdash \left\{ \mathsf{nil}\ \ [\mathsf{zero} \;/^?\; \mathsf{zero}] \right\}$$

$$(A : \mathsf{Set})(n' : \mathbb{N}) \vdash$$

$$\left\{ \mathsf{cons}\ \ (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m)\ \ [\mathsf{suc}\ m \;/^?\; \mathsf{suc}\ n'] \right\}$$

Elaboration step: solve constraint

$$(A : \mathsf{Set})(n : \mathbb{N}) \vdash$$
$$\left\{\ \mathsf{cons}\ \ (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m)\ \ [\mathsf{suc}\ m\ /^{?}\ \mathsf{suc}\ n'\,]\ \right\}$$

$$\Downarrow$$

$$(A : \mathsf{Set})(n : \mathbb{N}) \vdash \left\{\ \mathsf{cons}\ \ (x : A)(xs : \mathbb{V}\ A\ n')\ \right\}$$

Elaboration step: finish splitting

$$(A : \mathsf{Set})(n : \mathbb{N}) \vdash$$

$$\left\{\ \mathsf{cons}\ \ (m : \mathbb{N})(x : A)(xs : \mathbb{V}\ A\ m)\ \ [\mathsf{suc}\ m \ /^{?}\ \mathsf{suc}\ n']\ \right\}$$

$$\Downarrow$$

$$(A : \mathsf{Set})(n : \mathbb{N}) \vdash \left\{\ \mathsf{cons}\ \ (x : A)(xs : \mathbb{V}\ A\ n')\ \right\}$$

$$\Downarrow$$

$$\mathsf{cons}\ (x : A)(xs : \mathbb{V}\ A\ n')$$

Elaboration step: introduce equality proof

$$(A\ B : \mathsf{Set})(f : A \to B)(y : B) \vdash \left\{\ \mathsf{image}\ (x : A)\ [f\ x\ /^{?}\ y]\ \right\}$$

$$\Downarrow$$

$$(A\ B : \mathsf{Set})(f : A \to B)(y : B) \vdash \left\{\ \mathsf{image}\ (x : A)\ (e : f\ x \equiv_B y)\ \right\}$$

Ongoing & future work

- Implement translation in Coq (WIP)
- Generate constructors & eliminator
- Generate case trees for Agda datatypes
- User syntax to control splitting?

Conclusion

Datatypes have long been denied features of record types such as η-equality.

Conclusion

Datatypes have long been denied features of record types such as η-equality.

We can *automatically* transform a datatype into an equivalent definition with η-laws.

Conclusion

Datatypes have long been denied features of record types such as η-equality.

We can *automatically* transform a datatype into an equivalent definition with η-laws.

Now you can both have the cake and eat it: vectors are records, too!