Vectors are records, too

Jesper Cockx[1], Gaëtan Gilbert[2], and Nicolas Tabareau[2]

[1] Gothenburg University, Sweden
[2] INRIA, France

When talking about dependently typed programming languages such as Coq and Agda, it is traditional to start with an example involving vectors, i.e. length-indexed lists:

$$\begin{array}{l} \mathsf{data}\ \mathsf{Vector}\ (A : \mathsf{Set}) : (n : \mathbb{N}) \to \mathsf{Set}\ \mathsf{where} \\ \quad \mathsf{nil} \quad : \quad \mathsf{Vector}\ A\ \mathsf{zero} \\ \quad \mathsf{cons} \quad : \quad (m : \mathbb{N})(x : A)(xs : \mathsf{Vector}\ A\ m) \to \mathsf{Vector}\ A\ (\mathsf{suc}\ m) \end{array} \tag{1}$$

This definition of vectors as an indexed family of datatypes is very intuitive: we take the definition of lists and ornament them with their length. Alternatively, we can also define vectors by recursion on the length:

$$\begin{array}{lcl} \multicolumn{3}{l}{\mathsf{Vector} : (A : \mathsf{Set})(n : \mathbb{N}) \to \mathsf{Set}} \\ \mathsf{Vector}\ A\ \mathsf{zero} & = & \top \\ \mathsf{Vector}\ A\ (\mathsf{suc}\ n) & = & A \times \mathsf{Vector}\ A\ n \end{array} \tag{2}$$

This transformation of an indexed family of inductive datatypes (or *indexed datatype* for short) into a recursive definition has a number of benefits:

- Vector inherits η-laws from the record types $\top$ and $A \times B$: every vector of length zero is definitionally equal to $\mathsf{tt} : \top$, and every vector $xs : \mathsf{Vec}\ A\ (\mathsf{suc}\ n)$ is definitionally equal to the pair $(\mathsf{fst}\ xs, \mathsf{snd}\ xs)$ where $\mathsf{fst}\ xs : A$ and $\mathsf{snd}\ xs : \mathsf{Vector}\ A\ n$.
- We get the forcing and detagging optimizations from Brady et al. (2004) for free: we do not have to store the length of a vector, and not even whether it is a nil or a cons.
- There are no restrictions on the sorts of the types of forced indices; they can be in a bigger sort than the datatype itself. In particular, this allows us to define indexed datatypes in a proof-irrelevant universe such as Prop, as long as the constructor can be uniquely determined from the indices and all non-forced constructor arguments are in the proof-irrelevant universe themselves.
- The recursive occurrences of the datatype do not have to be strictly positive: they only have to use a structurally smaller index. This allows us to define stratified types as in Beluga (Pientka, 2015).

While this transformation works for vectors, it is not possible for all datatypes. For example, the recursive definition of $\mathbb{N}$ by the equation $\mathbb{N} = \top \uplus \mathbb{N}$ is invalid since it is not terminating. This explains why we cannot allow η-laws for all datatypes.

As another counterexample, consider the datatype $\mathsf{Image}\ (A\ B : \mathsf{Set})(f : A \to B) : B \to \mathsf{Set}$ with one constructor $\mathsf{image} : (x : A) \to \mathsf{Image}\ A\ B\ f\ (f\ x)$. Image cannot be defined by pattern matching on $y : B$ since $f\ x$ is not a pattern. We can instead transform the index into a parameter by introducing an equality proof: $\mathsf{Image}\ A\ B\ f\ y = \Sigma_{(x:A)}(f\ x \equiv_B y)$. This transformation removes the non-pattern index and hence allows us to match against an element of $\mathsf{Image}\ A\ B\ f\ u$ even when u is not a variable. On the other hand, this transformation does not enable us to have large indices: we cannot define Image in Prop since both A and $f\ x \equiv_B y$ have to fit in the sort of Image.

Both these transformations for removing the indices from a datatype definition as described above are well known, but so far the only way to get their benefits was to apply them by hand. This means that we also have to define terms for the constructors and the elimination principle ourselves, and we cannot rely on built-in support for indexed datatypes such as dependent pattern matching.

We present a fully automatic and general transformation of an indexed datatype to an equivalent definition of a type as a case tree. This transformation generates not just the type itself but also terms for the constructors and the elimination principle. It exposes eta laws for datatypes when there is only a single possible constructor for the given indices, and removes non-pattern indices by introducing equality proofs as new constructor arguments.

Our transformation is similar to the elaboration of dependent pattern matching (Goguen et al., 2006). It uses pattern matching on the indices where it can, and introduces equality proofs where it must. First we elaborate the datatype declaration to a case tree where each internal node indicates a case split on one of the indices, and each leaf node contains some (possibly zero) telescopes for the arguments of each constructor. For Vector we get:

$$\mathsf{Vector} = \lambda A, n.\ \mathsf{case}_n \left\{ \begin{array}{lll} \mathsf{zero} & \mapsto & () \\ \mathsf{suc}\ m & \mapsto & (x : A)(xs : \mathsf{Vector}\ A\ m) \end{array} \right\} \qquad (3)$$

Any non-constructor patterns and non-linear variables are dealt with by replacing them with fresh variables and introducing equality types on the right-hand side. For Image we get:

$$\mathsf{Image} = \lambda y.\ (x : A)(p : f\ x \equiv_B y) \qquad (4)$$

Once we have a case tree, it is straightforward to construct a definition for the datatype itself, as well as for the constructors and the eliminator.

Our approach is similar to the notion of case-splitting datatypes of Dagand and McBride (2014), but we do not require any annotations from the user. One could however imagine extending our approach with some user annotations to guide the elaboration process, similar to the inaccessible patterns from dependent pattern matching.

For a long time datatypes have been discriminated against by refusing to give them eta equality and restricting the sort of their indices. We say: no more! Let vectors be records, too.

References

Edwin Brady, Conor McBride, and James McKinna. Inductive families need not store their indices. In *Types for Proofs and Programs*, 2004.

Pierre-Evariste Dagand and Conor McBride. Transporting functions across ornaments. *Journal of functional programming*, 24(2-3):316–383, 2014.

Healfdene Goguen, Conor McBride, and James McKinna. Eliminating dependent pattern matching. In *Algebra, Meaning, and Computation: Essays dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday*. 2006.

Brigitte Pientka. *Beluga 0.8.2 Reference Guide*, 2015. URL `http://complogic.cs.mcgill.ca/beluga/userguide2/userguide.pdf`.