Species Constructors

Brent A. Yorgey
Stephanie Weirich

Dept. of Computer and Information Science
The University of Pennsylvania
Philadelphia, Pennsylvania, USA
{byorgey,sweirich}@cis.upenn.edu

Jacques Carette

Dept. of Computing and Software
McMaster University
Hamilton, Ontario, Canada
carette@mcmaster.ca

Abstract

[TODO: Abstract goes here.]

Categories and Subject Descriptors D.3.2 [*Programming Languages*]: Applicative (functional) languages

General Terms Languages, Types

1. Introduction

TODO: Motivation. "An answer looking for a question." Note symmetries were original motivation, but drawn to labels instead. "Follow the theory" and see what pops out.

Take-home points:

- Labelled structures capture a wide range of data structures.
- Combinators! ($\times 2!$ — type level and value level)

Other interesting but not take-home points:

- fun with isos
- labels as abstract model of memory
- labels make sharing easy

The idea of separating shapes and data is not new [TODO: citations: containers, shapely types, etc.]. However, previous approaches have left the labels *implicit*. Bringing the labels to the fore enables cool stuff like

- include a bunch of disparate stuff under one framework
- let us talk about relabelling as a separate operation
- put structure on the labels themselves, e.g. L-species
- [TODO: more?]

2. Labelled Structures

Rather than diving immediately into species, we begin with an intuitive definition of "labelled structures" and some examples.

The essential idea of labelled structures is to separate the notions of container shapes and the data stored in those shapes. Labels provide the missing link between shapes and data, allowing one to specify which data goes where.

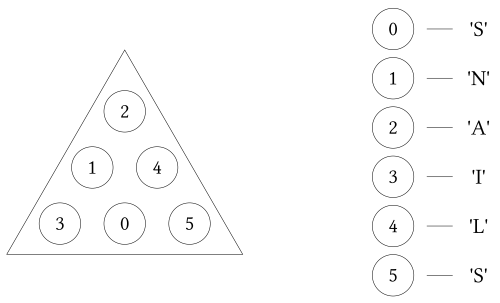

Figure 1. A labelled structure with six labels

Informally, a *labelled structure* is specified by:

- a finite type of labels L;
- a type of data elements A;
- some sort of "shape" containing each label from L exactly once; and
- a function $v : L \to A$ which maps labels to data values.

See Figure 1 for an abstract example. A *family* of labelled structures refers to a class of structures parameterized over the label type L and data type A.

Note that shapes must contain each label exactly once, but the function $L \to A$ need not be injective. As illustrated in Figure 1, it is perfectly valid to have the same value of type A occurring multiple times, each matched to a different label. The requirement that shapes contain all the labels, with no duplicates, may seem overly restrictive; we will have more to say about this later. The notion of "shape" is also left vague for now; a precise definition will be given in [TODO: where?].

Algebraic data types All the usual algebraic data types can be viewed as families of labelled structures. For example, [TODO: example]. Note, however, that the family of labelled tree structures is quite a bit larger than the usual algebraic type of trees: every possible different way of labelling a given tree shape results in a different labelled structure. For algebraic data types, this added structure is uninteresting, in a way that we will make precise later [TODO: when?]. [Idea here is that for regular species we can always recover a canonical labelling from the shape; and moreover there are always precisely $n!$ different labellings for a shape of size n (given a fixed set of labels). —BAY]

Finite maps Since the definition of a labelled structure already includes the notion of a mapping from labels to data, we may encode finite maps simply by using *sets* of labels as shapes, *i.e.* shapes with no structure other than containing some labels.

[TODO: picture?]

Vectors and arrays Vectors, and multi-dimensional arrays more generally, [TODO: from one point of view are just finite maps with some nontrivial structure on the labels. Can also move the structure around between labels and shape (???).]

Symmetric shapes We have not yet defined precisely what counts as a "shape", but one interesting possibility is the use of shapes with some sort of *symmetry*. For example, a *cycle* is like a list, except that it is invariant under cyclic rotation of its labels. One area where cycles are especially useful is in computational geometry: we can represent an (oriented) polygon, for example, as a labelled cycle shape, with each label mapping to a point in space.

[TODO: picture of a polygon represented with labelled cycle]

[should we include cycles at all? Our system can't handle them although they fit from a theoretical point of view… —BAY]

An *unordered pair* is another sort of shape with symmetry: it is like an ordered pair but invariant under swapping. Unordered pairs can be used to represent undirected graph edges, [TODO: other stuff?]

Value-level sharing [e.g. $\mathsf{L} \times \mathsf{F}$ —BAY]

Graphs [can we do graphs? —BAY]

3. Combinatorial Species

3.1 Species, set-theoretically

We begin with a standard set-theoretic definition of species [**??**] (we will upgrade to a *type*-theoretic definition in §3.2).

Definition 1. A *species* F is a pair of mappings which

- sends any finite set L (of *labels*) to a finite set $F[L]$ (of *shapes*), and
- sends any bijection on finite sets $\sigma : L \leftrightarrow L'$ (a *relabelling*) to a function $F[\sigma] : F[L] \to F[L']$ (illustrated in Figure **??**),

satisfying the following functoriality conditions:

- $F[id_L] = id_{F[L]}$, and
- $F[\sigma \circ \tau] = F[\sigma] \circ F[\tau]$.

Using the language of category theory, this definition can be pithily summed up by saying that a species is a functor $F : \mathbb{B} \to \mathsf{FinSet}$, where $\mathbb{B}$ is the category of finite sets whose morphisms are bijections, and FinSet is the category of finite sets whose morphisms are arbitrary (total) functions.

We call $F[L]$ the set of "F-shapes with labels drawn from L", or simply "F-shapes on L", or even (when L is clear from context) just "F-shapes. $F[\sigma]$ is called the "transport of σ along F", or sometimes the "relabelling of F-shapes by σ".

Note that in the combinatorial literature, elements of $F[L]$ are usually called "F-structures" rather than "F-shapes". To a combinatorialist, labelled shapes are themselves the primary objects of interest; however, in a computational context, we must be careful to distinguish between labelled structures (which have data associated with the labels) and bare labelled shapes (which do not).

Here we see that the formal notion of "shape" is actually quite broad, so broad as to make one squirm: a shape is just an element of some arbitrary set! In this context our informal insistance from the previous section that a shape "contain each label exactly once" is completely meaningless, because there is no sense in which we can say that a shape "contains" labels.

In practice, however, we are interested not in arbitrary species but in ones built up algebraically from a set of primitives and operations. In that case the corresponding shapes will have more structure as well. Before we get there, however, we need to give the definition of species a firmer computational basis.

3.2 Species, constructively

The foregoing set-theoretic definition of species is perfectly serviceable in the context of classical combinatorics, but in order to use it as a foundation for data structures, it is useful to first "port" the definition from set theory to constructive type theory.

In the remainder of this paper, we work within a standard variant of Martin-Löf dependent type theory [**?**] (precisely *which* variant we use probably does not matter very much), equipped with an empty type $\mathbf{0}$, unit type $\mathbf{1}$, coproducts, dependent pairs, dependent functions, a universe Type of types, and a notion of propositional equality. For convenience, instead of writing the traditional $\sum_{x:A} B(x)$ and $\prod_{x:A} B(x)$ for dependent pair and function types, we will use the Agda-like [**?**] notations $(x : A) \times B(x)$ and $(x : A) \to B(x)$, respectively. We continue to use the standard abbreviations $A \times B$ and $A \to B$ for non-dependent pair and function types, that is, when x does not appear free in B.

[TODO: Need to pick a notation for implicit arguments (∀? subscript? Agda braces?), and explain it. *e.g.* see types of fO and fS below.]

We use $\mathbb{N} : \mathsf{Type}$ to denote the usual inductively defined type of natural numbers, with constructors $\mathsf{O} : \mathbb{N}$ and $\mathsf{S} : \mathbb{N} \to \mathbb{N}$. We also make use of the usual indexed type of canonical finite sets $\mathsf{Fin} : \mathbb{N} \to \mathsf{Type}$, with constructors $\mathsf{fO} : \forall (n : \mathbb{N}).\, \mathsf{Fin} (\mathsf{S}\, n)$ and $\mathsf{fS} : \forall (n : \mathbb{N}).\, \mathsf{Fin}\, n \to \mathsf{Fin} (\mathsf{S}\, n)$.

[TODO: define $\leftrightarrow$] [Have to be careful: don't want this section to just become a big Agda file! Want to define things precisely enough so that people get the idea but omit all the sufficiently obvious parts. *e.g.* we probably don't actually need to write out the definition of $\leftrightarrow$ but just say in English what it denotes (*i.e.* a pair of inverse functions). —BAY]

The first concept we need to port is that of a finite set. Constructively, a finite set is one with an isomorphism to $\mathsf{Fin}\ n$ for some natural number n. That is,

$$\mathsf{IsFinite}\, A :\equiv (n : \mathbb{N}) \times (\mathsf{Fin}\, n \leftrightarrow A).$$

[Note there are other notions of finiteness but this is the one we want/need? See *e.g.* `http://ncatlab.org/nlab/show/finite+set`. —BAY] Then we can define $\mathsf{FinType} :\equiv (A : \mathsf{Type}) \times \mathsf{IsFinite}\, A$ as the universe of finite types.

[TODO: need some nice notation for dependent n-tuples, *i.e.* records.]

[Should we write it out like this? Or just use some sort of $\mathsf{IsFunctor}$ which we leave undefined? —BAY]

$$\begin{aligned}\mathsf{Species} :\equiv\ & (\mathsf{shapes} : \mathsf{FinType} \to \mathsf{Type}) \\ & \times (\mathsf{relabel} : (\mathsf{FinType} \leftrightarrow \mathsf{FinType}) \to (\mathsf{Type} \leftrightarrow \mathsf{Type})) \\ & \times ((L : \mathsf{FinType}) \to \mathsf{relabel}\, \mathsf{id}_L = \mathsf{id}_{(\mathsf{shapes}\ L)}) \\ & \times ((L_1, L_2, L_3 : \mathsf{FinType}) \to (\sigma : L_2 \leftrightarrow L_3) \to (\tau : L_1 \leftrightarrow L_2) \to (\mathsf{rel}\end{aligned}$$

Where the meaning is clear from context, we will use simple application to denote the action of a species on both objects and arrows. That is, if $F : \mathsf{Species}$, instead of writing $\pi_1\ F\ L$ or $\pi_1\ (\pi_2\ F)\ \sigma$ we will just write $F\ L$ or $F\ \sigma$.

3.3 The algebra of species

We now return to the observation from §3.1 that we do not really want to work directly with the definition of species, but rather with an algebraic theory. [TODO: say a bit more]

Zero The *zero* or *empty* species, denoted $\mathsf{0}$, is the unique species with no shapes whatsoever, defined by its action on finite types and bijections as

$$\mathsf{0}\ L = \mathbf{0}$$
$$\mathsf{0}\ \sigma = \mathsf{id}_{\mathbf{0}}$$

We omit the straightforward proofs of functoriality.

One The *one* or *unit* species, denoted $\mathsf{1}$, is the species with a single shape of size 0. Set-theoretically, we could write

$$\mathsf{1}\ L = \begin{cases} \{\star\} & |U| = 0 \\ \varnothing & \text{otherwise} \end{cases}$$

But this is confusing [TODO: explain]. Type-theoretically,

$$\mathsf{1}\ L = (\mathbf{0} \leftrightarrow L)$$
$$\mathsf{1}\ \sigma = (\lambda\tau.\sigma \circ \tau) \leftrightarrow (\lambda\tau.\sigma^{-1} \circ \tau)$$

which gives a better idea of what is going on: in order to construct something of type $\mathsf{1}\ L$, one must provide a proof that L is empty (that is, isomorphic to $\mathbf{0}$).

Singleton The *singleton* species, denoted X, is defined by

Bags The species of *bags*

[TODO: Note that a lot of the power of the theory for combinatorics comes from homomorphisms to rings of formal power series; we won't use that in this paper.]

3.4 Labelled structures, formally

Formally, we may define a labelled structure as a dependent five-tuple with the type

$$(F : \mathsf{Species}) \times (L : \mathsf{FinType}) \times (A : \mathsf{Type}) \times F\ L \times (L \to A),$$

that is,

- a species F,
- a constructively finite type L of *labels*,
- a type A of *data*,
- a shape of type $F\ L$, *i.e.* an L-labelled F-shape,
- a mapping from labels to data, $m : L \to A$.

[TODO: formal intro and elim forms for labelled structures? operations on labelled structures?]

[TODO: how formal do we want/need to make this?]

4. Labelled Structures in Haskell

[TODO: Describe our implementation. Note that actually compiling such things to efficient runtime code is future work.]

5. Programming with Labelled Structures

[TODO: Give some examples of using our implementation. e.g. n-dimensional vectors.]

6. Related Work

- containers, naturally
- shapely types
- HoTT?

7. Conclusion