Labelled structures and combinatorial species

Brent A. Yorgey Stephanie Weirich
Dept. of Computer and Information Science
The University of Pennsylvania
Philadelphia, Pennsylvania, USA

Jacques Carette
Dept. of Computing and Software
McMaster University
Hamilton, Ontario, Canada

We describe a theory of *labelled structures*, which intuitively consist of a labelled shape together with a mapping from labels to data. Labelled structures thus subsume algebraic data types as well as "labelled" types such as arrays and finite maps. This idea of decomposing container structures into shapes and data is an old one; our novel contribution is to explicitly mediate the decomposition with arbitrary labels. We demonstrate the benefits of this approach, showing how it can be used, for example, to explicitly model composition of container shapes, to model value-level sharing via superimposing multiple structures on the same data, and to model issues of memory allocation and layout.

The theory of labelled structures is built directly on the foundation of *combinatorial species*, which serve to describe labelled shapes. The theory of species bears striking similarities to the theory of algebraic data types, and it has long been suspected that a more precise and fruitful connection could be made between the two. In a larger sense, the aim of this paper is to serve as a first step in connecting combinatorial species to the theory and practice of programming. In particular, we describe a "port" of the theory of species into constructive type theory, justifying its use as a basis for computation.

1 Introduction

The theory of combinatorial species [Joyal, 1981, Bergeron et al., 1998], as it relates to the theory and practice of programming languages, has long seemed to the authors "an answer looking for a question": the theory is too beautiful, and too "obviously" related to algebraic data types, to have no applications whatsoever. Teasing out the precise relationship between species and data types, however, has proved challenging, for two reasons. First, combinatorialists are mainly concerned with enumerating and generating abstract structures, not with storing and computing with data. Thus, in order to apply species in a computational setting, there are many hidden assumptions and glossed distinctions that must first be made explicit. Second, being situated in traditional mathematical practice rooted in set theory, species are usually described in ways that are *untyped* and *nonconstructive*, both of which hinder adoption and understanding in a computational context.

From a computational point of view, the right way to think about species is as *labelled shapes* which do not contain any data. To recover *data* structures, one must add a notion of mapping from labels to data. This leads to the familiar idea of decomposing data structures as shapes plus data [Banger and Skillicorn, 1993, Jay and Cockett, 1994, Hoogendijk and De Moor, 2000, Abbott et al., 2003a], with the new twist that arbitrary labels are used to mediate between the two. Informally, this pairing of a labelled shape (corresponding to a species) and a mapping from labels to data values is what we call a *labelled structure*[1]. For example, Figure 1 illustrates a labelled tree shape paired with a mapping from labels to data. A *family* of labelled structures refers to a class of structures parameterized over the label type L and (typically) the data type A.

[1] Following Flajolet *et al.*'s lead [1991, 1994].

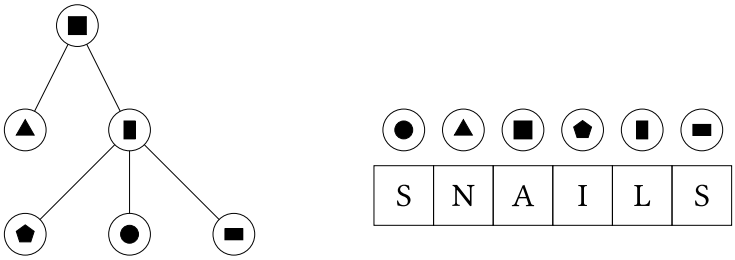

Figure 1: A labelled structure with six labels

Note that the mapping from labels to data values need not be injective, so the same value of type A may be associated to multiple labels. However, the mapping must of course be functional, that is, each label is assigned to exactly one value.

For now, we leave the notion of "labelled shape" abstract; we will return to define it more precisely in §3. Instead, we turn to a collection of examples showing the possibilities afforded by this framework.

Algebraic data types All the usual algebraic data types have corresponding families of labelled structures, where values of the algebraic data type are used as labelled shapes (§5.1). Given such a labelled structure we can "collapse" it back to an algebraic data structure by substituting data for labels. For example, the labelled tree structure in Figure 1 represents the tree containing `A` at its root, `N` as the left child, and so on. Note that the family of labelled tree structures is quite a bit larger than the usual type of trees: every possible labelling of a given tree shape results in a different labelled structure, whereas there are many labelled tree structures that will "collapse" to the same algebraic data structure, which differ only in the way they are labelled.

Composition The indirection afforded by labels makes it easy to explicitly model and reason about the *composition* of container shapes,*e.g.* lists of trees (§5.2).

Finite maps and bags 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 (§5.3), *i.e.* shapes with no structure other than containing some labels. If we ignore the labels, the same implementation gives us *bags*, also known as multisets. Furthermore, we can directly model multiple finite map implementations.

Value-level sharing Structures with shared labels can be used to model (value-level) *sharing* (§5.4). For example, we can superimpose both a binary tree and a list structure on some data, as shown in Figure 2. This leads to modular and generic implementations of familiar classes of operations such as filtering and folding (§6).

Memory allocation and layout Labels function as pointers, allowing us to model issues of memory allocation and layout (§7). Moreover, we can easily "swap out" one model for another, allowing us to model the particular features we care about.

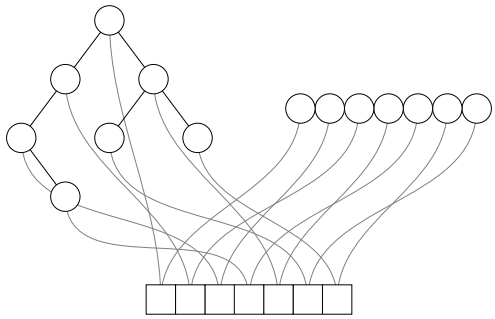

Figure 2: Superimposing a tree and a list on shared data

1.1 Contributions

Though our work bears similarities to previous approaches that separate shapes and data [Banger and Skillicorn, 1993, Jay and Cockett, 1994, Hoogendijk and De Moor, 2000, Abbott et al., 2003a], there is a key difference, directly inspired by the theory of species. Whereas previous approaches use a fixed, canonical set of labels (or left the labels entirely implicit), species naturally lead us to work directly with the labels, giving them a much more prominent role. Bringing the mediating labels to the fore in this way is, to our knowledge, novel, and leads to some interesting benefits.

Furthermore, species are defined over *finite* sets of labels. In a classical setting, while finiteness is a crucial part of the definition, it is an otherwise fairly implicit feature of the actual theory. Combinatorialists do not need to remind themselves of this finiteness condition, as it is a pervasive axiom that you can only "count" finite collections of objects. When ported to a constructive setting, however, the notion of finiteness takes on nontrivial computational content and significance. In particular, we are naturally led to work up to computationally relevant *equivalences* on labels. Working up to equivalence in this way confers additional expressive power, allowing us to model efficient label operations (*e.g.* partition) without copying. This is also one of the key ingredients in modeling memory layout and allocation (§7).

In more detail, our contributions are as follows:

- We describe a "port" of combinatorial species from set theory to constructive type theory (§3.2), making the theory more directly applicable in a programming context, more accessible to functional programmers, and incidentally illuminating some new features of the theory.
- We define a generic framework for *labelled types* on top of this basis (§4).
- We show how to unify "implicitly labelled" structures (such as algebraic data types) and "explicitly labelled" structures (such as vectors and finite maps) under the same framework.
- We show how to explicitly model and program with compositions of container shapes.
- We model value-level *sharing* via shared labels (§5.4)—in contrast, this is not possible if every structure has a fixed set of canonical labels.
- We give examples of programming with labelled types, focusing particularly on the ways that explicit sharing enables more modular, generic implementations.
- Labels share some of the properties of memory addresses, *i.e.* pointers, and taking this analogy seriously lets us reason about memory allocation and layout for stored data structures (§7).

We have an implementation of these ideas in Haskell, available at `http://github.com/byorgey/labelled-structures`.

It is worth mentioning that in previous work [Carette and Uszkay, 2008, Yorgey, 2010] we conjectured that the benefits of the theory of species would lie primarily in its ability to describe data types with *symmetry* (*i.e.* quotient types [Hofmann, 1995, Abbott et al., 2004]). That promise has not gone away; but we now believe that a solid understanding of labelled structures is a prerequisite for tackling symmetry. In retrospect, this is unsurprising, since introducing species as explicitly labelled objects was one of Joyal's great insights, allowing the encoding of symmetric and unlabelled structures via equivalence classes of labelled ones.

2 Theoretical setting

We have found it convenient to work within *homotopy type theory* (HoTT). However, we do not need much complex machinery from the theory, and simply summarize the most important ideas and notation here; interested readers should consult the HoTT book [Univalent Foundations Program, 2013] for more details. Everything in this paper could be formalized in most any standard constructive type theory; we choose to work in HoTT because of its emphasis on equality and isomorphism, which meshes well with the theory of species. In fact, it seems likely that there are deeper connections between the two theories, but exploring these connections is left to future work.

The concept of *finiteness* plays a central (but implicit) role in the theory of combinatorial species, primarily through the pervasive use of generating functions. As it remains important in our setting, we give the precise definition we use, seeing as there are multiple constructive interpretations of finiteness.

2.1 A fragment of homotopy type theory

The type theory we work with is equipped with an empty type $\bot$, a unit type $\top$(with inhabitant $\star$), coproducts (with constructors inl and inr), dependent pairs (with projections outl and outr), dependent functions, a hierarchy of type universes $\mathscr{U}_0$, $\mathscr{U}_1$, $\mathscr{U}_2\ldots$ (we usually omit the subscripts), and a notion of propositional equality. The theory also allows inductive definitions. We use $\mathbb{N} : \mathscr{U}_0$ to denote the type of natural numbers, $\mathsf{Fin} : \mathbb{N} \to \mathscr{U}_0$ the usual indexed type of canonical finite sets, and $[-] : \mathscr{U} \to \mathscr{U}$ the inductive type of polymorphic lists, with constructors $[\,] : \prod_{A:\mathscr{U}} [A]$ and $- :: - : \prod_{A:\mathscr{U}} A \to [A] \to [A]$.

Instead of writing the traditional $\sum_{x:A} B(x)$ for the type of dependent pairs and $\prod_{x:A} B(x)$ for dependent functions, we will often use the Agda-like [Norell, 2007] notations $(x : A) \times B(x)$ and $(x : A) \to B(x)$, respectively (though we still occasionally use Σ and Π for emphasis). 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. Also, to reduce clutter, we sometimes make use of implicit quantification: free type variables in a type—like A and B in $A \times (B \to \mathbb{N})$—are implicitly universally quantified, like $(A : \mathscr{U}) \to (B : \mathscr{U}) \to A \times (B \to \mathbb{N})$.

$A \simeq B$ is the type of *equivalences* between A and B; intuitively, an equivalence is a pair of inverse functions $f : A \to B$ and $g : B \to A$.[2] We overload the notations id and $\circ$ to denote the identity equivalence and equivalence composition respectively; we also allow equivalences of type $A \simeq B$ to be implicitly used as functions $A \to B$ where it does not cause confusion. We use the notation $\rightleftharpoons$ for constructing equivalences from a pair of functions. That is, if $f : A \to B$ and $g : B \to A$ are inverse, then $f \rightleftharpoons g : A \simeq B$; the proof that f and g are inverse is left implicit.

[2] The precise details are more subtle [Univalent Foundations Program, 2013, chap. 4], but unimportant for our purposes.

A few remarks about propositional equality are also in order. First, the structure of the type theory guarantees that functions are always functorial with respect to equality, that is, if $e : x = y$ and f is a function of an appropriate type, then $f(x) = f(y)$. Given e we also have $P(x) \to P(y)$ for any type family P, called the *transport* of $P(x)$ along e. Finally, a consequence of the *univalence axiom* is that an equivalence $A \simeq B$ can be converted to the propositional equality $A = B$ (and vice versa). The intuitive idea is to formally encode the common mathematical practice of treating isomorphic things as identical. It is important to keep in mind that an equality $e : A = B$ can thus have nontrivial computational content. In other words, $A = B$ means not that A and B are identical, but merely that they can be used interchangeably—and moreover, interchanging them may require some work, computationally speaking.

2.2 Finiteness

There are many possible constructive interpretations of finiteness [nLab, 2013]; we adopt the simplest: a finite set is one which is in bijection to a canonical set of a known size. That is,

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

Other constructive characterizations of finiteness may have roles to play as well, but we leave exploring them to future work.

It is tempting to use mechanisms for implicit evidence, such as Haskell's *type class* mechanism, to record the finiteness of types. That is, we could imagine defining a type class

```
class IsFinite a where
  isFinite :: Finite a
```

The idea is that the statement "the type A is finite" translates to "A is an instance of the IsFinite class". However, this is not what we want. An instance IsFinite A corresponds to a *canonical choice* of an inhabitant of Finite$A$, but inhabitants of Finite$A$ have nontrivial computational content; it really matters *which* inhabitant we have. Thus, instead of simply passing around types and requiring them to have an implicit, canonical finiteness proof, we will in general pass around types *together with* some specific finiteness proof. We can encapsulate this by defining

$$\mathscr{U}_{\mathrm{Fin}} :\equiv (A : \mathscr{U}) \times \mathsf{Finite}A$$

as the universe of finite types. We use $\lfloor - \rfloor : \mathscr{U}_{\mathrm{Fin}} \to \mathscr{U}$ to project out the underlying type from a finite type, forgetting the finiteness evidence. We also use $\lceil \mathsf{Fin}\, n \rceil : \mathscr{U}_{\mathrm{Fin}}$ to denote the type $\mathsf{Fin}\, n : \mathscr{U}$ paired with the identity equivalence.

It is not hard to see that the size of a finite type is determined uniquely. That is, if (n_1, e_1) and $(n_2, e_2) : \mathsf{Finite}A$ are any two witnesses that A is finite, then $n_1 = n_2$. As proof, note that $e_2^{-1} \circ e_1 : \mathsf{Fin}\, n_1 \simeq \mathsf{Fin}\, n_2$, from which we can derive $n_1 = n_2$ by double induction. In a slight abuse of notation, we write $|A|$ to denote this size. Computationally, this corresponds to applying outl to some finiteness proof; but since it does not matter which proof we use, we simply leave it implicit, being careful to only use $|-|$ in a context where a suitable finiteness proof can be obtained. We also write $|L|$ when $L : \mathscr{U}_{\mathrm{Fin}}$, to denote the projection of the natural number size stored in L.

As a final remark, we note that an inhabitant of Finite$A$ contains quite a lot of information, much more than one usually means by saying "A is finite". For example, it encodes a total order and decidable equality on A, by transferring these properties along the equivalence from $\mathsf{Fin}\, n$. This is often useful, but

occasionally it gets us into trouble (§5.3). It may be that the right evidence for the finiteness of A is not $(n : \mathbb{N}) \times (\mathsf{Fin}\, n \simeq A)$ but the *propositional truncation*

$$(n : \mathbb{N}) \times \| \mathsf{Fin}\, n \simeq A \|,$$

or something like it [Univalent Foundations Program, 2013, sect. 3.7]. In any case, we are reasonably certain that a complete story of labelled structures with symmetries will require a more nuanced conception of evidence for finiteness; we leave this to future work.

3 Combinatorial Species

Our theory of labelled structures is inspired by, and directly based upon, the theory of *combinatorial species* [Joyal, 1981]. We give a brief introduction to it here; the reader interested in a fuller treatment should consult Bergeron et al. [1998]. Functional programmers may wish to start with Yorgey [2010].

3.1 Species, set-theoretically

We begin with a standard set-theoretic definition of species (essentially what one finds in Bergeron et al. [1998], but with slightly different terminology). We will upgrade to a *type*-theoretic definition in §3.2, but it is worth seeing both definitions and the relationship between them.

Definition 1. A *species* is a pair of mappings, both called F, that

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

additionally 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 $F : \mathbb{B} \to \mathsf{Set}$, where $\mathbb{B}$ is the category of finite sets whose morphisms are bijections, and Set is the usual category of sets whose morphisms are arbitrary (total) functions. Note that we could have chosen FinSet as codomain with very few changes, but Set is now customary.

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".

In the existing 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, in our terminology, have data associated to 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 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; we explore this in §5. For the moment, we turn to a constructive, type-theoretic definition of species.

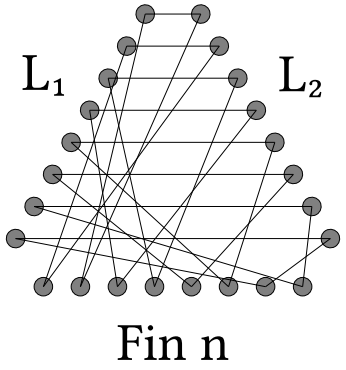

Figure 3: An equivalence between constructively finite types contains only triangles

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 necessary to first "port" the definition from set theory to constructive type theory. In fact, doing so results in a greatly simplified definition: we merely define

$$\mathsf{Species} :\equiv \mathscr{U}_{\mathrm{Fin}} \to \mathscr{U}.$$

The rest of the definition comes "for free" from the structure of our type theory! In particular, we have

$$\mathsf{relabel} : (F : \mathsf{Species}) \to (L_1 = L_2) \to (F\ L_1 \to F\ L_2)$$

via transport, where relabel automatically respects identity and composition. This is one of the great strengths of type theory as a foundation for mathematics: everything is functorial, natural, continuous, *etc.*, and we do not have to waste time ruling out bizarre constructions which violate these obvious and desirable properties, or proving that our constructions do satisfy them.

It is important to note that an equality $L_1 = L_2$ between constructively finite types $L_1, L_2 : \mathscr{U}_{\mathrm{Fin}}$, as required by relabel, contains more than first meets the eye. Since

$$\mathscr{U}_{\mathrm{Fin}} :\equiv (L : \mathscr{U}) \times (n : \mathbb{N}) \times (\mathsf{Fin}\, n \simeq L),$$

such equalities contain not just an equality $\lfloor L_1 \rfloor = \lfloor L_2 \rfloor$ between the underlying types (typically constructed from an equivalence $\lfloor L_1 \rfloor \simeq \lfloor L_2 \rfloor$ via univalence), but also an equality between their sizes, and a second-order equality-between-equivalences requiring the types to be isomorphic to $\mathsf{Fin}\, n$ "in the same way", that is, to yield the same equivalence with $\mathsf{Fin}\, n$ after mapping from one to the other. The situation can be pictured as shown in Figure 3, where the diagram necessarily contains only triangles: corresponding elements of L_1 and L_2 on the sides correspond to the same element of $\mathsf{Fin}\, n$ on the bottom row. Intuitively, this means that if $L_1, L_2 : \mathscr{U}_{\mathrm{Fin}}$, an equality $L_1 = L_2$ cannot contain "too much" information: it only tells us how the underlying types of L_1 and L_2 relate, preserving the fact that they can both be put in correspondence with $\mathsf{Fin}\, n$ for some n. In particular, it cannot also encode a nontrivial permutation on $\mathsf{Fin}\, n$.

4 Labelled structures and mappings

To recover a notion of *data structure*, we must pair species, *i.e.* labelled shapes, with mappings from labels to data. Formally, we define families of labelled structures by

$$\langle - \rangle_-(-) : \mathsf{Species} \to \mathscr{U} \to \mathscr{U} \to \mathscr{U}$$
$$\langle F \rangle_L(A) = F\ L \times (L \Rightarrow A)$$

where $- \Rightarrow - : \mathscr{U}_{\mathrm{Fin}} \to \mathscr{U} \to \mathscr{U}$ constructs the type of *mappings*. We need not pin down a particular implementation for $- \Rightarrow -$; we require only that it come equipped with the following operations:

$$\begin{aligned}
\mathit{allocate} &: (\lfloor L \rfloor \to A) \to (L \Rightarrow A) \\
\mathit{index} &: (L \Rightarrow A) \to \lfloor L \rfloor \to A \\
\mathit{map} &: (A \to B) \to (L \Rightarrow A) \to (L \Rightarrow B) \\
\mathit{reindex} &: (L' = L) \to (L \Rightarrow A) \to (L' \Rightarrow A) \\
\mathit{zipWith} &: (A \to B \to C) \to (L \Rightarrow A) \to (L \Rightarrow B) \to (L \Rightarrow C) \\
\mathit{append} &: (\lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to (L_1 \Rightarrow A) \to (L_2 \Rightarrow A) \to (L \Rightarrow A) \\
\mathit{concat} &: (\lfloor L_1 \rfloor \times \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to (L_1 \Rightarrow (L_2 \Rightarrow A)) \to (L \Rightarrow A)
\end{aligned}$$

One could also imagine requiring other operations like $\mathit{replace} : L \to A \to (L \Rightarrow A) \to A \times (L \Rightarrow A)$, but these are the operations we need in the context of this paper. The semantics of these operations can be specified by various laws (for example, *allocate* and *index* are inverse; *index* and *reindex* commute appropriately with the other operations; and so on). For now, we will content ourselves with some informal descriptions of the semantics.

- First, *allocate* is the sole means of constructing $(L \Rightarrow A)$ values, taking a function $\lfloor L \rfloor \to A$ as a specification of the mapping. Note that since $L : \mathscr{U}_{\mathrm{Fin}}$, implementations of *allocate* also have access to an equivalence $\lfloor L \rfloor \simeq \mathsf{Fin}\,|L|$. Intuitively, this is important because allocation may require some intensional knowledge about the type L. For example, as explained in §7, we may implement $(L \Rightarrow A)$ using a vector of A values; allocating such a vector requires knowing the size of L.
- *index* allows looking up data by label.
- *map* ensures that $(L \Rightarrow -)$ is functorial.
- $\mathit{reindex} : (L' = L) \to (L \Rightarrow A) \to (L' \Rightarrow A)$ expresses the functoriality of $(- \Rightarrow A)$: we can change from one type of labels to another by specifying an equivalence between them. *map* and *reindex* together thus give $(- \Rightarrow -)$ the structure of a profunctor.
- *zipWith* gives us a way to combine the contents of two mappings labelwise.
- *append* and *concat* are "structural" operations, allowing us to combine two mappings into one, or collapse nested mappings, respectively. One might naïvely expect them to have types like $\mathit{append} : (L_1 \Rightarrow A) \to (L_2 \Rightarrow A) \to ((L_1 + L_2) \Rightarrow A)$, but this is not even well-typed: L_1 and L_2 are elements of $\mathscr{U}_{\mathrm{Fin}}$, not $\mathscr{U}$, and there is no obvious way to lift $+$ to operate over $\mathscr{U}_{\mathrm{Fin}}$. In particular, there are many possible ways to combine equivalences $\lfloor L_1 \rfloor \simeq \mathsf{Fin}\,|L_1|$ and $\lfloor L_2 \rfloor \simeq \mathsf{Fin}\,|L_2|$ into an equivalence $\lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \mathsf{Fin}\,(|L_1| + |L_2|)$, and no canonical way to pick one. (Should the elements of L_1 come first? L_2? Should they be interleaved somehow?) Intuitively, the extra argument to *append* provides precisely this missing information (and similarly for *concat*).

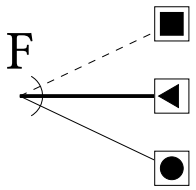

Figure 4: Schematic of a typical $(F\ L)$-structure

We can give a particularly simple implementation with $(L \Rightarrow A) :\equiv \lfloor L \rfloor \to A$, presented here using Haskell-like notation:

$$\begin{aligned}
&\mathit{allocate} && = \mathit{id}\\
&\mathit{index} && = \mathit{id}\\
&\mathit{map} && = (\circ)\\
&\mathit{reindex}\ i\ f && = f \circ i\\
&\mathit{zipWith}\ z\ f\ g && = \lambda l \to z\ (f\ l)\ (g\ l)\\
&\mathit{append}\ e\ f\ g && = \mathit{either}\ f\ g \circ e^{-1}\\
&\mathit{concat}\ e\ f && = \mathit{curry}\ f \circ e^{-1}
\end{aligned}$$

Note that this implementation of *allocate* does not take into account the finiteness of L at all. In §7 we explore a more interesting implementation which does make use of the finiteness of L.

5 The algebra of species and labelled structures

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. In this section we explain such a theory. At its core, this theory is not new; what is new is porting it to a constructive setting, and the introduction and elimination forms for labelled structures built on top of these species.

5.1 Algebraic data types

We begin by exhibiting species, *i.e.* labelled shapes, which correspond to familiar algebraic data types. As a visual aid, throughout the following section we will use schematic illustrations as typified in Figure 4. The edges of the tree visually represent different labels; the leaves of the tree represent data associated with those labels. The root of the tree shows the species shape applied to the labels (in this case, F).

Zero The *zero* or *empty* species, denoted 0, is the unique species with no shapes whatsoever, defined by

$$0\ L :\equiv \bot.$$

One The *one* or *unit* species, denoted 1, is the species with a single shape of size 0 (that is, containing no labels), defined by

$$1\ L :\equiv (\bot \simeq \lfloor L \rfloor).$$

That is, a 1-shape with labels drawn from L consists solely of a proof that L is empty.[3] (Note that there is at most one such proof.)

There is a trivial introduction form for 1, also denoted 1, which creates a 1-shape using the canonical label set $\lceil \mathsf{Fin}\ 0 \rceil : \mathscr{U}_{\mathrm{Fin}}$, that is,

$$1 : 1\ \lceil \mathsf{Fin}\ 0 \rceil .$$

We also have an introduction form for labelled 1-structures,

$$\langle 1 \rangle : \langle 1 \rangle_{\lceil \mathsf{Fin}\, 0 \rceil}(A).$$

Note that the usual set-theoretic definition is

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

However, this is confusing to the typical type theorist. First, it seems strange that the definition of 1 gets to "look at" L, since species are supposed to be functorial. In fact, the definition does not violate functoriality—because it only "looks at" the size of L, not its contents, and bijections preserve size—but this is not manifestly obvious. It's also strange that we have to pull some arbitrary one-element set out of thin air.

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

$$\mathsf{X}\ L :\equiv (\top \simeq \lfloor L \rfloor),$$

that is, an X-shape is just a proof that L has size 1. Again, there is at most one such proof. Unlike 1, we may also think of an X-shape as "containing" a single label of type L, which we may recover by applying the equivalence to $\star$.

X-shapes, as with 1, have a trivial introduction form,

$$\mathsf{x} : \mathsf{X}\ \lceil \mathsf{Fin}\ 1 \rceil .$$

To introduce an X-structure, one must provide the single value of type A which is to be stored in the single location:

$$\langle \mathsf{x} \rangle : A \to \langle \mathsf{X} \rangle_{\lceil \mathsf{Fin}\, 1 \rceil}(A).$$

Combinatorialists often regard the species X as a "variable". Roughly speaking, this can be justified by thinking of the inhabitant of L as the actual variable, and the species X then *represents* the action of subtituting an arbitrary value for that label in the structure. In that sense X does act operationally as a variable. However, X does *not* act like a binder.

Sum Given two species F and G, we may form their sum. We use $\boxplus$ to denote the sum of two species to distinguish it from $+$, which denotes a sum of types. The definition is straightforward:

$$(F \boxplus G)\ L :\equiv F\ L + G\ L.$$

That is, a labelled $(F \boxplus G)$-shape is either a labelled F-shape or a labelled G-shape (Figure 5).

[3] Yeh [1986] mentions something equivalent, namely, that the unit species can be defined as the hom-functor $\mathbb{B}(\varnothing, -)$, though he certainly does not have constructive type theory in mind.

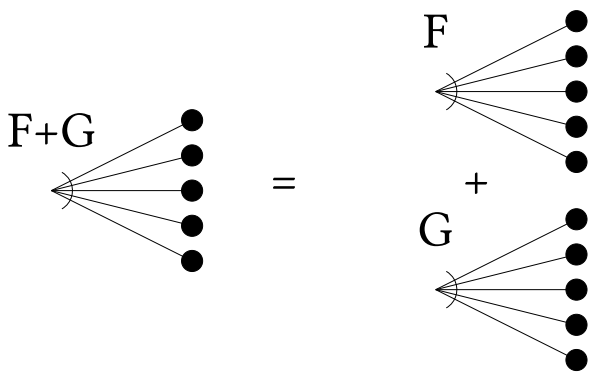

Figure 5: Species sum

As the reader is invited to check, $(\boxplus, 0)$ forms a commutative monoid structure on species, up to species isomorphism. That is, one can define equivalences

$$\begin{aligned}
\mathsf{plusAssoc} &: (F \boxplus G) \boxplus H \simeq F \boxplus (G \boxplus H) \\
\mathsf{zeroPlusL} &: 0 \boxplus F \simeq F \\
\mathsf{plusComm} &: F \boxplus G \simeq G \boxplus F
\end{aligned}$$

We remark that unfolding definitions, an equivalence $F \simeq G$ between two Species is seen to be a natural isomorphism between F and G as functors; this is precisely the usual definition of isomorphism between species.

As expected, there are two introduction forms for $(F \boxplus G)$-shapes and -structures:

$$\begin{aligned}
&\mathsf{inl} : F\ L \to (F \boxplus G)\ L \\
&\mathsf{inr} : G\ L \to (F \boxplus G)\ L \\
&\langle \mathsf{inl} \rangle : \langle F \rangle_L(A) \to \langle F \boxplus G \rangle_L(A) \\
&\langle \mathsf{inr} \rangle : \langle G \rangle_L(A) \to \langle F \boxplus G \rangle_L(A)
\end{aligned}$$

As a simple example, the species $1 \boxplus \mathsf{X}$ corresponds to the familiar Maybe type from Haskell, with $\langle \mathsf{inl} \rangle \langle 1 \rangle$ playing the role of Nothing and $\langle \mathsf{inr} \rangle \circ \langle \mathsf{x} \rangle$ playing the role of Just. Note that $\langle 1 \boxplus \mathsf{X} \rangle_L(A)$ is only inhabited for certain L (those of size 0 or 1), and moreover that this size restriction determines the possible structure of an inhabitant.

Product The product of two species F and G consists of paired F- and G-shapes, but with a twist: the label types L_1 and L_2 used for F and G are not necessarily the same as the label type L used for $(F \boxdot G)$. In fact, they must constitute a partition of L, in the sense that their sum is isomorphic to L (Figure 6).

$$(F \boxdot G)\ L :\equiv \sum_{L_1, L_2 : \mathscr{U}_{\mathrm{Fin}}} (\lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \times F\ L_1 \times G\ L_2$$

For comparison, in set theory the definition is usually presented as

$$(F \boxdot G)\ L = \sum_{L_1 \uplus L_2 = L} F\ L_1 \times G\ L_2$$

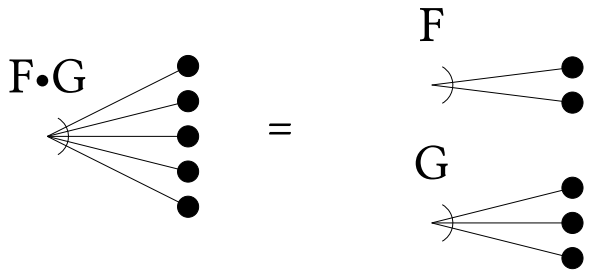

Figure 6: Species product

which is obviously similar, but lacks any computational evidence for the relationship of L_1 and L_2 to L.

The intuition behind partitioning the labels in this way is that each label represents a unique "location" which can hold a data value, so the locations in the two paired shapes should be disjoint. Another good way to gain intuition is to imagine indexing species not by label types, but by natural number sizes. Then it is easy to see that we would have

$$(F \boxdot G)_n :\equiv \sum_{n_1,n_2:\mathbb{N}} (n_1 + n_2 = n) \times F_{n_1} \times G_{n_2},$$

that is, an $(F \boxdot G)$-shape of size n consists of an F-shape of size n_1 and a G-shape of size n_2, where $n_1 + n_2 = n$. Indexing by labels is a generalization (a *categorification*, in fact) of this size-indexing scheme, where we replace natural numbers with finite types, addition with coproduct, and multiplication with product.

Finally, this definition highlights a fundamental difference between *container types* and *labelled shapes*. Given two functors representing container types, their product is defined as $(F \times G)\,A = F\,A \times G\,A$—that is, an $(F \times G)$-structure containing values of type A is a pair of an F-structure and a G-structure, both containing values of type A. On the other hand, when dealing with labels instead of data values, we have to carefully account for the way the labels are distributed between the two shapes.

One introduces a labelled $(F \boxdot G)$-shape by pairing a labelled F-shape and a labelled G-shape, using a label set isomorphic to the coproduct of the two label types:

$$\begin{aligned} &- \boxdot_- - : (\lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to F\ L_1 \to G\ L_2 \to (F \boxdot G)\ L \\ &- \langle \boxdot \rangle_- - : (\lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to \langle F \rangle_{L_1}(A) \to \langle G \rangle_{L_2}(A) \to \langle F \boxdot G \rangle_L(A) \end{aligned}$$

We use here (and for the rest of the paper) the notational convention that the isomorphism arguments are given first, but are written as subscripts in mathematical notation.

As an example, we may now encode the standard algebraic data type of lists, represented by the inductively-defined species satisfying $\mathsf{List} \simeq 1 \boxplus (\mathsf{X} \boxdot \mathsf{List})$ (for convenience, in what follows we leave implicit the constructor witnessing this equivalence). We can then define the usual constructors nil and cons as follows:

$$\begin{aligned} &\mathsf{nil} : \langle \mathsf{List} \rangle_{\mathsf{Fin}\,0}(A) \\ &\mathsf{nil} :\equiv \langle \mathsf{inl} \rangle\, \langle 1 \rangle \\ &\mathsf{cons} : A \to \langle \mathsf{List} \rangle_L(A) \to (\mathsf{Fin}\,1 + \lfloor L \rfloor \simeq \lfloor L' \rfloor) \to \langle \mathsf{List} \rangle_{L'}(A) \\ &\mathsf{cons}\ a\ (\mathit{shape}, \mathit{elts})\ e :\equiv (\mathsf{inr}\ (\mathsf{x} \boxdot_e \mathit{shape}), \mathit{append}\ e\ (\mathit{allocate}\ (\lambda x.a))\ \mathit{elts}) \end{aligned}$$

The interesting thing to note here is the extra equivalence passed as an argument to cons, specifying the precise way in which the old label type augmented with an extra distinguished label is isomorphic to the new label type. Again, one might intuitively expect something like

$$\mathsf{cons} : A \to \langle \mathsf{List} \rangle_L(A) \to \langle \mathsf{List} \rangle_{\lceil \mathsf{Fin}\, 1 \rceil + L}(A),$$

but this is nonsensical: we cannot take the coproduct of two elements of $\mathscr{U}_{\mathrm{Fin}}$, as it is underspecified. For implementations of $- \Rightarrow\!\!\!\!\!\!\Rightarrow -$ which make use of the equivalence to $\mathsf{Fin}\, n$ stored in $\mathscr{U}_{\mathrm{Fin}}$ values (we give an example of one such implementation in §7), the extra equivalence given as an argument to cons allows us to influence the particular way in which the list elements are stored in memory. For lists, this is not very interesting, and we would typically use a variant $\mathsf{cons}' : A \to \langle \mathsf{List} \rangle_L(A) \to \langle \mathsf{List} \rangle_{\mathsf{inc}(L)}(A)$ making use of a canonical construction $\mathsf{inc}(-) : \mathscr{U}_{\mathrm{Fin}} \to \mathscr{U}_{\mathrm{Fin}}$ with $\mathsf{Fin}\, 1 + \lfloor L \rfloor \simeq \lfloor \mathsf{inc}(L) \rfloor$.

5.2 Composition

We may also define the *composition* of two species. Intuitively, $(F \boxdot G)$-shapes consist of a single top-level F-shape, which itself contains labelled G-shapes in place of the usual labels, as illustrated in Figure 7. Set-theoretically, we have

$$(F \boxdot G)\, L = \sum_{\pi \in \mathsf{Par}(L)} F\, \pi \times \prod_{L' \in \pi} G\, L',$$

where $\mathsf{Par}(L)$ denotes the set of all partitions of L into nonempty subsets. Note how this uses the elements of the partition π itself as labels on the F-structure. A more natural type-theoretic encoding is to use an arbitrary type of F-labels, and then store a mapping from these labels to the label types used for the G-shapes. Additionally, we store an equivalence witnessing the fact that the G-labels constitute a partition of the overall label type. Formally,

$$(F \boxdot G)\, L :\equiv \sum_{L_F : \mathscr{U}} F\, L_F \times (Ls_G : L_F \Rightarrow\!\!\!\!\!\!\Rightarrow \mathscr{U}_{\mathrm{Fin}}) \times (\lfloor L \rfloor \simeq \mathit{sum}\ (\mathit{map}\ \lfloor - \rfloor\ Ls_G)) \times \mathit{prod}\ (\mathit{map}\ G\ Ls_G).$$

We assume functions $\mathit{sum}, \mathit{prod} : (J \Rightarrow\!\!\!\!\!\!\Rightarrow \mathscr{U}) \to \mathscr{U}$ which compute the sum and product, respectively, of all the types in the range of a mapping. Note how the presence of explicit labels and mappings work together to make this possible.

Composition ($\boxdot$), unlike sum ($\boxplus$) and product ($\boxdot$), is not commutative[4]: an F-shape of G-shapes is quite different from a G-shape of F-shapes. It is, however, still associative, and in fact $(\boxdot, \mathsf{X})$ forms a monoid.

The space of introduction forms for composition structures is nontrivial. We will not separately consider introduction forms for composition shapes, but study introduction forms for composition structures directly. At the simplest end of the spectrum, we can define an operator $\langle \otimes \rangle$ ("cross") as a sort of cartesian product of structures, copying the provided G structure into every location of the F structure and pairing up both their labels and data (Figure 8):

$$- \langle \otimes \rangle_{-} - : (\lfloor L_1 \rfloor \times \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to \langle F \rangle_{L_1}(A) \to \langle G \rangle_{L_2}(B) \to \langle F \boxdot G \rangle_L(A \times B)$$

The isomorphism argument is notated as a subscript to $\langle \otimes \rangle$. Of course, this is far from being a general

[4] Interestingly, a relatively recent paper of Maia and Méndez [2008] introduces a new monoidal structure on species, the *arithmetic product*, which according to one intuition represents a sort of "commutative composition". Incorporating this into our framework will, we conjecture, have important applications to multidimensional arrays.

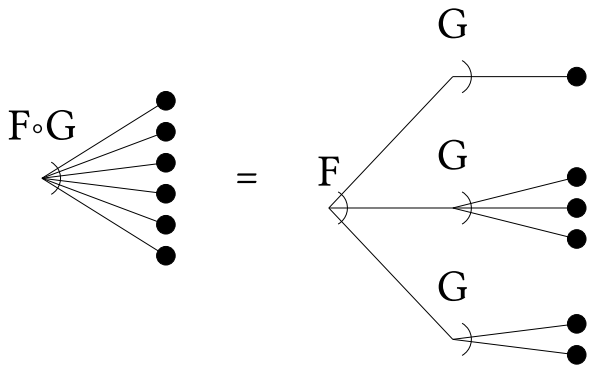

Figure 7: Species composition

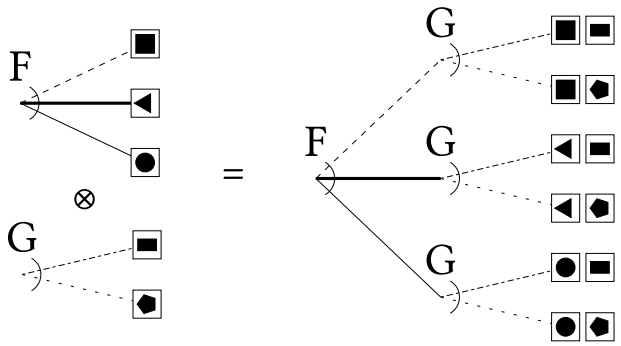

Figure 8: Constructing a composition with $\langle\otimes\rangle$

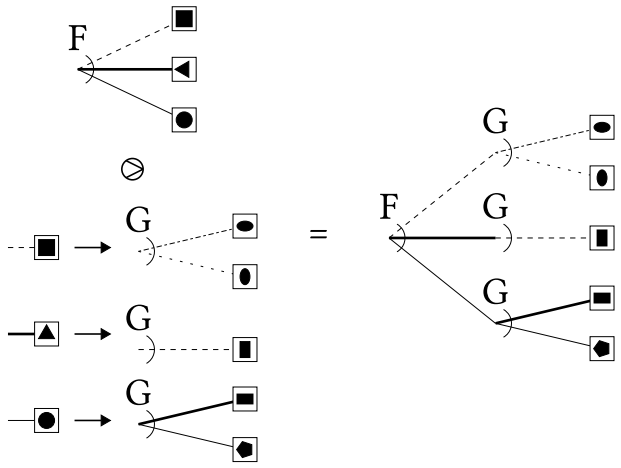

Figure 9: Constructing a composition with $\langle \circledcirc \rangle$

introduction form for $\boxdot$, since it only allows us to construct composition structures of a special form, but is convenient when it suffices.

We also have $\langle \circledast \rangle$ ("ap"), with type

$$- \langle \circledast \rangle_{-} - : (\lfloor L_1 \rfloor \times \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to \langle F \rangle_{L_1} (A \to B) \to \langle G \rangle_{L_2} (A) \to \langle F \boxdot G \rangle_L (B).$$

$\langle \circledast \rangle$ is equivalent in power to $\langle \otimes \rangle$: in particular, $x \ \langle \otimes \rangle \ y = (map \ (,) \ x) \ \langle \circledast \rangle \ y$, where $(,) : A \to B \to A \times B$ denotes the constructor for pair types, and $x \ \langle \circledast \rangle \ y = map \ eval \ (x \ \langle \otimes \rangle \ y)$, where $eval : (A \to B) \times A \to B$.

There is another introduction form for composition ($\langle \circledcirc \rangle$, "join") which is a generalization of the *join* (μ) function of a monad:

$$- \langle \circledcirc \rangle - : (\lfloor L_1 \rfloor \times \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to \langle F \rangle_{L_1} (\langle G \rangle_{L_2} (A)) \to \langle F \boxdot G \rangle_L (A)$$

$\langle \circledcirc \rangle$ takes a labelled F-structure filled with labelled G-structures, and turns it into a labelled $(F \boxdot G)$-structure.

$\langle \circledcirc \rangle$, unlike $\langle \otimes \rangle$ and $\langle \circledast \rangle$, allows constructing an $(F \boxdot G)$-structure where the G-shapes are not all the same. Note, however, that all the G-structures are restricted to use the same label set, L_1, so they still must all be equal in size.

Most generally, of course, it should be possible to compose G-structures of different shapes and sizes inside an F-structure, which is made possible by $\langle \circledcirc \rangle$ ("bind"), the last and most general introduction form for composition, which can be seen as a generalization of a monadic bind operation $(\gg\!=)$.

$$- \langle \circledcirc \rangle_{-} - : \left(\sum_{l : \lfloor L_1 \rfloor} \lfloor L_2 \ l \rfloor \right) \simeq \lfloor L \rfloor \to \langle F \rangle_{L_1} (A) \to \left(\prod_{l : L_1} A \to \langle G \rangle_{L_2 \ l} (B) \right) \to \langle F \boxdot G \rangle_L (B)$$

Here, L_2 is actually a *family* of types, indexed over L_1, so each G subshape can have a different type of labels, and hence a different size (Figure 9).

As an example using composition, we can directly encode the type of ordered, rooted n-ary trees, sometimes known as *rose trees*, as $\mathsf{R} \simeq \mathsf{X} \boxdot (\mathsf{List} \boxdot \mathsf{R})$. This corresponds to the Haskell type Rose defined as **data** $\mathsf{Rose}\ a = \mathsf{Node}\ a\ [\mathsf{Rose}\ a]$, but the composition is more explicit. The explicit use of composition

is useful when doing generation of such structures, as it allows switching of generation strategies at those points [Uszkay, 2014].

The most general type for the node constructor is complex, since it must deal with a list of subtrees all having different label types. As a compromise, we can make use of a variant type representing labelled structures with an existentially quantified label type:

$$\langle F \rangle_\bullet(A) :\equiv \sum_{L:\mathscr{U}_{\mathrm{Fin}}} \langle F \rangle_L(A)$$

Using $\langle \mathsf{R} \rangle_\bullet(A)$, we can write a constructor for R with the type

$$\mathit{nodeE} : A \to [\langle \mathsf{R} \rangle_\bullet(A)] \to \langle \mathsf{R} \rangle_\bullet(A)$$

which converts the list into a labelled List-structure of R-structures, uses the join operator $\langle \textcircled{\scriptsize V} \rangle$ to collapse it into a single $(\mathsf{List} \boxdot \mathsf{R})$-structure, and finally prepends the A value.

5.3 Sets, bags, and maps

The species of *sets*, denoted E, is defined by

$$\mathsf{E}\, L :\equiv \top.$$

That is, there is a single E-shape for every label type. Intuitively, E-shapes impose no structure whatsoever; that is, a labelled E-shape can be thought of simply as a *set* of labels. This is the first example of a species with nontrivial *symmetry*, *i.e.* which is invariant under some nontrivial permutations of the labels. In fact, E is invariant under *all* label permutations. It is thus the "quintessential" symmetric species. Anecdotally, introducing E alone seems to go a very long way towards enabling the sorts of symmetric structures that actually arise in programming; we give some examples below. (Adding the species C of *cycles* covers almost all the rest, but we do not consider cycles in this paper.)

Note that if E-shapes are sets, then labelled E-structures (E-shapes plus mappings from labels to data) are *bags*, or *multisets*: any particular data element may occur multiple times (each time associated with a different, unique label), and the collection of data elements has no structure imposed on it.

E-shapes have a trivial introduction form, $\mathsf{e} : \mathsf{E}\, L$, along with a corresponding introduction form for E-structures which simply requires the mapping from labels to values:

$$\langle \mathsf{e} \rangle : (\lfloor L \rfloor \to A) \to \langle \mathsf{E} \rangle_L(A)$$
$$\langle \mathsf{e} \rangle\, f = (\star, \mathit{allocate}\ f)$$

Eliminating E-structures, on the other hand, is somewhat problematic. At the end of the day, the data need to be stored in some particular order in memory, but we do not want to allow any such ordering to be observed. We can require E-structures to be eliminated using a commutative monoid, but if an eliminator has access to the finiteness proof for the label type, it can still observe a linear ordering on the labels and hence on the data elements as well. As a "solution", we could forbid eliminators from being able to observe labels, but this largely defeats the purpose of having explicitly labelled structures in the first place. In the end, this is a problem needing more study, likely requiring a rethinking of the way we represent evidence of finiteness.

Leaving the problems with the mechanics of elimination aside for the moment, we highlight here a few particular examples of the use of E:

Rooted, unordered trees If we replace the List in the definition of rose trees with E, we obtain a species of rooted, arbitrary-arity trees where the children of each node are *unordered*:

$$\mathsf{T} \simeq \mathsf{X} \boxdot (\mathsf{E} \circledcirc \mathsf{T}).$$

Hierarchies without an ordering on sibling nodes arise quite often in practice: for example, filesystem directory hierarchies and typical company organizational charts are both of this type.

Finite maps Formally, there is no difference between bags (multisets) and finite maps: both may be represented by $\langle \mathsf{E} \rangle_L(A)$. The difference is the role played by the labels. With bags, the labels are implicit; indeed, we might wish to define $\mathsf{Bag}\ A :\equiv \sum_{L:\mathscr{U}_{\mathrm{Fin}}} \langle \mathsf{E} \rangle_L(A)$. With finite maps, on the other hand, the labels play a more explicit role.

There are many possible implementations of finite maps, with attendant performance tradeoffs. We can explicitly model different implementations with suitable implementations of $- \Mapsto -$. §7 gives one implementation, and hints at another, corresponding to finite maps stored as arrays or tries. Another common class of finite map implementations involve a balanced tree, making use of a required total ordering on the labels. It should be easy to model such implementations as well, by extending the framework in this paper to allow arbitrary extra structure on label types.

Partition We may define the species Part of *partitions* by

$$\mathsf{Part} :\equiv \mathsf{E} \boxdot \mathsf{E}.$$

$(\mathsf{Part}\ L)$-shapes consist of a (disjoint) pair of sets of labels. Thus $(\mathsf{Part}\ L)$-shapes represent *predicates* on labels, *i.e.* they are isomorphic to $\lfloor L \rfloor \to 2$. In conjunction with Cartesian product (§5.4), this allows us to generalize operations like *filter* and *partition* to arbitrary labelled structures, as described in §6.

5.4 Cartesian product

As we saw earlier, the definition of the standard product operation on species partitioned the set of labels between the two subshapes. However, there is nothing to stop us from defining a different product-like operation, known as *Cartesian product*, which does not partition the labels:

$$(F \boxtimes G)\ L = F\ L \times G\ L$$

This is the "naïve" version of product that one might initially expect. However, Cartesian product works very differently with labelled shapes. It is important to remember that a mapping $(L \Mapsto A)$ still only assigns a single A value to each label; but labels can occur twice (or more) in an $(F \times G)$-shape. This lets us *explicitly* model value-level sharing, that is, multiple parts of the same shape can all "point to" the same data. In pure functional languages such as Haskell or Agda, sharing is a (mostly) unobservable operational detail; with a labelled structure we can directly model and observe it. Figure 10 illustrates the Cartesian product of a binary tree and a list.

To introduce a Cartesian product shape, one simply pairs two shapes on the same set of labels. Introducing a Cartesian product structure is more interesting. One way to do it is to overlay an additional shape on top of an existing structure:

$$\mathsf{cprodL} : F\ L \to \langle G \rangle_L(A) \to \langle F \boxtimes G \rangle_L(A).$$

There is also a corresponding cprodR which combines an F-structure and a G-shape.

$(\boxtimes, \mathsf{E})$ forms a commutative monoid up to species isomorphism; superimposing an E-shape has no effect, since the E-shape imposes no additional structure.

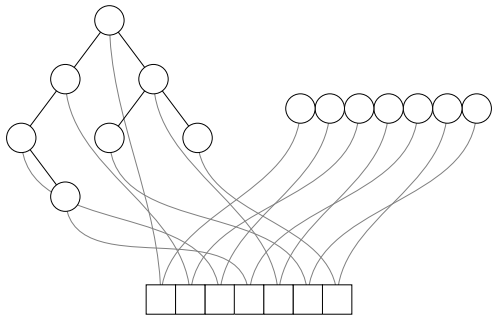

Figure 10: Superimposing a tree and a list on shared data

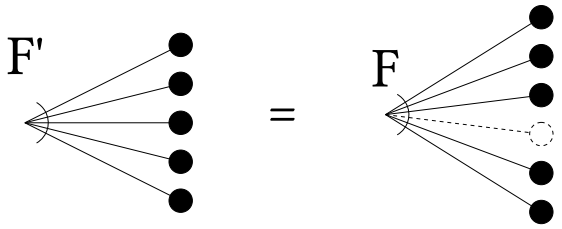

Figure 11: Species differentiation

5.5 Other operations

There are many other operations on species. We mention a few here which we have not yet explored in detail, but seem promising in a computational context.

Cardinality restriction Another important operation on species is *cardinality restriction*, which simply restricts a given species to only have shapes of certain sizes. For example, if L is the species of lists, L_3 is the species of lists with length exactly three, and $\mathsf{L}_{\geqslant 1}$ is the species of non-empty lists. We can formalize a simple version of this, for restricting only to particular sizes, as follows:

$$\mathsf{OfSize} : \mathsf{Species} \to \mathbb{N} \to \mathsf{Species}$$
$$\mathsf{OfSize}\ F\ n\ L :\equiv (|L| = n) \times F\ L$$

The introduction form for OfSize is simple enough, allowing one to observe that an existing label type has the size that it has:

$$\mathsf{sized} : \langle F \rangle_L(A) \to \langle \mathsf{OfSize}\ F\ |L| \rangle_L(A).$$

Derivative and pointing The *derivative* is a well-known operation on shapes in the functional programming community [Huet, 1997, McBride, 2001, Abbott et al., 2003b, Morris et al., 2006, McBride, 2008], and it works in exactly the way one expects on species. That is, F'-shapes consist of F-shapes with one distinguished location (a "hole") that contains no data (Figure 11). Formally, we may define

$$F'\ L :\equiv (L' : \mathscr{U}) \times (\lfloor L' \rfloor \simeq \top + \lfloor L \rfloor) \times F\ L'$$

Note that a mapping $(L \Rightarrow A)$ associates data to every label in the underlying $F\ L'$ structure but one, since $\lfloor L' \rfloor \simeq \top + \lfloor L \rfloor$.

To introduce a derivative structure, we require an input structure whose label type is already in the form $\top + L$:

$$\begin{aligned} \mathsf{d} &: (\lfloor L' \rfloor \simeq \top + \lfloor L \rfloor) \to F\ L' \to F'\ L \\ \langle \mathsf{d} \rangle &: (\lfloor L' \rfloor \simeq \top + \lfloor L \rfloor) \to \langle F \rangle_{L'}(A) \to A \times \langle F' \rangle_L(A) \end{aligned}$$

The idea behind $\langle \mathsf{d} \rangle$ is that we get back the A that used to be labelled by $\top$, paired with a derivative structure with that value missing.

A related operation is that of *pointing*. A pointed F-shape is an F-shape with a particular label distinguished. Formally,

$$F^\bullet\ L :\equiv L \times F\ L.$$

Introducing a pointed structure simply requires specifying which label should be pointed:

$$\begin{aligned} \mathsf{p} &: L \to F\ L \to F^\bullet\ L \\ \langle \mathsf{p} \rangle &: L \to \langle F \rangle_L(A) \to \langle F^\bullet \rangle_L(A) \end{aligned}$$

The relationship bewteen pointing and derivative is given by the equivalence

$$F^\bullet \simeq \mathsf{X} \boxdot F'.$$

The right-to-left direction is straightforward to implement, requiring only some relabelling. The left-to-right direction, on the other hand, requires modelling an analogue of "subtraction" on types: the given label type L must be decomposed as "$(L - l) + l$" for some $l : L$, that is,

$$L \simeq \left(\sum_{l' : L} l' \neq l \right) + \left(\sum_{l' : L} l' = l \right).$$

Functorial composition It is worth mentioning the operation of *functorial composition*, which set-theoretically is defined as the "naïve" composition

$$(F \boxdot G)\ L :\equiv F\ (G\ L).$$

Just as with Cartesian product, functorial composition allows encoding structures with sharing—for example, the species of simple, undirected graphs can be specified as

$$\mathscr{G} :\equiv (\mathsf{E} \boxdot \mathsf{E}) \boxdot (\mathsf{X}^2 \boxdot \mathsf{E}),$$

describing a graph as a subset $(\mathsf{E} \boxdot \mathsf{E})$ of all $(\boxdot)$ ordered pairs chosen from the complete set of vertex labels $(\mathsf{X}^2 \boxdot \mathsf{E})$.

However, functorial composition mixes up labels and shapes in the most peculiar way—and while this is perfectly workable in an untyped, set-theoretic setting, we do not yet know how to interpret it in a typed, constructive way.

6 Programming with Labelled Structures

This section gives some examples of programming with labelled structures. In particular, there are a number of standard functions on vectors, lists, sets and bags, finite maps, and similar structures, which we can generalize to work over all labelled structures. We present a small sampling here, using type-theoretic notation; these and additional examples, concretely implemented in Haskell, are available from `http://github.com/byorgey/labelled-structures`.

6.1 Partition

We begin with a *partition* function, implemented using the species Part of partitions (§5.3) and Cartesian product (§5.4). This turns out to be a key component of some of the later examples as well. The idea is to use a predicate on data to divide the labels into two disjoint sets, and to then *superimpose* (via Cartesian product) a second structure on the old, recording this new information about the labels. The original structure is not modified in any way.

$$\begin{aligned}
&\mathit{partition} : \langle F \rangle_L(A) \to (A \to 2) \to \langle F \boxtimes \mathsf{Part} \rangle_L(A) \\
&\mathit{partition}\ (f, \mathit{elts})\ p :\equiv ((f, \star \boxdot_e \star), \mathit{elts}) \\
&\quad \textbf{where} \\
&\qquad e : \left(\sum_{l : \lfloor L \rfloor} p\ (\mathit{elts}\,!\,l) = \mathsf{True} \right) + \left(\sum_{l : \lfloor L \rfloor} p\ (\mathit{elts}\,!\,l) = \mathsf{False} \right) \simeq \lfloor L \rfloor \\
&\qquad e = \ldots
\end{aligned}$$

We omit the implementation of the equivalence e in the interest of clarity; the details are fiddly but there is nothing particularly interesting about it. The new superimposed Part structure contains nothing but this equivalence e (the two E-shapes are just $\star$).

At this point, we might want to actually take this information and "extract" the result (in the usual meaning of splitting the structure into two distinct pieces). To do that we will first have to explore the idea of eliminators for labelled structures.

6.2 Eliminators

We use a framework of eliminators of type $(\langle F \rangle_L(A) \rightsquigarrow R)$, which can be "run" with

$$\mathit{runElim} : (\langle F \rangle_L(A) \rightsquigarrow R) \to \langle F \rangle_L(A) \to R,$$

and built with combinators like

$$\begin{aligned}
&\mathit{elim}_{\mathsf{1}} : R \to (\langle \mathsf{1} \rangle_L(A) \rightsquigarrow R) \\
&\mathit{elim}_{\mathsf{X}} : (A \to R) \to (\langle \mathsf{X} \rangle_L(A) \rightsquigarrow R) \\
&\mathit{elim}_{\boxplus} : (\langle F \rangle_L(A) \rightsquigarrow R) \to (\langle G \rangle_L(A) \rightsquigarrow R) \to (\langle F \boxplus G \rangle_L(A) \rightsquigarrow R) \\
&\mathit{elim}_{\boxdot} : \left(\prod_{L_1, L_2 :\, \mathscr{U}_{\mathrm{Fin}}} \lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \lfloor L \rfloor \to (\langle F \rangle_{L_1}(A) \rightsquigarrow ((\langle G \rangle_{L_2}(A) \rightsquigarrow R))) \right) \to (\langle F \boxdot G \rangle_L(A) \rightsquigarrow R) \\
&\mathit{elim}_{\mathsf{E}} : (\{\!|\, L \times A \,|\!\} \to R) \to (\langle \mathsf{E} \rangle_L(A) \rightsquigarrow R) \\
&\mathit{elim}_{\mathsf{List}} : (L \times A \to R \to R) \to R \to (\langle \mathsf{List} \rangle_L(A) \rightsquigarrow R)
\end{aligned}$$

For $elim_{\mathsf{E}}$ we assume a type $\wr - \int : \mathscr{U} \to \mathscr{U}$ of bags with an appropriate elimination principle. We omit the implementations of these combinators in the interest of space, and refer the interested reader to our Haskell implementation, available at `http://github.com/byorgey/labelled-structures`.

Using this eliminator framework, we can now use the information added by *partition* to split the data elements of a list into two sublists. We use the List structure for its ordering, but use the information from the superimposed partition to make our choices of where to put each element. Note how the elements themselves take no part in this choice, but the isomorphism which is stored in the partition plays a key role.

$$
\begin{aligned}
&\mathit{splitPartition} : \langle \mathsf{List} \boxtimes \mathsf{Part} \rangle_L(A) \to [A] \times [A] \\
&\mathit{splitPartition}((\mathit{lst}, \star \boxdot_e \star), \mathit{elts}) = \mathit{runElim}\ (\mathit{elim}_{\mathsf{List}}\ \mathit{cons}\ ([\,],[\,]))\ (\mathit{lst}, \mathit{elts}) \\
&\quad \textbf{where} \\
&\qquad \mathit{cons}\ (l,a)\ (\mathit{ll}, \mathit{rl}) = \\
&\qquad\quad \textbf{case}\ e^{-1}\ l\ \textbf{of} \\
&\qquad\qquad \mathsf{inl} - \to (a :: \mathit{ll}, \mathit{rl}) \\
&\qquad\qquad \mathsf{inr} - \to (\mathit{ll}, a :: \mathit{rl})
\end{aligned}
$$

Using similar techniques we can easily implement other standard list functions like *filter*, membership testing, and linear search.

6.3 Traversing and folding

We can also implement more general functions which work over all labelled structures. For example, any functions which traditionally rely on Haskell's Traversable type class can be implemented straightforwardly. We give *all* as an example, which computes whether all the data in a structure satisfies a predicate, assuming a suitable function $\wr \int$-*isEmpty* : $\wr A \int \to 2$:

$$
\begin{aligned}
&\mathit{all} : \langle F \rangle_L(A) \to (A \to 2) \to 2 \\
&\mathit{all}\ s\ p = \mathit{runElim}\ \mathit{el}\ (\mathit{part}, \mathit{elts}) \\
&\quad \textbf{where} \\
&\qquad ((- , \mathit{part}), \mathit{elts}) = \mathit{partition}\ s\ p \\
&\qquad \mathit{el} : \langle \mathsf{Part} \rangle_L(A) \rightsquigarrow 2 \\
&\qquad \mathit{el} = \mathit{elim}_{\boxdot}\ (\lambda \_ . \mathit{elim}_{\mathsf{E}}\ (\lambda \_ . \mathit{elim}_{\mathsf{E}}\ \wr \int\text{-}\mathit{isEmpty}))
\end{aligned}
$$

This relies on the fact that *all* is equivalent to having the second set of a partition be empty.

We can also implement *product*, which finds the product of all the numbers contained in a labelled structure:

$$
\begin{aligned}
&\mathit{product} : \langle F \rangle_L(\mathbb{N}) \to \mathbb{N} \\
&\mathit{product}\ (-, \mathit{elts}) = \mathit{runElim}\ k\ (\star, \mathit{elts}) \\
&\quad \textbf{where} \\
&\qquad k : \langle \mathsf{E} \rangle_L(\mathbb{N}) \rightsquigarrow \mathbb{N} \\
&\qquad k = \mathit{elim}_{\mathsf{E}}\ (\wr \int\text{-}\mathit{fold}\ (*)\ 0 \circ \wr \int\text{-}\mathit{map}\ \mathsf{outr})
\end{aligned}
$$

The idea is that we simply "forget" the F-shape, replacing it by an E-shape. Since $\mathbb{N}$ forms a commutative monoid under multiplication we are allowed to eliminate a bag of natural numbers in this way.

We cannot use the same technique to implement more general folds, where we do want to observe some order, because we are not allowed to observe any order on the elements of a bag. However, it suffices to have a superimposed List-shape; we just forget the other shape and eliminate the list:

$$\begin{aligned}&\mathit{foldr} : (L \times A \to A \to R) \to R \to \langle F \boxtimes \mathsf{List}\rangle_L(A) \to R\\ &\mathit{foldr}\ f\ z\ ((-, \mathit{lst}), \mathit{elts}) = \mathit{runElim}\ (\mathit{elim}_{\mathsf{List}}\ f\ z)\ (\mathit{lst}, \mathit{elts})\end{aligned}$$

Furthermore, we can always canonically superimpose a List-shape on species with no symmetries, *e.g.* $\mathit{traverse}_{\mathsf{R}} : \langle \mathsf{R}\rangle_L(A) \to \langle \mathsf{R} \boxtimes \mathsf{List}\rangle_L(A)$. In combination with *foldr*, this gives us the ability to do ordered folds over the data stored using such species, and corresponds to Haskell's Foldable type class.

7 Vector mappings

Section 4 introduced the requirements that a mapping $- \Mapsto -$ from labels to data must satisfy, and showed that functions can be used as mappings. Such an implementation is somewhat degenerate, however, in that it does not make use of the evidence that label sets are finite. The isomorphisms provided to *reindex*, *append* and *concat* are used, but only in a superficial manner.

Our goal here is to show that we can model low-level concerns such as memory layout, allocation and manipulation, in a uniform manner for all labelled structures. To model a consecutive block of memory, we will implement a mapping using finite vectors to store A values. More precisely, we use length-indexed vectors; this gives a very detailed view of the memory layout, allocation and manipulation required for storing the data associated with labelled structures. As we will see, for such mappings, *finiteness* is crucial, and the finiteness proofs are all computationally relevant.

Concretely, we assume a type $\mathsf{Vec} : \mathbb{N} \to \mathscr{U} \to \mathscr{U}$ of length-indexed vectors, supporting operations

$$\begin{aligned}\mathit{allocate}_V &: (n : \mathbb{N}) \to (\mathsf{Fin}\, n \to A) \to \mathsf{Vec}\ n\ A\\ (!) &: \mathsf{Vec}\ n\ A \to \mathsf{Fin}\, n \to A\\ \mathit{map}_V &: (A \to B) \to (\mathsf{Vec}\ n\ A \to \mathsf{Vec}\ n\ B)\\ \mathit{fold}_V &: R \to (R \to A \to R) \to \mathsf{Vec}\ n\ A \to R\\ \mathit{append}_V &: \mathsf{Vec}\ m\ A \to \mathsf{Vec}\ n\ A \to \mathsf{Vec}\ (m+n)\ A \times (\mathsf{Fin}\, m + \mathsf{Fin}\, n \simeq \mathsf{Fin}\, (m+n))\\ \mathit{concat}_V &: \mathsf{Vec}\ m\ (\mathsf{Vec}\ n\ A) \to \mathsf{Vec}\ (m \cdot n)\ A \times (\mathsf{Fin}\, m \times \mathsf{Fin}\, n \simeq \mathsf{Fin}(m \cdot n))\end{aligned}$$

Note that in addition to computing new vectors, append_V and concat_V also yield equivalences which encode the precise relationship bewteen the indices of the input and output vectors. For example, if $\mathit{append}_V\ v_1\ v_2 = (v, e)$, then it must be the case that $v_1\,!\,m = v\,!\,(e\ (\mathsf{inl}\ m))$. Similarly, $v\,!\,m\,!\,n = v'\,!\,(e\ (m,n))$ when $\mathit{concat}_V\ v = (v', e)$.

Given such a type Vec, we may define

$$(L \Mapsto A) :\equiv \sum_{n:\mathbb{N}} (\lfloor L \rfloor \simeq \mathsf{Fin}\, n) \times \mathsf{Vec}\ n\ A.$$

The idea is that we store not only a vector of the appropriate length, but also an equivalence recording how the labels correspond to vector indices. We can then implement the required operations as follows:

- The implementation of *allocate* uses the (implicitly provided) proof $(n, iso) : \mathsf{Finite}\,\lfloor L \rfloor$ to determine the size of the vector to be allocated, as well as the initial layout of the values.

$$\begin{aligned} &\mathit{allocate} : (\lfloor L \rfloor \to A) \to (L \Rightarrow A) \\ &\mathit{allocate}\ \{n, iso\}\ f = (n, iso^{-1}, \mathit{allocate}_V\ n\ (f \circ iso)) \end{aligned}$$

- To reindex, there is no need to allocate a new vector; *reindex* simply composes the given equality with the stored equivalence (we elide a technically necessary application of univalence):

$$\mathit{reindex}\ i'\ (n, i, v) = (n, i \circ i', v)$$

  This illustrates why we did not simply define $(L \Rightarrow A) :\equiv \mathsf{Vec}\ |L|\ A$, using L's associated finiteness proof for the indexing. Decoupling the finiteness evidence of the labels from the indexing scheme for the vector in this way allows us to reindex without any shuffling of data values.

- *index* is implemented in terms of $(!)$, using the stored equivalence to convert an external label L into an internal index of type $\mathsf{Fin}\,n$.

- *map* is implemented straightforwardly in terms of map_V; since the type L and the length of the underlying vector are not affected, the proof $(\lfloor L \rfloor \simeq \mathsf{Fin}\,n)$ can be carried through unchanged.

- At first blush it may seem that *zipWith* would be equally straightforward to implement in terms of a function $\mathit{zipWithV} : (A \to B \to C) \to \mathsf{Vec}\ n\ A \to \mathsf{Vec}\ n\ B \to \mathsf{Vec}\ n\ C$ (if we had such a function). The problem, however, is that the $(\lfloor L \rfloor \simeq \mathsf{Fin}\,n)$ proofs have real computational content: zipping on labels may not coincide with zipping on indices. Since we want to zip on indices, *zipWith* must compose the given equivalences to obtain the correspondence between the label mappings used by the two input vectors:

$$\begin{aligned} &\mathit{zipWith} f\ (n, i_1, v_1)\ (\_, i_2, v_2) = (n, i_2, v) \\ &\quad \textbf{where}\ v = \mathit{allocate}_V\ n\ (\lambda k \to f\ (v_1\,!\,(i_1 \circ i_2^{-1})\ k)\ (v_2\,!\,k)) \end{aligned}$$

  Note that the output of $\mathit{zipWith} f\ s_1\ s_2$ reuses the label equivalence i_2 from s_2. Of course we could instead have chosen to reuse i_1 instead, but these are the only possible choices. Given the choice of i_2, an optimizing compiler can compile *zipWith* into in-place update on s_2 when it can prove that the old value is no longer needed.

- *append* is straightforward to implement via append_V:

$$\begin{aligned} &\mathit{append} : (\lfloor L_1 \rfloor + \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to (L_1 \Rightarrow A) \to (L_2 \Rightarrow A) \to (L \Rightarrow A) \\ &\mathit{append}\ e\ (n_1, i_1, v_1)\ (n_2, i_2, v_2) = (n_1 + n_2, e^{-1} \circ (i_1 + i_2) \circ f, v) \\ &\quad \textbf{where}\ (v, f) = \mathit{append}_V\ v_1\ v_2 \end{aligned}$$

  Note that we construct the required label equivalence as the composite

$$\lfloor L \rfloor \overset{e^{-1}}{\simeq} \lfloor L_1 \rfloor + \lfloor L_2 \rfloor \overset{i_1+i_2}{\simeq} \mathsf{Fin}\,n_1 + \mathsf{Fin}\,n_2 \overset{f}{\simeq} \mathsf{Fin}\,(n_1 + n_2),$$

  using the provided equivalence e and the index equivalence f returned by append_V.

- *concat* is implemented similarly to *append*: we multiply the sizes and use $concat_V$ on the input vector-of-vectors.

$$
\begin{aligned}
&concat : (\lfloor L_1 \rfloor \times \lfloor L_2 \rfloor \simeq \lfloor L \rfloor) \to (L_1 \Rightarrow (L_2 \Rightarrow A)) \to (L \Rightarrow A) \\
&concat\ e\ (n_1, i_1, v_1) = (n_1 \cdot n_2, eqv, v') \\
&\quad \textbf{where} \\
&\qquad n_2 = |L_2| \\
&\qquad (v', f) = concat_V\ (map_V\ (\lambda(-,-,v_2).v_2)\ v) \\
&\qquad is_2 : \mathsf{Vec}\ n_1\ (\lfloor L_2 \rfloor \simeq \mathsf{Fin}\, n_2) \\
&\qquad is_2 = map_V\ (\lambda(-,i_2,-).i_2)\ v
\end{aligned}
$$

  We construct the required index equivalence *eqv* as

$$\lfloor L \rfloor \overset{e^{-1}}{\simeq} \lfloor L_1 \rfloor \times \lfloor L_2 \rfloor \overset{i_1 \times \mathsf{id}}{\simeq} \mathsf{Fin}\, n_1 \times \lfloor L_2 \rfloor \overset{\sum is_2}{\simeq} \mathsf{Fin}\, n_1 \times \mathsf{Fin}\, n_2 \overset{f}{\simeq} \mathsf{Fin}(n_1 \cdot n_2),$$

  with the only nontrivial part indicated by the notation $\sum is_2$: each inner $(L_2 \Rightarrow A)$ might contain a *different* equivalence $\lfloor L_2 \rfloor \simeq \mathsf{Fin}\, n_2$. Given a vector of n_1-many such equivalences, we "sum" them to produce an equivalence $\mathsf{Fin}\, n_1 \times \lfloor L_2 \rfloor \simeq \mathsf{Fin}\, n_1 \times \mathsf{Fin}\, n_2$ which uses the $\mathsf{Fin}\, n_1$ value as an index into the vector.

In this instance, the labels are acting like (generalized) "pointers", and the label equivalences yield some built-in "pointer indirection", allowing us to manipulate the labels without incurring a lot of (potentially expensive) allocation and copying. Data structures ultimately have to be stored in memory somehow, and this gives us a nice "end-to-end" theory that is able to model both high-level concerns as well as low-level operational details.

Note that when $append_V$ and $concat_V$ cannot be optimized to in-place updates, they must allocate an entire new vector in memory. To avoid this, in exchange for some bookkeeping overhead, we could make a deep embedding out of the vector operations, turning $append_V$ and $concat_V$ (and possibly $allocate_V$ and map_V as well) into *constructors* in a new data type. This results in something that looks very much like generalized tries [Hinze, 2000].

8 Labelled Structures in Haskell

Although a language like Agda might have been more appropriate in some ways, we used Haskell because of its greater emphasis on computation, and the possibility of demonstrating our framework with "real-world" examples. Another definite benefit of the choice of Haskell is the availability of the *lens* library [Kmett], which we use extensively to model equivalences.

On the other hand, doing dependently typed programming in Haskell certainly has its quirks [Lindley and McBride, 2013]. We make heavy use of GHC's DataKinds extension [Yorgey et al., 2012] and existential wrappers in order to simulate dependent types. Often, we are forced to use approximations to the types we really want, which occasionally gets us into trouble! Porting our code to Agda would likely be enlightening.

9 Related work

The work on *containers* [Abbott et al., 2003a,b, 2004, 2005, Morris and Altenkirch, 2009] also aims to find a more general theory of data structures which captures a large set of "containers". The resulting theory is quite elegant. It involves *shapes* and a family of *position* types indexed by shapes. More formally, it is a dependent pair of types $A : \mathscr{U}$ and $B : A \to \mathscr{U}$ (which they write $A \lhd B$) which yields a functor $T_{A \lhd B} X$ defined as $\Sigma a : A.X^{B(a)}$. Roughly, their positions correspond to our labels, their shapes correspond to our labelled shapes, and the associated functor maps positions to data values, much as our mappings associate data values to labels. They have developed the theory quite far; as of yet, however, there is no implementation of containers, nor is there a fully developed dictionary linking concrete structures to the corresponding abstract container. It is thus difficult to do a deeper comparison of the approaches. We can nevertheless make a few simple observations. One significant difference is that in the containers work, each shape is associated with a fixed, inherent set of positions, whereas in our approach a shape can be used with any type of labels. Furthermore, for them shape is an input, while for us it is part of what is generated. As a result, with containers, it does not seem that the positions can easily be given extra structure (the work on quotient containers [Abbott et al., 2004] is quite involved). There are fewer combinators for containers than for labelled structures: for example, neither the Cartesian product nor functorial composition seem to be present. Thus there is as of yet no theory of sharing for containers, nor is there a fine grained theory of storage. Having said all of that, however, containers are not restricted to finite sets of labels, which makes them more general than labelled structures: there are useful types (such as streams) which are containers but not labelled structures. And therein seems to be the main difference: the extra generality allows containers to encapsulate fancier types, while our concreteness lets us uniformly and easily model low-level concerns.

Shapely types [Jay and Cockett, 1994] are closely related to containers— see Abbott et al. [2003a, section 8] for a careful explanation of the details. Their results show that shapely types are those containers which are closest to labelled structures: in many settings of importance, shapely types are *discretely finite* containers, which essentially amounts to saying that all shapes give rise to a finite number of positions (*i.e.* labels). Shapely types do not explicitly make use of labels at all, but since they involve *lists* of data values, one may say that they implicitly make use of labels from $\mathsf{Fin}\, n$. There is thus a close relationship to our constructive finiteness proofs for label types. Furthermore, there are claims [Jay and Cockett, 1994] that this also corresponds to information about memory allocation and layout, however this is not detailed anywhere in the literature.

Another approach is that of *Container Types Categorically* [Hoogendijk and De Moor, 2000]. They define containers as monotone endofunctors F on Rel (*i.e. relators*) which have a *membership relation*; this latter concept turns out to be a special kind of lax natural transformation from F to the identity functor. This approach is again rather difficult to adequately compare to ours. There is again overlap, but no inclusion in either direction.

From the categorical perspective, *stuff types* [Baez and Dolan, 2001, Morton, 2006], brilliantly explained in Byrne's master's thesis [2006], are directly related to species. Stuff types are functors from some arbitrary groupoid X to the groupoid of finite sets and bijections. Faithful stuff types are equivalent to species. But these work much like containers: stuff types map a structure to its underlying set (which can be thought of as positions), instead of mapping labels to structures. In a different direction, *polynomial functors* also generalize species [Kock, 2012], and seem a categorically solid foundation for an even more general approach to data type constructors. Unfortunately, no one has yet to unravel these definitions into something suitable for implementation. Similarly, *generalised species of structures* [Fiore et al., 2008] may also be another interesting direction. But in all these cases, there remains much work

to be done to bridge theory and practice.

Species have been the basis for many implementations in the area of enumerative combinatorics, such as Darwin [Bergeron and Pichet, 1985], $\Lambda\Upsilon^{\Omega}$ [Flajolet et al., 1989], combstruct [Flajolet and Salvy, 1995], Aldor-Combinat [Hemmecke and Rubey, 2006] and MuPAD-Combinat [Hivert and Thiéry, 2004]. Most do not model the full spectrum of species combinators, but make up for it by implementing very sophisticated algorithms for enumeration and generation, both exhaustive and random. The Haskell species package [Yorgey, 2010, 2012] is a fairly direct implementation of the theory of species, without attempting to use this theory as a foundation for data structures.

Lastly, we should note that we have used but a small fraction of the theory of species. Bergeron et al. [1998] alone still contains a vast trove of further examples (sometimes buried deep in the exercises!) of relevance to programming. We have also not yet really touched the *calculus* aspects of the theory; while the derivative is by now well-known, integration [Rajan, 1993] has not really been explored. There are also new variants on species [Schmitt, 1993, Menni, 2008, Maia and Méndez, 2008, Aguiar and Mahajan, 2010] with nontrivial applications to combinatorics, and possible applications to programming as well. Species have even been applied to the study of attribute grammars [Mishna, 2003].

10 Future work

We have only started our translation of the theory of species to constructive type theory, but already many different threads of work are clear to us.

Capture more extant theory. Several of the species operations (such as pointing, functorial composition and arithmetic product) seem quite powerful, but we have yet to leverage them properly. Similarly, we have made very little use of *symmetry* beyond the extreme cases (ADTs have none, and E has all symmetries). 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.

We have also not yet pursued weighted species, multisorted species, nor virtual species, all of which look quite promising for applications. We can also investigate other categories of labels: for example, $\mathbb{L}$-species [Joyal, 1986], [Bergeron et al., 1998, chap. 5] use linearly ordered labels; the link with Girard's normal functors [Girard, 1988] is also worth pursuing in more detail. We firmly believe that alternate categories of labels will have significant benefits in a computational setting.

It is worth noting that much of the power of the theory of species in the context of combinatorics can be traced to fruitful homomorphisms between algebraic descriptions of species and rings of formal power series. It is worth exploring the computational content of these homomorphisms when ported to a constructive setting.

Another route of investigation are *tensorial species* [Joyal, 1986, chap. 4], which are functors to Vect rather than Set. These seem to be directly related to our vector mappings (section 7).

Lastly, there are several powerful theorems (like the molecular decomposition and the implicit species theorem) that we have yet to leverage. In the case of (small) finitary species, the molecular decomposition theorem could be used as a powerful means to specialize complex species code down to much simpler operations on a few well understood cases.

Memory allocation One of the most intriguing aspects of this elaboration of labelled structures are the close links with memory allocation and layout. This could lead to a uniform mechanism for *unboxing* of

algebraic data types, at least when their size is statically known (or even statically known to be bounded and small). We feel like we have just scratched the surface of this link. Combined with an appropriate theory of structured labels (to handle multi-dimensional arrays in a principled manner), we hope to be able to give a more uniform explanation for various memory layout strategies commonly used in high-performance linear algebra computations.

Along with this, there is a particular labelled structure, *subset*, which is particularly interesting. Combinatorially, it is equivalent to a partition, *i.e.* $\mathsf{E} \boxdot \mathsf{E}$. However, semantically a subset corresponds to only the *left* component of that product, and the right component should be ignored. In other words, we can use subset to indicate that only a subset of labels need be stored.

Categorical requirements As is clear from the species literature, there are quite a number of useful variations in the exact categories used for species. We have not been able to find a systematic treatment giving minimal assumptions required for the validity of the various constructions (sum, product, cartesian product, etc). We plan to pursue this quite formally, by first porting our implementation to Agda, as a means to prove the various properties.

In particular, outside of combinatorial uses, it is unclear exactly where *finiteness* is crucial.

HoTT and reversible programming The links with homotopy type theory run deeper than what we have used here, and deserved to be explored. For example, lists as ADTs are unique (for each size), whereas here there are many lists as labelled structures (for each size), although all of them are *equivalent*. This joins up nicely with HoTT, which teaches us to use equivalences rather than quotients. The groupoid of equivalences of labels is related to the identity type of the label set – though details obviously need to be worked out.

Another link is with reversible programming, and more particularly with the language Π of [James and Sabry, 2012]. While we use arbitrary isomorphisms between finite sets, Π is a convenient *language* in which to write (and reason about) those isomorphisms.

References

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Categories of containers. In Andrew Gordon, editor, *Proceedings of FOSSACS 2003*, number 2620 in Lecture Notes in Computer Science, pages 23–38. Springer-Verlag, 2003a.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Derivatives of Containers. In *Typed Lambda Calculi and Applications, TLCA*, volume 2701 of *LNCS*. Springer-Verlag, 2003b.

Michael Abbott, Thorsten Altenkirch, Neil Ghani, and Conor McBride. Constructing Polymorphic Programs with Quotient Types. In *7th International Conference on Mathematics of Program Construction (MPC 2004)*, volume 3125 of *LNCS*. Springer-Verlag, 2004.

Michael Abbott, Thorsten Altenkirch, and Neil Ghani. Containers - constructing strictly positive types. *Theoretical Computer Science*, 342:3–27, September 2005. Applied Semantics: Selected Topics.

M. Aguiar and S.A. Mahajan. *Monoidal Functors, Species and Hopf Algebras*. CRM monograph series / Centre de recherches mathématiques, Montréal. American Mathematical Society, 2010. ISBN 9780821847763. URL `http://books.google.ca/books?id=tXnPbwAACAAJ`.

John C. Baez and James Dolan. From finite sets to Feynman diagrams. In *Mathematics Unlimited—2001 And Beyond*, pages 29–50. Springer-Verlag, 2001.

CR Banger and DB Skillicorn. A foundation for theories of arrays. *Queen's University, Canada*, 1993.

F. Bergeron and C. Pichet. Darwin, a system designed for the study of enumerative combinatorics. In *EUROCAL '85*, volume 204 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg, 1985.

F. Bergeron, G. Labelle, and P. Leroux. *Combinatorial species and tree-like structures*. Number 67 in Encyclopedia of Mathematics and its Applications. Cambridge University Press, Cambridge, 1998.

Simon Byrne. On groupoids and stuff. Master's thesis, Macquarie University, 2006.

Jacques Carette and Gordon Uszkay. Species: making analytic functors practical for functional programming. Available at `http://www.cas.mcmaster.ca/~carette/species/`, 2008.

M. Fiore, N. Gambino, M. Hyland, and G. Winskel. The cartesian closed bicategory of generalised species of structures. *J. London Math. Soc.*, 77(1):203–220, 2008. doi: 10.1112/jlms/jdm096. URL `http://jlms.oxfordjournals.org/cgi/content/abstract/77/1/203`.

P. Flajolet, B. Salvy, and P. Zimmermann. Lambda-upsilon-omega: The 1989 cookbook. Technical Report 1073, Institut National de Recherche en Informatique et en Automatique, August 1989. URL `http://www.inria.fr/rrrt/rr-1073.html`. 116 pages.

P. Flajolet, B. Salvy, and P. Zimmermann. Automatic average-case analysis of algorithms. *Theoretical Computer Science A*, 79(1):37–109, February 1991. doi: 10.1016/0304-3975(91)90145-R.

Philippe Flajolet and Bruno Salvy. Computer algebra libraries for combinatorial structures. *Journal of Symbolic Computation*, 20(5-6):653–671, 1995. doi: 10.1006/jsco.1995.1070.

Philippe Flajolet, Paul Zimmermann, and Bernard Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *Theor. Comput. Sci.*, 132(2):1–35, 1994.

J. Girard. Normal functors, power series and λ-calculus. *Annals of Pure and Applied Logic*, 37(2): 129–177, February 1988. ISSN 01680072. doi: 10.1016/0168-0072(88)90025-5. URL `http://dx.doi.org/10.1016/0168-0072(88)90025-5`.

Ralf Hemmecke and Martin Rubey. Aldor-combinat: An implementation of combinatorial species, 2006. Available at `http://www.risc.uni-linz.ac.at/people/hemmecke/aldor/combinat/`.

Ralf Hinze. Generalizing generalized tries. *J. Funct. Program.*, 10(4):327–351, 2000.

F. Hivert and N.M Thiéry. MuPAD-Combinat, an open-source package for research in algebraic combinatorics. *Séminaire Lotharingien de Combinatoire*, 51, 2004. 70 pp.

Martin Hofmann. A simple model for quotient types. In *Typed lambda calculi and applications*, pages 216–234. Springer, 1995.

Paul Hoogendijk and Oege De Moor. Container types categorically. *J. Funct. Program.*, 10(2):191–225, March 2000. ISSN 0956-7968. doi: 10.1017/S0956796899003640. URL `http://dx.doi.org/10.1017/S0956796899003640`.

Gérard Huet. Functional pearl: The zipper. *J. Functional Programming*, 7:7–5, 1997.

Roshan P. James and Amr Sabry. Information effects. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '12, pages 73–84, New York, NY, USA, 2012. ACM. ISBN 978-1-4503-1083-3. doi: 10.1145/2103656.2103667. URL `http://doi.acm.org/10.1145/2103656.2103667`.

C. Barry Jay and J. Robin B. Cockett. Shapely types and shape polymorphism. In *ESOP '94: Proceedings of the 5th European Symposium on Programming*, pages 302–316, London, UK, 1994. Springer-Verlag. ISBN 3-540-57880-3.

André Joyal. Une théorie combinatoire des Séries formelles. *Advances in Mathematics*, 42(1):1–82, 1981.

André Joyal. Foncteurs analytiques et espèces de structures. In Gilbert Labelle and Pierre Leroux, editors, *Combinatoire Énumérative*, volume 1234 of *Lecture Notes in Mathematics*, pages 126–159. Springer Berlin Heidelberg, 1986. ISBN 978-3-540-17207-9. doi: 10.1007/BFb0072514. URL `http://dx.doi.org/10.1007/BFb0072514`.

Edward Kmett. The —lens— package. URL `http://hackage.haskell.org/package/lens`.

Joachim Kock. Data types with symmetries and polynomial functors over groupoids. In *Proceedings of the 28th Conference on the Mathematical Foundations of Programming Semantics (Bath, 2012)*, 2012.

Sam Lindley and Conor McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Proceedings of the 2013 ACM SIGPLAN Symposium on Haskell*, Haskell '13, pages 81–92, New York, NY, USA, 2013. ACM. ISBN 978-1-4503-2383-3. doi: 10.1145/2503778.2503786. URL `http://doi.acm.org/10.1145/2503778.2503786`.

Manuel Maia and Miguel Méndez. On the arithmetic product of combinatorial species. *Discrete Mathematics*, 308(23):5407 – 5427, 2008. ISSN 0012-365X. doi: http://dx.doi.org/10.1016/j.disc.2007.09.062. URL `http://www.sciencedirect.com/science/article/pii/S0012365X07007960`.

Conor McBride. The Derivative of a Regular Type is its Type of One-Hole Contexts. Available at `http://www.cs.nott.ac.uk/~ctm/diff.ps.gz`, 2001.

Conor McBride. Clowns to the left of me, jokers to the right (pearl): dissecting data structures. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 287–295, San Francisco, California, USA, 2008. ACM. ISBN 978-1-59593-689-9. doi: 10.1145/1328438.1328474. URL `http://portal.acm.org/citation.cfm?id=1328474`.

Matías Menni. Combinatorial functional and differential equations applied to differential posets. *Discrete Mathematics*, 308(10):1864–1888, May 2008. doi: 10.1016/j.disc.2007.04.035.

Marni Mishna. Attribute grammars and automatic complexity analysis. *Advances in Applied Mathematics*, 30(1):189–207, 2003.

Peter Morris and Thorsten Altenkirch. Indexed containers. In *Twenty-Fourth IEEE Symposium in Logic in Computer Science (LICS 2009)*, 2009.

Peter Morris, Thorsten Altenkirch, and Conor Mcbride. Exploring the regular tree types. In Christine Paulin-Mohring Jean-Christophe Filliatre and Benjamin Werner, editors, *Types for Proofs and Programs (TYPES 2004)*, Lecture Notes in Computer Science. Springer-Verlag, 2006.

Jeffrey Morton. Categorified algebra and quantum mechanics. *Theory Appl. Categ.*, 16:785–854, 2006.

The nLab. Finite sets, 2013. URL `http://ncatlab.org/nlab/show/finite+set`.

Ulf Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

Dayanand S. Rajan. The adjoints to the derivative functor on species. *J. Comb. Theory, Ser. A*, 62(1): 93–106, 1993.

William R. Schmitt. Hopf algebras of combinatorial structures. *Canadian Journal of Mathematics*, pages 412–428, 1993.

The Univalent Foundations Program. *Homotopy Type Theory: Univalent Foundations of Mathematics*. `http://homotopytypetheory.org/book`, Institute for Advanced Study, 2013.

Gordon Uszkay. *Modular Property Based Testing*. PhD thesis, McMaster University, 2014. (in preparation).

Yeong-Nan Yeh. The calculus of virtual species and K-species. In Gilbert Labelle and Pierre Leroux, editors, *Combinatoire énumérative*, volume 1234 of *Lecture Notes in Mathematics*, pages 351–369. Springer Berlin Heidelberg, 1986. ISBN 978-3-540-17207-9. doi: 10.1007/BFb0072525. URL `http://dx.doi.org/10.1007/BFb0072525`.

Brent Yorgey. The `species` package, 2012. URL `http://hackage.haskell.org/package/species`.

Brent A. Yorgey. Species and Functors and Types, Oh My! In *Proceedings of the third ACM Haskell symposium on Haskell*, Haskell '10, pages 147–158, New York, NY, USA, 2010. ACM. ISBN 978-1-4503-0252-4. doi: http://doi.acm.org/10.1145/1863523.1863542. URL `http://doi.acm.org/10.1145/1863523.1863542`.

Brent A Yorgey, Stephanie Weirich, Julien Cretin, Simon Peyton Jones, Dimitrios Vytiniotis, and José Pedro Magalhães. Giving haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, pages 53–66. ACM, 2012.