all in all

guillaume, georgi, fred, conor,...

July 10, 2023

1 introduction

The humble cons-list was, for many years, the workhorse of functional programming, being a means to capture free monoid structure in some (arbitrarily biased) canonical form. (We write its values [] for them empty list and x,-xs for the list with head x and tail xs. Everyone else uses a *symmetric* symbol for ,- — their teaching assistants weep as their students mistake it for a symmetric operation. We retain the standard $[a, b, c]$ notation for []-terminated lists a,-b,-c,-[]) With the advent of fancy (dependent or otherwise) type systems, the world of list-like types has exploded, with many dimensions of variation, e.g.,

dimensions two-dimensional matrices, and even higher-dimensional cuboids, can be seen as nested lists, with types imposing varying degrees of

coherence meaning the extent to which we enforce non-raggedness, by whatever means, but notably

indexing with the over-deployed length-indexed vectors being but one example (of a list with interesting elements with positions corresponding to those in a list with dull elements, otherwise a 'natural number'), modelling sized arrays and admitting safe

access which is not always left-to-right—right-to-left is also useful, but people get up to all sorts of things to support random access or healthy amortized complexity.

The archetype of these structures is (borrowing the language of propositions to talk about data-bearing types) the 'predicate transformer'

$$\mathbf{All} \ \in \ (X \to \mathbf{Ty}) \ \to \ (\mathbf{List}\ X \to \mathbf{Ty})$$

where $\mathbf{All}\ P\ xs$ stores a 'proof' (possibly involving nontrivial data) of $P\ x$ for each element x of xs. More specifically,

$$\mathbf{All}\ P\ [] \ni [] \qquad \mathbf{All}\ P\ (x,\text{-}xs) \ni p,\text{-}ps \ \Leftarrow \ P\ x \ni p,\ \mathbf{All}\ P\ xs \ni ps$$

The type X gives some notion of *individual*; position in the list xs is a proxy for the distinction between individuals; the way $P\ x$ varies with x tracks a notion

of *pertinence* to the individual. The length-indexed vectors is thus the special case where $X = 1$ — the list individuals are indistinct and the vector elements are impertinent. That is, **All** characterizes a kind of *table* where xs represents a row of *headers*, while the *cells* of the table have types dependent on the header which *governs* them.

Stepping onwards, we have at least three choices of direction.

- We might ask what sort of table has $\mathbf{All}\ P\ xs$ as its type of headers. Consider 'three-layer **All**'

$$\mathbf{Alll} \in ((x : X) \to P\,x \to \mathbf{Ty}) \to ((xs : \mathbf{List}\ X) \to \mathbf{All}\ P\ xs \to \mathbf{Ty})$$

  where data construction is in even tighter lock-step:

$$\mathbf{Alll}Q[][] \ni [] \qquad \mathbf{Alll}Q(x,\text{-}xs)(p,\text{-}ps) \ni q,\text{-}qs \Leftarrow Qxp \ni q, \mathbf{Alll}Qxsps \ni qs$$

- We might recognize **All** as the *predicate* lifting of **List** and ask what is the *relational* lifting.

$$\mathbf{ListR} \in (X \to Y \to \mathbf{Ty}) \to (\mathbf{List}\ X \to \mathbf{List}\ Y \to \mathbf{Ty})$$

  which requires that the two lists have the same length, or more pertinently a sensible notion of 'corresponding position', and that elements in corresponding positions are related:

$$\mathbf{ListR}R[][] \ni [] \qquad \mathbf{ListR}R(x,\text{-}xs)(y,\text{-}ys) \ni r,\text{-}rs \Leftarrow Rxy \ni r, \mathbf{ListR}Rxsys \ni rs$$

- We might reach into more dimensions, characterizing rectangular matrices with independent 'row' and 'column' headers, after the manner of a *spreadsheet*.

$$\mathbf{Matrix} \in (X \to Y \to \mathbf{Ty}) \to (\mathbf{List}\ X \to \mathbf{List}\ Y \to \mathbf{Ty})$$

  which is readily definable using **All**:

$$\mathbf{Matrix}\ R\ xs\ ys = \mathbf{All}\ (\lambda x \mapsto \mathbf{All}\ (\lambda y \mapsto R\ x\ y)\ ys)\ xs$$

The fact that **ListR** and **Matrix** have the same type but mean rather different things ought to prompt us to consider how to make the distinction plain. For **ListR**, the two lists are in the 'same dimension' and yoked together, with corresponding positions that are somehow cognate; for **Matrix**, the lists are in 'orthogonal dimensions', and any point in the cartesian product of their positions is in some way meaningful. Meanwhile, **ListR** and **Alll** both describe one-dimensional sequences yoked to two other one-dimensional sequences, but with differing patterns of type dependency.

The fact that **Matrix** is definable in terms of **All** should prompt us to ask whether we can manage the same for **Alll** and **ListR**. To avoid undue suspense,

category theorists can but type theorists cannot. By dependent pairing, we may form

$$\mathbf{ListR}\ xs\ ys\ =\ (rs : \mathbf{All}\ (\lambda x \mapsto (y : Y) \times R\ x\ y)\ xs) \times\ \mathbf{map\ fst}\ rs = ys$$

That is, we forget that we know in advance the $\mathbf{List}\ Y$ we want, yoke each x to any old y such that $R\ x\ y$, then, as an afterthought, demand the outrageous coincidence that the ys we collected turned out to be exactly the ys we now remember that we wanted. Such inverse image constructions create troublesome pedantry when one's audience is too mechanical to be readily intimidated. Similarly,

$$\mathbf{Alll}\ Q\ xs\ ps\ =\ (qs : \mathbf{All}\ (\lambda x \mapsto (p : P\ x) \times Q\ x\ p)\ xs) \times \mathbf{map\ fst}\ qs = ps$$

where again the prior indexing is rendered as posterior projection. If we want to retain the intrinsic coherence that comes from knowing the indices in advance, the current state of the art requires us to invent **Alll** and **ListR** afresh, as if they have nothing to do with one another.

We have seen five structures — **List**, **All**, **Alll**, **ListR** and **Matrix** — but we might imagine many others, e.g., rectangular collections of cells where the length and height are not explicit in the type but their uniformity is enforced, or cuboids of data indexed by three matrices on the three faces meeting at the origin themselves indexed by two of three lists on their axes. Let us seek a common scheme for describing these m-dimensional data 'bricks' indexed over various n-dimensional 'bricks', where the n are selected from the m. The humble cons-list is then the simplest form of one-dimensional brick, in which the cell type is unindexed and so is the brick type, but each of the rest is just another brick, rather than something we must invent anew.

2 telescopes with explicit dependency

Let us start in *one* dimension, seeking to characterize the distinctions between **List**, **All**, **Alll** and **ListR**. Look at how the cell types are indexed by 'headers' in each case

type former	cell indices	brick indices
List		
All	$x : X$	$xs : \mathbf{List}\ X$
Alll	$x : X, p : P\ x$	$xs : \mathbf{List}\ X, ps : \mathbf{All}\ P\ xs$
ListR	$x : X, y : Y$	$xs : \mathbf{List}\ X, ys : \mathbf{List}\ Y$

These indices form *telescopes*, in the sense of de Bruijn, in which every entry type can depend on the entry values to its left. For **List**, **All** and **Alll**, we see fully dependent telescopes of lengths 0, 1 and 2, respectively. But for **ListR**, we note that the type Y may *not* depend on the value of x. The pattern of permitted dependencies determines the type of brick from which the cell indices are drawn. Above, the non-dependent xs and y are drawn from **List**s, while the dependent

p must be drawn from an **All**. We may therefore specify the headers of a one-dimensional brick by giving a telescope with explicit dependency permissions.

Definition 2.1 (telescope with explicit dependency) *Inductively, the empty telescope ε is a telescope with explicit dependency, and an existing such telescope Δ may be extended to the right by some $\Delta, y : \vec{x}.T$ where y is a name fresh for Δ, and $\vec{x}$ is a dependency-closed selection of names bound in Δ (in the order that Δ binds them), and T is a type with respect to the subtelescope $\vec{x} \lfloor \Delta$ of Δ selected by $\vec{x}$. Such a selection is dependency-closed if every x selected in $\vec{x}$ has all its dependencies included in $\vec{x}$. Dependency closure ensures the existence of the selected subtelescope.*

$$\begin{array}{rll} \lfloor\varepsilon & = \varepsilon & \\ (\vec{z}y)\lfloor(\Delta, y:\vec{x}.T) & = (\vec{z} \lfloor \Delta), y:\vec{x}.T & \textit{noting } \vec{x} \subseteq \vec{z} \\ \vec{z}\rfloor(\Delta, y:\vec{x}.T) & = \vec{z} \lfloor \Delta & \textit{if } y \notin \vec{z} \end{array}$$

If a dependency sequence $\vec{x}$ is empty, it is permitted to omit the . before the T.

Remark 2.1.1 (fresh names) *Of course, telescopes should be considered invariant under α-equivalence. Formally, telescopes are given in some nameless representation, with selections being given as bit vectors.*

We now have the means to distinguish the header telescope for **Alll**, namely $\varepsilon, x:X, p:x.T$, from that for **ListR**, namely $\varepsilon, x:X, y:Y$. However, we have yet to clarify how to specify these one-dimensional brick types as a whole.

3 one-dimensional bricks

Let us start by giving some examples, then back-fill the general scheme. Brick types look a little like *parallel* comprehensions, but with the generators left of the value. **List** X is a trivial comprehension, which makes it hard to notice that it is a comprehension at all.

$$[\varepsilon \mid X]$$

Indeed, we could even allow the omission of the $\mid$ when the header context is empty, to fool Haskell programmers into thinking that nothing unusual is happening. Let us do so, and also omit "ε,", wherever it would otherwise clutter the left end of a nonempty context.

However, **All** P xs is more clearly a comprehension

$$[x:X \leftarrow xs \mid P\ x] \quad \text{where} \quad [X] \ni xs$$

ListR R xs ys is

$$[x:X \leftarrow xs, y:Y \leftarrow ys \mid R\ x\ y] \quad \text{where} \quad [X] \ni xs, [Y] \ni ys$$

and **Alll** Q xs ps is

$$[x:X \leftarrow xs, p:x.P\ x \leftarrow ps \mid Q\ x\ p] \quad \text{where} \quad [X] \ni xs, [x:X \leftarrow xs \mid P\ x] \ni ps$$

That is, we annotate each entry in the header context with the brick it is drawn from, so $\Delta \leftarrow \vec{ys}$ and extend the $\lfloor$ operator accordingly. Whenever we extend to $\Delta \leftarrow \vec{ys}, y{:}\vec{x}.S \leftarrow ys$, we should have

$$[\vec{x} \lfloor (\Delta \leftarrow \vec{ys}) \mid S] \ni ys$$

We have come good on the idea that the explicit dependencies are what distinguish the header brick types in **ListR** and **Alll**.

Definition 3.1 (one-dimensional brick type) *The type*

$$[\Delta \leftarrow \vec{ys} \mid T]$$

is a valid one-dimensional brick type if $\Delta \leftarrow \vec{ys}$ is a valid sequence of headers and $\Delta \vdash \mathbf{Ty} \ni T$. The empty sequence ε is a valid sequence of headers. If $\Delta \leftarrow \vec{ys}$ is a valid sequence of headers, we may validly extend it to $\Delta \leftarrow \vec{ys}, y{:}\vec{x}.S \leftarrow ys$ if $\vec{x} \lfloor \Delta \vdash \mathbf{Ty} \ni S$ and $[\vec{x} \lfloor (\Delta \leftarrow \vec{ys}) \mid S] \ni ys$.

Now, what is a brick in such a type? The rules may seem straightforward, but they disguise a mathematician's vacuity joke.

Definition 3.2 (one-dimensional brick) *One-dimensional bricks are constructed from $[]$ and $\cdot$,-$\cdot$ in accordance with the rules:*

$$[\Delta \leftarrow \vec{[]} \mid T] \ni []$$

(i.e., the brick may be $[]$ if all the headers are)

$$[\Delta \leftarrow (\vec{s}\text{,-}\vec{ss}) \mid T] \ni t\text{,-}ts \;\Leftarrow\; T\{\vec{s}/\Delta\} \ni t, [\Delta \leftarrow \vec{ss} \mid T] \ni ts$$

(i.e., the brick may be a $\cdot$,-$\cdot$ if all the headers are, with the brick head's type being the cell type instantiated with the header heads)

Note that whenever the header sequence is ε, *both* of these conditions are vacuously satisfied, so the brick may be *either* $[]$ or $\cdot$,-$\cdot$, recovering exactly the unconstrained cons-lists from the mud of our childhoods. Meanwhile, we recover **All**, **Alll**, and **ListR** exactly as declared above.

4 bricks in more (or fewer) dimensions

Suitably warmed up, let us now extend the notion of brick to any number of dimensions including zero. We open up the *left* panel of the triptych and write **Matrix** $R\ xs\ ys$ as

$$[i\ j \mid i\ x{:}X \leftarrow xs, j\ y{:}Y \leftarrow ys \mid R\ x\ y]$$

The names i and j are the binding sites of the dimensions used to annotate the headers, and they are subject to α-equivalence. Again, it would suffice to identify the *number* of dimensions and give for each header a bit vector indicating which dimensions it exists in, but that would be less friendly to humans.

Definition 4.1 (brick type) *A brick type has the form*

$$[\vec{i} \mid \Gamma \mid T]$$

where each entry in Γ *is of form*

$$\vec{j}\, y{:}\vec{x}.S \leftarrow ss$$

with

- *the dimensions* $\vec{j}$ *of* y *selected from* $\vec{i}$
- *the dependencies* $\vec{x}$ *of* y *selected from* Γ *left of* y *such that the dimensions of each* x *are selected from* $\vec{j}$ *and the dependencies of each* x *are selected from* $\vec{x}$
- $\vec{x} \downarrow \Gamma \vdash \mathbf{Ty} \ni S$
- $[\vec{j} \mid \vec{x} \downarrow \Gamma \mid S] \ni ss$

and where $\Gamma \vdash \mathbf{Ty} \ni T$.

We note that $[\mid \mid T]$ is a brick type when $\mathbf{Ty} \ni T$ and consider it to be the same type as X. Indeed, this allows us to have *zero*-dimensional headers instantiated by single values. $[\mid \Delta \leftarrow \vec{s} \mid T] = T\{\vec{s}/\Delta\}$.

For notational convenience, let us adopt the following conventions for omitting $\mid$:

- The left $\mid$ may be omitted if there is exactly *one* dimension and all headers align with it. (Hence, we preserve our prior notation for one-dimensional brick types.)
- If the header context is ε, we may omit $\varepsilon \mid$.
- We forbid the situation $[|T]$, where it is not clear whether the omitted $\mid$ is the left (yielding lists of T — $[T]$ does that job) or the right (yielding T itself — use $[||T]$ or just plain T).

Headerless types exist at higher dimensions. $[i\ j \mid \mid T]$ is the type of rectangular arrays of T cells with unspecified size. This raises two problems:

- How do we exclude *ragged* lists of lists?
- How do we ensure that the *empty* list of lists is invertibly transposable?

All of our bricks should have *one* size in each of their dimensions, whether or not that size is fixed by a size of a header. In theory, every brick should be given as the dependent pair of an explicit size vector and a nested list giving the contents. In practice, the typechecker can guess these sizes, for the most part, by a particularly degenerate form of unification. However, these sizes are accessible at run-time: you can always compute the number of ,-es in the outermost layer,

and you should always be able to reorder the layers by any permutation of the dimensions. In Hindley-Milner polymorphism, parametricity allows us to generalize over anything which survives type inference unconstrained, but here (as with Haskell type classes in the notorious `show.read` example), failure to infer sizes is an error.

Correspondingly, there must be a way to give the missing sizes whenever a brick is degenerate. Whenever an outer layer is of size zero, we should be able to write its 'nil' as $[\times n_i \ldots \times n_k]$ to fill in the sizes of the inner layers. That is, $[\times 2]$ is a 0×2 rectangle, and its transpose is $[[], []]$.

We are nearly in a position to define, formally, what constitutes an n-dimensional brick for positive n. The 0-dimensional bricks are given already by the types to which they degenerate. We shall need some auxiliary operations which compute the types of heads and tails of bricks.

Definition 4.2 (head brick type) *The partial operation* Head *maps* $(1+n)$*-dimensional brick types to* n*-dimensional brick types. In particular, it maps*

$$[i\,\vec{i} \mid \Gamma \mid T] \quad \mapsto \quad [\vec{i} \mid \Gamma_i \mid T\sigma] \quad \textit{where} \quad (\Gamma_i, \sigma) = \Gamma^{i,}$$

The head selector operator $\Gamma^{i,}$ *computes a subcontext* Γ_i *and a substitution* σ *recursively thus:*

$$\begin{array}{rcl} \varepsilon^{i,} & = & (\varepsilon, \{\}) \\ (\Gamma, i\,\vec{j}\,y{:}\vec{x}.S \leftarrow (1+n)\textit{,-ns\#s,-ss})^{i,} & = & (\Gamma_i, \sigma\{s/y\}) \\ (\Gamma, \vec{j}\,y{:}\vec{x}.S \leftarrow \textit{ns\#ss})^{i,} & = & ((\Gamma_i, \vec{j}\,y{:}\vec{x}.S \leftarrow \textit{ns\#ss}), \sigma) \end{array}$$

That is, in the head brick type, all the variables in the header context marked with the outermost dimension must be selected from conses, so they may be removed from the header context and substituted with the heads of those conses. The headers not in the outermost dimension stay just as they are.

Definition 4.3 (tail brick type) *The partial operation* Tail *maps* $(1+n)$*-dimensional brick types to* $1+n$*-dimensional brick types. In particular, it maps*

$$[i\,\vec{i} \mid \Gamma \mid T] \quad \mapsto \quad [i\,\vec{i} \mid \Gamma^{i-} \mid T]$$

The tail selector operator Γ^{i-} *decapitates a context thus:*

$$\begin{array}{rcl} \varepsilon^{i-} & = & \varepsilon \\ (\Gamma, i\,\vec{j}\,y{:}\vec{x}.S \leftarrow (1+n)\textit{,-ns\#s,-ss})^{i-} & = & \Gamma^{i-}, i\,\vec{j}\,y{:}\vec{x}.S \leftarrow n\textit{,-ns\#ss} \\ (\Gamma, \vec{j}\,y{:}\vec{x}.S \leftarrow \textit{ns\#ss})^{i-} & = & \Gamma^{i-}, \vec{j}\,y{:}\vec{x}.S \leftarrow \textit{ns\#ss} \end{array}$$

That is, in the tail brick type, the headers in the outermost dimension lose their heads, while the headers not in that dimension stay just as they are.

Definition 4.4 (brick) *The* n*-dimensional brick type*

$$[\vec{i} \mid \Gamma \mid T]$$

admits bricks of form

$$\textit{ms\#tss}$$

where

- *ms is a vector of natural numbers of length n (also the length of* $\vec{i}$*);*
- *tss is n-layer nested vector of values (whose types will be given shortly);*
- *for each* $\vec{j}$ $y{:}\vec{x}.S \leftarrow ns\#ss$ *in* Γ, $\vec{j} \downharpoonleft ms = ns$*;*
- $tss = []$ *if* $ms = 0$,-ms'*;*
- $tss = ts$,-tss' *if* $ms = (1 + m)$,-ms' *and* $\vec{i} = i\ \vec{i'}$ *and* $\mathrm{Head}([\vec{i} \mid \Gamma \mid T]) \ni ms'\#ts$ *and* $\mathrm{Tail}([\vec{i} \mid \Gamma \mid T]) \ni m$,-$ms'\#tss'$*.*