How to Keep Your Neighbours in Order

Conor McBride
University of Strathclyde
Conor.McBride@strath.ac.uk

Abstract

I present a datatype-generic treatment of recursive container types whose elements are guaranteed to be stored in increasing order, with the ordering invariant rolled out systematically. Intervals, lists and binary search trees are instances of the generic treatment. On the journey to this treatment, I report a variety of failed experiments and the transferable learning experiences they triggered. I demonstrate that a *total* element ordering is enough to deliver insertion and flattening algorithms, and show that (with care about the formulation of the types) the implementations remain as usual. Agda's *instance arguments* and *pattern synonyms* maximize the proof search done by the typechecker and minimize the appearance of proofs in program text, often eradicating them entirely. Generalizing to indexed recursive container types, invariants such as *size* and *balance* can be expressed in addition to *ordering*. By way of example, I implement insertion and deletion for 2-3 trees, ensuring both order and balance by the discipline of type checking.

Categories and Subject Descriptors D.1.1 [*Programming Techniques*]: Applicative (Functional) Programming; D.3.3 [*Language Constructs and Features*]: Data types and structures

Keywords dependent types; Agda; ordering; balancing; sorting

1. Introduction

It has taken years to see what was under my nose. I have been experimenting with ordered container structures for a *long* time [12]: how to keep lists ordered, how to keep binary search trees ordered, how to flatten the latter to the former. Recently, the pattern common to the structures and methods I had often found effective became clear to me. Let me tell you about it. Patterns are, of course, underarticulated abstractions. Correspondingly, let us construct a *universe* of container-like datatypes ensuring that elements are in increasing order, good for intervals, ordered lists, binary search trees, and more besides.

This paper is a literate Agda development, available online at `https://github.com/pigworker/Pivotal`. As well as contributing

- a datatype-generic treatment of ordering invariants and operations which respect them
- a technique for hiding proofs from program texts
- a precise implementation of insertion and deletion for 2-3 trees

I take the time to explore the design space, reporting a selection of the wrong turnings and false dawns I encountered on my journey to these results. I try to extrapolate transferable design principles, so that others in future may suffer less than I.

ICFP '14, September 1–6, 2014, Gothenburg, Sweden.

http://dx.doi.org/10.1145/2628136.2628163

2. How to Hide the Truth

If we intend to enforce invariants, we shall need to mix a little bit of logic in with our types and a little bit of proof in with our programming. It is worth taking some trouble to set up our logical apparatus to maximize the effort we can get from the computer and to minimize the textual cost of proofs. We should prefer to encounter logic only when it is dangerously absent!

Our basic tools are the types representing falsity and truth by virtue of their number of inhabitants:

```
data 0 : Set where                          -- no constructors!
record 1 : Set where constructor ⟨⟩          -- no fields!
```

Dependent types allow us to compute sets from data. E.g., we can represent evidence for the truth of some Boolean expression which we might have tested.

```
data 2 : Set where tt ff : 2

So : 2 → Set
So tt = 1
So ff = 0
```

A set P which evaluates to 0 or to 1 might be considered 'propositional' in that we are unlikely to want to *distinguish* its inhabitants. We might even prefer not even to *see* its inhabitants. I define a wrapper type for propositions whose purpose is to hide proofs.

```
record ⌜_⌝ (P : Set) : Set where
  constructor !
  field {{prf}} : P
```

Agda uses braces to indicate that an argument or field is to be suppressed by default in program texts and inferred somehow by the typechecker. Single-braced variables are solved by unification, in the tradition of Milner [16]. Doubled braces indicate *instance arguments*, inferred by *contextual search*: if just one hypothesis can take the place of an instance argument, it is silently filled in, allowing us a tiny bit of proof automation [6]. If an inhabitant of $\ulcorner \mathsf{So}\ b \urcorner$ is required, we may write ! to indicate that we expect the truth of b to be known.

Careful positioning of instance arguments seeds the context with useful information. We may hypothesize over them quietly,

```
_⇒_ : Set → Set → Set
P ⇒ T = {{p : P}} → T
infixr 3 _⇒_
```

and support forward reasoning with a 'therefore' operator.

```
_∴_ : ∀{P T} → ⌜P⌝ → (P ⇒ T) → T
! ∴ t = t
```

This apparatus can give the traditional conditional a subtly more informative type, thus:

```
¬ : 2 → 2; ¬ tt = ff; ¬ ff = tt
if_then_else_ :
  ∀{X} b → (So b ⇒ X) → (So (¬ b) ⇒ X) → X
if tt then t else f = t
if ff then t else f = f
infix 1 if_then_else_
```

If ever there is a proof of 0 in the context, we should be able to ask for anything we want. Let us define

```
magic : {X : Set} → 0 ⇒ X
magic {{()}}
```

using Agda's *absurd pattern* to mark the impossible instance argument which shows that no value need be returned. E.g., if tt then ff else magic : 2.

Instance arguments are not a perfect fit for proof search: they were intended as a cheap alternative to type classes, hence the requirement for exactly one candidate instance. For proofs we might prefer to be less fussy about redundancy, but we shall manage perfectly well for the purposes of this paper.

3. Barking Up the Wrong Search Trees

David Turner [17] notes that whilst *quicksort* is often cited as a program which defies structural recursion, it performs the same sorting algorithm (although not with the same memory usage pattern) as building a binary search tree and then flattening it. The irony is completed by noting that the latter sorting algorithm is the archetype of structural recursion in Rod Burstall's development of the concept [4]. Binary search trees have empty leaves and nodes labelled with elements which act like *pivots* in quicksort: the left subtree stores elements which precede the pivot, the right subtree elements which follow it. Surely this invariant is crying out to be a dependent type! Let us search for a type for search trees.

We could, of course, define binary search trees as ordinary node-labelled trees with parameter P giving the type of pivots:

```
data Tree : Set where
  leaf : Tree;  node : Tree → P → Tree → Tree
```

We might then define the invariant as a predicate IsBST : Tree → Set, implement insertion in our usual way, and prove separately that our program maintains the invariant. However, the joy of dependently typed programming is that refining the types of the data themselves can often alleviate or obviate the burden of proof. Let us try to bake the invariant in.

What should the type of a subtree tell us? If we want to check the invariant at a given node, we shall need some information about the subtrees which we might expect comes from their type. We require that the elements left of the pivot precede it, so we could require the whole set of those elements represented somehow, but of course, for any order worthy of the name, it suffices to check only the largest. Similarly, we shall need to know the smallest element of the right subtree. It would seem that we need the type of a search tree to tell us its extreme elements (or that it is empty).

```
data STRange : Set where
  ∅ : STRange;  _–_ : P → P → STRange
infix 9 _–_
```

From checking the invariant to enforcing it. Assuming we can test the order on P with some $le : P \to P \to 2$, we could write a recursive function to check whether a Tree is a valid search tree and compute its range if it has one. Of course, we must account for the possibility of invalidity, so let us admit failure in the customary manner.

```
data Maybe (X : Set) : Set where
  yes : X → Maybe X;  no : Maybe X
_?⟩_ : ∀{X} → 2 → Maybe X → Maybe X
b ?⟩ mx = if b then mx else no
infixr 4 _?⟩_
```

The guarding operator ?⟩ allows us to attach a Boolean test. We may now validate the range of a Tree.

```
valid : Tree → Maybe STRange
valid leaf = yes ∅
valid (node l p r) with valid l | valid r
... | yes ∅       | yes ∅       = yes (p – p)
... | yes ∅       | yes (c – d) = le p c ?⟩ yes (p – d)
... | yes (a – b) | yes ∅       = le b p ?⟩ yes (a – p)
... | yes (a – b) | yes (c – d)
    = le b p ?⟩ le p c ?⟩ yes (a – d)
... | _           | _           = no
```

As valid is a *fold* over the structure of Tree, we can follow my colleagues Bob Atkey, Neil Ghani and Patricia Johann in computing the *partial refinement* [2] of Tree which valid induces. We seek a type BST : STRange → Set such that BST $r \cong \{t : \mathsf{Tree} \mid \mathsf{valid}\ t = \mathsf{yes}\ r\}$ and we find it by refining the type of each constructor of Tree with the check performed by the corresponding case of valid, assuming that the subtrees yielded valid ranges. We can calculate the conditions to check and the means to compute the output range if successful.

```
lOK : STRange → P → 2
lOK ∅       p = tt
lOK (_ – u) p = le u p

rOK : P → STRange → 2
rOK p ∅       = tt
rOK p (l – _) = le p l

outRan : STRange → P → STRange → STRange
outRan ∅       p ∅       = p – p
outRan ∅       p (_ – u) = p – u
outRan (l – _) p ∅       = l – p
outRan (l – _) _ (_ – u) = l – u
```

We thus obtain the following refinement from Tree to BST:

```
data BST : STRange → Set where
  leaf : BST ∅
  node : ∀{l r} → BST l → (p : P) → BST r →
    So (lOK l p) ⇒ So (rOK p r) ⇒ BST (outRan l p r)
```

Attempting to implement insertion. Now that each binary search tree tells us its type, can we implement insertion? Rod Burstall's implementation is as follows

```
insert : P → Tree → Tree
insert y leaf = node leaf y leaf
insert y (node lt p rt) =
  if le y p then node (insert y lt) p rt
            else node lt p (insert y rt)
```

but we shall have to try a little harder to give a type to insert, as we must somehow negotiate the ranges. If we are inserting a new extremum, then the range will be wider afterwards than before.

```
insRan : STRange → P → STRange
insRan ∅       y = y − y
insRan (l − u) y =
  if le y l then y − u else if le u y then l − y else l − u
```

So, we have the right type for our data and for our program. Surely the implementation will go like clockwork!

```
insert : ∀{r} y → BST r → BST (insRan r y)
insert y leaf            = node leaf y leaf
insert y (node lt p rt) =
  if le y p then (node (insert y lt) p rt)
            else (node lt p (insert y rt))
```

The leaf case checks easily, but alas for node! We have lt : BST l and rt : BST r for some ranges l and r. The then branch delivers a BST (outRan (insRan l y) p r), but the type required is BST (insRan (outRan l p r) y), so we need some theorem-proving to fix the types, let alone to discharge the obligation So (lOK (insRan l y) p). We could plough on with proof and, coughing, push this definition through, but tough work ought to make us ponder if we might have thought askew.

We have defined a datatype which is logically correct but which is pragmatically disastrous. Is it thus inevitable that all datatype definitions which enforce the ordering invariant will be pragmatically disastrous? Or are there lessons we can learn about dependently typed programming that will help us to do better?

4. Why Measure When You Can Require?

Last section, we got the wrong answer because we asked the wrong question: "What should the type of a subtree tell us?" somewhat presupposes that information bubbles outward from subtrees to the nodes which contain them. In Milner's tradition, we are used to synthesizing the type of a thing. Moreover, the very syntax of **data** declarations treats the index delivered from each constructor as an output. It seems natural to treat datatype indices as measures of the data. That is all very well for the length of a vector, but when the measurement is intricate, as when computing a search tree's extrema, programming becomes vexed by the need for theorems about the measuring functions. The presence of 'green slime'—defined functions in the return types of constructors—is a danger sign.

We can take an alternative view of types, not as synthesized measurements of data, bubbled outward, but as checked *requirements* of data, pushed *inward*. To enforce the invariant, let us rather ask "What should we tell the type of a subtree?".

The elements of the left subtree must precede the pivot in the order; those of the right must follow it. Correspondingly, our requirements on a subtree amount to an *interval* in which its elements must fall. As any element can find a place somewhere in a search tree, we shall need to consider unbounded intervals also. We can extend any type with top and bottom elements as follows.

```
data _⊤⊥ (P : Set) : Set where
  ⊤ : P⊤⊥; # : P → P⊤⊥; ⊥ : P⊤⊥
```

and extend the order accordingly:

```
_⊤⊥ : ∀{P} → (P → P → 2) → P⊤⊥ → P⊤⊥ → 2
le⊤⊥ _     ⊤     = tt
le⊤⊥ (#x) (#y) = le x y
le⊤⊥ ⊥     _     = tt
le⊤⊥ _     _     = ff
```

We can now index search trees by a pair of *loose bounds*, not measuring the range of the contents exactly, but constraining it sufficiently. At each node, we can require that the pivot falls in the interval, then use the pivot to bound the subtrees.

```
data BST (l u : P⊤⊥) : Set where
  leaf  : BST l u
  pnode : (p : P) → BST l (#p) → BST (#p) u →
    So (le⊤⊥ l (#p)) ⇒ So (le⊤⊥ (#p) u) ⇒ BST l u
```

In doing so, we eliminate all the 'green slime' from the indices of the type. The leaf constructor now has many types, indicating all its elements satisfy any requirements. We also gain BST ⊥ ⊤ as the general type of binary search trees for P. Unfortunately, we have been forced to make the pivot value p, the first argument to pnode, as the type of the subtrees now depends on it. Luckily, Agda now supports *pattern synonyms*, allowing linear macros to abbreviate both patterns on the left and pattern-like expressions on the right [1]. We may fix up the picture:

```
pattern node lp p pu = pnode p lp pu
```

Can we implement insert for this definition? We can certainly give it a rather cleaner type. When we insert a new element into the left subtree of a node, we must ensure that it precedes the pivot: that is, we expect insertion to *preserve* the bounds of the subtree, and we should already know that the new element falls within them.

```
insert : ∀{l u} y → BST l u →
  So (le⊤⊥ l (#y)) ⇒ So (le⊤⊥ (#y) u) ⇒ BST l u
insert y leaf            = node leaf y leaf
insert y (node lt p rt) =
  if le y p then node (insert y lt) p rt
            else node lt p (insert y rt)
```

We have no need to repair type errors by theorem proving, and most of our proof obligations follow directly from our assumptions. The recursive call in the then branch requires a proof of So (le y p), but that is just the evidence delivered by our evidence-transmitting conditional. However, the else case snatches defeat from the jaws of victory: the recursive call needs a proof of So (le p y), but all we have is a proof of So (¬ (le y p)). For any given total ordering, we should be able to fix this mismatch up by proving a theorem, but this is still more work than I enjoy. The trouble is that we couched our definition in terms of the truth of bits computed in a particular way, rather than the ordering *relation*. Let us now tidy up this detail.

5. One Way Or The Other

We can recast our definition in terms of relations—families of sets Rel P indexed by pairs.

```
Rel : Set → Set1
Rel P = P × P → Set
```

giving us types which directly make statements about elements of P, rather than about bits.

I must, of course, say how such pairs are defined: the habit of dependently typed programmers is to obtain them as the degenerate case of dependent pairs: let us have them.

```
record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field
    π1 : S
    π2 : T π1
open Σ
_×_ : Set → Set → Set
S × T = Σ S λ _ → T
infixr 5 _×_ _,_
```

Now, suppose we have some 'less or equal' ordering $L : \mathsf{Rel}\ P$. Let us have natural numbers by way of example,

```
data ℕ : Set where 0 : ℕ; s : ℕ → ℕ

L_ℕ : Rel ℕ
L_ℕ (x , y) = x ≤ y where
  _≤_ : ℕ → ℕ → Set
  0   ≤ y   = 1
  s x ≤ 0   = 0
  s x ≤ s y = x ≤ y
```

The information we shall need is exactly the totality of L: for any given x and y, L must hold *One Way Or The Other*, as captured by the disjoint sum type, $\mathsf{OWOTO}\ L\ (x, y)$, defined as follows:

```
data _+_ (S T : Set) : Set where
  ◁ : S → S + T;   ▷ : T → S + T
infixr 4 _+_

OWOTO : ∀{P} (L : Rel P) → Rel P
OWOTO L (x , y) = ⌜L (x , y)⌝ + ⌜L (y , x)⌝

pattern le = ◁ !
pattern ge = ▷ !
```

I have used pattern synonyms to restore the impression that we are just working with a Boolean type, but the ! serves to unpack evidence when we test and to pack it when we inform. We shall usually be able to keep silent about ordering evidence, even from the point of its introduction. For $\mathbb{N}$, let us have

```
owoto : ∀x y → OWOTO L_ℕ (x , y)
owoto 0     y     = le
owoto (s x) 0     = ge
owoto (s x) (s y) = owoto x y
```

Note that we speak only of the crucial bit of information. Moreover, we especially benefit from type-level computation in the step case: $\mathsf{OWOTO}\ \mathsf{L}_{\mathbb{N}}\ (\mathsf{s}\ x\ ,\ \mathsf{s}\ y)$ is the very same type as $\mathsf{OWOTO}\ \mathsf{L}_{\mathbb{N}}\ (x, y)$.

Any ordering relation on elements lifts readily to bounds: I have overloaded the notation for lifting in the typesetting of this paper, but sadly not in the Agda source code. Let us take the opportunity to add propositional wrapping, to help us hide ordering proofs.

```
_⊤⊥ ⌐ : ∀{P} → Rel P → Rel P⊤⊥
L⊤⊥ (_ , ⊤)   = 1
L⊤⊥ (#x , #y) = L (x , y)
L⊤⊥ (⊥ , _)   = 1
L⊤⊥ (_ , _)   = 0
⌜L⌝ xy = ⌜L⊤⊥ xy⌝
```

The type $\ulcorner L \urcorner\ (x, y)$ thus represents ordering evidence on bounds with matching and construction by !, unaccompanied.

6. Equipment for Relations and Other Families

Before we get back to work in earnest, let us build a few tools for working with relations and other such indexed type families: a relation is a family which happens to be indexed by a pair. We shall have need of pointwise truth and falsity.

```
0̇ i : {I : Set} → I → Set
0̇ i = 0
1̇ i = 1
```

We shall also need to lift disjunction, conjunction and implication to to their pointwise counterparts.

```
_+̇_ _×̇_ _→̇_ : {I : Set} →
  (I → Set) → (I → Set) → I → Set
(S +̇ T) i = S i + T i
(S ×̇ T) i = S i × T i
(S →̇ T) i = S i → T i
infixr 3 _+̇_; infixr 4 _×̇_; infixr 2 _→̇_
```

Pointwise implication will be useful for writing *index-respecting* functions, e.g., bounds-preserving operations. It is useful to be able to state that something holds at every index (i.e., 'always works').

```
[_] : {I : Set} → (I → Set) → Set
[F] = ∀{i} → F i
```

With this apparatus, we can quite often talk about indexed things without mentioning the indices, resulting in code which almost looks like its simply typed counterpart. You can check that for any S and T, $\triangleleft : [S \to S +̇ T]$ and so forth.

7. Working with Bounded Sets

It will be useful to consider sets indexed by bounds in the same framework as relations on bounds: *propositions-as-types* means we have been doing this from the start! Useful combinator on such sets is the *pivoted pair*, $S\ \dot\wedge\ T$, indicating that some pivot value p exists, with S holding before p and T afterwards. A pattern synonym arranges the order neatly.

```
_∧̇_ : ∀{P} → Rel P⊤⊥ → Rel P⊤⊥ → Rel P⊤⊥
_∧̇_ {P} S T (l , u) = Σ P λ p → S (l , #p) × T (#p , u)
pattern _,,_ s p t = p , s , t
infixr 5 _,,_
```

Immediately, we can define an *interval* to be the type of an element proven to lie within given bounds.

```
_• : ∀{P} (L : Rel P) → Rel P⊤⊥
L• = ⌜L⌝ ∧̇ ⌜L⌝
pattern _° p = !, p, !
```

With habitual tidiness, a pattern synonym conceals the evidence.

Let us then parametrize over some

$$owoto\ :\ \forall x\ y\ \to\ \mathsf{OWOTO}\ L\ (x, y)$$

and reorganise our development.

```
data BST (lu : P⊤⊥ × P⊤⊥) : Set where
  leaf  : BST lu
  pnode : ((⌜L⌝ ×̇ BST) ∧̇ (⌜L⌝ ×̇ BST) →̇ BST) lu
pattern node lt p rt = pnode (p , (!, lt) , (!, rt))
```

Reassuringly, the standard undergraduate error, arising from thinking about *doing* rather than *being*, is now ill typed.

```
insert : [L• →̇ BST →̇ BST]
insert y° leaf            = node leaf y leaf
insert y° (node lt p rt) with owoto y p
... | le = (insert y° lt)
... | ge = (insert y° rt)
```

However, once we remember to restore the unchanged parts of the tree, we achieve victory, at last!

```
insert : [L• →̇ BST →̇ BST]
insert y° leaf            = node leaf y leaf
insert y° (node lt p rt) with owoto y p
... | le = node (insert y° lt) p rt
... | ge = node lt p (insert y° rt)
```

The evidence generated by testing $owoto\ y\ p$ is just what is needed to access the appropriate subtree. We have found a method which seems to work! But do not write home yet.

8. The Importance of Local Knowledge

Our current representation of an ordered tree with n elements contains $2n$ pieces of ordering evidence, which is $n-1$ too many. We should need only $n+1$ proofs, relating the lower bound to the least element, then comparing neighbours all the way along to the greatest element (one per element, so far) which must then fall below the upper bound (so, one more). As things stand, the pivot at the root is known to be greater than every element in the right spine of its left subtree and less than every element in the left spine of its right subtree. If the tree was built by iterated insertion, these comparisons will surely have happened, but that does not mean we should retain the information.

Suppose, for example, that we want to rotate a tree, perhaps to keep it balanced, then we have a little local difficulty:

```
rotR : [BST ⇢ BST]
rotR (node (node lt m mt) p rt)
  = node lt m (node mt p rt)
rotR t = t
```

Agda rejects the outer node of the rotated tree for lack of evidence. I expand the pattern synonyms to show what is missing.

```
rotR : [BST ⇢ BST]
rotR (pnode
  ((! {{lp}}) , pnode ((! {{lm}}) , lt) , m ,
   (! {{mp}}) , mt)))
  , p , (! {{pu}}) , rt))) = pnode ((! {{lm}}) , lt) , m ,
  (! {{ ?0 }}) , pnode ((! {{mp}}) , mt) , p , (! {{pu}}) , rt))))
rotR t = t
```

We can discard the non-local ordering evidence $lp : L^{\top}_{\bot}\ (l, \#p)$, but now we need the non-local $?_0 : L^{\top}_{\bot}\ (\#m, u)$ that we lack. Of course, we can prove this goal from mp and pu if L is transitive, but if we want to make less work, we should rather not demand non-local ordering evidence in the first place.

Looking back at the type of node, note that the indices at which we demand *ordering* are the same as the indices at which we demand *subtrees*. If we strengthen the invariant on trees to ensure that there is a sequence of ordering steps from the lower to the upper bound, we could dispense with the sometimes non-local evidence stored in nodes, at the cost of a new constraint for leaf.

```
data BST (lu : P⊤⊥ × P⊤⊥) : Set where
  pleaf : (⌜L⌝ ⇢ BST) lu
  pnode : (BST ∧̇ BST ⇢ BST) lu
pattern leaf       = pleaf !
pattern node lt p rt = pnode (lt , p , rt)
```

Indeed, a binary tree with n nodes will have $n+1$ leaves. An in-order traversal of a binary tree is a strict alternation, leaf-node-leaf-...-node-leaf, making a leaf the ideal place to keep the evidence that neighbouring nodes are in order! Insertion remains easy.

```
insert : [L• ⇢ BST ⇢ BST]
insert y° leaf = node leaf y leaf
insert y° (node lt p rt) with owoto y p
... | le = node (insert y° lt) p rt
... | ge = node lt p (insert y° rt)
```

Rotation becomes very easy: the above code now typechecks, with no leaves in sight, so no proofs to rearrange!

```
rotR : [BST ⇢ BST]
rotR (node (node lt m mt) p rt)
  = node lt m (node mt p rt)
rotR t = t
```

We have arrived at a neat way to keep a search tree in order, storing pivot elements at nodes and proofs in leaves. Phew!

But it is only the end of the beginning. To complete our sorting algorithm, we need to flatten binary search trees to ordered *lists*. Are we due another long story about the discovery of a good definition of the latter? Fortunately not! The key idea is that an ordered list is just a particularly badly balanced binary search tree, where every left subtree is a leaf. We can nail that down in short order, just by inlining leaf's data in the left subtree of node, yielding a sensible cons.

```
data OList (lu : P⊤⊥ × P⊤⊥) : Set where
  nil  : (⌜L⌝ ⇢ OList) lu
  cons : (⌜L⌝ ∧̇ OList ⇢ OList) lu
```

These are exactly the ordered lists Sam Lindley and I defined in Haskell [11], but now we can see where the definition comes from.

By figuring out how to build ordered binary search trees, we have actually discovered how to build quite a variety of in-order data structures. We simply need to show how the data are built from particular patterns of BST components. So, rather than flattening binary search trees, let us pursue a generic account of in-order datatypes, then flatten them *all*.

9. Jansson and Jeuring's PolyP Universe

If we want to see how to make the treatment of ordered container structures systematic, we shall need some datatype-generic account of recursive types with places for elements. A compelling starting point is the 'PolyP' system of Patrik Jansson and Johan Jeuring [8], which we can bottle as a universe—a system of codes for types—in Agda, as follows:

```
data JJ : Set where
  `R `P `1 : JJ
  _`+_ _`×_ : JJ → JJ → JJ
infixr 4 _`+_
infixr 5 _`×_
```

The `R stands for 'recursive substructure' and the `P stands for 'parameter'—the type of elements stored in the container. Given meanings for these, we interpret a code in JJ as a set.

```
⟦_⟧JJ : JJ → Set → Set → Set
⟦ `R ⟧JJ     R P = R
⟦ `P ⟧JJ     R P = P
⟦ `1 ⟧JJ     R P = 1
⟦ S `+ T ⟧JJ R P = ⟦ S ⟧JJ R P + ⟦ T ⟧JJ R P
⟦ S `× T ⟧JJ R P = ⟦ S ⟧JJ R P × ⟦ T ⟧JJ R P
```

When we 'tie the knot' in $\mu_{JJ}\ F\ P$, we replace F's `Ps by some actual $P$ and its `Rs by recursive uses of $\mu_{JJ}\ F\ P$.

```
data μJJ (F : JJ) (P : Set) : Set where
  ⟨_⟩ : ⟦ F ⟧JJ (μJJ F P) P → μJJ F P
```

Being finitary and first-order, all of the containers encoded by JJ are *traversable* in the sense defined by Ross Paterson and myself [14]. We shall need to introduce the interface for Applicative functors

```
record Applicative (H : Set → Set) : Set₁ where
  field
    pure : ∀{X} → X → H X
    ap   : ∀{S T} → H (S → T) → H S → H T
open Applicative
```

and then abstract over Applicative to compute the datatype generic treatment of traverse.

```
traverse : ∀{H F A B} → Applicative H →
  (A → H B) → μJJ F A → H (μJJ F B)
traverse {H} {F} {A} {B} AH h t = go ‘R t where
  pu = pure AH; _⊛_ = ap AH
  go : ∀G →
    ⟦ G ⟧JJ (μJJ F A) A → H (⟦ G ⟧JJ (μJJ F B) B)
  go ‘R       ⟨ t ⟩   = pu ⟨_⟩ ⊛ go F t
  go ‘P       a       = h a
  go ‘1       ⟨⟩      = pu ⟨⟩
  go (S ‘+ T) (◁ s)   = pu ◁ ⊛ go S s
  go (S ‘+ T) (▷ t)   = pu ▷ ⊛ go T t
  go (S ‘× T) (s , t) = (pu _,_ ⊛ go S s) ⊛ go T t
```

We can specialise traverse to standard functorial map by choosing the identity functor.

```
idApp : Applicative (λ X → X)
idApp = record {pure = id; ap = id}
map : ∀{F A B} →
  (A → B) → μJJ F A → μJJ F B
map = traverse idApp
```

We can equally well specialise traverse to a monoidal crush by choosing a constant functor.

```
record Monoid (X : Set) : Set where
  field
    neutral : X
    combine : X → X → X
  monApp : Applicative (λ _ → X)
  monApp = record
    {pure = λ _ → neutral; ap = combine}
  crush : ∀{P F} → (P → X) → μJJ F P → X
  crush = traverse {B = 0} monApp
open Monoid
```

Perversely, the fact that the constant functor discards the return value type, B in traverse's type signature, results in the absence of constraints on B in the definition of crush, and hence the need to give B explicitly. I choose 0 merely to emphasize that B-values are not involved.

Endofunctions on a given set form a monoid with respect to composition, which allows us a generic foldr-style operation.

```
compMon : ∀{X} → Monoid (X → X)
compMon = record
  {neutral = id; combine = λ f g → f ∘ g}
foldr : ∀{F A B} →
  (A → B → B) → B → μJJ F A → B
foldr f b t = crush compMon f t b
```

We can use foldr to build up Bs from any structure containing As, given a way to 'insert' an A into a B, and an 'empty' B to start with. Let us check that our generic machinery is fit for purpose.

10. The Simple Orderable Subuniverse of JJ

The quicksort algorithm divides a sorting problem in two by partitioning about a selected *pivot* element the remaining data. Rendered as the process of building then flattening a binary search tree [4], the pivot element clearly marks the upper bound of the lower subtree and the lower bound of the upper subtree, giving exactly the information required to guide insertion.

We can require the presence of pivots between substructures by combining the parameter ‘P and pairing ‘× constructs of the PolyP universe into a single pivoting construct, ‘∧, with two substructures and a pivot in between. We thus acquire the simple orderable universe, SO, a subset of JJ picked out as the image of a function, ⟦_⟧SO. Now, ‘P stands also for pivot!

```
data SO : Set where
  ‘R ‘1   : SO
  _‘+_ _‘∧_ : SO → SO → SO
infixr 5 _‘∧_

⟦_⟧SO : SO → JJ
⟦ ‘R ⟧SO       = ‘R
⟦ ‘1 ⟧SO       = ‘1
⟦ S ‘+ T ⟧SO   = ⟦ S ⟧SO ‘+ ⟦ T ⟧SO
⟦ S ‘∧ T ⟧SO   = ⟦ S ⟧SO ‘× ‘P ‘× ⟦ T ⟧SO

μSO : SO → Set → Set
μSO F P = μJJ ⟦ F ⟧SO P
```

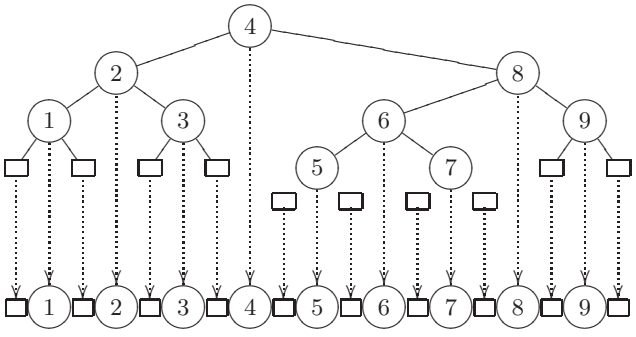
Let us give SO codes for structures we often order and bound:

```
‘List ‘Tree ‘Interval : SO
‘List     = ‘1 ‘+ (‘1 ‘∧ ‘R)
‘Tree     = ‘1 ‘+ (‘R ‘∧ ‘R)
‘Interval = ‘1 ‘∧ ‘1
```

Every data structure described by SO is a regulated variety of node-labelled binary trees. Let us check that we can turn anything into a tree, preserving the substructure relationship. The method[1] is to introduce a helper function, go, whose type separates G, the structure of the top node, from F the structure of recursive subnodes, allowing us to take the top node apart: we kick off with $G = F$.

```
tree : ∀{P F} → μSO F P → μSO ‘Tree P
tree {P} {F} ⟨ f ⟩ = go F f where
  go : ∀G → ⟦ ⟦ G ⟧SO ⟧JJ (μSO F P) P → μSO ‘Tree P
  go ‘R       f       = tree f
  go ‘1       ⟨⟩      = ⟨ ◁ ⟨⟩ ⟩
  go (S ‘+ T) (◁ s)   = go S s
  go (S ‘+ T) (▷ t)   = go T t
  go (S ‘∧ T) (s , p , t) = ⟨ ▷ (go S s , p , go T t) ⟩
```

All tree does is strip out the ◁s and ▷s corresponding to the structural choices offered by the input type and instead label the void leaves ◁ and the pivoted nodes ▷. Note well that a singleton tree has void leaves as its left and right substructures, and hence that the inorder traversal is a strict alternation of leaves and pivots, beginning with the leaf at the end of the left spine and ending with the leaf at the end of the right spine. As our tree function preserves the leaf/pivot structure of its input, we learn that *every* datatype we can define in SO stores such an alternation of leaves and pivots.



We are now in a position to roll out the "loose bounds" method to the whole of the SO universe. We need to ensure that each pivot

[1] If you try constructing the division operator as a primitive recursive function, this method will teach itself to you.

is in order with its neighbours and with the outer bounds, and the alternating leaf/pivot structure gives us just what we need: let us store the ordering evidence at the leaves!

```
⟦_⟧≤SO : SO → ∀{P} → Rel P⊤⊥ → Rel P → Rel P⊤⊥
⟦ 'R ⟧≤SO        R L = R
⟦ '1 ⟧≤SO        R L = ⌜L⌝
⟦ S '+ T ⟧≤SO R L = ⟦ S ⟧≤SO R L ∔ ⟦ T ⟧≤SO R L
⟦ S '∧ T ⟧≤SO R L = ⟦ S ⟧≤SO R L ∧̇ ⟦ T ⟧≤SO R L

data μ≤SO (F : SO) {P : Set} (L : Rel P)
          (lu : P⊤⊥ × P⊤⊥) : Set where
  ⟨⟩ : ⟦ F ⟧≤SO (μ≤SO F L) L lu → μ≤SO F L lu
```

We have shifted from sets to relations, in that our types are indexed by lower and upper bounds. The leaves demand evidence that the bounds are in order, whilst the nodes require the pivot first, then use it to bound the substructures appropriately.

Meanwhile, the need in nodes to bound the left substructure's type with the pivot value disrupts the left-to-right spatial ordering of the data, but we can apply a little cosmetic treatment, thanks to the availability of pattern synonyms.

With these two devices available, let us check that we can still turn any ordered data into an ordered tree, writing $L^{\Delta}\ (l, u)$ for $\mu^{\leq}_{SO}$ 'Tree $L\ (l, u)$, and redefining intervals accordingly.

```
_Δ _• : ∀{P} → Rel P → Rel P⊤⊥

L^Δ = μ≤SO 'Tree    L
pattern leaf          = ⟨◁ !⟩
pattern node lp p pu  = ⟨▷ (lp ,p, pu)⟩

L^• = μ≤SO 'Interval L
pattern _° p = ⟨(p ,!, !)⟩

tree : ∀{P F} {L : Rel P} → [μ≤SO F L ⇢ L^Δ]
tree {P} {F} {L} ⟨f⟩ = go F f where
  go : ∀G → [⟦ G ⟧≤SO (μ≤SO F L) L ⇢ L^Δ]
  go 'R       f       = tree f
  go '1       !       = leaf
  go (S '+ T) (◁ s)   = go S s
  go (S '+ T) (▷ t)   = go T t
  go (S '∧ T) (s ,p, t) = node (go S s) p (go T t)
```

We have acquired a collection of orderable datatypes which all amount to specific patterns of node-labelled binary trees: an interval is a singleton node; a list is a right spine. All share the treelike structure which ensures that pivots alternate with leaves bearing the evidence the pivots are correctly placed with respect to their immediate neighbours.

Let us check that we are where we were, so to speak. Hence we can rebuild our binary search tree insertion for an element in the corresponding interval:

```
insert : [L^• ⇢ L^Δ ⇢ L^Δ]
insert y° leaf            = node leaf y leaf
insert y° (node lt p rt) with owoto y p
... | le = node (insert y° lt) p rt
... | ge = node lt p (insert y° rt)
```

The constraints on the inserted element are readily expressed via our 'Interval type, but at no point need we ever name the ordering evidence involved. The *owoto* test brings just enough new evidence into scope that all proof obligations on the right-hand side can be discharged by search of assumptions. We can now make a search tree from any input container.

```
makeTree : ∀{F} → μJJ F P → L^Δ (⊥, ⊤)
makeTree = foldr (λ p → insert p°) leaf
```

11. Digression: Merging Monoidally

Let us name our family of ordered lists L^{+}, as the leaves form a nonempty chain of $\ulcorner L \urcorner$ ordering evidence.

```
_+ : ∀{P} → Rel P → Rel P⊤⊥
L^+ = μ≤SO 'List L
pattern []       = ⟨◁ !⟩
pattern _::_ x xs = ⟨▷ (x ,!, xs)⟩
infixr 6 _::_
```

The next section addresses the issue of how to *flatten* ordered structures to ordered lists, but let us first consider how to *merge* them. Merging sorts differ from flattening sorts in that order is introduced when 'conquering' rather than 'dividing'.

We can be sure that whenever two ordered lists share lower and upper bounds, they can be merged within the same bounds. Again, let us assume a type P of pivots, with *owoto* witnessing the totality of order L. The familiar definition of merge typechecks but falls just outside the class of lexicographic recursions accepted by Agda's termination checker. I have locally expanded pattern synonyms to dig out the concealed evidence which causes the trouble.

```
merge : [L^+ ⇢ L^+ ⇢ L^+]
merge []        ys       = ys
merge xs        []       = xs
merge ⟨▷ (! {{_}}) ,x, xs⟩ (y :: ys) with owoto x y
... | le = x :: merge xs (y :: ys)
... | ge = y :: merge ⟨▷ (! {{_}}) ,x, xs⟩ ys
```

In one step case, the first list gets smaller, but in the other, where we decrease the second list, the first does not remain the same: it contains fresh evidence that x is above the tighter lower bound, y. Separating the recursion on the second list is sufficient to show that both recursions are structural.

```
merge : [L^+ ⇢ L^+ ⇢ L^+]
merge       []       = id
merge {l, u} (x :: xs) = go where
  go : ∀{l} {{_ : L^⊤⊥ (l, #x)}} → (L^+ ⇢ L^+) (l, u)
  go []       = x :: xs
  go (y :: ys) with owoto x y
  ... | le = x :: merge xs (y :: ys)
  ... | ge = y :: go ys
```

The helper function go inserts x at its rightful place in the second list, then resumes merging with xs.

Merging equips ordered lists with monoidal structure.

```
olMon : ∀{lu} → L^⊤⊥ lu ⇒ Monoid (L^+ lu)
olMon = record {neutral = []; combine = merge}
```

An immediate consequence is that we gain a family of sorting algorithms which amount to depth-first merging of a given intermediate data structure, making a singleton from each pivot.

```
mergeJJ : ∀{F} → μJJ F P → L^+ (⊥, ⊤)
mergeJJ = crush olMon λ p → p :: []
```

The instance of merge_{JJ} for *lists* is exactly *insertion* sort: at each cons, the singleton list of the head is merged with the sorted tail. To obtain an efficient mergeSort, we should arrange the inputs as a leaf-labelled binary tree.

```
'qLTree : JJ
'qLTree = ('1 '+ 'P) '+ 'R '× 'R
```

As ever, pattern synonyms prove invaluable for restoring readability.

```
pattern none    = ⟨◁ (◁ ⟨⟩)⟩
pattern one p   = ⟨◁ (▷ p)⟩
pattern fork l r = ⟨▷ (l , r)⟩
```

We can add each successive elements to the tree with a twisting insertion, placing the new element at the bottom of the left spine, but swapping the subtrees at each layer along the way to ensure fair distribution.

```
twistIn : P → μJJ ‘qLTree P → μJJ ‘qLTree P
twistIn p none       = one p
twistIn p (one q)    = fork (one p) (one q)
twistIn p (fork l r) = fork (twistIn p r) l
```

If we notice that twistIn maps elements to endofunctions on trees, we can build up trees by a monoidal crush, obtaining an efficient generic sort for any container in the JJ universe.

```
mergeSort : ∀{F} → μJJ F P → L⁺ (⊥ , ⊤)
mergeSort = mergeJJ ∘ foldr twistIn none
```

12. Flattening With Concatenation

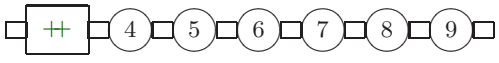
Several sorting algorithms amount to building an ordered intermediate structure, then flattening it to an ordered list. As all of our orderable structures amount to trees, it suffices to flatten trees to lists. Let us take the usual naïve approach as our starting point. In Haskell, we might write

```
flatten Leaf           = []
flatten (Node l p r) = flatten l ++ p : flatten r
```

so let us try to do the same in Agda with ordered lists. We shall need concatenation, so let us try to join lists with a shared bound p in the middle.

```
infixr 8 _++_
_++_ : ∀{P} {L : Rel P} {l p u} →
  L⁺ (l , p) → L⁺ (p , u) → L⁺ (l , u)
[]        ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

The ‘cons’ case goes without a hitch, but there is trouble at ‘nil’. We have $ys : \mu^{\leq}_{\mathsf{SO}}$ ‘List $L\ (p , u)$ and we know $L^{\top}_{\bot}\ (l , p)$, but we need to return a $\mu^{\leq}_{\mathsf{SO}}$ ‘List $L\ (l , u)$.



“The trouble is easy to fix,” one might confidently assert, whilst secretly thinking, “What a nuisance!”. We can readily write a helper function which unpacks ys, and whether it is nil or cons, extends its leftmost order evidence by transitivity. And this really is a nuisance, because, thus far, we have not required transitivity to keep our code well typed: all order evidence has stood between neighbouring elements. Here, we have two pieces of ordering evidence which we must join, because we have nothing to put in between them. Then, the penny drops. Looking back at the code for flatten, observe that p is the pivot and the whole plan is to put it between the lists. You can’t always get what you want, but you can get what you need.

```
sandwich : ∀{P} {L : Rel P} → [(L⁺ ∧̇ L⁺) ⇾̇ L⁺]
sandwich ([]       , p , ys) = p :: ys
sandwich (x :: xs , p , ys) = x :: sandwich (xs , p , ys)
```

We are now ready to flatten trees, thence any ordered structure:

```
flatten : ∀{P} {L : Rel P} → [L^Δ ⇾̇ L⁺]
flatten leaf          = []
flatten (node l p r) = sandwich (flatten l , p , flatten r)
```

```
flatten^≤_SO : ∀{P} {L : Rel P} {F} → [μ^≤_SO F L ⇾̇ L⁺]
flatten^≤_SO = flatten ∘ tree
```

For a little extra speed we might fuse that composition, but it seems frivolous to do so as the benefit is outweighed by the quadratic penalty of left-nested concatenation. The standard remedy applies: we can introduce an accumulator [18], but our experience with ++ should alert us to the possibility that it may require some thought.

13. Faster Flattening, Generically

We may define flatten generically, and introduce an accumulator yielding a combined flatten-and-append which works right-to-left, growing the result with successive conses. But what should be the bounds of the accumulator? If we have not learned our lesson, we might be tempted by

```
flapp : ∀{F P} {L : Rel P} {l p u} →
  μ^≤_SO F L (l , p) → L⁺ (p , u) → L⁺ (l , u)
```

but again we face the question of what to do when we reach a leaf. We should not need transitivity to rearrange a tree of ordered neighbours into a sequence. We can adopt the previous remedy of inserting the element p in the middle, but we shall then need to think about where p will come from in the first instance, for example when flattening an empty structure.

```
flapp : ∀{F P} {L : Rel P} G →
  [[ G ]]^≤_SO (μ^≤_SO F L) L ∧̇ L⁺ ⇾̇ L⁺]
flapp {F} ‘R (⟨ t ⟩     , p , ys) = flapp F (t , p , ys)
flapp ‘1     (!         , p , ys) = p :: ys
flapp (S ‘+ T) (◁ s     , p , ys) = flapp S (s , p , ys)
flapp (S ‘+ T) (▷ t     , p , ys) = flapp T (t , p , ys)
flapp (S ‘∧ T) ((s , p′ , t) , p , ys)
  = flapp S (s , p′ , flapp T (t , p , ys))
```

To finish the job, we need to work our way down the right spine of the input in search of its rightmost element, which initialises p.

```
flatten : ∀{F P} {L : Rel P} → [μ^≤_SO F L ⇾̇ L⁺]
flatten {F} {P} {L} {l , u} ⟨ t ⟩ = go F t where
  go : ∀{l} G → [[ G ]]^≤_SO (μ^≤_SO F L) L (l , u) → L⁺ (l , u)
  go ‘R     t     = flatten t
  go ‘1     !     = []
  go (S ‘+ T) (◁ s) = go S s
  go (S ‘+ T) (▷ t) = go T t
  go (S ‘∧ T) (s , p , t) = flapp S (s , p , go T t)
```

This is effective, but it is more complicated than I should like. It is basically the same function twice, in two different modes, depending on what is to be affixed *after* the rightmost order evidence in the structure being flattened: either a pivot-and-tail in the case of flapp, or nothing in the case of flatten. The problem is one of parity: the thing we must affix to one odd-length leaf-node-leaf alternation to get another is an even-length node-leaf alternation. Correspondingly, it is hard to express the type of the accumulator cleanly. Once again, I begin to suspect that this is a difficult thing to do because it is the wrong thing to do. How can we reframe the problem, so that we work only with odd-length leaf-delimited data?

14. A Replacement for Concatenation

My mathematical mentor, Tom Körner, is fond of remarking “A mathematician is someone who knows that 0 is 0 + 0”. It is often difficult to recognize the structure you need when the problem in front of you is a degenerate case of it. If we think again about concatenation, we might realise that it does not amount to *affixing*

one list to another, but rather *replacing* the 'nil' of the first list with the whole of the second. We might then notice that the *monoidal* structure of lists is in fact degenerate *monadic* structure.

Any syntax has a monadic structure, where 'return' embeds variables as terms and 'bind' is substitution. Quite apart from their 'prioritised choice' monadic structure, lists are the terms of a degenerate syntax with one variable (called 'nil') and only unary operators ('cons' with a choice of element). Correspondingly, they have this substitution structure: substituting nil gives concatenation, and the monad laws are the monoid laws.

Given this clue, let us consider concatenation and flattening in terms of *replacing* the rightmost leaf by a list, rather than affixing more data to it. We replace the list to append with a function which maps the contents of the rightmost leaf—some order evidence—to its replacement. The type looks more like that of 'bind' than 'append', because in some sense it is!

```
infixr 8 _++_
RepL : ∀{P} → Rel P → Rel P⊤⊥
RepL L (n, u) = ∀{m} → L⊤⊥ (m, n) ⇒ L⁺ (m, u)
_++_ : ∀{P}{L : Rel P}{l n u} →
  L⁺ (l, n) → RepL L (n, u) → L⁺ (l, u)
[]        ++ ys = ys
(x :: xs) ++ ys = x :: xs ++ ys
```

Careful use of instance arguments leaves all the manipulation of evidence to the machine. In the [] case, ys is silently instantiated with exactly the evidence exposed in the [] pattern on the left.

Let us now deploy the same technique for flatten.

```
flapp : ∀{P}{L : Rel P}{F}{l n u} →
  μ≤SO F L (l, n) → RepL L (n, u) → L⁺ (l, u)
flapp {P}{L}{F}{u = u} t ys = go 'R t ys where
  go : ∀{l n} G → ⟦ G ⟧≤SO (μ≤SO F L) L (l, n) →
       RepL L (n, u) → L⁺ (l, u)
  go 'R      ⟨ t ⟩     ys = go F t ys
  go '1      !         ys = ys
  go (S '+ T) (◁ s)    ys = go S s ys
  go (S '+ T) (▷ t)    ys = go T t ys
  go (S '∧ T) (s ,p, t) ys = go S s (p :: go T t ys)
flatten : ∀{P}{L : Rel P}{F} → [μ≤SO F L ·→ L⁺]
flatten t = flapp t []
```

15. An Indexed Universe of Orderable Data

Ordering is not the only invariant we might want to enforce on orderable data structures. We might have other properties in mind, such as size, or balancing invariants. It is straightforward to extend our simple universe to allow general indexing as well as orderability. We can extend our simple orderable universe SO to an indexed orderable universe IO, just by marking each recursive position with an index, then computing the code for each node as a function of its index. We may add a '0 code to rule out some cases as illegal.

```
data IO (I : Set) : Set where
  'R      : I → IO I
  '0 '1   : IO I
  _'+_ _'∧_ : IO I → IO I → IO I
```

When interpreting such a code, we now require the family of relations which sit in recursive positions, one for each element of the index set, I. However, the interpretation function is not concerned with indexing the overall node. The function mapping each index to the code for the appropriate node structure appears only when we tie the recursive knot.

```
⟦_⟧≤IO : ∀{I P} → IO I →
         (I → Rel P⊤⊥) → Rel P → Rel P⊤⊥
⟦ 'R i ⟧≤IO     R L = R i
⟦ '0 ⟧≤IO       R L = λ _ → 0
⟦ '1 ⟧≤IO       R L = ⌜L⌝
⟦ S '+ T ⟧≤IO R L = ⟦ S ⟧≤IO R L ∔ ⟦ T ⟧≤IO R L
⟦ S '∧ T ⟧≤IO R L = ⟦ S ⟧≤IO R L ∧̇ ⟦ T ⟧≤IO R L
data μ≤IO {I P : Set} (F : I → IO I) (L : Rel P)
          (i : I) (lu : P⊤⊥ × P⊤⊥) : Set where
  ⟨⟩ : ⟦ F i ⟧≤IO (μ≤IO F L) L lu → μ≤IO F L i lu
```

We recover all our existing data structures by trivial indexing.

```
'List 'Tree 'Interval : 1 → IO 1
'List     _ = '1 '+ ('1 '∧ 'R ⟨⟩)
'Tree     _ = '1 '+ ('R ⟨⟩ '∧ 'R ⟨⟩)
'Interval _ = '1 '∧ '1
```

We also lift our existing type-forming abbreviations:

```
_⁺ _^Δ _• : ∀{P} → Rel P → Rel P⊤⊥
L⁺ = μ≤IO 'List     L ⟨⟩
L^Δ = μ≤IO 'Tree    L ⟨⟩
L• = μ≤IO 'Interval L ⟨⟩
```

However, we may also make profitable use of indexing: here are ordered *vectors*.

```
'Vec : ℕ → IO ℕ
'Vec 0     = '1
'Vec (s n) = '1 '∧ 'R n
```

Note that we need no choice of constructor or storage of length information: the index determines the shape. If we want, say, even-length tuples, we can use '0 to rule out the odd cases.

```
'Even : ℕ → IO ℕ
'Even 0       = '1
'Even (s 0)   = '0
'Even (s (s n)) = '1 '∧ '1 '∧ 'R n
```

We could achieve a still more flexible notion of data structure by allowing a general Σ-type rather than our binary '+, but we have what we need for finitary data structures with computable conditions on indices.

The tree operation carries over unproblematically, with more indexed input but plain output.

```
tree : ∀{I P F}{L : Rel P}{i : I} →
  [μ≤IO F L i ·→ L^Δ]
```

Similarly, flatten works (efficiently) just as before.

```
flatten : ∀{I P F}{L : Rel P}{i : I} →
  [μ≤IO F L i ·→ L⁺]
```

We now have a universe of indexed orderable data structures with efficient flattening. Let us put it to work.

16. Balanced 2-3 Trees

To ensure a logarithmic access time for search trees, we can keep them balanced. Maintaining balance as close to perfect as possible is rather fiddly, but we can gain enough balance by allowing a little redundancy. A standard way to achieve this is to insist on uniform height, but allow internal nodes to have either one pivot and two subtrees, or two pivots and three subtrees. We may readily encode these *2-3 trees* and give pattern synonyms for the three kinds of structure. This approach is much like that of *red-black* (effectively,

2-3-4) trees, for which typesafe balancing has a tradition going back to Hongwei Xi and Stefan Kahrs [9, 19].

As with ‘Vec, case analysis on the index, now representing height, tells us whether we are at a leaf or an internal node.

```
‘Tree23 : ℕ → IO ℕ
‘Tree23 0       = ‘1
‘Tree23 (s h)   = ‘R h ‘∧ (‘R h ‘+ (‘R h ‘∧ ‘R h))

_23 : ∀{P} (L : Rel P) → ℕ → Rel P⊤⊥
L23 = μIO≤ ‘Tree23 L

pattern no0                = ⟨!⟩
pattern no2 lt p rt        = ⟨p, lt, ◁ rt⟩
pattern no3 lt p mt q rt   = ⟨p, lt, ▷ (q, mt, rt)⟩
```

When we map a 2-3 tree of height n back to binary trees, we get a tree whose left spine has length n and whose right spine has a length between n and $2n$.

Insertion is quite similar to binary search tree insertion, except that it can have the impact of increasing height. The worst that can happen is that the resulting tree is too tall but has just one pivot at the root. Indeed, we need this extra wiggle room immediately for the base case!

```
ins23 : ∀h {lu} → L• lu → L23 h lu →
        L23 h lu +
        Σ P λ p → L23 h (π1 lu, #p) × L23 h (#p, π2 lu)
ins23 0 y° no0 = ▷ (⟨!⟩, y, ⟨!⟩)
```

In the step case, we must find our way to the appropriate subtree by suitable use of comparison.

```
ins23 (s h) y° ⟨ lt, p, rest ⟩       with owoto y p
ins23 (s h) y° ⟨ lt, p, rest ⟩       | le = ?0
ins23 (s h) y° (no2 lt p rt)         | ge = ?1
ins23 (s h) y° (no3 lt p mt q rt)    | ge with owoto y q
ins23 (s h) y° (no3 lt p mt q rt)    | ge | le = ?2
ins23 (s h) y° (no3 lt p mt q rt)    | ge | ge = ?3
```

Our ?0 covers the case where the new element belongs in the left subtree of either a 2- or 3-node; ?1 handles the right subtree of a 2-node; ?2 and ?3 handle middle and right subtrees of a 3-node after a further comparison. Note that we inspect *rest* only after we have checked the result of the first comparison, making real use of the way the **with** construct brings more data to the case analysis but keeps the existing patterns open to further refinement, a need foreseen by the construct's designers [13].

Once we have identified the appropriate subtree, we can make the recursive call. If we are lucky, the result will plug straight back into the same hole. Here is the case for the left subtree.

```
ins23 (s h) y° ⟨ lt, p, rest ⟩ | le
  with ins23 h y° lt
ins23 (s h) y° ⟨ lt, p, rest ⟩ | le
  | ◁ lt′ = ◁ ⟨ lt′, p, rest ⟩
```

However, if we are unlucky, the result of the recursive call is too big. If the top node was a 2-node, we can accommodate the extra data by returning a 3-node. Otherwise, we must rebalance and pass the ‘too big’ problem upward. Again, we gain from delaying the inspection of *rest* until we are sure reconfiguration will be needed.

```
ins23 (s h) y° (no2 lt p rt)         | le
  | ▷ (llt, r, lrt) = ◁ (no3 llt r lrt p rt)
ins23 (s h) y° (no3 lt p mt q rt)    | le
  | ▷ (llt, r, lrt) = ▷ (no2 llt r lrt, p, no2 mt q rt)
```

For the ?1 problems, the top 2-node can always accept the result of the recursive call somehow, and the choice offered by the return type conveniently matches the node-arity choice, right of the pivot. For completeness, I give the middle (?2) and right (?3) cases for 3-nodes, but it works just as on the left.

```
ins23 (s h) y° (no3 lt p mt q rt) | ge | le
  with ins23 h y° mt
ins23 (s h) y° (no3 lt p mt q rt) | ge | le
  | ◁ mt′          = ◁ (no3 lt p mt′ q rt)
ins23 (s h) y° (no3 lt p mt q rt) | ge | le
  | ▷ (mlt, r, mrt) = ▷ (no2 lt p mlt, r, no2 mrt q rt)
ins23 (s h) y° (no3 lt p mt q rt) | ge | ge
  with ins23 h y° rt
ins23 (s h) y° (no3 lt p mt q rt) | ge | ge
  | ◁ rt′          = ◁ (no3 lt p mt q rt′)
ins23 (s h) y° (no3 lt p mt q rt) | ge | ge
  | ▷ (rlt, r, rrt) = ▷ (no2 lt p mt, q, no2 rlt r rrt)
```

Pleasingly, the task of constructing suitable return values in each of these cases is facilitated by Agda's type directed search gadget, *Agsy* [10]. There are but two valid outputs constructible from the pieces available: the original tree reconstituted, and the correct output.

To complete the efficient sorting algorithm based on 2-3 trees, we can use a Σ-type to hide the height data, giving us a type which admits iterative construction.

```
Tree23 = Σ ℕ λ h → L23 h (⊥, ⊤)
insert : P → Tree23 → Tree23
insert p (h, t) with ins23 h p° t
... | ◁ t′          = h  , t′
... | ▷ (lt, r, rt) = s h, no2 lt r rt
sort : ∀{F} → μJJ F P → L+ (⊥, ⊤)
sort = flatten ∘ π2 ∘ foldr insert (0, no0)
```

17. Deletion from 2-3 Trees

Might is right: the omission of *deletion* from treatments of balanced search trees is always a little unfortunate [15]. Deletion is a significant additional challenge because we can lose a key from the *middle* of the tree, not just from the *fringe* of nodes whose children are leaves. Insertion acts always to extend the fringe, so the problem is only to bubble an anomaly up from the fringe to the root. Fortunately, just as nodes and leaves alternate in the traversal of a tree, so do middle nodes and fringe nodes: whenever we need to delete a middle node, it always has a neighbour at the fringe which we can move into the gap, leaving us once more with the task of bubbling a problem up from the fringe.

Our situation is further complicated by the need to restore the neighbourhood ordering invariant when one key is removed. At last, we shall need our ordering to be transitive. We shall also need a decidable equality on keys.

```
data _≡_ {X : Set} (x : X) : X → Set where
  ⟨⟩ : x ≡ x
infix 6 _≡_

trans : ∀{x} y {z} → L (x, y) ⇒ L (y, z) ⇒ ⌜L (x, z)⌝

eq? : (x y : P) → x ≡ y + (x ≡ y → 0)
```

Correspondingly, a small amount of theorem proving is indicated, ironically, to show that it is sound to throw information about local ordering away.

Transitivity for bounds. Transitivity we may readily lift to bounds with a key in the middle:

```
pattern via p = p,!,!

trans⊤⊥ : [(⌜L⌝ ∧̇ ⌜L⌝) ⇢̇ ⌜L⌝]
trans⊤⊥ {_ , ⊤} _        = !
trans⊤⊥ {⊥ , ⊥} _        = !
trans⊤⊥ {⊥ , #u} _       = !
trans⊤⊥ {⊤ , _} (via _)  = magic
trans⊤⊥ {#l , #u} (via p) = trans p ∴ !
trans⊤⊥ {#l , ⊥} (via _) = magic
```

What is the type of deletion? When we remove an element from a 2-3 tree of height n, the tree will often stay the same height, but there will be situations in which it must get shorter, becoming a 3-node or a leaf, as appropriate.

```
Del²³ Short²³ : ℕ → Rel P⊤⊥
Del²³  h lu     = Short²³ h lu + L²³ h lu
Short²³ 0 lu    = 0
Short²³ (s h) lu = L²³ h lu
```

The task of deletion has three phases: finding the key to delete; moving the problem to the fringe; plugging a short tree into a tall hole. The first of these will be done by our main function,

```
del²³ : ∀{h} → [L• ⇢̇ L²³ h ⇢̇ Del²³ h]
```

and the second by extracting the extreme right key from a nonempty left subtree,

```
extr : ∀{h} → [L²³ (s h) ⇢̇ (Del²³ (s h) ∧̇ L⊤⊥)]
```

recovering the (possiy short) remainder of the tree and the evidence that the key is below the upper bound (which will be the deleted key). Both of these operations will need to reconstruct trees with one short subtree, so let us build 'smart constructors' for just that purpose, then return to the main problem.

Rebalancing reconstructors. If we try to reconstruct a 2-node with a possibly-short subtree, we might be lucky enough to deliver a 2-node, or we might come up short. We certainly will not deliver a 3-node of full height and it helps to reflect that in the type. Shortness can be balanced out if we are adjacent to a 3-node, but if we have only a 2-node, we must give a short answer.

```
Re2 : ℕ → Rel P⊤⊥
Re2 h = Short²³ (s h) ∔̇ (L²³ h ∧̇ L²³ h)

d2t : ∀{h} → [(Del²³ h ∧̇ L²³ h) ⇢̇ Re2 h]
d2t {h}   (▷ lp , p , pu)              = ▷ (lp , p , pu)
d2t {0}   (◁ () , p , pu)
d2t {s h} (◁ lp , p , no₂ pq q qu)     = ◁ (no₃ lp p pq q qu)
d2t {s h} (◁ lp , p , no₃ pq q qr r ru)
  = ▷ (no₂ lp p pq , q , no₂ qr r ru)

t2d : ∀{h} → [(L²³ h ∧̇ Del²³ h) ⇢̇ Re2 h]
t2d {h}   (lp , p , ▷ pu)              = ▷ (lp , p , pu)
t2d {0}   (lp , p , ◁ ())
t2d {s h} (no₂ ln n np , p , ◁ pu)     = ◁ (no₃ ln n np p pu)
t2d {s h} (no₃ lm m mn n np , p , ◁ pu)
  = ▷ (no₂ lm m mn , n , no₂ np p pu)

rd : ∀{h} → [Re2 h ⇢̇ Del²³ (s h)]
rd (◁ s)          = (◁ s)
rd (▷ (lp , p , pu)) = ▷ (no₂ lp p pu)
```

The adaptor rd allows us to throw away the knowledge that the full height reconstruction must be a 2-node if we do not need it, but the extra detail allows us to use 2-node reconstructors in the course of 3-node reconstruction. To reconstruct a 3-node with one possibly-short subtree, rebuild a 2-node containing the suspect, and then restore the extra subtree. We thus need to implement the latter.

```
r3t : ∀{h} → [(Re2 h ∧̇ L²³ h) ⇢̇ Del²³ (s h)]
r3t (▷ (lm , m , mp) , p , pu) = ▷ (no₃ lm m mp p pu)
r3t (◁ lp , p , pu)            = ▷ (no₂ lp p pu)

t3r : ∀{h} → [(L²³ h ∧̇ Re2 h) ⇢̇ Del²³ (s h)]
t3r (lp , p , ▷ (pq , q , qu))  = ▷ (no₃ lp p pq q qu)
t3r (lp , p , ◁ pu)             = ▷ (no₂ lp p pu)
```

Cutting out the extreme right. We may now implement extr, grabbing the rightmost key from a tree. I use

```
pattern _△ lr r = r , lr , !
```

to keep the extracted element on the right and hide the ordering proofs.

```
extr : ∀{h} → [L²³ (s h) ⇢̇ (Del²³ (s h) ∧̇ ⌜L⌝)]
extr {0} (no₂ lr r no₀)          = ◁ lr △ r
extr {0} (no₃ lp p pr r no₀)     = ▷ (no₂ lp p pr) △ r
extr {s h} (no₂ lp p pu)         with extr pu
... | pr △ r = rd (t2d (lp , p , pr)) △ r
extr {s h} (no₃ lp p pq q qu)    with extr qu
... | qr △ r = t3r (lp , p , t2d (pq , q , qr)) △ r
```

To delete the pivot key from between two trees, we extract the rightmost key from the left tree, then weaken the bound on the right tree (traversing its left spine only). Again, we are sure that if the height remains the same, we shall deliver a 2-node.

```
delp : ∀{h} → [(L²³ h ∧̇ L²³ h) ⇢̇ Re2 h]
delp {0} {lu} (no₀ , p , no₀) = trans⊤⊥ {lu} (via p) ∴ ◁ no₀
delp {s h}    (lp , p , pu)   with extr lp
... | lr △ r = d2t (lr , r , weak pu) where
  weak : ∀{h u} → L²³ h (#p , u) → L²³ h (#r , u)
  weak {0} {u} no₀ = trans⊤⊥ {#r , u} (via p) ∴ no₀
  weak {s h} ⟨ pq , q , qu ⟩ = ⟨ weak pq , q , qu ⟩
```

A remark on weakenings. It may seem regrettable that we have to write weak, which is manifestly an obfuscated identity function, and programmers who do not wish the ordering guarantees are entitled not to pay and not to receive. If we took an extrinsic approach to managing these invariants, weak would still be present, but it would just be the proof of the proposition that you can lower a lower bound that you know for a tree. Consequently, the truly regrettable thing about weak is not that it is written but that it is *executed*. The 'colored' analysis of Bernardy and Moulin offers a suitable method to ensure that the weakening operation belongs to code which is erased at run time [3]. An alternative might be a notion of 'propositional subtyping', allowing us to establish coercions between types which are guaranteed erasable at runtime because all they do is fix up indexing and the associated content-free proof objects.

The completion of deletion. Now that we can remove a key, we need only find the key to remove. I have chosen to delete the topmost occurrence of the given key, and to return the tree unscathed if the key does not occur at all.

As with insertion, the discipline of indexing by bounds and height is quite sufficient to ensure in silence that rebalancing works as required. Indeed, no further explicit proof effort is needed: once delp reestablishes the local ordering invariant around the deleted element, the rest of the ordering evidence stays intact from input to output.

```
del^23 : ∀{h} → [L• ⇢ L^23 h ⇢ Del^23 h]
del^23 {0} _    no0                  = ▷ no0
del^23 {s h} y° ⟨ lp , p , pu ⟩      with eq? y p
del^23 {s h} .p° (no2 lp p pu)       | ◁ ⟨⟩
  = rd (delp (lp , p , pu))
del^23 {s h} .p° (no3 lp p pq q qu)  | ◁ ⟨⟩
  = r3t (delp (lp , p , pq) , q , qu)
del^23 {s h} y° ⟨ lp , p , pu ⟩      | ▷ _ with owoto y p
del^23 {s h} y° (no2 lp p pu)        | ▷ _ | le
  = rd (d2t (del^23 y° lp , p , pu))
del^23 {s h} y° (no2 lp p pu)        | ▷ _ | ge
  = rd (t2d (lp , p , del^23 y° pu))
del^23 {s h} y° (no3 lp p pq q qu)   | ▷ _ | le
  = r3t (d2t (del^23 y° lp , p , pq) , q , qu)
del^23 {s h} y° (no3 lp p pq q qu)   | ▷ _ | ge with eq? y q
del^23 {s h} .q° (no3 lp p pq q qu)  | ▷ _ | ge | ◁ ⟨⟩
  = t3r (lp , p , delp (pq , q , qu))
... | ▷ _ with owoto y q
... | le = r3t (t2d (lp , p , del^23 y° pq) , q , qu)
... | ge = t3r (lp , p , t2d (pq , q , del^23 y° qu))
```

At no point did we need to construct trees with the invariant broken. Rather, we chose types which expressed with precision the range of possible imbalances arising locally from a deletion. It is exactly this precision which allowed us to build and justify the rebalancing reconstruction operators we reused so effectively to avoid an explosion of cases.

18. Discussion

We have seen *intrinsic* dependently typed programming at work. Internalizing ordering and balancing invariants to our datatypes, we discovered not an explosion of proof obligations, but rather that unremarkable programs check at richer types because they *accountably* do the testing which justifies their choices.

Of course, to make the programs fit neatly into the types, we must take care of how we craft the latter. I will not pretend for one moment that the good definition is the first to occur to me, and it is certainly the case that one is not automatically talented at designing dependent types, even when one is an experienced programmer in Haskell or ML. There is a new skill to learn. Hopefully, by taking the time to explore the design space for ordering invariants, I have exposed some transferable lessons. In particular, we must overcome our type inference training and learn to see types as pushing requirements inwards, as well as pulling guarantees out.

It is positive progress that work is shifting from the program definitions to the type definitions, cashing out in our tools as considerable mechanical assistance in program construction. A precise type structures its space of possible programs so tightly that an interactive editor can often offer us a small choice of plausible alternatives, usually including the thing we want. It is exhilarating being drawn to one's code by the strong currents of a good design. But that happens only in the last iteration: we are just as efficiently dashed against the rocks by a bad design, and the best tool to support recovery remains, literally, the drawing board. We should give more thought to machine-assisted exploration.

A real pleasure to me in doing this work was the realisation that I not only had 'a good idea for ordered lists' and 'a good idea for ordered trees', but that they were the *same* idea, and moreover that I could implement the idea in a datatype-generic manner. The key underpinning technology is first-class datatype description. By the end of the paper, we had just one main datatype $\mu^{\leq}_{\mathsf{IO}}$, whose sole role was to 'tie the knot' in a recursive node structure determined by a computable code. The resulting raw data are strewn with artefacts of the encoding, but pattern synonyms do a remarkably good job of recovering the appearance of bespoke constructors whenever we work specifically to one encoded datatype.

Indeed, there is clearly room for even more datatype-generic technology in the developments given here. On the one hand, the business of finding the substructure in which a key belongs, whether for insertion or deletion, is crying out for a generic construction of Gérard Huet's 'zippers' [7]. Moreover, the treatment of ordered structures as variations on the theme of the binary search tree demands consideration in the framework of 'ornaments', as studied by Pierre-Évariste Dagand and others [5]. Intuitively, it seems likely that the IO universe corresponds closely to the ornaments on node-labelled binary trees which add only finitely many bits (because IO has '$+$ rather than a general Σ). Of course, one node of a $\mu^{\leq}_{\mathsf{IO}}$ type corresponds to a region of nodes in a tree: perhaps ornaments, too, should be extended to allow the unrolling of recursive structure.

Having developed a story about ordering invariants to the extent that our favourite sorting algorithms silently establish them, we still do not have total correctness intrinsically. *What about permutation?* It has always maddened me that the insertion and flattening operations manifestly construct their output by rearranging their input: the proof that sorting permutes should thus be *by inspection*. Experiments suggest that many sorting algorithms can be expressed in a domain specific language whose type system is linear for keys. We should be able to establish a general purpose permutation invariant for this language, once and for all, by a logical relations argument. We are used to making sense of programs, but it is we who make the sense, not the programs. It is time we made programs make their own sense.

Acknowledgements. I should like to thank my father for filling my childhood with the joy of precision in programming. His sad and sudden death somewhat disrupted my writing schedule, so I am grateful to Manuel Chakravarty and Lisa Tolles for their forbearance and support. Thanks are due also to Amanda Clare and colleagues for arranging emergency electricity by the sea.

References

[1] William Aitken and John Reppy. Abstract value constructors. Technical Report TR 92-1290, Cornell University, 1992.

[2] Robert Atkey, Patricia Johann, and Neil Ghani. Refining inductive types. *Logical Methods in Computer Science*, 8(2), 2012.

[3] Jean-Philippe Bernardy and Guilhem Moulin. Type-theory in color. In Greg Morrisett and Tarmo Uustalu, editors, *ICFP*, pages 61–72. ACM, 2013. ISBN 978-1-4503-2326-0.

[4] Rod Burstall. Proving properties of programs by structural induction. *Computer Journal*, 12(1):41–48, 1969.

[5] Pierre-Évariste Dagand and Conor McBride. Transporting functions across ornaments. In Peter Thiemann and Robby Bruce Findler, editors, *ICFP*, pages 103–114. ACM, 2012. ISBN 978-1-4503-1054-3.

[6] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in Agda. In Manuel M. T. Chakravarty, Zhenjiang Hu, and Olivier Danvy, editors, *ICFP*, pages 143–155. ACM, 2011. ISBN 978-1-4503-0865-6.

[7] Gérard P. Huet. The zipper. *J. Funct. Program.*, 7(5):549–554, 1997.

[8] Patrik Jansson and Johan Jeuring. PolyP - a polytypic programming language. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *POPL*, pages 470–482. ACM Press, 1997. ISBN 0-89791-853-3.

[9] Stefan Kahrs. Red-black trees with types. *J. Funct. Program.*, 11(4): 425–432, 2001.

[10] Fredrik Lindblad and Marcin Benke. A Tool for Automated Theorem Proving in Agda. In Jean-Christophe Filliâtre, Christine Paulin-Mohring, and Benjamin Werner, editors, *TYPES*, volume 3839 of *Lec-*

ture Notes in Computer Science, pages 154–169. Springer, 2004. ISBN 3-540-31428-8.

[11] Sam Lindley and Conor McBride. Hasochism: the pleasure and pain of dependently typed haskell programming. In Chung-chieh Shan, editor, *Haskell*, pages 81–92. ACM, 2013. ISBN 978-1-4503-2383-3.

[12] Conor McBride. A Case For Dependent Families. LFCS Theory Seminar, Edinburgh, 2000. URL http://strictlypositive.org/a-case/.

[13] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[14] Conor McBride and Ross Paterson. Applicative programming with effects. *J. Funct. Program.*, 18(1):1–13, 2008.

[15] Matthew Might. The missing method: Deleting from Okasaki's red-black trees. Blog post, 2010. http://matt.might.net/articles/red-black-delete/.

[16] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[17] David Turner. Elementary strong functional programming. 1987. URL http://sblp2004.ic.uff.br/papers/turner.pdf.

[18] Philip Wadler. The concatenate vanishes. Technical report, 1987.

[19] Hongwei Xi. Dependently Typed Data Structures. In *Proceedings of Workshop of Algorithmic Aspects of Advanced Programming Languages (WAAAPL '99)*, pages 17–32, Paris, September 1999.