How to Keep Your Neighbours in Order

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

Abstract

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 [McBride(2000)]: 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.

2. Preliminaries

```
data 0 : Set where
record 1 : Set where constructor ⟨⟩

data 2 : Set where tt ff : 2

¬ : 2 → 2
¬ tt = ff
¬ ff = tt

if_then_else_ : {X : Set} → 2 → X → X → X
if tt then t else f = t
if ff then t else f = f
infix 1 if_then_else_

So : 2 → Set
So tt = 1
So ff = 0

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

data Maybe (X : Set) : Set where
  yes : X → Maybe X
  no  : Maybe X

so : ∀{X} → 2 → Maybe X → Maybe X
so tt mx = mx
so ff _  = no

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

_×_ : Set → Set → Set
S × T = Σ S λ _ → T
infixr 5 _×_

_∘_ : {A : Set} {B : A → Set} {C : (a : A) → B a → Set}
  (f : {a : A} (b : B a) → C a b) (g : (a : A) → B a) →
  (a : A) → C a (g a)
(f ∘ g) x = f (g x)
infixr 3 _∘_

id : {A : Set} → A → A
id a = a

_→̇_ : {I : Set} → (I → Set) → (I → Set) → I → Set
(F →̇ G) i = F i → G i
infixr 2 _→̇_

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

3. Searching for Search Trees (and Barking up the Wrong One)

David Turner [Turner(1987)] 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 [Burstall(1969)]. 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 in the order, the right subtree elements which follow it. Surely this invariant is crying out to be captured in a dependent type! Let us search for a type for search trees.

We could, of course, choose to 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 introduce the invariant as a predicate IsBST : Tree → Set. We could then implement insertion in our usual way, and then prove separately that our program maintains the invariant. However, the joy of dependently typed programming is that working with refined types for the data themselves can often alleviate and sometimes 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 → P → 2, we could write a recursive function to check whether a Tree is a valid search tree and compute its range:

```
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)  = so (le p c) (yes (p – d))
... | yes (a – b)  | yes ∅        = so (le b p) (yes (a – p))
... | yes (a – b)  | yes (c – d)
    = so (le b p) (so (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* [Atkey et al.(2012)Atkey, Johann, and Ghani] of Tree which valid induces. We seek a type BST : STRange → Set such that BST r ≅ {t : Tree | valid t = 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

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

The So function maps tt to 1 and ff to 0, requiring that the tests on left and right ranges succeed. When a test passes, Agda can infer the value ⟨⟩, hence we may safely leave this evidence implicit. If a test fails, Agda will complain that it cannot synthesize the implicit argument, for a very good reason!

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 output range will be wider than the input range.

```
oRange : STRange → P → STRange
oRange ∅       y = y – y
oRange (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 (oRange 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 (rOut (oRange l y) p r), but the type required is BST (oRange (rOut l p r) y), so we need some theorem-proving to fix the types, let alone to discharge the obligation So (lOK (oRange l y) p). Of course, we could plough on, despite the slough of proof, and force this definition through, but I have had enough and so have you!

We have written a datatype definition 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?

In the previous 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. As functional programmers in Milner's tradition, we are used to synthesizing the type of a thing. Moreover, the very syntax we use for **data** declarations treats the index delivered from each constructor as some sort of output. It seems natural to take datatype indices as some sort of measure of the data, which is all very well for the length of a vector, but when the measurement is computationally intricate, as in the case of computing a search tree's extrema, programming becomes vexed by the need to prove theorems about the measuring functions. The presence of 'green slime'—defined functions in the return types of constructors—is a danger sign in type design.

We can, however, 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 the question "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⊤⊥
```

Correspondingly, we can extend the order, putting ⊤ at the top and ⊥ at the bottom.

```
_⊤⊥ : ∀{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
  node : (p : P) → So (le⊤⊥ l (#p)) → So (le⊤⊥ (#p) u) →
         BST l (#p) → BST (#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.

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 → So (le⊤⊥ l (#y)) → So (le⊤⊥ (#y) u) →
         BST l u → BST l u
insert y ly yu leaf = node y ly yu leaf leaf
insert y ly yu (node p lp pu lt rt) =
  if le y p then node p lp pu (insert y ly ?0 lt) rt
            else node p lp pu lt (insert y ?1 yu rt)
```

We have no need to repair type errors by theorem proving, and most of our proof obligations follow directly from our assumptions. Working interactively, we can use Agda's proof search helper, Agsy, to fill them in for us. Our only outstanding goals are

```
?0 : So (le y p)   -- in the then branch
?1 : So (le p y)   -- in the else branch
```

The first of these is the very thing our conditional expression has found to be true! We could choose to work with an evidence-producing version of if.

```
if_then_else_ : ∀{X : Set} b →
  (So b → X) → (So (¬ b) → X) → X
if tt then t else f = t ⟨⟩
if ff then t else f = f ⟨⟩
```

We can now *learn* by testing: the then branch has a type which is reassuringly distinct from that of the else branch, and both are more informative than the target type, X. We have made a little progress:

```
insert y ly yu (node p lp pu lt rt) = if le y p
  then (λ yp → node p lp pu (insert y ly yp lt) rt)
  else (λ py → node p lp pu lt (insert y py yu rt))
```

However, we are now defeated by the fact that py : So (¬ (le y p)), which is not the evidence we need for ?1. For any given total ordering, we should be able to fix this up by proving a theorem, but this is still more work that 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

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

giving us types which directly make statements about elements of P, rather than about bits. Let us suppose we have some 'less or equal' ordering relation L : Rel P. For natural numbers, we can define

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

The information we shall need just corresponds to the totality of L: for any given x and y, L must hold *one way or the other*. For ℕ, we may define

```
owoto : ∀x y → Lℕ (x, y) + Lℕ (y, x)
owoto 0       0       = ◁ ⟨⟩
owoto 0       (suc y) = ◁ ⟨⟩
owoto (suc x) 0       = ▷ ⟨⟩
owoto (suc x) (suc y) = owoto x y
```

using only mechanical case-splitting and proof search.

Any such ordering relation on elements lifts readily to bounds.

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

Moreover, we obtain a notion of *interval*—a set of elements within given bounds.

```
_• : ∀{P} → Rel P → Rel P⊤⊥
L• (l, u) = Σ _ λ p → L⊤⊥ (l, #p) × L⊤⊥ (#p, u)
```

Let us then parametrize over some

```
owoto : ∀x y → L x y + L y x
```

and reorganise our development.

```
data BST (lu : P⊤⊥ × P⊤⊥) : Set where
  leaf : BST lu
  node : (p : P) → L⊤⊥ (π₁ lu, #p) → L⊤⊥ (#p, π₂ lu) →
         BST (π₁ lu, #p) → BST (#p, π₂ lu) → BST lu
```

```
insert : [L• ∸→ BST ∸→ BST]
insert (y, ly, yu) leaf = node y ly yu leaf leaf
insert (y, ly, yu) (node p lp pu lt rt) with owoto y p
... | ◁ yp = node p lp pu (insert (y, ly, yp) lt) rt
... | ▷ py = node p lp pu lt (insert (y, py, yu) rt)
```

The evidence generated by testing *owoto y p* is just what is needed to enable insertion in the appropriate subtree. We have found a method which seems to work! However, I fear we should not get too excited.

6. 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 problem:

```
rotater : [BST ∸→ BST]
rotater (node p lp pu (node m lm mp lt mt) rt)
  = node m lm ?2 lt (node p mp pu mt rt)
rotater t = t
```

We have discarded the non-local ordering evidence $lp : L^\top_\bot\ l\ (\#p)$, but now we need the non-local $?_2 : L^\top_\bot\ (\#\,m)\ u$ and we do not have it. Of course, we can prove this goal from mp and pu if we know that L is transitive, but if we want to make less work for ourselves, 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
  leaf : L⊤⊥ lu → BST lu
  node : (p : P) →
         BST (π₁ lu, #p) → BST (#p, π₂ lu) → BST lu
```

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, ly, yu) (leaf _) = node y (leaf ly) (leaf yu)
insert (y, ly, yu) (node p lt rt) with owoto y p
... | ◁ yp = node p (insert (y, ly, yp) lt) rt
... | ▷ py = node p lt (insert (y, py, yu) rt)
```

Rotation becomes very easy, with not a proof in sight!

```
rotater : [BST ∸→ BST]
rotater (node p (node m lt mt) rt) =
  node m lt (node p mt rt)
rotater t = t
```

We have arrived at a neat way to keep a search tree in order, storing pivot elements at nodes and ordering evidence 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⊤⊥ lu → OList lu
  cons : (p : P) →
         L⊤⊥ (π₁ lu, #p) → OList (#p, π₂ lu) → OList lu
```

By figuring out how to build ordered binary search trees, we have actually discovered how to build all sorts of in-order data structures. We simply need to show how the data are build 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*.

7. Jansson and Jeuring's PolyP Universe

```
data JJ : Set where
  ‘R ‘P ‘1 : JJ
  _‘+_ _‘×_ : JJ → JJ → JJ
infixr 4 _‘+_
infixr 5 _‘×_
⟦_⟧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
data μJJ (F : JJ) (P : Set) : Set where
  ⟨_⟩ : ⟦ F ⟧JJ (μJJ F P) P → μJJ F P
```

The ‘R stands for ‘recursive substructure’ and the ‘P stands for ‘parameter’—the type of elements stored in the container. When we ‘tie the knot’ in $\mu_{JJ}\ F\ P$, we replace interpret F's ‘Ps by some actual P and its ‘Rs by $\mu_{JJ}\ F\ P$.

Being finitary and first-order, all of the containers in the JJ universe are *traversable* in the sense defined by Ross Paterson and myself [McBride and Paterson(2008)].

```
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

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.

```
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

```
record Monoid (X : Set) : Set where
  field
    neutral : X
    combine : X → X → X
open Monoid

monApp : ∀{X} → Monoid X → Applicative (λ _ → X)
monApp m = record {pure = λ _ → neutral m; ap = combine m}

crush : ∀{P X F} → Monoid X → (P → X) → μJJ F P → X
crush m = traverse {B = 0} (monApp m)
```

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.

8. The Simple Orderable Universe

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 [Burstall(1969)], 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
```

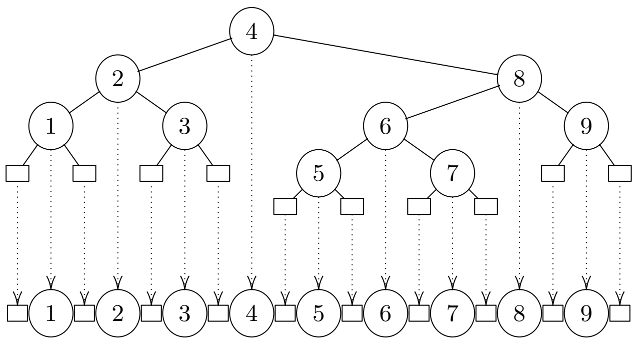
Let us give SO codes for structures we often order and bound:

```
‘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 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 lu      = R lu
⟦ ‘1 ⟧≤SO       R L lu      = L⊤⊥ lu
⟦ S ‘+ T ⟧≤SO R L lu      = ⟦ S ⟧≤SO R L lu + ⟦ T ⟧≤SO R L lu
⟦ S ‘∧ T ⟧≤SO R L (l, u) = Σ _ λ p →
                  ⟦ S ⟧≤SO R L (l, #p) × ⟦ T ⟧≤SO R L (#p, u)

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. I promise that I shall never name the evidence: I shall always match it with the _ pattern and construct it by means of the following device, making use of *instance arguments*:

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

```
! : ∀{X : Set} {{x : X}} → X
! {X} {{x}} = x
```

When we use ! at type X, Agda treats the x as an implicit argument, but rather than solving for x by *unification*, Agda seeks an *assumption* of type X in the context, succeeding if there is exactly one.

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* [Aitken and Reppy(1992)].

```
pattern _‚_‚_ s p t = p, s, t
infixr 5 _‚_‚_
```

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 $\mu^{\leq}_{\mathsf{SO}}$ ‘Tree $L\ l\ u$, and redefining intervals accordingly.

```
_^Δ _^• : ∀{P} → Rel P → Rel P_⊥^⊤
L^Δ = μ^≤_SO ‘Tree     L
L^• = μ^≤_SO ‘Interval L

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         _         = ⟨◁ !⟩
  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)⟩
```

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 ‚ _⟩ ⟨◁ _⟩ = ⟨▷ (⟨◁ !⟩ ‚ y ‚ ⟨◁ !⟩)⟩
insert ⟨_ ‚ y ‚ _⟩ ⟨▷ (lt ‚ p ‚ rt)⟩ with owoto y p
... | ◁ _ = ⟨▷ (insert ⟨! ‚ y ‚ !⟩ lt ‚ p ‚ rt)⟩
... | ▷ _ = ⟨▷ (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} → μ_JJ F P → L^Δ (⊥, ⊤)
makeTree = foldr (λ p → insert ⟨! ‚ p ‚ !⟩) ⟨◁ !⟩
```

9. Digression: Merging Monoidally

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

```
_^+ : ∀{P} → Rel P → Rel P_⊥^⊤
L^+ = μ^≤_SO ‘List L
```

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.

```
merge : [L^+ →̇ L^+ →̇ L^+]
merge ⟨◁ _⟩ ys    = ys
merge xs    ⟨◁ _⟩ = xs
merge ⟨▷ (_ ‚ x ‚ xs)⟩ ⟨▷ (_ ‚ y ‚ ys)⟩ with owoto x y
... | ◁ _ = ⟨▷ (! ‚ x ‚ merge xs ⟨▷ (! ‚ y ‚ ys)⟩)⟩
... | ▷ _ = ⟨▷ (! ‚ 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, u) → L^+ (l, u)
  go ⟨◁ _⟩             = ⟨▷ (! ‚ x ‚ xs)⟩
  go ⟨▷ (_ ‚ y ‚ ys)⟩ with owoto x y
  ... | ◁ _ = ⟨▷ (! ‚ x ‚ merge xs ⟨▷ (! ‚ y ‚ ys)⟩)⟩
  ... | ▷ _ = ⟨▷ (! ‚ 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.

```
merge_JJ : ∀{F} → μ_JJ F P → L^+ (⊥, ⊤)
merge_JJ = 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
```

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 → μ_JJ ‘qLTree P → μ_JJ ‘qLTree P
twistIn p ⟨◁ (◁ ⟨⟩)⟩ = ⟨◁ (▷ p)⟩
twistIn p ⟨◁ (▷ q)⟩  = ⟨▷ (⟨◁ (▷ p)⟩, ⟨◁ (▷ q)⟩)⟩
twistIn p ⟨▷ (l, r)⟩ = ⟨▷ (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} → μ_JJ F P → L^+ (⊥, ⊤)
mergeSort = merge_JJ ∘ foldr twistIn ⟨◁ (◁ ⟨⟩)⟩
```

10. 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 $L\ l\ u$.

draw a diagram showing the — —o—o— situation

"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 u} p →
  L^+ (l, #p) → L^+ (#p, u) → L^+ (l, u)
sandwich p ⟨◁ _⟩          ys  =  ⟨▷ (!, p, ys)⟩
sandwich p ⟨▷ (_, x, xs)⟩ ys  =  ⟨▷ (!, x, sandwich p xs ys)⟩
```

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

```
flatten : ∀{P} {L : Rel P} → [L^△ ⇾ L^+]
flatten ⟨◁ _⟩          = ⟨◁ !⟩
flatten ⟨▷ (l, p, r)⟩  = sandwich p (flatten l) (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 then benefit is outweighed by the quadratic penalty of left-nested concatenation. The standard remedy applies: we can introduce an accumulator [Wadler(1987)], but our experience with ++ should alert us to the possibility that it may require some thought.

11. 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 : ∀{P} {L : Rel P} {F} {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 : ∀{P} {L : Rel P} {F} {l u} G p →
  ⟦ G ⟧^≤_SO (μ^≤_SO F L) L (l, #p) →
  L^+ (#p, u) → L^+ (l, u)
flapp {F = F} 'R p ⟨t⟩        ys  =  flapp F p t ys
flapp '1         p _          ys  =  ⟨▷ (!, p, ys)⟩
flapp (S '+ T)   p (◁ s)      ys  =  flapp S p s ys
flapp (S '+ T)   p (▷ t)      ys  =  flapp T p t ys
flapp (S '∧ T)   p (s, p', t) ys  =  flapp S p' s (flapp T p t 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 : ∀{P} {L : Rel P} {F} → [μ^≤_SO F L ⇾ L^+]
flatten {P} {L} {F} {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 p s (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?

12. 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 ‘nil’ case, ys is silently instantiated with exactly the evidence exposed in the ‘nil’ 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 ⟨◁ !⟩
```

13. 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

  ⟦_⟧≤IO  :   ∀{I P}  →  IO I  →
               (I  →  Rel P⊤⊥)  →  Rel P  →  Rel P⊤⊥
  ⟦ ‘R i ⟧≤IO        R L lu      =  R i lu
  ⟦ ‘0 ⟧≤IO          R L lu      =  0
  ⟦ ‘1 ⟧≤IO          R L lu      =  L⊤⊥ lu
  ⟦ S ‘+ T ⟧≤IO  R L lu      =  ⟦ S ⟧≤IO R L lu + ⟦ T ⟧≤IO R L lu
  ⟦ S ‘∧ T ⟧≤IO  R L (l, u)  =  Σ _ λ p  →
                    ⟦ S ⟧≤IO R L (l, #p) × ⟦ T ⟧≤IO R L (#p, u)
  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:

```
  _⁺ _△ _• : ∀{P}  →  Rel P  →  Rel P⊤⊥
  L⁺  =  μ≤IO ‘List      L ⟨⟩
  L△  =  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 (suc 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 (suc 0)          =  ‘0
  ‘Even (suc (suc 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△]
```

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.

14. 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*.

```
  ‘Tree23 : ℕ  →  IO ℕ
  ‘Tree23 0          =  ‘1
  ‘Tree23 (suc n)  =  ‘R n ‘∧ (‘R n ‘+ (‘R n ‘∧ ‘R n))
  _²³ : ∀{P} (L : Rel P)  →  ℕ  →  Rel P⊤⊥
  L²³  =  μ≤IO ‘Tree23 L
```

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

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

```
  ins23 (suc n) ⟨_, y, _⟩ ⟨lt, p, rest⟩ with owoto y p
  ins23 (suc n) ⟨_, y, _⟩ ⟨lt, p, rest⟩
     | ◁ _  =  ?0
  ins23 (suc n) ⟨_, y, _⟩ ⟨lt, p, ◁ rt⟩
     | ▷ x  =  ?1
```

```
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x with owoto y q
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x |   ◁ _ = ?₂
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x |   ▷ _ = ?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 [McBride and McKinna(2004)].

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 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ rest ⟩
   | ◁ _ with ins23 n ⟨ !‚ y‚ ! ⟩ lt
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ rest ⟩
   | ◁ _ | ◁ 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 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ◁ rt ⟩
   | ◁ _ | ▷ (llt‚ r‚ lrt)
   = ◁ ⟨ llt‚ r‚ ▷ (lrt‚ p‚ rt) ⟩
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ◁ _ | ▷ (llt‚ r‚ lrt)
   = ▷ (⟨ llt‚ r‚ ◁ lrt ⟩‚ p‚ ⟨ 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 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x | ◁ _ with ins23 n ⟨ !‚ y‚ ! ⟩ mt
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x | ◁ _ | ◁ mt′  = ◁ ⟨ lt‚ p‚ ▷ (mt′‚ q‚ rt) ⟩
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x | ◁ _ | ▷ (mlt‚ r‚ mrt)
   = ▷ (⟨ lt‚ p‚ ◁ mlt ⟩‚ r‚ ⟨ mrt‚ q‚ ◁ rt ⟩)

ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x | ▷ _ with ins23 n ⟨ !‚ y‚ ! ⟩ rt
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x | ▷ _ | ◁ rt′  = ◁ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt′) ⟩
ins23 (suc n) ⟨ _‚ y‚ _ ⟩ ⟨ lt‚ p‚ ▷ (mt‚ q‚ rt) ⟩
   | ▷ x | ▷ _ | ▷ (rlt‚ r‚ rrt)
   = ▷ (⟨ lt‚ p‚ ◁ mt ⟩‚ q‚ ⟨ rlt‚ r‚ ◁ rrt ⟩)
```

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 n → L²³ n (⊥, ⊤)

insert : P → Tree23 → Tree23
insert p (n, t) with ins23 n ⟨ _‚ p‚ _ ⟩ t
... | ◁ t′           = n     , t′
... | ▷ (lt‚ r‚ rt)  = suc n, ⟨ lt‚ r‚ ◁ rt ⟩

sort : ∀{F} → μJJ F P → L⁺ (⊥, ⊤)
sort = flatten ∘ π₂ ∘ foldr insert (0, ⟨ _ ⟩)
```

15. Discussion

References

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

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

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

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

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

[McBride and Paterson(2008)] Conor McBride and Ross Paterson. Applicative programming with effects. *JFP*, 2008.

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

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