Computing with Generic Trees in Agda

Stephen Dolan
stedolan@stedolan.net

Abstract

Dependently-typed programming languages offer powerful new means of abstraction, allowing the programmer to work generically across data structures. However, using the standard generic encoding of tree-like data structures (the *W-types*), we soon notice a caveat: the computational behaviour of W-types does not quite match their first-order counterparts. Here, we show how a tweak to the definition of W-types avoids this caveat, making the generic definition work just as well as the direct one.

ACM Reference Format:
Stephen Dolan. 2022. Computing with Generic Trees in Agda. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Part of the promise of dependent types is that the ability to abstract over types just as easily as one can abstract over values makes generic programming straightforward. One example is the definition of W-types, which in a single stroke defines a whole family of tree-like data structures:

```
data W (Sh : Set) (Pl : Sh → Set) : Set where
  sup : (s : Sh) → (Pl s → W Sh Pl) → W Sh Pl
```

This type represents tree-shaped data generically. A tree datatype is given by a set *Sh* of *shapes*, describing the possible kinds of node, and for each shape *s* a set *Pl s* of *places*, listing the subtrees of such nodes. A tree of such a type is then given as a node of a specified shape, with one subtree per place of that shape. (This is the least fixed point of a *container* [2], from where the "shapes and places" terminology arises)

This is the simplest form of W-type, generically representing a single recursive datatype with no parameters and no indices. While this is enough to illustrate the point of this paper, note that the idea has been generalised much further, covering nested types [1], indices [3, 5], and more)

The trouble with this definition, at least in standard intensional type theory, is that what would normally be a record of several values ("one subtree for each place") is instead encoded as a function ("a function from places to subtrees"), and this causes difficulties with equality.

Conference'17, July 2017, Washington, DC, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1.1 Encoding records as functions

A pair $A \times A$ can be encoded as a function $2 \longrightarrow A$, where 2 is the type containing two elements. (Indeed, both are often written A^2).

We can try this in Agda, implementing construction and projection functions for pairs-as-functions:

```
Pair : Set → Set
Pair A = (Bool → A)

make : {A : Set} → A → A → Pair A
make a b = λ { false → a; true → b }

proj₁ proj₂ : {A : Set} → Pair A → A
proj₁ p = p false
proj₂ p = p true
```

The problem arises when we consider equality. The η-equality rule for pairs states:

$$p \equiv (\mathsf{proj}_1\ p, \mathsf{proj}_2\ p)$$

But in our functional encoding in Agda, we'd need to show

$$p \equiv \lambda\,\{\ \mathsf{false} \longrightarrow p\ \mathsf{false};\ \mathsf{true} \longrightarrow p\ \mathsf{true}\ \}$$

This does not hold definitionally. Agda compares the two functions by comparing them after applying an abstract argument x : Bool, but there is no η rule for Bool which would allow it to continue by case analysis on x.

If function extensionality is available propositionally (e.g. because it is postulated, or because we're working in a system like HoTT where it is provable), then we can prove η-equality for functional pairs. However, this is less useful than the definitional equality of native pairs, since it is not used in computation and must be explicitly appealed to.

One could imagine adding special-case rules to Agda for definitional equality at type $2 \longrightarrow A$, by comparing two functions at arguments `true` and `false`. This approach does not generalise, however, because of the following example due to McBride [7]:

Consider the functional encoding of the empty tuple, or the unit type. An tuple of no A is encoded as a function $0 \longrightarrow A$ (where 0 is the empty type), and by the η rule for empty records we expect any two such functions to be equal. In particular, this means that in an arbitrary context Γ:

$$\Gamma \vdash (\lambda x.\mathsf{true}) \equiv (\lambda x.\mathsf{false}) : 0 \longrightarrow 2$$

If there is some e such that $\Gamma \vdash e : 0$ (that is, if Γ is an inconsistent context), then we have:

$$\Gamma \vdash (\lambda x.\mathsf{true})e \equiv (\lambda x.\mathsf{false})e : 2$$
$$\Gamma \vdash \mathsf{true} \equiv \mathsf{false} : 2$$

So, deciding whether true ≡ false means first deciding whether Γ is inconsistent. Since the latter is undecidable, we have broken decidability of definitional equality.

1.2 Induction on Nat

These difficulties with functional encoding of records crop up when we try to write the induction principles for W-types. For Nat, for instance, we expect to end up with:

```
nat-ind :
  (R : Nat → Set)
  (Rzero : R zero)
  (Rsucc : ∀ n → R n → R (succ n)) →
  ∀ x → R x
```

But when we try to write the case for the zero shape, with an empty set of places, we find that our provided *Rzero* does not apply directly, because the two empty collections of subtrees are not definitionally equal. We can prove them equal propositionally, but then we lose the computation rule:

```
nat-ind R Rzero Rsucc zero ≡ Rzero
```

1.3 Contribution

The contribution of this paper is to write the induction principle for W-types generically, and have it compute, inside intensional type theory.

It is not a new result that this can be done by restricting to *finitary* W-types, where each tree has only finitely many subtrees. (These are fixed points of what Girard calls *normal functors* [6], rather than fixed points of general containers). In this case, one can encode the subtrees as a finite vector (cf. McBride [8]), with the expected computation rules.

While this approach is not novel, we present an alternative construction of it in section 3, based on a universe of finite types defined in section 2.

The new result here is to show that a small generalisation of the technique can work for infinitary W-types as well. In section 4, we introduce *partitioned sets*, which are disjoint unions of finitely many sets (which need not themselves be finite). By using partitioned sets rather than finite sets as the set of places, we can describe even infinitary W-types that still compute, as demonstrated in section 5.

The above takes place using the proof assistant Agda, and this paper is a literate Agda script.

2 Codes for Finite Types

Our goal for this section is to encode function types $A \to B$ as records, for finite A, and we begin by writing a type of codes for finite types, following the universes approach explored by Benke et al. [4].

The most straightforward choice is to use $\mathbb{N}$, which has exactly one representative for each finite cardinality. However, we do not want our finite types to be unique up to cardinality, as a type of exactly two elements is not necessarily Bool. So instead, we define finite codes as containing the empty type, singletons, and being closed under sum, and we allow singletons to be named:

```
infixr 20 _+_
data Fin : Set where
  none : Fin
  one : String → Fin
  _+_ : Fin → Fin → Fin
```

The finite types themselves are defined by interpreting Fin into Set:

```
record NamedUnit (Name : String) {l} : Set l where
  constructor tt

⟦_⟧ : Fin → Set
⟦ none ⟧ = ⊥
⟦ one name ⟧ = NamedUnit name
⟦ A + B ⟧ = ⟦ A ⟧ ⊎ ⟦ B ⟧
```

We use the NamedUnit type to ensure that distinct codes have distinct interpretations. This condition is not semantically important, but aids Agda's typechecking since it many cases it is then able to uniquely deduce the code from the interpretation, allowing us to mostly leave codes as implicit arguments to be filled in automatically.

Our named singletons mean we can write finite types with named members:

```
foobarbaz : Fin
foobarbaz = one "foo" + one "bar" + one "baz"
```

Sadly, its inhabitants have names like inj_2 (inj_1 tt) instead of "bar". To let us make use of the names, we add a convenience function for looking up elements by name:

```
lookup : (A : Fin) → String → Maybe ⟦ A ⟧
lookup none s = nothing
lookup (one t) s with primStringEquality t s
... | false = nothing
... | true = just tt
lookup (A + B) s with lookup A s
... | just x = just (inj₁ x)
... | nothing with lookup B s
... | just x = just (inj₂ x)
... | nothing = nothing
```

as well as some syntactic trickery for making use of it:

```
data Found : Set where
  ⟩ : Found

inhab : ∀ {A : Set} → Maybe A → Set
inhab nothing = ⊥
inhab (just _) = Found

⟨ : ∀ {A : Fin} → (s : String) → inhab (lookup A s) → ⟦ A ⟧
⟨ {A} s with lookup A s
```

```
... | nothing = λ ()
... | just x = λ _ → x
```

Now, we can refer to inhabitants of foobarbaz compactly:

```
bar : ⟦ foobarbaz ⟧
bar = ⟨ "bar" ⟩
```

The trick here is that the second argument to ⟨ is of type inhab (lookup A s), which is uninhabited if lookup fails, but inhabited by ⟩ if it succeeds.

2.1 Universe polymorphism and generalization

We are going to use these finite codes to describe both values and types, and to allow the same definitions to be used for both we employ Agda's *universe polymorphism*. We are not making much use of this powerful feature: the only universe levels we actually use are 0 and 1, and we could get the same effect by duplicating most definitions.

From here on, the number of quantified variables in our types increases, so to remove some clutter we allow Agda to implicitly generalise "l" as a universe level and "A" as a finite code:

```
variable l : Level
variable A : Fin
```

2.2 Function types with finite domain

Next, we define functions with finite domain, by recursion on the code of their domain:

```
record One (Name : String) (S : Set l) : Set l where
  constructor v
  field contents : S

_→°_ : Fin → Set l → Set l
none →° S = ⊤
one name →° S = One name S
(A + B) →° S = (A →° S) × (B →° S)
```

As before, using the One type ensures distinct codes have distinct interpretations, improving inference. We also write an alternative constructor for One, and a convenience function for proving equations on it.

```
_↦_ : (Name : String) → {S : Set l} → S → One Name S
_ ↦ x = v x

≡/v : ∀ {n} {A : Set l} {a a' : A} →
      (a ≡ a') → n ↦ a ≡ n ↦ a'
≡/v refl = refl
```

The purpose of ↦ is to let us use explicit names when writing functions with finite domain:

```
is-bar : foobarbaz →° Bool
is-bar =
  ("foo" ↦ false),
  ("bar" ↦ true),
  ("baz" ↦ false)
```

These names are redundant, since typechecking works by position rather than by name: in each occurrence of ↦, there is only one string that can appear on the left and Agda already knows what it is. However, being able to write these names (and have them checked) makes the code readable.

Having defined →°, we now define generic introduction and elimination forms (lambda-abstraction and application):

```
λ° : {S : Set l} → (⟦ A ⟧ → S) → (A →° S)
λ° {A = none} f = tt
λ° {A = one _} f = v (f tt)
λ° {A = A + B} f = λ° (f ∘ inj₁) , λ° (f ∘ inj₂)

_◃°_ : {S : Set l} → (A →° S) → (⟦ A ⟧ → S)
_◃°_ {A = one _} (v f) tt = f
_◃°_ {A = A + B} (f , g) (inj₁ x) = f ◃° x
_◃°_ {A = A + B} (f , g) (inj₂ x) = g ◃° x
```

These function types have slightly different computation rules than the usual, as their underlying implementation is as records rather than as functions. In particular, the β and η rules for functions no longer hold: we do not have $(\lambda x.f)e \equiv f[x/e]$ nor $f \equiv \lambda x.fx$ definitionally in general. However, these rules are provable (that is, they hold propositionally):

```
beta° : {S : Set l} (f : ⟦ A ⟧ → S) (x : ⟦ A ⟧) →
        (λ° f ◃° x) ≡ f x
beta° {A = one _} f tt = refl
beta° {A = A + B} f (inj₁ x) = beta° (f ∘ inj₁) x
beta° {A = A + B} f (inj₂ x) = beta° (f ∘ inj₂) x

eta° : {S : Set l} (f : A →° S) →
       f ≡ λ° λ x → f ◃° x
eta° {A = none} tt = refl
eta° {A = one _} (v x) = refl
eta° {A = A + B} (f , g) = ≡/, (eta° f) (eta° g)
```

Additionally, β-equality holds definitionally as long as the code of the domain and the argument are both in canonical form, while η-equality holds definitionally when the code of the domain is canonical. So these functions do compute, but not when their types or arguments are stuck. That is, while (λ° f ◃° x) does not reduce with x a variable, an application to a concrete argument like λ° f ◃° (⟨"bar"⟩) reduces to f (⟨"bar"⟩) for any f of type ⟦ foobarbaz ⟧ → Set.

A useful property that these functions also have is that extensionality is provable:

```
ext° : {S : Set l} (f g : ⟦ A ⟧ → S) →
       (eq : ∀ x → f x ≡ g x) →
       λ° f ≡ λ° g
ext° {A = none} f g eq = refl
ext° {A = one _} f g eq = ≡/v (eq tt)
ext° {A = A + B} f g eq =
```

```
  ≡/, (ext° (f ∘ inj1) (g ∘ inj1) (eq ∘ inj1))
      (ext° (f ∘ inj2) (g ∘ inj2) (eq ∘ inj2))
```

Again, this holds definitionally for canonical domain codes.

2.3 Dependent functions with finite domain

Next, we generalise from simple function types →° to dependent ones Π°, where the type of the result may depend on the argument. These are semantically tuples, consisting of finitely many values of different types.

We need to define →° and Π° separately, because the former is used in the definition of the latter: the result type of Π° is given as a finitary function into Set:

```
Π° : (A : Fin) (U : A →° Set l) → Set l
Π° none U = ⊤
Π° (one name) (v U) = One name U
Π° (A + B) (U , V) = (Π° A U) × (Π° B V)
```

As above, we have abstraction and application:

```
Λ° : {U : A →° Set l} → ((x : ⟦ A ⟧) → U ◂° x) → (Π° A U)
Λ° {A = none} f = tt
Λ° {A = one _} f = v (f tt)
Λ° {A = A + B} f = Λ° (f ∘ inj1) , Λ° (f ∘ inj2)

_◃°_ : {U : A →° Set l} → (Π° A U) → (a : ⟦ A ⟧) → U ◂° a
_◃°_ {A = one _} (v f) x = f
_◃°_ {A = A + B} (f , g) (inj1 x) = f ◃° x
_◃°_ {A = A + B} (f , g) (inj2 x) = g ◃° x
```

The β and η rules hold (with the same caveats about computation as before), as does extensionality:

```
Beta° : {U : A →° Set l} → (f : ((x : ⟦ A ⟧) → U ◂° x)) →
        (a : ⟦ A ⟧) → Λ° f ◃° a ≡ f a
Eta° : {U : A →° Set l} → (f : Π° A U) →
       f ≡ Λ° λ x → f ◃° x
Ext° : {U : A →° Set l} → (f g : (x : ⟦ A ⟧) → U ◂° x) →
       (eq : ∀ x → f x ≡ g x) → Λ° f ≡ Λ° g
```

The proofs are omitted, as they are identical to those for simple function types.

3 Finitary W-types

Having function types with finite domains available, we are now able to generically describe finite trees, by taking the definition of W-types above and replacing → with →°:

```
data W° (Sh : Set) (Pl : Sh → Fin) : Set where
  sup : (sh : Sh) → (Pl sh →° W° Sh Pl) → W° Sh Pl
```

Note that this being accepted as a strictly positive inductive type relies on the precise definition of →°.

The full eliminator for W-types is a bit of a mouthful. It states that to compute a result R for all trees of a given W-type, it suffices to compute R for every tree of the form sup *sh sub*, given R is already computed for each subtree in *sub*.

The type is almost that of the standard eliminator, except uses finitary function spaces →° and Π° instead of the usual ones. The implementation is slightly different, doing an explicit recursion on the set of places to ensure that the recursion is structural:

```
elim° : ∀ {Sh Pl} (R : W° Sh Pl → Set) →
  (F : (sh : Sh) →
      (sub : Pl sh →° W° Sh Pl) →
      (subR : Π° (Pl sh) (λ° λ p → R (sub ◂° p))) →
      R (sup sh sub)) →
  (x : W° Sh Pl) → R x
elim° {Sh} {Pl} R F (sup sh t) = F sh t (IH t)
  where
    IH : ∀ {Ps} → (t : Ps →° W° Sh Pl) →
         Π° Ps (λ° (λ p → R (t ◂° p)))
    IH {none} t = tt
    IH {one n} (v t) = n ↦ elim° R F t
    IH {Ps1 + Ps2} (t1 , t2) = IH t1 , IH t2
```

As an example, we implement the Peano natural numbers, which are written in Agda directly as:

```
data Nat : Set where
  zero : Nat
  succ : (x : Nat) → Nat
```

The set of shapes of a W°-type is an arbitrary Set, so we are not obliged to use Fin. In this case it happens to be finite, making it convenient to use Fin anyway. We use a helper function to easily eliminate Fin in ordinary functions:

```
cases : {B : ⟦ A ⟧ → Set l} → (Π° A (λ° B)) →
  (a : ⟦ A ⟧) → B a
cases f a = transp (beta° _ a) (f ◃° a)
```

Then, the definition of Nat is:

```
Nat = W° ⟦ one "zero" + one "succ" ⟧
  (cases (
    "zero" ↦ none ,
    "succ" ↦ one "x"))
```

and the constructors are:

```
zero : Nat
zero = sup (⟨"zero"⟩) tt

succ : Nat → Nat
succ x = sup (⟨"succ"⟩) ("x" ↦ x)
```

The usual induction principle for ℕ is now definable by appeal to elim°:

```
nat-ind :
  (R : Nat → Set)
  (Pzero : R zero)
  (Psucc : ∀ n → R n → R (succ n)) →
  ∀ x → R x
```

```
nat-ind R Pzero Psucc =
  elim∘ R (cases (
    "zero" ↦ (λ _ _ → Pzero) ,
    "succ" ↦ λ { (v x) (v Rx) → Psucc x Rx }))
```

The point of this exercise is that unlike with plain W-types, the induction principle so defined has the right computation behaviour, in particular having the right behaviour on zero:

```
nat-ind-zero : ∀ { R Pzero Psucc } →
  nat-ind R Pzero Psucc zero ≡ Pzero
nat-ind-zero = refl

nat-ind-succ : ∀ { R Pzero Psucc x } →
  nat-ind R Pzero Psucc (succ x)
  ≡ Psucc x (nat-ind R Pzero Psucc x)
nat-ind-succ = refl
```

The important thing is not that these are true, but that they are true by refl: our definition of nat-ind computes.

4 Partitioned Sets

Next, we generalise from finite sets to *partitioned sets*, which are disjoint unions of finitely many components, where the components themselves need not be finite:

```
record PSet : Set₁ where
  constructor pset
  field
    parts : Fin
    elems : parts →∘ Set

⟦_⟧* : PSet → Set
⟦ pset none E ⟧* = ⊥
⟦ pset (one name) (v E) ⟧* = NamedUnit name × E
⟦ pset (P + Q) (E , F) ⟧* = ⟦ pset P E ⟧* ⊎ ⟦ pset Q F ⟧*
```

An element of a partitioned set can be constructed by specifying a component and a member of that component:

```
el : ∀ {P E} → (p : ⟦ P ⟧) → (E ◃∘ p) → ⟦ pset P E ⟧*
el {P = one x} p e = tt , e
el {P = P + Q} (inj₁ p) e = inj₁ (el p e)
el {P = P + Q} (inj₂ q) e = inj₂ (el q e)
```

4.1 Functions on Partitioned Sets

Mirroring the definitions of →∘ and Π∘ earlier, we define →* and Π* as functions with partitioned rather than finite domain. The definitions are essentially curried, where a function from a partitioned set P to a set S is a finitary function from the components of P to an ordinary function from the members of that component to S:

```
_→*_ : ∀ {l} → PSet → Set l → Set l
pset none tt →* S = ⊤
pset (one name) (v E) →* S =
  One name (E → S)
pset (P + Q) (E , F) →* S =
  ((pset P E) →* S) × ((pset Q F) →* S)

Π* : (X : PSet) (M : X →* Set l) → Set l
Π* (pset none tt) M = ⊤
Π* (pset (one name) (v E)) (v M) =
  One name ((x : E) → M x)
Π* (pset (P + Q) (E , F)) (M , N) =
  Π* (pset P E) M × Π* (pset Q F) N
```

As before, these can be introduced and eliminated with abstraction and application operators:

```
variable X : PSet

λ* : {S : Set l} → (⟦ X ⟧* → S) → (X →* S)
λ* {X = pset none tt} f = tt
λ* {X = pset (one n) (v E)} f = n ↦ λ x → f (tt , x)
λ* {X = pset (P + Q) (E , F)} f = λ* (f ∘ inj₁) , λ* (f ∘ inj₂)

_◃*_ : {S : Set l} → (X →* S) → ⟦ X ⟧* → S
_◃*_ {X = pset (one _) (v E)} (v f) (tt , e) = f e
_◃*_ {X = pset (P + Q) (E , F)} (f , g) (inj₁ x) = f ◃* x
_◃*_ {X = pset (P + Q) (E , F)} (f , g) (inj₂ x) = g ◃* x

Λ* : {M : X →* Set l} → ((x : ⟦ X ⟧*) → M ◃* x) →
       Π* X M
Λ* {X = pset none tt} f = tt
Λ* {X = pset (one n) E} f = n ↦ λ x → f (tt , x)
Λ* {X = pset (P + Q) (E , F)} f = Λ* (f ∘ inj₁) , Λ* (f ∘ inj₂)

_⊲*_ : {M : X →* Set l} → (Π* X M) →
       (x : ⟦ X ⟧*) → M ◃* x
_⊲*_ {X = pset (one n) (v E)} (v f) (tt , e) = f e
_⊲*_ {X = pset (P + Q) (E , F)} (f , g) (inj₁ x) = f ⊲* x
_⊲*_ {X = pset (P + Q) (E , F)} (f , g) (inj₂ x) = g ⊲* x
```

Agda's termination checker is relatively generous here, by accepting `pset P E` as smaller than `pset (P + Q) (E , F)` – with other typecheckers, we may have had to use a separate recursion on `P` to make the recursion count as structural.

We again have β and η, with essentially identical proofs to the finite case:

```
beta* : {S : Set l} (f : ⟦ X ⟧* → S) →
        (x : ⟦ X ⟧*) → (λ* f ◃* x) ≡ f x
eta* : {S : Set l} (f : X →* S) →
        f ≡ λ* λ x → f ◃* x
Beta* : {U : X →* Set l} → (f : ((x : ⟦ X ⟧*) → U ◃* x)) →
        (a : ⟦ X ⟧*) → Λ* f ⊲* a ≡ f a
Eta* : {U : X →* Set l} → (f : Π* X U) →
        f ≡ Λ* λ x → f ⊲* x
```

However, lacking function extensionality, we no longer have the ext and Ext rules, as showing equality of functions on partitioned sets requires both equality of their finite, record-based part and their possibly-infinite, functional part.

5 Infinitary W-types with Partitioned Sets

Finally, we are ready to implement W-types with a partitioned set of places:

```
data W* (Sh : Set) (Pl : Sh → PSet) : Set where
  sup : (sh : Sh) → (Pl sh →* W* Sh Pl) → W* Sh Pl
```

The eliminator is identical to that of W°, with * replacing °:

```
elim* : ∀ {Sh Pl} (R : W* Sh Pl → Set) →
  (F : (sh : Sh) →
      (sub : (Pl sh) →* W* Sh Pl) →
      (subR : Π* (Pl sh) (λ* λ p → R (sub ◂* p))) →
      R (sup sh sub)) →
  (x : W* Sh Pl) → R x
elim* {Sh} {Pl} R F (sup sh t) = F sh t (IH t)
  where
    IH : ∀ {Ps} → (t : Ps →* W* Sh Pl) →
        Π* Ps (λ* (λ p → R (t ◂* p)))
    IH {pset none Es} t = tt
    IH {pset (one n) Es} (v t) = n ↦ λ e → elim* R F (t e)
    IH {pset (Ps₁ + Ps₂) Es} (t₁ , t₂) = IH t₁ , IH t₂
```

As an example, we code the Brouwer ordinal trees, which are defined by two finitary shapes and one infinitary one:

```
data Ord : Set where
  ozero : Ord
  osucc : Ord → Ord
  olim : (Nat → Ord) → Ord
```

Using W*, this translates to:

```
Ord = W* ⟦ one "zero" + one "succ" + one "lim" ⟧
  (cases (
    "zero" ↦ pset none tt ,
    "succ" ↦ pset (one "x") ("x" ↦ ⊤) ,
    "lim" ↦ pset (one "f") ("f" ↦ Nat)))
```

with constructors:

```
ozero : Ord
ozero = sup (⟨"zero"⟩) tt

osucc : Ord → Ord
osucc x = sup (⟨"succ"⟩) ("x" ↦ λ _ → x)

olim : (Nat → Ord) → Ord
olim f = sup (⟨"lim"⟩) ("f" ↦ f)
```

and an induction principle (defined using the general elim*):

```
ord-ind :
  (R : Ord → Set)
  (Pzero : R ozero)
  (Psucc : ∀ n → R n → R (osucc n))
  (Plim : ∀ f → (∀ n → R (f n)) → R (olim f)) →
  ∀ x → R x
ord-ind R Pzero Psucc Plim =
  elim* R (cases (
    "zero" ↦ (λ _ _ → Pzero) ,
    "succ" ↦ (λ { (v x) (v Rx) → Psucc (x tt) (Rx tt) }) ,
    "lim" ↦ (λ { (v f) (v Rf) → Plim f Rf })))
```

Like nat-ind previously, this eliminator computes, as shown by the following properties (where again, the point is not so much that they are true but that they are true by refl):

```
ord-ind-zero : ∀ { R Pzero Psucc Plim } →
  ord-ind R Pzero Psucc Plim ozero ≡ Pzero
ord-ind-zero = refl

ord-ind-succ : ∀ { R Pzero Psucc Plim x } →
  ord-ind R Pzero Psucc Plim (osucc x)
  ≡ Psucc x (ord-ind R Pzero Psucc Plim x)
ord-ind-succ = refl

ord-ind-lim : ∀ { R Pzero Psucc Plim f } →
  ord-ind R Pzero Psucc Plim (olim f)
  ≡ Plim f (λ n → ord-ind R Pzero Psucc Plim (f n))
ord-ind-lim = refl
```

6 Conclusion

The powerful abstraction capabilities of dependently-typed programming languages make it possible to write extremely general definitions of families of data structures. Yet even when it is fairly straightforward to define a type with the desired inhabitants, it can be much trickier to ensure that the type has the right computation rules.

However, computation rules are fundamentally finite things, and as we've seen here careful attention to which parts of a definition are finitary can yield definitions that compute as expected.

References

[1] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2004. Representing nested inductive types using W-types. In *International Colloquium on Automata, Languages, and Programming*. Springer, 59–71.

[2] Michael Abbott, Thorsten Altenkirch, and Neil Ghani. 2005. Containers: Constructing strictly positive types. *Theoretical Computer Science* 342, 1 (2005), 3–27.

[3] Thorsten Altenkirch, Neil Ghani, Peter Hancock, Conor McBride, and Peter Morris. 2015. Indexed containers. *Journal of Functional Programming* 25 (2015).

[4] Marcin Benke, Peter Dybjer, and Patrik Jansson. 2003. Universes for generic programs and proofs in dependent type theory. *Nord. J. Comput.* 10, 4 (2003), 265–289.

[5] James Chapman, Pierre-Evariste Dagand, Conor Mcbride, and Peter Morris. 2010. The gentle art of levitation. In *ICFP 2010*. 3–14.

[6] Jean-Yves Girard. 1988. Normal functors, power series and λ-calculus. *Annals of pure and applied logic* 37, 2 (1988), 129–177.

[7] Conor McBride. 2009. Grins from my Ripley Cupboard. http://strictlypositive.org/Ripley.pdf.

[8] Conor McBride. 2015. Datatypes of datatypes. *Summer School on Generic and Effectful Programming, St Anne's College, Oxford* (2015).