Everybody's Got To Be Somewhere

Conor McBride

Mathematically Structured Programming Group
Department of Computer and Information Sciences
University of Strathclyde, Glasgow

conor.mcbride@strath.ac.uk

This literate Agda paper gives a nameless *co*-de-Bruijn representation of generic (meta)syntax with binding. It owes much to the work of Sato et al. [6] on representation of variable binding by mapping variable use sites. The key to any nameless representation of syntax is how it indicates the variables we choose to use and thus, implicitly those we neglect. The business of *selecting* is what we shall revisit with care in the sequel. The definition leads to a new construction of hereditary substitution.

1 Basic Equipment

We shall need finite types Zero, One, and Two, named for their cardinality, and the reflection of Two as a set of evidence for 'being tt'. Dependent pairing is by means of the Σ type, abbreviated by × when non-dependent. The *pattern synonym* !_ allows the first component to be determined by the second: making it a right-associative prefix operator lets us write ! ! *expression* rather than ! (! (*expression*)).

```
data   Zero : Set where
record One  : Set where constructor ⟨⟩
data   Two  : Set where tt ff : Two

Tt : Two → Set
Tt tt = One
Tt ff = Zero

record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field fst : S; snd : T fst

_×_ : Set → Set → Set
S × T = Σ S λ _ → T
pattern !_ t = _ , t
```

We shall also need to reason equationally. For all its imperfections in matters of *extensionality*, it will be convenient to define equality inductively, enabling the **rewrite** construct in equational proofs.

```
data _==_ {X : Set} (x : X) : X → Set where refl : x == x
```

2 OPE$_K$: The Category of Order-Preserving Embeddings

No category theorist would mistake me for one of their own. However, the key technology in this paper can be helpfully conceptualised categorically. Category theory is the study of compositionality for anything, not just sets-and-functions: here, we have an opportunity to develop categorical structure with an example apart from the usual functional programming apparatus, as emphasized particularly in the Haskell community. This strikes me as a good opportunity to revisit the basics.

Category (I): Objects and Morphisms. A *category* is given by a class of *objects* and a family of *morphisms* (or *arrows*) indexed by two objects: *source* and *target*. Abstractly, we may write $\mathbb{C}$ for a given category, $|\mathbb{C}|$ for its objects, and $\mathbb{C}(S,T)$ for its morphisms with given source and target, $S,T \in |\mathbb{C}|$.

Submitted to:
MSFP 2018

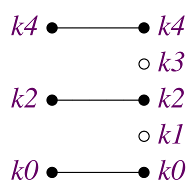
The rest of the definition will follow shortly, but let us fix these notions for our example, the category, $\mathbf{OPE}_K$, of *order-preserving embeddings* between variable scopes. The objects of $\mathbf{OPE}_K$ are *scopes*, which we may represent concretely backward (or 'snoc') lists giving the *kinds*, K, of variables. (I shall habitually suppress the kind subscript and just write **OPE** for the category.) I use backward lists, because it is traditional to write contexts to the left of judgements in typing rules and extend them on the right. However, I write 'scope' rather than 'context' as we are characterizing at least which variables we may refer to, but perhaps not all the information one would find in a context.

```
data Bwd (K : Set) : Set where          data _≤_ : Bwd K → Bwd K → Set where
  _-,_ : Bwd K → K → Bwd K                _o' : iz ≤ jz → iz        ≤ (jz -, k)
  []   : Bwd K                            _os : iz ≤ jz → (iz -, k) ≤ (jz -, k)
                                          oz  :           []        ≤  []
```

The morphisms, $iz \leq jz$, of **OPE** give an embedding from a source to a target scope. Colloquially, we may call them 'thinnings', as they dilute the variables of the source scope with more. Dually, we may see such a morphism as expelling variables from the target scope, leaving a particular selection as the source. I write the step constructors postfix, so thinnings (like scopes) grow on the right.

Now, where I give myself away as a type theorist is that I do not consider the notion of 'morphism' to make sense without prior source and target objects. The type $iz \leq jz$ (which is a little more mnemonic than $\mathbf{OPE}(iz, jz)$) is the type of 'thinnings from iz to jz': there is no type of 'thinnings' *per se*.

Let us have an example thinning: here, we embed a scope with three variables into a scope with five.



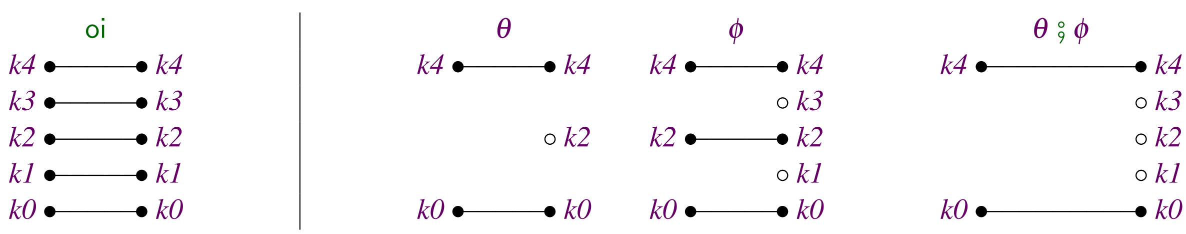

```
              os   : [] -, k0 -, k2 -, k4 ≤ [] -, k0 -, k1 -, k2 -, k3 -, k4
            o'
          os
        o'
      os
    oz
```

Category (II): Identity and Composition. In any category, certain morphisms must exist. Each object $X \in |\mathbb{C}|$ has an *identity* $\iota_X \in \mathbb{C}(X, X)$, and wherever the target of one morphism meets the source of another, their *composite* makes a direct path: if $f \in \mathbb{C}(R, S)$ and $g \in \mathbb{C}(S, T)$, then $(f; g) \in \mathbb{C}(R, T)$.

For example, every scope has the identity thinning, oi, and thinnings compose via ⨾. (For functions, it is usual to write $g \cdot f$ for 'g *after* f' than $f; g$ for 'f *then* g', but for thinnings I retain spatial intuition.)

```
oi : kz ≤ kz                                     _⨾_ : iz ≤ jz → jz ≤ kz → iz ≤ kz
oi {kz = iz -, k} = oi os   -- os preserves oi   θ    ⨾ φ o' = (θ ⨾ φ) o'
oi {kz = []}      = oz                           θ o' ⨾ φ os = (θ ⨾ φ) o'
                                                 θ os ⨾ φ os = (θ ⨾ φ) os   -- os preserves ⨾
                                                 oz   ⨾ oz   = oz
```

By way of example, let us plot specific uses of identity and composition.



Category (III): Laws. to complete the definition of a category, we must say which laws are satisfied by identity and composition. Composition *absorbs* identity on the left and on the right. Moreover, composition is *associative*, meaning that any sequence of morphisms which fit together target-to-source can be composed without the specific pairwise grouping choices making a difference. That is, we have three laws which are presented as *equations*, at which point any type theorist will want to know what is meant by 'equal': I shall always be careful to say. Our thinnings are first-order, so $==$ will serve. With this definition in place, we may then state the laws. I omit the proofs, which go by functional induction.

$$\mathsf{law{-}oi}⨾ : \mathsf{oi} ⨾ \theta == \theta \qquad \mathsf{law{-}}⨾\mathsf{oi} : \theta ⨾ \mathsf{oi} == \theta \qquad \mathsf{law{-}}⨾⨾ : \theta ⨾ (\phi ⨾ \psi) == (\theta ⨾ \phi) ⨾ \psi$$

As one might expect, order-preserving embeddings have a strong antisymmetry property that one cannot expect of categories in general. The *only* invertible arrows are the identities. Note that we must match on the proof of $iz == jz$ even to claim that θ and ϕ are the identity.

$$\mathsf{antisym} : (\theta : iz \le jz)\ (\phi : jz \le iz) \to \Sigma\ (iz == jz)\ \lambda\ \{\mathsf{refl} \to \theta == \mathsf{oi} \times \phi == \mathsf{oi}\}$$

3 De Bruijn Syntax via OPE

We often see numbers as de Bruijn indices for variables [4], perhaps with some bound enforced by typing, as shown in principle by Bellegarde and Hook [2], and in practice by Bird and Paterson [3], and (for simple types) by Altenkirch and Reus [1]. To grow the set of free variables under a binder, use option types or some 'finite set' construction, often called 'Fin'. We can use singleton embedding,

$$k \leftarrow kz \ = \ [] \text{ -, } k \le kz$$

and then give the well scoped but untyped de Bruijn λ-terms, readily seen to admit thinning:

$$\begin{array}{lll} \textbf{data}\ \mathsf{Lam}\ (iz : \mathsf{Bwd\ One}) : \mathsf{Set}\ \textbf{where} & & \\ \quad \# \quad : (x : \langle\rangle \leftarrow iz) \to & \mathsf{Lam}\ iz & \\ \quad \_\$\_ : (f\ s : \mathsf{Lam}\ iz) \to & \mathsf{Lam}\ iz & \\ \quad \lambda \quad : (t : \mathsf{Lam}\ (iz \text{ -, } \langle\rangle)) \to & \mathsf{Lam}\ iz & \end{array} \qquad \begin{array}{l} \_{\uparrow}\_ : \mathsf{Lam}\ iz \to iz \le jz \to \mathsf{Lam}\ jz \\ \#\, i \quad \uparrow \theta = \#\,(i ⨾ \theta) \\ (f\ \$\ s) \uparrow \theta = (f \uparrow \theta)\ \$\ (s \uparrow \theta) \\ \lambda\, t \quad \uparrow \theta = \lambda\,(t \uparrow \theta\ \mathsf{os}) \end{array}$$

Variables are represented by pointing, eliminating redundant choice of names, but it is only when we point to one variable that we exclude the others. Thus de Bruijn indexing effectively uses thinnings to discard unwanted variables as *late* as possible, in the *leaves* of syntax trees.

Note how the scope index iz is the target of a thinning in # and weakened in λ. Hence, thinnings act on terms ultimately by postcomposition, but because terms keep their thinnings at their leaves, we must hunt the entire tree to find them. Now consider the other canonical placement of thinnings, nearest the *root*, discarding unused variables as *early* as possible.

4 Slices of Thinnings

If we fix the target of thinnings, $(\_ \le kz)$, we obtain the notion of *subscopes* of a given kz. Fixing a target is a standard way to construct a new category whose objects are given by morphisms of the original.

Slice Category. If $\mathbb{C}$ is a category and I one of its objects, the *slice category* $\mathbb{C}/I$ has as its objects pairs (S, f), where S is an object of $\mathbb{C}$ and $f : S \to I$ is a morphism in $\mathbb{C}$. A morphism in $(S, f) \to (T, g)$ is some $h : S \to T$ such that $f = h;g$. (The dotted regions in the diagram show the objects in the slice.)

$$\begin{tikzcd} S \arrow[rr, "h"] \arrow[dr, "f"'] & & T \arrow[dl, "g"] \\ & I & \end{tikzcd}$$

That is, the morphisms are *triangles*. A seasoned dependently typed programmer will be nervous at a definition like the following (where the _ after Σ asks Agda to compute the type $iz \le jz$ of θ):

$$\psi \to_/ \phi \;=\; \Sigma\ \_\ \lambda\ \theta \to (\theta \fatsemi \phi) \equiv \psi \qquad \text{-- beware of } \fatsemi!$$

because the equation restricts us when it comes to manipulating triangles. Dependent pattern matching relies on *unification* of indices, but defined function symbols like $\fatsemi$ make unification difficult, obliging us to reason about the *edges* of the triangles. It helps at this point to define the *graph* of $\fatsemi$ inductively.

```
data Tri : iz ≤ jz → jz ≤ kz → iz ≤ kz → Set where
  _t-”  : Tri θ φ ψ → Tri  θ      (φ o’) (ψ o’)
  _t’s’ : Tri θ φ ψ → Tri (θ o’) (φ os) (ψ o’)
  _tsss : Tri θ φ ψ → Tri (θ os) (φ os) (ψ os)
  tzzz  :             Tri  oz     oz     oz
```

```
tri  : (θ : iz ≤ jz) (φ : jz ≤ kz) →
       Tri θ φ (θ ⨟ φ)
comp : Tri θ φ ψ → ψ ≡ (θ ⨟ φ)
```

The indexing is entirely in constructor form, which will allow easy unification. Moreover, all the *data* in a Tri structure comes from its *indices*. Easy inductions show that Tri is precisely the graph of $\fatsemi$.

The example composition given above can be rendered a triangle, as follows:

```
egTri : Tri {kz = [] -, k0 -, k1 -, k2 -, k3 -, k4} (oz os o’ os) (oz os o’ os o’ os) (oz os o’ o’ o’ os)
egTri = tzzz tsss t-” t’s’ t-” tsss
```

We obtain a definition of morphisms in the slice as triangles.

$$\psi \to_/ \phi \;=\; \Sigma\ \_\ \lambda\ \theta \to \mathsf{Tri}\ \theta\ \phi\ \psi$$

A useful property specific to thinnings is that morphisms in the slice category are *unique*. It is straightforward to formulate this property in terms of triangles with edges in common, and then to work by induction on the triangles rather than their edges. As a result, it will be cheap to establish *universal properties* in the slices of **OPE**, asserting the existence of unique morphisms: uniqueness comes for free!

$$\mathsf{triU} : \mathsf{Tri}\ \theta\ \phi\ \psi \to \mathsf{Tri}\ \theta'\ \phi\ \psi \to \theta \equiv \theta'$$

5 Functors, a densely prevalent notion

Haskell makes considerable use of the type class `Functor` and its many subclasses, but this is only to scratch the surface: Haskell's functors are *endo*functors, mapping the 'category' of types-and-functions into itself. Once we adopt the appropriate level of generality, functoriality sprouts everywhere, and the same structures can be usefully functorial in many ways.

Functor. A *functor* is a mapping from a source category $\mathbb{C}$ to a target category $\mathbb{D}$ which preserves categorical structure. To specify a structure, we must give a function $F_o : |\mathbb{C}| \to |\mathbb{D}|$ from source objects to target objects, together with a family of functions $F_m : \mathbb{C}(S,T) \to \mathbb{D}(F_o(S), F_o(T))$. The preserved structure amounts to identity and composition: we must have that $F_m(\iota_X) = \iota_{F_o(X)}$ and that $F_m(f;g) = F_m(f);F_m(g)$. Note that there is an identity functor $\mathbf{I}$ (whose actions on objects and morphisms are the identity) from $\mathbb{C}$ to itself and that functors compose (componentwise).

E.g., every $k : K$ induces a functor (*weakening*) from **OPE** to itself by scope extension, $(\_ \mathbin{-\!,} k)$ on objects and os on morphisms. The very definitions of oi and $\fatsemi$ show that os preserves oi and $\fatsemi$.

Before we can have more examples, we shall need some more categories. Let Set be the category whose objects are Agda types in the Set universe and whose morphisms in $\mathsf{Set}(S,T)$ are functions of type $S \to T$, with the usual identity and composition. Consider morphisms equal if they agree *pointwise*. As an exercise, find the action on morphisms of show the Set to Set functor which is Bwd on objects, and check that $(\mathsf{Lam}, \uparrow)$ gives a functor from **OPE** to Set.

Our work takes us in a different direction, profiting from the richness of dependent types: let us construct new categories by *indexing*. If $I : \mathsf{Set}$, we may then take $I \to \mathsf{Set}$ to be the category whose objects are *families* of objects in Set, $S, T : I \to \mathsf{Set}$ with morphisms being the corresponding (implicitly indexed) families of functions:

$$S \dot{\to} T \;=\; \forall \{i\} \to S\ i \to T\ i$$

Consider morphisms equal if they map each index to pointwise equal functions. We may define a functor from $K \to \mathsf{Set}$ to $\mathsf{Bwd}\ K \to \mathsf{Set}$ as follows:

$$\begin{array}{lll}
\mathbf{data}\ \mathsf{All}\ (P : K \to \mathsf{Set}) : \mathsf{Bwd}\ K \to \mathsf{Set}\ \mathbf{where} & & \\
\quad [] \quad\ : & & \mathsf{All}\ P\ [] \\
\quad _\mathbin{-\!,}_ : \mathsf{All}\ P\ kz \to P\ k \to & & \mathsf{All}\ P\ (kz \mathbin{-\!,} k)
\end{array}$$

$$\begin{array}{lcl}
\mathsf{all} : (P \dot{\to} Q) \to (\mathsf{All}\ P \dot{\to} \mathsf{All}\ Q) & & \\
\mathsf{all}\ f\ [] & = & [] \\
\mathsf{all}\ f\ (pz \mathbin{-\!,} p) & = & \mathsf{all}\ f\ pz \mathbin{-\!,} f\ p
\end{array}$$

For a given K, All acts on objects, giving for each k in a scope, some value in $P\ k$, thus giving us a notion of *environment*. The action on morphisms, all, lifts *kind-respecting* operations on values to *scope-respecting* operations on environments. Identity and composition are readily preserved. In the sequel, it will be convenient to abbreviate $\mathsf{Bwd}\ K \to \mathsf{Set}$ as $\overline{K}$, for types indexed over scopes.

However, All gives more functorial structure. Fixing kz, we obtain $\lambda\ P \to \mathsf{All}\ P\ kz$, a functor from $K \to \mathsf{Set}$ to Set, again with the instantiated all acting on morphisms. And still, there is more.

Opposite Category. For a given category $\mathbb{C}$, its *opposite* category is denoted $\mathbb{C}^{\mathrm{op}}$ and defined thus:

$$|\mathbb{C}^{\mathrm{op}}| = |\mathbb{C}| \qquad \mathbb{C}^{\mathrm{op}}(S,T) = \mathbb{C}(T,S) \qquad \iota_X^{\mathrm{op}} = \iota_X \qquad f;^{\mathrm{op}} g = g;f$$

Note that a functor from $\mathbb{C}^{\mathrm{op}}$ to $\mathbb{D}$ is sometimes called a *contravariant functor* from $\mathbb{C}$ to $\mathbb{D}$.

For example, $\mathbf{OPE}^{\mathrm{op}}(jz, iz) = iz \le jz$ allows us to see the category of thinnings as the category of *selections*, where we choose just iz from the available jz. If we have an environment for all of the jz, we should be able to whittle it down to an environment for just the iz by throwing away the values we no longer need. That is to say, $\mathsf{All}\ P$ is a *functor* from $\mathbf{OPE}^{\mathrm{op}}$ to Set, whose action on morphisms is

$$\begin{array}{lcl}
_\le?_ : iz \le jz \to \mathsf{All}\ P\ jz \to \mathsf{All}\ P\ iz & & \\
\mathsf{oz} \quad\ \le?\ [] & = & [] \\
(\theta\ \mathsf{os}) \le? (pz \mathbin{-\!,} p) & = & (\theta \le? pz) \mathbin{-\!,} p \\
(\theta\ \mathsf{o'}) \le? (pz \mathbin{-\!,} p) & = & \theta \le? pz
\end{array}$$

Natural Transformation. Given functors F and G from $\mathbb{C}$ to $\mathbb{D}$, a *natural transformation* is a family of $\mathbb{D}$-morphisms $k_X \in \mathbb{D}(F_o(X), G_o(X))$ indexed by $\mathbb{C}$-objects, $X \in |\mathbb{C}|$, satisfying the *naturality* condition, which is that for any $h \in \mathbb{C}(S,T)$, we have $k_S; G_m(h) = F_m(h); k_T$, amounting to a kind of *uniformity*. It tells us that k_X does not care what X is, but only about the additional structure imposed by F and G.

Parametric polymorphism famously induces naturality [7], in that ignorance of a parameter imposes uniformity, and the same is true in our more nuanced setting. We noted that $\lambda\ P \rightarrow \mathsf{All}\ P\ kz$ is a functor (with action all) from $K \rightarrow \mathsf{Set}$ to Set. Accordingly, if $\theta : iz \leq jz$ then $(\theta \leq ?_)$is a natural transformation from $\lambda\ P \rightarrow \mathsf{All}\ P\ jz$ to $\lambda\ P \rightarrow \mathsf{All}\ P\ iz$, which is as much as to say that the definition of $\leq ?$ is uniform in P, and hence that if $f : \{k : K\} \rightarrow P\ k \rightarrow Q\ k$, then $\mathsf{all}\ f\ (\theta \leq ?\ pz) = \theta \leq ?\ \mathsf{all}\ f\ pz$.

Dependently typed programming thus offers us a much richer seam of categorical structure than the basic types-and-functions familiar from Haskell (which demands some negotiation of totality even to achieve that much). For me, at any rate, this represents an opportunity to make sense of the categorical taxonomy by relating it to concrete programming examples, and at the same time, organising those programs and giving healthy indications for *what to prove*.

6 Things-with-Thinnings (a Monad)

Let us develop the habit of packing terms with an object in the slice category of thinnings, selecting the support of the term and discarding unused variables at the root. Note that $/$ is a functor from $\overline{K}$ to itself.

```
record _/_ {K} (T : K̄) (scope : Bwd K) : Set where    -- (T /_) : K̄
  constructor _↑_
  field {support} : Bwd K; thing : T support; thinning : support ≤ scope
map/ : ∀ {K} {S T : K̄} → (S ∸→ T) → ((S /_) ∸→ (T /_))
map/ f (s ↑ θ) = f s ↑ θ
```

In fact, the categorical structure of **OPE** makes $/$ a *monad*. Let us recall the definition.

Monad. A functor M from $\mathbb{C}$ to $\mathbb{C}$ gives rise to a *monad* (M, η, μ) if we can find a pair of natural transformations, respectively 'unit' ('add an M layer') and 'multiplication' ('merge M layers').

$$\eta_X : \mathbf{I}(X) \to M(X) \qquad\qquad \mu_X : M(M(X)) \to M(X)$$

subject to the conditions that merging an added layer yields the identity (whether the layer added is 'outer' or 'inner'), and that adjacent M layers may be merged pairwise in any order.

$$\eta_{M(X)}; \mu_X = \iota_{M(X)} \qquad\qquad M(\eta_X); \mu_X = \iota_{M(X)} \qquad\qquad \mu_{M(X)}; \mu_X = M(\mu_X); \mu_X$$

The categorical structure of thinnings makes $/$ a monad. Here, 'adding a layer' amounts to 'wrapping with a thinning'. The proof obligations to make $(/, \mathsf{unit/}, \mathsf{mult/})$ a monad are exactly those required to make **OPE** a category in the first place. In particular, things-with-thinnings are easy to thin further, indeed, parametrically so. In other words, $(T\ /)$ is uniformly a functor from **OPE** to Set.

```
unit/ : T ∸→ (T /_)        mult/ : ((T /_) /_) ∸→ (T /_)        thin/ : iz ≤ jz → T / iz → T / jz
unit/ t = t ↑ oi           mult/ ((t ↑ θ) ↑ φ) = t ↑ (θ ⨾ φ)     thin/ θ t = mult/ (t ↑ θ)
```

Shortly, we shall give an operation to discover the variables in scope on which a term syntactically depends. However, merely *allowing* a thinning at the root, Lam / *iz*, yields a poor representation of terms over *iz*, as we may choose whether to discard unwanted variables either at root or at leaves. To eliminate redundancy, we must *insist* that a term's support is *relevant*: if a variable is not discarded by the thinning, it *must* be used in the thing. Or as Spike Milligan put it, 'Everybody's got to be somewhere.'.

7 The Curious Case of the Coproduct in Slices of OPE

The / construction makes crucial use of objects in the slice category **OPE**/*scope*, which exhibit useful additional structure: they are *bit vectors*, with one bit per variable telling whether it has been selected. Bit vectors inherit Boolean structure, via the 'Naperian' array structure of vectors [5].

Initial object. A category $\mathbb{C}$ has initial object 0, if there is a unique morphism in $\mathbb{C}(0,X)$ for every X.

We are used to the *empty type* playing this rôle for types-and-functions: empty case analysis gives the vacuously unique morphism. In **OPE**, the empty *scope* plays the same rôle, with the 'constant 0' bit vector as unique morphism. By return of post, we obtain that $([],oe)$ is the initial object in the slice category $\_ \le kz$.

```
oe : ∀ {K} {kz : Bwd K} → [] ≤ kz
oe {kz = iz -, k} = oe o'
oe {kz = []}      = oz

law−oe : (θ : [] ≤ kz) → θ == oe
```

```
oe/ : (θ : iz ≤ kz) → oe →/ θ
oe/ θ with tri oe θ
... | t rewrite law−oe (oe ⨾ θ) = oe , t
```

We can now make *constants* with empty support, i.e., noting that no variable is ($\cdot_R$ for) *relevant*.

```
data One_R {K} : K̄ where ⟨⟩ : One_R []
```

```
⟨⟩_R : One_R / kz
⟨⟩_R = ⟨⟩ ↑ oe
```

We should expect the constant to be the trivial case of some notion of *relevant pairing*, induced by *coproducts* in the slice category. If we have two objects in $(\_ \le kz)$ representing two subscopes, there should be a smallest subscope which includes both: pairwise disjunction of bit vectors.

Coproduct. Objects S and T of category $\mathbb{C}$ have a coproduct object $S+T$ if there are morphisms $l \in \mathbb{C}(S,S+T)$ and $r \in \mathbb{C}(T,S+T)$ such that every pair $f \in \mathbb{C}(S,U)$ and $g \in \mathbb{C}(T,U)$ factors through a unique $h \in \mathbb{C}(S+T,U)$ so that $f = l;h$ and $g = r;h$. In Set, we may take $S+T$ to be the *disjoint union* of S and T, with l and r its injections and h the *case analysis* whose branches are f and g.

However, we are not working in Set, but in a slice category. Any category theorist will tell you that slice categories $\mathbb{C}/I$ inherit *colimit* structure (characterized by universal out-arrows) from $\mathbb{C}$, as indeed we just saw with the initial object. If **OPE** has coproducts, too, we are done!

Curiously, though, **OPE** does *not* have coproducts. Taking K = One, let us seek the coproduct of two singletons, $S = T = [] \mathbin{-,} \langle\rangle$. Construct one diagram by taking $U = [] \mathbin{-,} \langle\rangle$ and $f = g =$ oi, ensuring that our only candidate for $S+T$ is again the singleton $[] \mathbin{-,} \langle\rangle$, with $l = r =$ oi, making $h =$ oi. Nothing else can sit between S,T and U. Now begin a different diagram, with $U' = [] \mathbin{-,} \langle\rangle \mathbin{-,} \langle\rangle$, allowing $f' =$ oz os o' and $g' =$ oz o' os. No h' post-composes l and r (both oi, making h' itself) to yield f' and g' respectively.

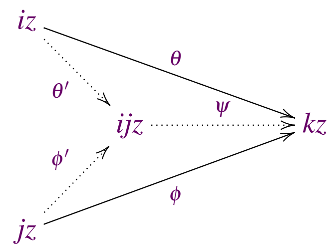
Fortunately, we shall get what we need. **OPE** may not have coproducts, but its *slices* do. Examine the data. We have two subscopes of some kz, $\theta : iz \le kz$ and $\phi : jz \le kz$. Their coproduct must be some

$\psi : ijz \le kz$, where our l and r must be triangles Tri $\theta'\ \psi\ \theta$ and Tri $\phi'\ \psi\ \phi$, giving us morphisms in $\theta \to_/ \psi$ and $\phi \to_/ \psi$, respectively. Intuitively, we should choose ψ to be the pointwise disjunction of θ and ϕ, so that ijz is as small as possible: θ' and ϕ' will then *cover* ijz. The flag, ov, determines whether *overlap* is permitted: this should be tt for coproducts, but ov = ff allows the notion of *partition*, too.

```
data Cover {K} (ov : Two) : {iz jz ijz : Bwd K} → iz ≤ ijz → jz ≤ ijz → Set where
  _c's : Cover ov θ ϕ → Cover ov (θ o') (ϕ os)
  _cs' : Cover ov θ ϕ → Cover ov (θ os) (ϕ o')
  _css : Cover ov θ ϕ → Cover ov (θ os) (ϕ os)
  czz  :                Cover ov    oz      oz
```

Note that we have no constructor which allows both θ and ϕ to omit a target variable: everybody's got to be somewhere. Let us compute the coproduct, then check its universal property.

```
cop : ∀ {K} {kz iz jz : Bwd K}
      (θ : iz ≤ kz) (ϕ : jz ≤ kz) →
      Σ _ λ ijz → Σ (ijz ≤ kz) λ ψ →
      Σ (iz ≤ ijz) λ θ' → Σ (jz ≤ ijz) λ ϕ' →
      Tri θ' ψ θ × Cover tt θ' ϕ' × Tri ϕ' ψ ϕ
```




```
cop (θ o') (ϕ o') = let ! ! ! ! tl , c , tr = cop θ ϕ in ! ! ! ! tl t-'' , c     , tr t-''
cop (θ o') (ϕ os) = let ! ! ! ! tl , c , tr = cop θ ϕ in ! ! ! ! tl t's' , c c's , tr tsss
cop (θ os) (ϕ o') = let ! ! ! ! tl , c , tr = cop θ ϕ in ! ! ! ! tl tsss , c cs' , tr t's'
cop (θ os) (ϕ os) = let ! ! ! ! tl , c , tr = cop θ ϕ in ! ! ! ! tl tsss , c css , tr tsss
cop    oz     oz  =                                      ! ! ! !   tzzz ,   czz ,    tzzz
```

To show that we have really computed a coproduct, we must show that any other pair of triangles from θ and ϕ to some ψ' must induce a (unique) morphism from ψ to ψ'.

```
copU : Tri θ' ψ θ → Cover tt θ' ϕ' → Tri ϕ' ψ ϕ →
       ∀ {ijz'} {ψ' : ijz' ≤ kz} → θ →/ ψ' → ϕ →/ ψ' → ψ →/ ψ'
```

The construction goes by induction on the triangles which share ψ' and inversion of the coproduct. The payoff from the coproduct construction is the type of *relevant pairs* — the co-de-Bruijn touchstone:

```
record _×R_ (S T : K̄) (ijz : Bwd K) : Set where
  constructor pair
  field outl  : S / ijz
        outr  : T / ijz
        cover : Cover tt (thinning outl) (thinning outr)
```

```
_,R_ : S / kz → T / kz → (S ×R T) / kz
(s ↑ θ) ,R (t ↑ ϕ) =
  let ! ψ , θ' , ϕ' , _ , c , _ = cop θ ϕ
  in  pair (s ↑ θ') (t ↑ ϕ') c ↑ ψ
```

8 Monoidal Structure of Order-Preserving Embeddings

To talk about binding, we need scope extension. We have seen single 'snoc', but binding is simultaneous in general. Concatenation induces monoidal structure on objects of **OPE**, extending to morphisms.

$$\begin{array}{lcl} _{+\!+}_ : (iz\ jz : \mathsf{Bwd}\ K) \to \mathsf{Bwd}\ K \\ kz +\!+ [] & = & kz \\ kz +\!+ (iz \mathbin{-\!,} j) & = & (kz +\!+ iz) \mathbin{-\!,} j \end{array}
\qquad
\begin{array}{lcl} _{+\!+_\le}_ : iz \le jz \to iz' \le jz' \to (iz +\!+ iz') \le (jz +\!+ jz') \\ \theta +\!+_\le \ \mathsf{oz} & = & \theta \\ \theta +\!+_\le (\phi\ \mathsf{os}) & = & (\theta +\!+_\le \phi)\ \mathsf{os} \\ \theta +\!+_\le (\phi\ \mathsf{o'}) & = & (\theta +\!+_\le \phi)\ \mathsf{o'} \end{array}$$

Moreover, given a embedding into a concatenation, we can split it into local and global parts.

$$\begin{array}{l} \_\dashv\_ : (jz : \mathsf{Bwd}\ K)\ (\psi : iz \le (kz +\!+ jz)) \to \Sigma\ (\mathsf{Bwd}\ K)\ \lambda\ kz' \to \Sigma\ (\mathsf{Bwd}\ K)\ \lambda\ jz' \to \\ \quad \Sigma\ (kz' \le kz)\ \lambda\ \theta \to \Sigma\ (jz' \le jz)\ \lambda\ \phi \to \Sigma\ (iz == (kz' +\!+ jz'))\ \lambda\ \{\mathsf{refl} \to \psi == (\theta +\!+_\le \phi)\} \\ [] \quad \dashv \psi \qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad = !\,!\ \psi , \mathsf{oz} , \mathsf{refl} , \mathsf{refl} \\ (kz \mathbin{-\!,} k) \dashv (\psi\ \mathsf{os}) \qquad\qquad \mathbf{with}\ kz \dashv \psi \\ (kz \mathbin{-\!,} k) \dashv (.\ (\theta +\!+_\le \phi)\ \mathsf{os}) \mid\ !\,!\ \theta , \phi , \mathsf{refl} , \mathsf{refl} = !\,!\ \theta , \phi\ \mathsf{os} , \mathsf{refl} , \mathsf{refl} \\ (kz \mathbin{-\!,} k) \dashv (\psi\ \mathsf{o'}) \qquad\qquad \mathbf{with}\ kz \dashv \psi \\ (kz \mathbin{-\!,} k) \dashv (.\ (\theta +\!+_\le \phi)\ \mathsf{o'}) \mid\ !\,!\ \theta , \phi , \mathsf{refl} , \mathsf{refl} = !\,!\ \theta , \phi\ \mathsf{o'} , \mathsf{refl} , \mathsf{refl} \end{array}$$

Thus equipped, we can say how to bind some variables. The key is to say at the binding site which of the bound variables will actually be used: if they are not used, we should not even bring them into scope.

$$\begin{array}{l} \mathbf{data}\ \_\vdash\_\ \{K\}\ (jz : \mathsf{Bwd}\ K)\ (T : \overline{K})\ (kz : \mathsf{Bwd}\ K) : \mathsf{Set}\ \mathbf{where} \quad \text{-- } jz \vdash T : \overline{K} \\ \quad \_\backslash\!\backslash\_ : \forall\ \{iz\} \to iz \le jz \to T\ (kz +\!+ iz) \to (jz \vdash T)\ kz \\ \_\backslash\!\backslash_R\_ : \forall\ \{K\ T\}\ \{kz\}\ (jz : \mathsf{Bwd}\ K) \to T\ /\ (kz +\!+ jz) \to (jz \vdash T)\ /\ kz \\ jz \backslash\!\backslash_R\ (t \uparrow \psi)\ \mathbf{with}\ jz \dashv \psi \\ jz \backslash\!\backslash_R\ (t \uparrow .\ (\theta +\!+_\le \phi)) \mid\ !\,!\ \theta , \phi , \mathsf{refl} , \mathsf{refl} = (\phi \backslash\!\backslash t) \uparrow \theta \end{array}$$

The monoid of scopes is generated from its singletons. By the time we *use* a variable, it should be the only thing in scope. The associated smart constructor computes the thinned representation of variables.

$$\begin{array}{l} \mathbf{data}\ \mathsf{Va}_R\ (k : K) : \overline{K}\ \mathbf{where} \\ \quad \mathsf{only} : \mathsf{Va}_R\ k\ ([] \mathbin{-\!,} k) \end{array}
\qquad
\begin{array}{l} \mathsf{va}_R : k \leftarrow kz \to \mathsf{Va}_R\ k\ /\ kz \\ \mathsf{va}_R\ x = \mathsf{only} \uparrow x \end{array}$$

Untyped λ-calculus. We can now give the λ-terms for which all *free* variables are relevant as follows. Converting de Bruijn to co-de-Bruijn representation is easy with smart constructors. E.g., compare de Bruijn terms for the $\mathbb{K}$ and $\mathbb{S}$ combinators with their co-de-Bruijn form.

$$\begin{array}{l} \mathbf{data}\ \mathsf{Lam}_R : \overline{\mathsf{One}}\ \mathbf{where} \\ \quad \# \quad\ : \mathsf{Va}_R\ \langle\rangle \qquad\qquad\quad \dot\to \mathsf{Lam}_R \\ \quad \mathsf{app} : (\mathsf{Lam}_R \times_R \mathsf{Lam}_R) \ \dot\to \mathsf{Lam}_R \\ \quad \lambda \quad\ : ([] \mathbin{-\!,} \langle\rangle \vdash \mathsf{Lam}_R) \quad \dot\to \mathsf{Lam}_R \end{array}
\qquad
\begin{array}{lcl} \mathsf{lam}_R : \mathsf{Lam} \ \dot\to (\mathsf{Lam}_R\ /_) \\ \mathsf{lam}_R\ (\#\ x) & = & \mathsf{map}/\ \# \quad (\mathsf{va}_R\ x) \\ \mathsf{lam}_R\ (f\ \$\ s) & = & \mathsf{map}/\ \mathsf{app}\ (\mathsf{lam}_R\ f\ ,_R\ \mathsf{lam}_R\ s) \\ \mathsf{lam}_R\ (\lambda\ t) & = & \mathsf{map}/\ \lambda \quad (_ \backslash\!\backslash_R\ \mathsf{lam}_R\ t) \end{array}$$

$$\begin{array}{l} \mathbb{K} \qquad\ = \lambda\ (\lambda\ (\#\ (\mathsf{oe\ os\ o'}))) \\ \mathsf{lam}_R\ \mathbb{K} = \lambda\ (\mathsf{oz\ os} \backslash\!\backslash\ \lambda\ (\mathsf{oz\ o'} \backslash\!\backslash\ \#\ \mathsf{only})) \uparrow \mathsf{oz} \\ \mathbb{S} \qquad\ = \lambda\ (\lambda\ (\lambda\ (\#\ (\mathsf{oe\ os\ o'\ o'})\ \$\ \#\ (\mathsf{oe\ os})\ \$\ (\#\ (\mathsf{oe\ os\ o'})\ \$\ \#\ (\mathsf{oe\ os}))))) \\ \mathsf{lam}_R\ \mathbb{S} = \lambda\ (\mathsf{oz\ os} \backslash\!\backslash\ \lambda\ (\mathsf{oz\ os} \backslash\!\backslash\ \lambda\ (\mathsf{oz\ os} \backslash\!\backslash \\ \quad \mathsf{app}\ (\mathsf{pair}\ (\mathsf{app}\ (\mathsf{pair}\ (\#\ \mathsf{only} \uparrow \mathsf{oz\ os\ o'})\ (\#\ \mathsf{only} \uparrow \mathsf{oz\ o'\ os})\ (\mathsf{czz\ cs'\ c's})) \uparrow \mathsf{oz\ os\ o'\ os}) \\ \qquad\qquad\quad (\mathsf{app}\ (\mathsf{pair}\ (\#\ \mathsf{only} \uparrow \mathsf{oz\ os\ o'})\ (\#\ \mathsf{only} \uparrow \mathsf{oz\ o'\ os})\ (\mathsf{czz\ cs'\ c's})) \uparrow \mathsf{oz\ o'\ os\ os}) \\ \qquad\qquad\quad (\mathsf{czz\ cs'\ c's\ css}))))) \uparrow \mathsf{oz} \end{array}$$

Staring bravely, we see that $\mathbb{K}$ uses its first argument to deliver a plainly constant function: the second λ discards its argument. Meanwhile, $\mathbb{S}$ clearly uses all three inputs ('function', 'argument', 'environment'): in the application, the function goes left, the argument goes right, and the environment is shared.

9 A Universe of Metasyntaxes-with-Binding

There is nothing specific to the λ-calculus about de Bruijn representation or its co-de-Bruijn counterpart. We may develop the notions generically for multisorted syntaxes. If the sorts of our syntax are drawn from set I, then we may characterize terms-with-binding as inhabiting Kinds $kz \Rightarrow i$, which specify an extension of the scope with new bindings kz and the sort i for the body of the binder.

```
record Kind (I : Set) : Set where inductive; constructor _⇒_
                                  field scope : Bwd (Kind I); sort : I
```

Kinds offer higher-order abstraction: a bound variable itself has a Kind, being an object sort parametrized by a scope, where the latter is, as in previous sections, a Bwd list, with K now fixed as Kind I. Object variables have sorts; *meta*-variables have Kinds. E.g., in the β-rule, t and s are not object variables like x

$$(\lambda x.t[x])\,s \;\rightsquigarrow\; t[s]$$

but placeholders, s for some term and $t[x]$ for some term with a parameter which can be and is instantiated, by x on the left and s on the right. The kind of t is $[] \mathbin{-\!,} ([] \Rightarrow \langle\rangle) \Rightarrow \langle\rangle$.

We may give the syntax of each sort as a function mapping sorts to Descriptions $D : I \to$ Desc I.

```
data Desc (I : Set) : Set₁ where
  Rec_D  : Kind I →                        Desc I
  One_D  :                                 Desc I
  _×_D_  : Desc I → Desc I →               Desc I
  Σ_D    : (S : Datoid) → (Data S → Desc I) → Desc I
```

We may ask for a subterm with a given Kind, so it can bind variables by listing their Kinds left of $\Rightarrow$. Descriptions are closed under unit and pairing. We may also ask for terms to be tagged by some sort of 'constructor' inhabiting some Datoid, i.e., a set with a decidable equality, given as follows:

```
data Decide (X : Set) : Set where        record Datoid : Set₁ where
  yes : X →             Decide X           field Data   : Set
  no  : (X → Zero) → Decide X                    decide : (x y : Data) → Decide (x == y)
```

Describing untyped λ-calculus. Define a tag enumeration, then a description.

```
data LamTag : Set where app λ : LamTag      decide LAMTAG app app = yes refl
LAMTAG : Datoid                             decide LAMTAG app λ   = no λ ()
Data LAMTAG = LamTag                        decide LAMTAG λ   app = no λ ()
                                            decide LAMTAG λ   λ   = yes refl

Lam_D : One → Desc One
Lam_D ⟨⟩ = Σ_D LAMTAG λ {app → Rec_D ([] ⇒ ⟨⟩) ×_D Rec_D ([] ⇒ ⟨⟩)
                        ;λ   → Rec_D ([] -, ([] ⇒ ⟨⟩) ⇒ ⟨⟩)}
```

Note that we do not and cannot include a tag or description for the use sites of variables in terms: use of variables in scope pertains not to the specific syntax, but to the general notion of what it is to be a syntax.

Interpreting Desc as de Bruijn Syntax. Let us give the de Bruijn interpretation of our syntax descriptions. We give meaning to Desc in the traditional manner, interpreting them as strictly positive operators in some R which gives the semantics to Rec_D. In recursive positions, the scope grows by the bindings demanded by the given Kind. At use sites, higher-kinded variables must be instantiated, just like $t[x]$ in the β-rule example: Sp_D computes the Description of the spine of actual parameters required.

$$
\begin{array}{lcl}
[\![\_|\_]\!] : \forall\,\{I\} \to \mathsf{Desc}\,I \to (I \to \overline{\mathsf{Kind}\,I}) \to \overline{\mathsf{Kind}\,I} & & \\
[\![\,\mathsf{Rec}_D\,k\;|\,R\,]\!]\,kz & = & R\,(\mathsf{sort}\,k)\,(kz +\!\!+ \mathsf{scope}\,k) \\
[\![\,\Sigma_D\,S\,T\;|\,R\,]\!]\,kz & = & \Sigma\,(\mathsf{Data}\,S)\,\lambda\,s \to [\![\,T\,s\,|\,R\,]\!]\,kz \\
[\![\,\mathsf{One}_D\;\;|\,R\,]\!]\,kz & = & \mathsf{One} \\
[\![\,S \times_D T\,|\,R\,]\!]\,kz & = & [\![\,S\,|\,R\,]\!]\,kz \times [\![\,T\,|\,R\,]\!]\,kz
\end{array}
$$

$$
\begin{array}{lcl}
\mathsf{Sp}_D : \mathsf{Bwd}\,(\mathsf{Kind}\,I) \to \mathsf{Desc}\,I & & \\
\mathsf{Sp}_D\;\;[] & = & \mathsf{One}_D \\
\mathsf{Sp}_D\,(kz \,\text{-,}\, k) & = & \mathsf{Sp}_D\,kz \times_D \mathsf{Rec}_D\,k
\end{array}
$$

Tying the knot, we find that a term is either a variable instantiated with its spine of actual parameters, or it is a construct of the syntax for the demanded sort, with subterms in recursive positions.

$$
\begin{array}{l}
\textbf{data}\;\mathsf{Tm}\,\{I\}\,(D : I \to \mathsf{Desc}\,I)\,(i : I)\,kz : \mathsf{Set}\;\textbf{where}\quad \text{-- } \mathsf{Tm}\,D\,i : \overline{\mathsf{Kind}\,I} \\
\quad \_\#\$\_ : \forall\,\{jz\} \to (jz \Rightarrow i) \leftarrow kz \to [\![\,\mathsf{Sp}_D\,jz\,|\,\mathsf{Tm}\,D\,]\!]\,kz \to \mathsf{Tm}\,D\,i\,kz \\
\quad [\_]\;\;\, : \qquad\qquad\qquad\qquad\qquad\quad [\![\,D\,i\quad\;|\,\mathsf{Tm}\,D\,]\!]\,kz \to \mathsf{Tm}\,D\,i\,kz
\end{array}
$$

Interpreting Desc as co-de-Bruijn Syntax. Now let us interpret Descriptions in co-de-Bruijn style, enforcing that all variables in scope are relevant, and that binding sites expose vacuity. We can compute co-de-Bruijn terms from de Bruijn terms, generically.

$$
\begin{array}{lcl}
[\![\_|\_]\!]_R : \forall\,\{I\} \to \mathsf{Desc}\,I \to (I \to \overline{\mathsf{Kind}\,I}) \to \overline{\mathsf{Kind}\,I} & & \\
[\![\,\mathsf{Rec}_D\,k\;|\,R\,]\!]_R & = & \mathsf{scope}\,k \vdash R\,(\mathsf{sort}\,k) \\
[\![\,\Sigma_D\,S\,T\;|\,R\,]\!]_R & = & \lambda\,kz \to \Sigma\,(\mathsf{Data}\,S)\,\lambda\,s \to [\![\,T\,s\,|\,R\,]\!]_R\,kz \\
[\![\,\mathsf{One}_D\;\;|\,R\,]\!]_R & = & \mathsf{One}_R \\
[\![\,S \times_D T\,|\,R\,]\!]_R & = & [\![\,S\,|\,R\,]\!]_R \times_R [\![\,T\,|\,R\,]\!]_R
\end{array}
$$

$$
\begin{array}{l}
\textbf{data}\;\mathsf{Tm}_R\,\{I\}\,(D : I \to \mathsf{Desc}\,I)\,(i : I) : \overline{\mathsf{Kind}\,I}\;\textbf{where} \\
\quad \#\;\;\; : \forall\,\{jz\} \to (\mathsf{Va}_R\,(jz \Rightarrow i) \times_R [\![\,\mathsf{Sp}_D\,jz\,|\,\mathsf{Tm}_R\,D\,]\!]_R) \mathrel{\dot\to} \mathsf{Tm}_R\,D\,i \\
\quad [\_] : \qquad\qquad\qquad\qquad\qquad\quad [\![\,D\,i\quad\;|\,\mathsf{Tm}_R\,D\,]\!]_R \mathrel{\dot\to} \mathsf{Tm}_R\,D\,i
\end{array}
$$

$$
\begin{array}{llcl}
\mathsf{code}\;\; : \forall\,\{I\}\,\{D : I \to \mathsf{Desc}\,I\}\,\{i\} \to \mathsf{Tm}\,D\,i & & \mathrel{\dot\to} & (\mathsf{Tm}_R\,D\,i\,/\_) \\
\mathsf{codes} : \forall\,\{I\}\,\{D : I \to \mathsf{Desc}\,I\}\,S \to [\![\,S\,|\,\mathsf{Tm}\,D\,]\!] & & \mathrel{\dot\to} & ([\![\,S\,|\,\mathsf{Tm}_R\,D\,]\!]_R\,/\_) \\
\mathsf{code} & (\_\#\$\_\,\{jz\}\,x\,ts) & = & \mathsf{map}/\;\#\;\;(\mathsf{va}_R\,x\;,_R\,\mathsf{codes}\,(\mathsf{Sp}_D\,jz)\,ts) \\
\mathsf{code}\,\{D = D\}\,\{i = i\}\,[\,ts\,] & & = & \mathsf{map}/\;[\_]\,(\mathsf{codes}\,(D\,i)\,ts) \\
\mathsf{codes}\,(\mathsf{Rec}_D\,k) & t & = & \mathsf{scope}\,k \mathbin{\backslash\!\backslash_R} \mathsf{code}\,t \\
\mathsf{codes}\,(\Sigma_D\,S\,T) & (s\,,\,ts) & = & \mathsf{map}/\;(s\,,\_)\,(\mathsf{codes}\,(T\,s)\,ts) \\
\mathsf{codes}\,\mathsf{One}_D & \langle\rangle & = & \langle\rangle_R \\
\mathsf{codes}\,(S \times_D T) & (ss\,,\,ts) & = & \mathsf{codes}\,S\,ss\;,_R\,\mathsf{codes}\,T\,ts
\end{array}
$$

10 Hereditary Substitution for Co-de-Bruijn Metasyntax

Let us develop the appropriate notion of substitution for our metasyntax, *hereditary* in the sense of Watkins et al. [8]. Substituting a higher-kinded variable requires us further to substitute its parameters.

There is some subtlety to the construction of the record type HSub of hereditary substitutions. We may partition the source scope into passive variables, which embed into the target, and active variables for which we have an environment images appropriate to their kinds. The HSub type is indexed by a third scope which bounds the active kinds, by way of ensuring *termination*.

```
record HSub {I} D (src trg bnd : Bwd (Kind I)) : Set where
  field pass act : Bwd (Kind I); passive : pass ≤ src; active : act ≤ src
        parti : Cover ff passive active; passTrg : pass ≤ trg; actBnd : act ≤ bnd
        images  : All (λ k → (scope k ⊢ Tm_R D (sort k)) / trg) act
```

Before we see how to perform a substitution, let us consider how to *weaken* one, as we shall certainly need to push under binders, where we have some $\phi : iz \le jz$ telling us which iz of the jz bound variables are used in the source term. Either way, bound variables are not substituted, so we add them to the passive side, at the same time keeping the active side below its bound.

```
wkHSub : HSub D src trg bnd → iz ≤ jz → HSub D (src ++ iz) (trg ++ jz) bnd
wkHSub {iz = iz} {jz = jz} h φ = record
  {parti = bindPassive iz; actBnd = actBnd h; passTrg = passTrg h ++≤ φ
  ;images = all (thin/ (oi ++≤ oe {kz = jz})) (images h)} where
  bindPassive : ∀ iz → Cover ff (passive h ++≤ oi {kz = iz}) (active h ++≤ oe {kz = iz})
```

As in a de Bruijn substitution, we must thin all the images, but the co-de-Bruijn representation avoids any need to traverse them — just compose thinnings at the root.

A second ancillary operation on HSub is to cut them down to just what is needed as variables are expelled from the source context by the thinnings stored in relevant pairs. We may select from an environment, but we must also refine the partition to cover just those source variables which remain, hence the selPart operation, which is a straightforward induction.

```
selHSub : src ≤ src′ → HSub D src′ trg bnd → HSub D src trg bnd
selHSub ψ (record {parti = c′; actBnd = θ′; images = tz′; passTrg = φ′}) =
  let ! ! ! ! φ , θ , c = selPart ψ c′ in record
    {parti = c; actBnd = θ ⨾ θ′; images = θ ≤? tz′; passTrg = φ ⨾ φ′}
selPart :  (ψ : kz ≤ kz′) → Cover ff θ′ φ′ → Σ _ λ iz → Σ _ λ jz →
  Σ (iz ≤ kz) λ θ → Σ (jz ≤ kz) λ φ → Σ (iz ≤ iz′) λ ψ₀ → Σ (jz ≤ jz′) λ ψ₁ → Cover ff θ φ
```

The definition of hereditary substitution is a mutual recursion, terminating because the active scope is always decreasing: hSub is the main operation on terms; hSubs and $\mathsf{hSubs}_{/}$ proceed structurally, in accordance with a syntax description; hered invokes hSub hereditarily.

```
hSub   :                HSub D src trg bnd → Tm_R D i src        → Tm_R D i / trg
hSubs  : (S : Desc I) → HSub D src trg bnd → ⟦ S | Tm_R D ⟧_R src   → ⟦ S | Tm_R D ⟧_R / trg
hSubs/ : (S : Desc I) → HSub D src trg bnd → ⟦ S | Tm_R D ⟧_R / src → ⟦ S | Tm_R D ⟧_R / trg
hered  : (jz ⊢ Tm_R D i) / trg → [] -, (jz ⇒ i) ≤ bnd → ⟦ Sp_D jz | Tm_R D ⟧_R / trg → Tm_R D i / trg
```

When hSub finds a variable, selHSub will reduce the parti to a single choice: if the variable is passive, embed it in target scope and reattach its substituted spine; if active, proceed hereditarily.

```
hSub {D = D} {i = i} h [ ts ] = map/ [_] (hSubs (D i) h ts)
hSub h (# {jz} (pair (only ↑ θ) ts _)) with selHSub θ h | hSubs/ (Sp_D jz) h ts
... | record {parti = _css {both = ()} _}                   | ts′
```

```
... | record {parti = czz cs’; passTrg = ϕ}                  | ts′ = map/ # (var_R ϕ ,_R ts′)
... | record {parti = czz c’s; actBnd = θ′; images = [] -, im} | ts′ = hered im θ′ ts′
```

To substitute a variable hereditarily, find it in the bound: the scope of its kind becomes the new, *structurally smaller* bound. Helper function part partitions passive free variables from active bound variables, while spAll converts the spine to an environment of images.

```
hered                        im          (θ′ o’) ts′ = hered im θ′ ts′
hered {D = D} {trg = trg} ((ϕ ⑊ t) ↑ ψ) (θ′ os) ts′ = let ! ! c = part _ _ in
  hSub (record {parti = c; actBnd = ϕ; images = ϕ ≤? spAll ts′; passTrg = ψ}) t where
    spAll : ∀ {kz} → ⟦ Sp_D kz | Tm_R D ⟧_R / trg → All _ kz
    part  : ∀ kz iz → Σ (kz ≤ (kz ++ iz)) λ θ → Σ (iz ≤ (kz ++ iz)) λ θ′ → Cover ff θ θ′
```

In the structural part of the algorithm, we may exploit our richer usage information to stop as soon as the active variables have all left scope, thinning the remaining passive variables with no further traversal. The lemma allLeft shows that if the right of a partition is empty, the left must be full.

```
hSubs (Rec_D k) h (ϕ ⑊ t)      = scope k ⑊_R hSub (wkHSub h ϕ) t
hSubs (Σ_D S T) h (s , ts)      = map/ (s , _) (hSubs (T s) h ts)
hSubs One_D     h ⟨⟩            = ⟨⟩_R
hSubs (S ×_D T) h (pair s t _)  = hSubs_/ S h s ,_R hSubs_/ T h t
hSubs_/ S h′ (ts ↑ θ) with selHSub θ h′
hSubs_/ S h′ (ts ↑ θ) | record {parti = c; images = []; passTrg = ϕ} rewrite allLeft c = ts ↑ ϕ
hSubs_/ S h′ (ts ↑ θ) | h = hSubs S h ts
```

References

[1] Thorsten Altenkirch & Bernhard Reus (1999): *Monadic presentations of lambda-terms using generalized inductive types*. In: Computer Science Logic 1999.

[2] Francoise Bellegarde & James Hook (1995): *Substitution: A formal methods case study using monads and transformations*. Science of Computer Programming.

[3] Richard Bird & Ross Paterson (1999): *de Bruijn notation as a nested datatype*. Journal of Functional Programming 9(1), pp. 77–92.

[4] Nicolas G. de Bruijn (1972): *Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation*. Indagationes Mathematicæ 34, pp. 381–392.

[5] Jeremy Gibbons (2017): *APLicative Programming with Naperian Functors*. In Hongseok Yang, editor: Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings, Lecture Notes in Computer Science 10201, Springer, pp. 556–583.

[6] Masahiko Sato, Randy Pollack, Helmut Schwichtenberg & Takafumi Sakurai (2013): *Viewing λ-terms through Maps*. Indagationes Mathematicæ 24(4).

[7] Philip Wadler (1989): *Theorems for Free!* In: Proceedings of the Fourth International Conference on Functional Programming Languages and Computer Architecture, FPCA ’89, ACM, New York, NY, USA, pp. 347–359.

[8] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2003): *A Concurrent Logical Framework: The Propositional Fragment*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers, Lecture Notes in Computer Science 3085, Springer, pp. 355–377.