

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 paper gives a nameless *co*-de-Bruijn representation of syntax with binding. It owes a great deal to the work of Sato, Pollack, Schwichtengberg and Sakurai [3] on canonical representation of variable binding by mapping the use sites of each variable.

The key to any nameless representation of syntax is how it manages the way, at a variable usage site, we choose to use *one* variable and thus, implicitly not any of the others in scope. The business of selecting from variables in scope may, in general, be treated by considering how to embed the selection into the whole scope.

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 \times in its non-dependent special case.

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

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))`.

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: 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 class of objects, and $\mathbb{C}(S, T)$ for its class of morphisms with source S and T , where $S, T \in |\mathbb{C}|$. 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 \mathbf{OPE} for the category.)

```
data Bwd (K : Set) : Set where
  []      : Bwd K
  _-, _ : Bwd K → K → Bwd K
```

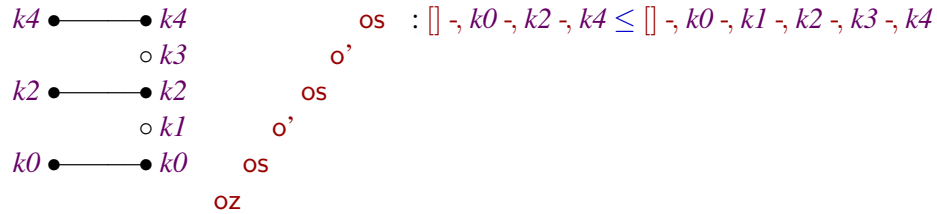
I use backward lists, because it is traditional to write contexts to the left of judgements in typing rules and extend them on the right. However, I use the word ‘scope’ rather than ‘context’ to suggest that we are characterizing at least which variables we may refer to, but perhaps not the full type information that one would find in a context.

The morphisms of \mathbf{OPE} give an embedding from a source scope to a target scope. Colloquially, we may call such embeddings ‘thinnings’, as they dilute the variables of the source scope with more variables. Equally, and usefully, we may see such a morphism as expelling variables from the target scope, leaving a particular selection as the source scope. I write the step constructors postfix, as thinnings (like contexts) grow on the right.

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

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.



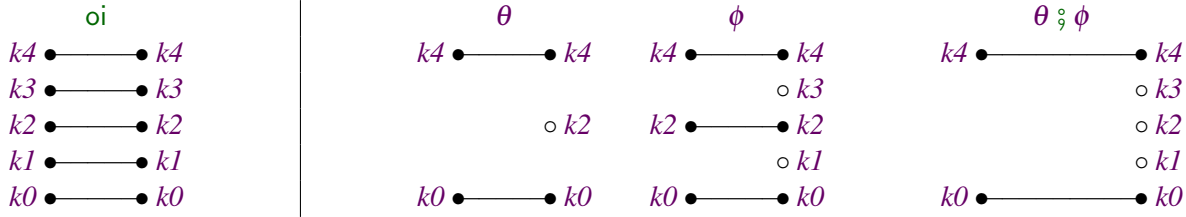
Category (II): Identity and Composition. The definition of a category insists on the existence of certain morphisms. Specifically, every object $X \in |\mathbb{C}|$ has an identity morphism $\iota_X \in \mathbb{C}(X, X)$ from that object to itself, and wherever the target of one morphism coincides with the source of another, a composite morphism must connect the source of the former to the target of the latter: 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 \circ . (Presentations of composition vary: for functions, it is much more common to write $g \circ f$ for ‘ g after f ’ than $f; g$ for ‘ f

then g' , but for thinnings, I prefer to retain a spatial intuition.)

$$\begin{array}{ll}
 \text{oi} : \forall \{K\} \{kz : \text{Bwd } K\} \rightarrow & \text{--}\circ\text{--} : \forall \{K\} \{iz\ jz\ kz : \text{Bwd } K\} \rightarrow \\
 kz \leq kz & iz \leq jz \rightarrow jz \leq kz \rightarrow iz \leq kz \\
 \text{oi} \{kz = iz \neg, k\} = \text{oi os} \quad \text{-- os preserves oi} & \theta \circ \phi \circ' = (\theta \circ \phi) \circ' \\
 \text{oi} \{kz = []\} = \text{oz} & \theta \circ' \circ \phi \text{ os} = (\theta \circ \phi) \circ' \\
 & \theta \text{ os} \circ \phi \text{ os} = (\theta \circ \phi) \text{ os} \quad \text{-- os preserves } \circ \\
 & \text{oz} \circ \text{oz} = \text{oz}
 \end{array}$$

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.

$$\begin{array}{ll}
 \text{law-oi}\circ : \forall \{K\} \{iz\ jz : \text{Bwd } K\} & (\theta : iz \leq jz) \rightarrow \text{oi} \circ \theta = \theta \\
 \text{law-}\circ\text{oi} : \forall \{K\} \{iz\ jz : \text{Bwd } K\} & (\theta : iz \leq jz) \rightarrow \theta \circ \text{oi} = \theta \\
 \text{law-}\circ\circ : \forall \{K\} \{iz\ jz\ kz\ lz : \text{Bwd } K\} & (\theta : iz \leq jz) (\phi : jz \leq kz) (\psi : kz \leq lz) \rightarrow \\
 & \theta \circ (\phi \circ \psi) = (\theta \circ \phi) \circ \psi
 \end{array}$$

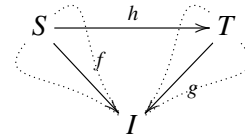
As one might expect, order-preserving embeddings have a strong antisymmetry property that one cannot expect of categories in general. The *only* invertible arrows are the identities. Note that we must match on the proof of $iz = jz$ even to claim that θ and ϕ are the identity.

$$\begin{array}{l}
 \text{antisym} : \forall \{K\} \{iz\ jz : \text{Bwd } K\} (\theta : iz \leq jz) (\phi : jz \leq iz) \rightarrow \\
 \Sigma (iz = jz) \lambda \{ \text{refl} \rightarrow \theta = \text{oi} \times \phi = \text{oi} \}
 \end{array}$$

3 Slices of Thinnings

If we fix the target of thinnings, $(_ \leq kz)$, 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 $f : S \rightarrow I$ is a morphism in \mathbb{C} . A morphism in $(S, f) \rightarrow (T, g)$ is some $h : S \rightarrow T$ such that $f = h \circ g$. (The dotted regions in the diagram show the objects in the slice.)



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 $iz \leq jz$ of θ):

$$\begin{aligned} \rightarrow_{/-} &: \forall \{K\} \{iz\ jz\ kz : \text{Bwd } K\} \rightarrow (iz \leq kz) \rightarrow (jz \leq kz) \rightarrow \text{Set} \\ \psi \rightarrow_{/-} \phi &= \Sigma _ \lambda \theta \rightarrow (\theta \circ \phi) = \psi \end{aligned}$$

because the equation gives us few options when it comes to manipulating triangles. Dependent pattern matching relies on *unification* of indices, but defined function symbols like \circ make unification difficult, obliging us to reason about the *edges* of the triangles. A helpful move at this point is to define triangles inductively, as the *graph* of \circ .

```
data Tri {K} : {iz jz kz : Bwd K} → iz ≤ jz → jz ≤ kz → iz ≤ kz → Set where
  _t-'' : ∀ {iz jz kz k} {θ : iz ≤ jz} {φ : jz ≤ kz} {ψ : iz ≤ kz} →
    Tri θ φ ψ → Tri {kz = -, k} θ (φ o') (ψ o')
  _t's' : ∀ {iz jz kz k} {θ : iz ≤ jz} {φ : jz ≤ kz} {ψ : iz ≤ kz} →
    Tri θ φ ψ → Tri {kz = -, k} (θ o') (φ os) (ψ o')
  _tsss : ∀ {iz jz kz k} {θ : iz ≤ jz} {φ : jz ≤ kz} {ψ : iz ≤ kz} →
    Tri θ φ ψ → Tri {kz = -, k} (θ os) (φ os) (ψ os)
  tzzz : Tri oz oz oz
infixl 8 _t-'' _t's' _tsss
```

Observe that the indexing is now entirely in constructor form, which will allow easy unification. Moreover, there is *no information* in a `Tri` structure, as its indexing (which derived from a pattern matching partition) completely determines its structure. The intended relationship between `Tri` and \circ thus holds.

$$\begin{aligned} \text{tri} &: \forall \{K\} \{iz\ jz\ kz : \text{Bwd } K\} (\theta : iz \leq jz) (\phi : jz \leq kz) \rightarrow \text{Tri } \theta \phi (\theta \circ \phi) \\ \text{comp} &: \forall \{K\} \{iz\ jz\ kz : \text{Bwd } K\} \{ \theta : iz \leq jz \} \{ \phi : jz \leq kz \} \{ \psi : iz \leq kz \} \rightarrow \\ &\quad \text{Tri } \theta \phi \psi \rightarrow \psi = (\theta \circ \phi) \end{aligned}$$

The proofs are easy inductions.

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

$$\begin{aligned} \text{egTri} &: \forall \{K\} \{k0\ k1\ k2\ k3\ k4 : K\} \rightarrow \text{Tri} \{kz = [], k0 -, k1 -, k2 -, k3 -, k4\} \\ &\quad (\text{oz os o' os}) (\text{oz os o' os o' os}) (\text{oz os o' o' o' os}) \\ \text{egTri} &= \text{tzzz tsss t-'' t's' t-'' tsss} \end{aligned}$$

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

$$\begin{aligned} \rightarrow_{/-} &: \forall \{K\} \{iz\ jz\ kz : \text{Bwd } K\} \rightarrow (iz \leq kz) \rightarrow (jz \leq kz) \rightarrow \text{Set} \\ \psi \rightarrow_{/-} \phi &= \Sigma _ \lambda \theta \rightarrow \text{Tri } \theta \phi \psi \end{aligned}$$

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!

$$\begin{aligned} \text{tri1} &: \forall \{K\} \{iz\ jz\ kz : \text{Bwd } K\} \{ \theta \theta' : iz \leq jz \} \{ \phi : jz \leq kz \} \{ \psi : iz \leq kz \} \rightarrow \\ &\quad \text{Tri } \theta \phi \psi \rightarrow \text{Tri } \theta' \phi \psi \rightarrow \theta = \theta' \end{aligned}$$

4 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 *endofunctors*, 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 $F_o : |\mathbb{C}| \rightarrow |\mathbb{D}|$ from source objects to target objects, together with a family of functions $F_m : \mathbb{C}(S, T) \rightarrow \mathbb{D}(F_o(S), F_o(T))$. 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 **I** (whose actions on objects and morphisms are the identity) from \mathbb{C} to itself and that functors compose (componentwise).

For example, every $k : K$ induces a functor from **OPE** to itself which acts by scope extension, $(-, \neg, k)$ on objects and **os** on morphisms. The very definitions of **oi** and **;** show that **os** preserves identity and composition. Let us refer to this functor as *weakening* by k .

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 $\mathbf{Set}(S, T)$ are functions of type $S \rightarrow T$, 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.

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

$$\begin{aligned} - \dot{\rightarrow} - &: \{I : \mathbf{Set}\} (S T : I \rightarrow \mathbf{Set}) \rightarrow \mathbf{Set} \\ S \dot{\rightarrow} T &= \forall \{i\} \rightarrow Si \rightarrow Ti \end{aligned}$$

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

```
data All {K} (P : K → Set) : Bwd K → Set where
  []      : All P []
  -, - : ∀ {kz k} → All P kz → P k → All P (kz ¬, k)
all : ∀ {K} {P Q : K → Set} → (P → Q) → (All P → All Q)
all f []      = []
all f (pz ¬, p) = all f pz ¬, f p
```

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 $\mathbf{Bwd} K \rightarrow \mathbf{Set}$ as \bar{K} , for types indexed over scopes.

However, **All** gives more functorial structure. Fixing kz , we obtain $\lambda P \rightarrow \mathbf{All} P kz$, a functor from $K \rightarrow \mathbf{Set}$ 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}^{op} and defined thus:

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

For example, $\mathbf{OPE}^{\text{op}}(jz, iz) = iz \leq jz$ 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, $\mathbf{All} P$ is a *functor* from \mathbf{OPE}^{op} to \mathbf{Set} , whose action on morphisms is

$$\begin{aligned} _ \leq? _ &: \forall \{K P\} \{jz iz : \mathbf{Bwd} K\} \rightarrow iz \leq jz \rightarrow \mathbf{All} P jz \rightarrow \mathbf{All} P iz \\ \mathbf{oz} \quad _ \leq? _ &= _ \\ (\theta \mathbf{os}) \leq? (pz \neg, p) &= (\theta \leq? pz) \neg, p \\ (\theta \mathbf{o}') \leq? (pz \neg, p) &= \theta \leq? pz \end{aligned}$$

It is not hard to check that the identity selection (selecting all elements) acts as the identity on environments, and that composition (making a subselection from a selection) is also respected.

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

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

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*.

5 De Bruijn Syntax via OPE

It is not unusual to find natural numbers as bound variables, perhaps with some bound enforced by typing, either by an inequality or by using a ‘finite set of size n ’ construction, often called ‘Fin’. However, we may also see scope membership as exactly singleton embedding, and then give the well scoped but untyped de Bruijn λ -terms using membership for variables [1].

$$\begin{aligned} _ \leftarrow _ &: \forall \{K\} \rightarrow K \rightarrow \overline{K} \quad \mathbf{data} \ \mathbf{Lam} \ (iz : \mathbf{Bwd} \ \mathbf{One}) : \mathbf{Set} \ \mathbf{where} \quad \text{-- } \mathbf{Lam} : \overline{\mathbf{One}} \\ k \leftarrow kz &= _ \neg, k \leq kz \quad \# \quad : (x : \langle \rangle \leftarrow iz) \rightarrow \mathbf{Lam} \ iz \quad \text{-- } \mathbf{finds} \ \text{a variable} \\ _ \mathbf{s} _ &: (f s : \mathbf{Lam} \ iz) \rightarrow \mathbf{Lam} \ iz \quad \text{-- } \mathbf{associates} \ \text{left} \\ \lambda \quad &: (t : \mathbf{Lam} \ (iz \neg, \langle \rangle)) \rightarrow \mathbf{Lam} \ iz \quad \text{-- } \mathbf{binds} \ \text{a variable} \end{aligned}$$

Variables are represented by pointing, which saves choosing names for them. We have but one way to encode a given term in a given scope. It is only when we point to one variable in scope that we

exclude the possibility of choosing the others. That is to say de Bruijn index representation effectively uses thinnings to discard unwanted variables as *late* as possible, in the *leaves* of syntax trees.

Note the way the scope index iz is used in the data type, as the target of a thinning in $\#$, and acted on by weakening in λ . Correspondingly, thinnings act functorially on terms, ultimately by postcomposition, but because terms keep their thinnings at their leaves, we must hunt the entire tree to find them.

$$\begin{aligned} _ \uparrow _ &: \forall \{iz\ jz\} \rightarrow \text{Lam } iz \rightarrow iz \leq jz \rightarrow \text{Lam } jz \\ \# i \quad \uparrow \theta &= \# (i \circ \theta) \\ (f \circ s) \uparrow \theta &= (f \uparrow \theta) \circ (s \uparrow \theta) \\ \lambda t \quad \uparrow \theta &= \lambda (t \uparrow \theta \circ s) \end{aligned}$$

The point of this paper is to consider the other canonical placement of the thinnings, nearest the *roots* of syntax trees, discarding unwanted variables at the *earliest* possible opportunity.

6 Things-with-Thinnings (a Monad)

Let us develop the habit of working with well scoped terms packed with a thinning at the root, discarding those variables from the scope which are not in the *support* of the term.

$$\begin{aligned} \text{record } _ / _ \{K\} (T : \overline{K}) (scope : \text{Bwd } K) : \text{Set where} \quad &-- (T / _) : \overline{K} \\ \text{constructor } _ \uparrow _ & \\ \text{field } \{support\} : \text{Bwd } K; \text{thing} : T \text{ support}; \text{thinning} : support \leq scope & \end{aligned}$$

Notice that $/$ is a functor from \overline{K} to itself, acting on morphisms as follows:

$$\begin{aligned} \text{map}/ : \forall \{K\} \{ST : \overline{K}\} \rightarrow (S \rightarrow T) \rightarrow ((S / _) \rightarrow (T / _)) \\ \text{map}/ f (s \uparrow \theta) = f s \uparrow \theta \end{aligned}$$

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’ and ‘multiplication’

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

which, respectively, ‘add an M layer’ and ‘merge M layers’, 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.

$$\eta_{M(X)}; \mu_X = \text{id}_{M(X)} \qquad M(\eta_X); \mu_X = \text{id}_{M(X)} \qquad \mu_{M(X)}; \mu_X = M(\mu_X); \mu_X$$

The categorical structure of thinnings induces a monadic structure on things-with-thinnings. Here, ‘adding a layer’ amounts to ‘wrapping with a thinning’, i.e., forgetting that only a selection from the variables in scope is needed.

$$\begin{aligned} \text{unit}/ : \forall \{K\} \{T : \overline{K}\} \rightarrow T \rightarrow (T / _) \quad & \text{mult}/ : \forall \{K\} \{T : \overline{K}\} \rightarrow ((T / _) / _) \rightarrow (T / _) \\ \text{unit}/ t = t \uparrow \text{id} \quad & \text{mult}/ ((t \uparrow \theta) \uparrow \phi) = t \uparrow (\theta \circ \phi) \end{aligned}$$

The proof obligations to make $(/ , \text{unit}/ , \text{mult}/)$ 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**.

$$\begin{aligned} \text{thin}/ : \forall \{K T\} \{iz jz : \text{Bwd } K\} &\rightarrow iz \leq jz \rightarrow T / iz \rightarrow T / jz \\ \text{thin}/ \theta t &= \text{mult}/ (t \uparrow \theta) \end{aligned}$$

Kleisli Category. Every monad (M, η, μ) on \mathbb{C} induces a category **Kleisli** $((M, \eta, \mu))$ with

$$|\mathbf{Kleisli}(M, \eta, \mu)| = |\mathbb{C}| \quad \mathbf{Kleisli}(M, \eta, \mu)(S, T) = \mathbb{C}(S, M(T)) \quad \iota_X^K = \eta_X \quad f;^K g = f; M(g); \mu$$

We sometimes call the morphisms in a Kleisli category *Kleisli arrows*.

The Kleisli arrows for $/$ are operations with types such as $S \rightarrow (T / _)$ which *discover dependency*: they turn an S over any scope kz into a T known to depend on at most some of the kz .

Shortly, we shall give exactly such an operation to discover the variables in scope on which a term syntactically depends. However, it is not enough to allow a thinning at the root: Lam / iz is a poor representation of terms over iz , as we may choose whether to discard unwanted variables either at root or at leaves. To recover a canonical positioning of thinnings, we must enforce the property 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 thing-with-thinning construction makes crucial use of ‘arrows into *scope* from some **support**’, which we might learn to recognize as objects in the slice category **OPE**/*scope*.

However, we also acquire some crucial additional structure. The key clue is that an object in the slice $(_ \leq kz)$ is effectively a *bit vector*, with one bit per variable telling whether or not it has been selected. Bit vectors inherit structure from Boolean algebra, via the ‘Naperian’ array structure of vectors [2].

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 in the category of types-and-functions, with the empty case analysis giving the morphism, which is vacuously pointwise unique. In **OPE**, the empty *scope* plays the same rôle, with the unique morphism given as the ‘constant false’ bit vector:

$$\begin{aligned} \text{oe} : \forall \{K\} \{kz : \text{Bwd } K\} &\rightarrow [] \leq kz & \text{law-oe} : \forall \{K\} \{kz : \text{Bwd } K\} \\ \text{oe} \{kz = iz \neg, k\} &= \text{oe } o' & (\theta : [] \leq kz) \rightarrow \theta = \text{oe} \\ \text{oe} \{kz = []\} &= \text{oz} \end{aligned}$$

By return of post, we obtain that $([], \text{oe})$ is the initial object in the slice category $_ \leq kz$.

$$\begin{aligned} \text{oe}/ : \forall \{K\} \{iz kz : \text{Bwd } K\} (\theta : iz \leq kz) &\rightarrow \text{oe} \rightarrow_ / \theta \\ \text{oe}/ \theta \text{ with tri oe } \theta & \\ \dots | t \text{ rewrite law-oe } (\text{oe} \circ \theta) &= \text{oe}, t \end{aligned}$$

The initial object gives us a way to make *constants* with empty support, i.e., noting that none of the available variables is *relevant*.


```

data OneR {K} :  $\bar{K}$  where  $\langle \rangle$  : OneR [] -- ‘R’ for relevant
 $\langle \rangle_R$  :  $\forall \{K\} \{kz : \text{Bwd } K\} \rightarrow \text{One}_R / kz$ 
 $\langle \rangle_R = \langle \rangle \uparrow \text{oe}$ 

```

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

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 $K = \text{One}$, let us try to construct the coproduct of two singleton scopes, $S = T = [] \neg, \langle \rangle$. We can begin one coproduct diagram by taking $U = [] \neg, \langle \rangle$ and $f = g = \text{oi}$: that forces the issue, as our only candidate for $S + T$ is once again the singleton $[] \neg, \langle \rangle$, with $l = r = \text{oi}$, making $h = \text{oi}$; a larger $S + T$ will not embed in U , a smaller will not embed S and T . However, we may begin a different coproduct diagram, taking $U' = [] \neg, \langle \rangle \neg, \langle \rangle$ with two variables, allowing us to choose $f' = \text{oz os o'}$ and $g' = \text{oz o' os}$, and now there is no way to choose an h' which 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 , $\theta : iz \leq kz$ and $\phi : jz \leq kz$. Their coproduct must be some $\psi : ijz \leq kz$, where our l and r must be triangles $\text{Tri } \theta' \psi \theta$ and $\text{Tri } \phi' \psi \phi$, giving us morphisms in $\theta \rightarrow_j \psi$ and $\phi \rightarrow_j \psi$, 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}  $\rightarrow iz \leq jz \rightarrow jz \leq ijz \rightarrow \text{Set}$  where
  -c's :  $\forall \{iz jz ijz k\} \{ \theta : iz \leq jz \} \{ \phi : jz \leq ijz \} \rightarrow$ 
    Cover ov  $\theta \phi \rightarrow \text{Cover ov } \{ijz = \neg, k\} (\theta \text{ o'}) (\phi \text{ os})$ 
  -cs' :  $\forall \{iz jz ijz k\} \{ \theta : iz \leq jz \} \{ \phi : jz \leq ijz \} \rightarrow$ 
    Cover ov  $\theta \phi \rightarrow \text{Cover ov } \{ijz = \neg, k\} (\theta \text{ os}) (\phi \text{ o'})$ 
  -css :  $\forall \{iz jz ijz k\} \{ \theta : iz \leq jz \} \{ \phi : jz \leq ijz \} \{ \text{both} : \text{Tt ov} \} \rightarrow$ 
    Cover ov  $\theta \phi \rightarrow \text{Cover ov } \{ijz = \neg, k\} (\theta \text{ os}) (\phi \text{ os})$ 
  czz : Cover ov oz oz

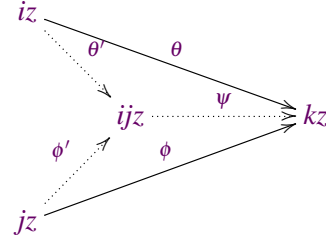
```

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 :  $\forall \{K\} \{kz \text{ } iz, jz : \text{Bwd } K\}$ 
       $(\theta : iz \leq kz) (\phi : jz \leq kz) \rightarrow$ 
       $\Sigma \_ \lambda \text{ } ijz \rightarrow \Sigma (ijz \leq kz) \lambda \text{ } \psi \rightarrow$ 
       $\Sigma (iz \leq ijz) \lambda \text{ } \theta' \rightarrow \Sigma (jz \leq ijz) \lambda \text{ } \phi' \rightarrow$ 
       $\text{Tri } \theta' \psi \theta \times \text{Cover } \text{tt } \theta' \phi' \times \text{Tri } \phi' \psi \phi$ 

```



```

cop (theta o') (phi o') = let !!!! tl, c, tr = cop theta phi in !!!! tl t-'', c'', tr t-'
cop (theta o') (phi os) = let !!!! tl, c, tr = cop theta phi in !!!! tl t's', c c's, tr tsss
cop (theta os) (phi o') = let !!!! tl, c, tr = cop theta phi in !!!! tl tsss, c cs', tr t's'
cop (theta os) (phi os) = let !!!! tl, c, tr = cop theta phi 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 :  $\forall \{K\} \{kz \text{ } iz, jz, ijz : \text{Bwd } K\}$ 
       $\{\theta : iz \leq kz\} \{\phi : jz \leq kz\} \{\psi : ijz \leq kz\} \{\theta' : iz \leq ijz\} \{\phi' : jz \leq ijz\} \rightarrow$ 
       $\text{Tri } \theta' \psi \theta \rightarrow \text{Cover } \text{tt } \theta' \phi' \rightarrow \text{Tri } \phi' \psi \phi \rightarrow$ 
       $\forall \{ijz'\} \{\psi' : ijz' \leq kz\} \rightarrow \theta \rightarrow_j \psi' \rightarrow \phi \rightarrow_j \psi' \rightarrow \psi \rightarrow_j \psi'$ 

```

The construction goes by induction on the triangles which share the edge ψ' and inversion of the coproduct construction. The payoff from this construction is the notion of *relevant pair*:

```

record  $\_ \times_R \_ \{K\} (ST : \overline{K}) (ijz : \text{Bwd } K) : \text{Set where}$  --  $S \times_R T : \overline{K}$ 
  constructor pair
  field outl :  $S / ijz$ ; outr :  $T / ijz$ ; cover :  $\text{Cover } \text{tt} (\text{thinning outl}) (\text{thinning outr})$ 
 $\rightarrow_R \_ : \forall \{K\} \{ST : \overline{K}\} \{kz\} \rightarrow S / kz \rightarrow T / kz \rightarrow (S \times_R T) / kz$ 
 $(s \uparrow \theta) ,_R (t \uparrow \phi) = \text{let } ! \psi, \theta', \phi', -, c, - = \text{cop } \theta \phi \text{ in pair } (s \uparrow \theta') (t \uparrow \phi') c \uparrow \psi$ 

```

8 Monoidal Structure of Order-Preserving Embeddings

In order to talk about binding, we need to talk about context extension. We have seen extension by a single ‘snoc’, but simultaneous binding also makes sense. Concatenation induces a monoidal structure on scopes, the objects of **OPE**, which extends to morphisms.

```

 $\_ ++ \_ : \forall \{K\} (kz \text{ } jz : \text{Bwd } K) \rightarrow \text{Bwd } K$ 
 $kz ++ [] = kz$ 
 $kz ++ (iz \neg j) = (kz ++ iz) \neg j$ 
 $\_ ++ \_ \leq \_ : \forall \{K\} \{iz, jz, iz', jz' : \text{Bwd } K\} \rightarrow$ 
 $iz \leq jz \rightarrow iz' \leq jz' \rightarrow (iz ++ iz') \leq (jz ++ jz')$ 
 $\theta ++ \_ \leq \text{oz} = \theta$ 
 $\theta ++ \_ \leq (\phi \text{ os}) = (\theta ++ \_ \leq \phi) \text{ os}$ 
 $\theta ++ \_ \leq (\phi \text{ o'}) = (\theta ++ \_ \leq \phi) \text{ o'}$ 

```

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

$$\begin{aligned}
& \neg _ : \forall \{K\} \{iz\} (jz : \text{Bwd } K) (\psi : iz \leq (kz ++ jz)) \rightarrow \\
& \quad \Sigma (\text{Bwd } K) \lambda kz' \rightarrow \Sigma (\text{Bwd } K) \lambda jz' \rightarrow \Sigma (kz' \leq kz) \lambda \theta \rightarrow \Sigma (jz' \leq jz) \lambda \phi \rightarrow \\
& \quad \Sigma (iz == (kz' ++ jz')) \lambda \{ \text{refl} \rightarrow \psi == (\theta ++ \leq \phi) \} \\
& \quad \square \quad \neg \psi \quad = !! \psi, \text{oz}, \text{refl}, \text{refl} \\
& (kz \neg, k) \neg (\psi \text{ os}) \quad \text{with } kz \neg \psi \\
& (kz \neg, k) \neg (\theta ++ \leq \phi \text{ os}) \mid !! \theta, \phi, \text{refl}, \text{refl} = !! \theta, \phi \text{ os}, \text{refl}, \text{refl} \\
& (kz \neg, k) \neg (\psi \text{ o}') \quad \text{with } kz \neg \psi \\
& (kz \neg, k) \neg (\theta ++ \leq \phi \text{ o}') \mid !! \theta, \phi, \text{refl}, \text{refl} = !! \theta, \phi \text{ o}', \text{refl}, \text{refl}
\end{aligned}$$

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.

$$\begin{aligned}
& \text{data } \neg _ : \{K\} (jz : \text{Bwd } K) (T : \overline{K}) (kz : \text{Bwd } K) : \text{Set where} \quad \neg jz \vdash T : \overline{K} \\
& \quad \neg _ : \forall \{iz\} \rightarrow iz \leq jz \rightarrow T (kz ++ iz) \rightarrow (jz \vdash T) kz \\
& \neg _ : \forall \{K T\} \{kz\} (jz : \text{Bwd } K) \rightarrow T / (kz ++ jz) \rightarrow (jz \vdash T) / kz \\
& jz \neg_R (t \uparrow \psi) \text{ with } jz \neg \psi \\
& jz \neg_R (t \uparrow (\theta ++ \leq \phi)) \mid !! \theta, \phi, \text{refl}, \text{refl} = (\phi \neg_R t) \uparrow \theta
\end{aligned}$$

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.

$$\begin{aligned}
& \text{data } \text{Va}_R \{K\} (k : K) : \overline{K} \text{ where only } : \text{Va}_R k (\square \neg, k) \\
& \text{va}_R : \forall \{K\} \{k\} \{kz : \text{Bwd } K\} \rightarrow k \leftarrow kz \rightarrow \text{Va}_R k / kz \\
& \text{va}_R x = \text{only } \uparrow x
\end{aligned}$$

Untyped λ -calculus. We can now give the lambda terms for which all *free* variables are relevant as follows. Converting de Bruijn to co-de-Bruijn representation is easy with smart constructors.

$$\begin{aligned}
& \text{data } \text{Lam}_R : \overline{\text{One}} \text{ where} & \text{lam}_R : \text{Lam} \rightarrow (\text{Lam}_R / _) \\
& \# : \text{Va}_R \langle \rangle & \rightarrow \text{Lam}_R & \text{lam}_R (\# x) = \text{map} / \# (\text{va}_R x) \\
& \text{app} : (\text{Lam}_R \times_R \text{Lam}_R) & \rightarrow \text{Lam}_R & \text{lam}_R (f \$ s) = \text{map} / \text{app} (\text{lam}_R f, \text{lam}_R s) \\
& \lambda : (\square \neg, \langle \rangle \vdash \text{Lam}_R) & \rightarrow \text{Lam}_R & \text{lam}_R (\lambda t) = \text{map} / \lambda (\neg \neg_R \text{lam}_R t)
\end{aligned}$$

E.g., compare de Bruijn terms for the \mathbb{K} and \mathbb{S} combinators with their co-de-Bruijn form.

$$\begin{aligned}
& \mathbb{K} \mathbb{S} : \text{Lam } \square \\
& \mathbb{K} = \lambda (\lambda (\# (\text{oe os o'}))) \\
& \text{lam}_R \mathbb{K} = \lambda (\text{oz os } \neg \lambda (\text{oz o'} \neg \# \text{only})) \uparrow \text{oz} \\
& \mathbb{S} = \lambda (\lambda (\lambda (\# (\text{oe os o'} o') \$ \# (\text{oe os}) \$ (\# (\text{oe os o'}) \$ \# (\text{oe os})))))) \\
& \text{lam}_R \mathbb{S} = \lambda (\text{oz os } \neg \lambda (\text{oz os } \neg \lambda (\text{oz os } \neg \\
& \quad \text{app (pair (app (pair (\# only } \uparrow \text{oz os o'}) (\# only } \uparrow \text{oz o' os}) (czz cs' c's)) } \uparrow \text{oz os o' os})} \\
& \quad \text{(app (pair (\# only } \uparrow \text{oz os o'}) (\# only } \uparrow \text{oz o' os}) (czz cs' c's)) } \uparrow \text{oz o' os os})} \\
& \quad \text{(czz cs' c's css)))))) \uparrow \text{oz}
\end{aligned}$$

Staring bravely, we can see that \mathbb{K} uses its first argument to deliver what is plainly a constant function: the second λ discards its argument, leaving only one variable in scope. Meanwhile, it is plain that \mathbb{S} uses all three inputs ('function', 'argument', 'environment'): in the subsequent 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 ( $I : \text{Set}$ ) : Set where inductive; constructor  $\_ \Rightarrow \_$ 
field scope : Bwd (Kind  $I$ ); sort :  $I$ 
```

Notice that **Kinds** offer higher-order abstraction: a bound variable itself has a **Kind**, being an object sort parametrized by a scope of bound variables, where the latter is, as in previous sections, a **Bwd** list, with the K parameter now fixed to be **Kind** I . Object variables have sorts; *meta*-variables have **Kinds**. For example, when we write the β -rule

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

the t and the s are not variables of the object calculus like x . They stand as placeholders, s for some term and $t[x]$ for some term with a parameter which can be instantiated, and is instantiated by x on the left and s on the right. The kind of t is $\square \neg, (\square \Rightarrow \langle \rangle) \Rightarrow \langle \rangle$.

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

```
data Desc ( $I : \text{Set}$ ) : Set1 where
  RecD : Kind  $I \rightarrow$  Desc  $I$ 
  OneD : Desc  $I$ 
   $\_ \times_D \_$  : Desc  $I \rightarrow$  Desc  $I \rightarrow$  Desc  $I$ 
   $\Sigma_D$  : ( $S : \text{Datoid}$ )  $\rightarrow$  (Data  $S \rightarrow$  Desc  $I$ )  $\rightarrow$  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 ( $X : \text{Set}$ ) : Set where
  yes :  $X \rightarrow$  Decide  $X$ 
  no : ( $X \rightarrow$  Zero)  $\rightarrow$  Decide  $X$ 

record Datoid : Set1 where
  field
    Data : Set
    decide : ( $x y : \text{Data}$ )  $\rightarrow$  Decide ( $x = y$ )
  open Datoid
```

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

```
data LamTag : Set where app  $\lambda$  : LamTag
LAMTAG : Datoid
Data LAMTAG = LamTag

LamD : One  $\rightarrow$  Desc One
LamD  $\langle \rangle$  =  $\Sigma_D$  LAMTAG  $\lambda$  { app  $\rightarrow$  RecD ( $\square \Rightarrow \langle \rangle$ )  $\times_D$  RecD ( $\square \Rightarrow \langle \rangle$ )
;  $\lambda \rightarrow$  RecD ( $\square \neg, (\square \Rightarrow \langle \rangle) \Rightarrow \langle \rangle$ ) }

decide LAMTAG app app = yes refl
decide LAMTAG app  $\lambda$  = no  $\lambda$  ()
decide LAMTAG  $\lambda$  app = no  $\lambda$  ()
decide LAMTAG  $\lambda$   $\lambda$  = yes refl
```

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**.

$$\begin{aligned}
\llbracket _ \mid _ \rrbracket &: \forall \{I\} \rightarrow \text{Desc } I \rightarrow (I \rightarrow \overline{\text{Kind } I}) \rightarrow \overline{\text{Kind } I} \\
\llbracket \text{Rec}_D k \mid R \rrbracket kz &= R (\text{sort } k) (kz ++ \text{scope } k) \\
\llbracket \Sigma_D ST \mid R \rrbracket kz &= \Sigma (\text{Data } S) \lambda s \rightarrow \llbracket T s \mid R \rrbracket kz \\
\llbracket \text{One}_D \mid R \rrbracket kz &= \text{One} \\
\llbracket S \times_D T \mid R \rrbracket kz &= \llbracket S \mid R \rrbracket kz \times \llbracket T \mid R \rrbracket kz
\end{aligned}$$

At use sites, higher-kinded variables must be instantiated, just like $t[x]$ in the β -rule example, above. We can compute from a given scope the **Description** of the spine of actual parameters required.

$$\begin{aligned}
\text{Sp}_D &: \forall \{I\} \rightarrow \text{Bwd } (\text{Kind } I) \rightarrow \text{Desc } I \\
\text{Sp}_D \quad [] &= \text{One}_D \\
\text{Sp}_D (kz \neg, k) &= \text{Sp}_D kz \times_D \text{Rec}_D k
\end{aligned}$$

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.

$$\begin{aligned}
\text{data Tm } \{I\} (D : I \rightarrow \text{Desc } I) (i : I) kz : \text{Set where} \quad & \text{-- Tm } D i : \overline{\text{Kind } I} \\
\text{\#\$} &: \forall \{jz\} \rightarrow (jz \Rightarrow i) \leftarrow kz \rightarrow \llbracket \text{Sp}_D jz \mid \text{Tm } D \rrbracket kz \rightarrow \text{Tm } D i kz \\
[-] &: \llbracket D i \mid \text{Tm } D \rrbracket kz \rightarrow \text{Tm } D i kz \\
\text{infixr 5 } \text{\#\$} &
\end{aligned}$$

Interpreting Desc as co-de-Bruijn Syntax. We may, of course, also interpret **Descriptions** in co-de-Bruijn style, enforcing that all variables in scope are relevant, and demanding that binding sites make clear which variables are to be used. Let us work in **Scope** $I \rightarrow \text{Set}$ where practical.

$$\begin{aligned}
\llbracket _ \mid _ \rrbracket_R &: \forall \{I\} \rightarrow \text{Desc } I \rightarrow (I \rightarrow \overline{\text{Kind } I}) \rightarrow \overline{\text{Kind } I} \\
\llbracket \text{Rec}_D k \mid R \rrbracket_R &= \text{scope } k \vdash R (\text{sort } k) \\
\llbracket \Sigma_D ST \mid R \rrbracket_R &= \lambda kz \rightarrow \Sigma (\text{Data } S) \lambda s \rightarrow \llbracket T s \mid R \rrbracket_R kz \\
\llbracket \text{One}_D \mid R \rrbracket_R &= \text{One}_R \\
\llbracket S \times_D T \mid R \rrbracket_R &= \llbracket S \mid R \rrbracket_R \times_R \llbracket T \mid R \rrbracket_R \\
\text{data Tm}_R \{I\} (D : I \rightarrow \text{Desc } I) (i : I) : \overline{\text{Kind } I} \text{ where} \\
\# &: \forall \{jz\} \rightarrow (\text{Va}_R (jz \Rightarrow i) \times_R \llbracket \text{Sp}_D jz \mid \text{Tm}_R D \rrbracket_R) \rightarrow \text{Tm}_R D i \\
[-] &: \llbracket D i \mid \text{Tm}_R D \rrbracket_R \rightarrow \text{Tm}_R D i
\end{aligned}$$

Let us compute co-de-Bruijn terms from de Bruijn terms, generically.

$$\begin{aligned}
\text{code} &: \forall \{I\} \{D : I \rightarrow \text{Desc } I\} \{i\} \rightarrow \text{Tm } D i \quad \rightarrow (\text{Tm}_R D i / _) \\
\text{codes} &: \forall \{I\} \{D : I \rightarrow \text{Desc } I\} S \rightarrow \llbracket S \mid \text{Tm } D \rrbracket \rightarrow (\llbracket S \mid \text{Tm}_R D \rrbracket_R / _) \\
\text{code} & \quad (\text{\#\$} \{jz\} x ts) = \text{map/ } \# \quad (\text{va}_R x ,_R \text{ codes } (\text{Sp}_D jz) ts) \\
\text{code } \{D = D\} \{i = i\} [ts] &= \text{map/ } [-] (\text{codes } (D i) ts) \\
\text{codes } (\text{Rec}_D (jz \Rightarrow i)) \quad t &= jz \setminus_R \text{code } t \\
\text{codes } (\Sigma_D ST) \quad (s , ts) &= \text{map/ } (s , _) (\text{codes } (T s) ts) \\
\text{codes } \text{One}_D &\langle \rangle = \langle \rangle_R \\
\text{codes } (S \times_D T) \quad (ss , ts) &= \text{codes } S ss ,_R \text{ codes } T ts
\end{aligned}$$

10 Hereditary Substitution for Co-de-Bruijn Metasyntax

Let us develop the appropriate notion of substitution for our metasyntax, which is *hereditary* in the sense of Watkins et al. [5], as substituting a higher-kinded variable requires us further to substitute its actual for its formal 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 ⊢ TmR 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 $\phi : iz \leq jz$ 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 : ∀ {I} {D : I → Desc I} {src trg bnd iz jz} →
  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})
  bindPassive [] = parti h
  bindPassive (iz ¬, _) = bindPassive iz cs'
```

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 : ∀ {I} {D : I → Desc I} {src src' trg bnd}
  (θ : 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 : ∀ {K} {iz' jz' kz kz' : Bwd K} {θ' : iz' ≤ kz'} {ϕ' : jz' ≤ kz'}
  (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 `hSubs/` proceed structurally, in accordance with a syntax description; `hered` invokes `hSub` hereditarily.

$\text{hSub} : \text{HSub } D \text{ src trg bnd} \rightarrow \text{Tm}_R D i \text{ src} \rightarrow \text{Tm}_R D i / \text{trg}$
 $\text{hSubs} : (S : \text{Desc } I) \rightarrow \text{HSub } D \text{ src trg bnd} \rightarrow \llbracket S \mid \text{Tm}_R D \rrbracket_R \text{ src} \rightarrow \llbracket S \mid \text{Tm}_R D \rrbracket_R / \text{trg}$
 $\text{hSubs}_/ : (S : \text{Desc } I) \rightarrow \text{HSub } D \text{ src trg bnd} \rightarrow \llbracket S \mid \text{Tm}_R D \rrbracket_R / \text{src} \rightarrow \llbracket S \mid \text{Tm}_R D \rrbracket_R / \text{trg}$
 $\text{hered} : (jz \vdash \text{Tm}_R D i) / \text{trg} \rightarrow \llbracket -, (jz \Rightarrow i) \leq \text{bnd} \rrbracket \rightarrow \llbracket \text{Sp}_D jz \mid \text{Tm}_R D \rrbracket_R / \text{trg} \rightarrow \text{Tm}_R D i / \text{trg}$

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

$\text{hSub } \{D = D\} \{i = i\} h [ts] = \text{map}/ [-] (\text{hSubs } (D i) h ts)$
 $\text{hSub } h (\# \{jz\} (\text{pair } (\text{only } \uparrow \theta) ts _)) \text{ with } \text{selHSub } \theta h \mid \text{hSubs}_/ (\text{Sp}_D jz) h ts$
 $\dots \mid \text{record } \{ \text{parti} = _ \text{css } \{ \text{both} = () \} _ \} \mid ts'$
 $\dots \mid \text{record } \{ \text{parti} = \text{czz } cs'; \text{passTrg} = \phi \} \mid ts' = \text{map}/ \# (\text{va}_R \phi, ts')$
 $\dots \mid \text{record } \{ \text{parti} = \text{czz } c's; \text{actBnd} = \theta'; \text{images} = \llbracket -, im \rrbracket \} \mid ts' = \text{hered } im \theta' ts'$

To substitute a variable hereditarily, find it in the bounding scope, for the scope of its kind becomes the new and *structurally smaller* bound. Helper function part computes the partition with free variables passive and bound variables active, while spAll converts the spine of actual parameters into the environment of images for the bound variables.

$\text{hered} \quad im \quad (\theta' o') ts' = \text{hered } im \theta' ts'$
 $\text{hered } \{D = D\} \{trg = trg\} ((\phi \searrow t) \uparrow \psi) (\theta' os) ts' = \text{let } !!c = \text{part } _ \text{ in}$
 $\quad \text{hSub } (\text{record } \{ \text{parti} = c; \text{actBnd} = \phi; \text{images} = \phi \leq ? \text{spAll } ts'; \text{passTrg} = \psi \}) t \text{ where}$
 $\quad \text{spAll} : \forall \{kz\} \rightarrow \llbracket \text{Sp}_D kz \mid \text{Tm}_R D \rrbracket_R / \text{trg} \rightarrow \text{All } _ kz$
 $\quad \text{part} : \forall kz iz \rightarrow \Sigma (kz \leq (kz ++ iz)) \lambda \theta \rightarrow \Sigma (iz \leq (kz ++ iz)) \lambda \theta' \rightarrow \text{Cover } \text{ff } \theta \theta'$

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.

$\text{hSubs } (\text{Rec}_D k) h (\phi \searrow t) = \text{scope } k \searrow_R \text{hSub } (\text{wkHSub } h \phi) t$
 $\text{hSubs } (\Sigma_D S T) h (s, ts) = \text{map}/ (s, _) (\text{hSubs } (T s) h ts)$
 $\text{hSubs } \text{One}_D \quad h \langle \rangle = \langle \rangle_R$
 $\text{hSubs } (S \times_D T) h (\text{pair } s t _) = \text{hSubs}_/ S h s, \text{hSubs}_/ T h t$
 $\text{hSubs}_/ S h' (ts \uparrow \theta) \text{ with } \text{selHSub } \theta h'$
 $\text{hSubs}_/ S h' (ts \uparrow \theta) \mid \text{record } \{ \text{parti} = c; \text{images} = \llbracket _, \text{passTrg} = \phi \rrbracket \} \text{ rewrite } \text{allLeft } c = ts \uparrow \phi$
 $\text{hSubs}_/ S h' (ts \uparrow \theta) \mid h = \text{hSubs } S h ts$

References

- [1] Nicolas G. de Bruijn (1972): *Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation*. *Indagationes Mathematicæ* 34, pp. 381–392.
- [2] Jeremy Gibbons (2017): *APLlicative 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, doi:10.1007/978-3-662-54434-1_21. Available at https://doi.org/10.1007/978-3-662-54434-1_21.

- [3] Masahiko Sato, Randy Pollack, Helmut Schwichtenberg & Takafumi Sakurai (2013): *Viewing λ -terms through Maps*. *Indagationes Mathematicæ* 24(4).
- [4] 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, doi:10.1145/99370.99404. Available at <http://doi.acm.org/10.1145/99370.99404>.
- [5] Kevin Watkins, Ilario 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, doi:10.1007/978-3-540-24849-1_23. Available at https://doi.org/10.1007/978-3-540-24849-1_23.