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 the means to construct tuples. The empty tuple is given by the *record* type with no fields, which Agda recognizes as uniquely inhabited. Dependent pairing is by means of the Sg type, abbreviated by $*$ in its non-dependent special case.

```
record One : Set where constructor ⟨⟩
```

```
record Σ (S : Set) (T : S → Set) : Set where
  constructor _,_
  field fst : S; snd : T fst
open Σ public
_×_ : 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.

Submitted to:
MSFP 2018

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 **OPE** for the category.)

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

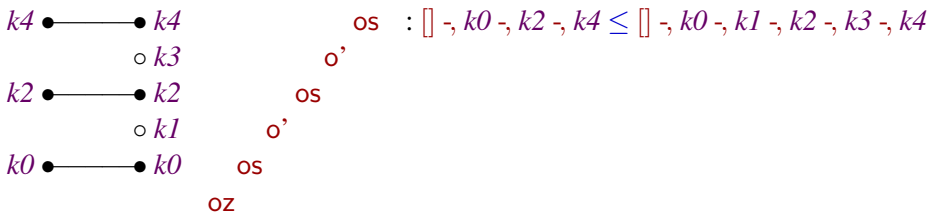
I use backward lists, because it is traditional to write contexts to the left of judgements in typing rules and extend them on the right. However, I use the word 'scope' rather than 'context' to suggest that we are characterizing at least which variables we may refer to, but perhaps not the full type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

The morphisms of **OPE** give an embedding from a source scope to a target scope. Colloquially, we may call such embeddings 'thinnings', as they dilute with 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 $(f;g) \in \mathbb{C}(R,T)$.

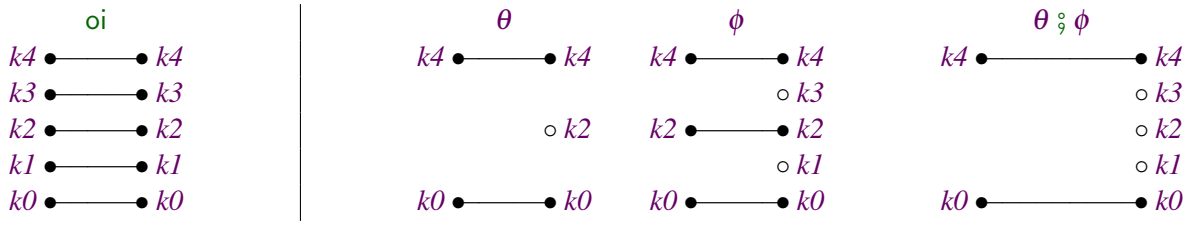
For example, every scope has the identity thinning, oi, and thinnings compose via ⨟. (Presentations of composition vary: for functions, it is much more common to write $g \cdot f$ for 'g *after* f' than $f;g$ for 'f

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

```
oi : ∀ {K} {kz : Bwd K} →
     kz ≤ kz
oi {kz = iz -, k} = oi os   -- os preserves oi
oi {kz = []}      = oz
```

```
_⨟_ : ∀ {K} {iz jz kz : Bwd K} →
      iz ≤ jz → jz ≤ kz → iz ≤ kz
θ     ⨟ ϕ o'  = (θ ⨟ ϕ) o'
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.

```
law−oi⨟  : ∀ {K} {iz jz : Bwd K}       (θ : iz ≤ jz) → oi ⨟ θ == θ
law−⨟oi  : ∀ {K} {iz jz : Bwd K}       (θ : iz ≤ jz) → θ ⨟ oi == θ
law−⨟⨟   : ∀ {K} {iz jz kz lz : Bwd K} (θ : iz ≤ jz) (ϕ : jz ≤ kz) (ψ : kz ≤ lz) →
                                        θ ⨟ (ϕ ⨟ ψ) == (θ ⨟ ϕ) ⨟ ψ
```

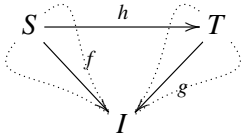
As one might expect, order-preserving embeddings have a strong antisymmetry property that one cannot expect of categories in general. The *only* invertible arrows are the identities. Note that we must match on the proof of $iz =\!= jz$ even to claim that θ and ϕ are the identity.

```
antisym : ∀ {K} {iz jz : Bwd K} (θ : iz ≤ jz) (ϕ : jz ≤ iz) →
          Σ (iz == jz) λ {refl → θ == oi × ϕ == oi}
```

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 $f : S \to I$ 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.)



That is, the morphisms are *triangles*. A seasoned dependently typed programmer will be nervous at the sight of a definition like

```
_→/_ : ∀ {K} {iz jz kz : Bwd K} → (iz ≤ kz) → (jz ≤ kz) → Set
θ →/ ϕ = Σ _ λ ψ → (ψ ⨟ ϕ) ≡ θ
```

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 ⨟ 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 ⨟.

```
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 ⨟ thus holds.

```
tri  : ∀ {K} {iz jz kz : Bwd K} (θ : iz ≤ jz) (ϕ : jz ≤ kz) → Tri θ ϕ (θ ⨟ ϕ)
comp : ∀ {K} {iz jz kz : Bwd K} {θ : iz ≤ jz} {ϕ : jz ≤ kz} {ψ : iz ≤ kz} →
       Tri θ ϕ ψ → ψ ≡ (θ ⨟ ϕ)
```

The proofs are easy inductions.

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

```
egTri : ∀ {K} {k0 k1 k2 k3 k4 : K} → 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. (The Σ _ tells Agda to infer the type of ψ, which is forced by its use as an index of Tri.)

```
_→/_ : ∀ {K} {iz jz kz : Bwd K} → (iz ≤ kz) → (jz ≤ kz) → Set
ψ →/ ϕ = Σ _ λ θ → Tri θ ϕ ψ
```

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!

```
tri1 : ∀ {K} {iz jz kz : Bwd K} {θ θ' : iz ≤ jz} {ϕ : jz ≤ kz} {ψ : iz ≤ kz} →
       Tri θ ϕ ψ → Tri θ' ϕ ψ → θ ≡ θ'
```

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 *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 $F_m(\imath_X) = \imath_{F_o(X)}$ 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).

For example, every $k : K$ induces a functor from **OPE** to itself which acts by scope extension, $(\_ \text{-}, k)$ 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 $\mathsf{Set}(S,T)$ 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.

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 : Set, we may then take $I \to$ Set to be the category whose objects are *families* of objects in Set, $S, T : I \to$ Set with morphisms being the corresponding (implicitly indexed) families of morphisms in $\{i : I\} \to S\,i \to T\,i$. Let us abbreviate:

```
_ -:> _ : {I : Set} (S T : I → Set) → Set
S -:> T = ∀ {i} → S i → T i
```

Consider morphisms equal if they map each index to pointwise equal functions. We may define a functor from $K \to$ Set to Bwd $K \to$ 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-repecting* operations on environments. Identity and composition are readily preserved. In the sequel, it will be convenient to abbreviate Bwd $K \to$ 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 $K \to$ 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:

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

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

```
_≤?_ : ∀ {K P} {jz iz : Bwd K} → iz ≤ jz → All P jz → All P iz
oz      ≤? []       = []
(θ os) ≤? (pz -, p) = (θ ≤? pz) -, p
(θ o') ≤? (pz -, p) = θ ≤? pz
```

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 $\mathbb{C}^{\mathrm{op}}$ 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 $\lambda\ P \to$ All $P\ kz$ is a functor (with action all) from $K \to$ Set to Set. Accordingly, if $\theta : iz \leq jz$ then $(\theta \leq ?_)$is a natural transformation from $\lambda\ P \to$ All $P\ jz$ to $\lambda\ P \to$ All $P\ iz$, which is as much as to say that the definition of $\leq ?$ is uniform in P, and hence that if $f : \{k : K\} \to P\ k \to Q\ k$, then all $f\ (\theta \leq ?\ pz) = \theta \leq ?$ 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].

```
_←_ : ∀ {K} → K → K̄
k ← kz = [] -, k ≤ kz

data Lam (iz : Bwd One) : Set where                -- Lam : One̅
  #   : (x : ⟨⟩ ← iz) →      Lam iz                -- finds a variable
  _$_ : (f s : Lam iz) →     Lam iz                -- associates left
  λ   : (t : Lam (iz -, ⟨⟩)) → Lam iz               -- binds a variable
```

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.

```
_↑_ : ∀ {iz jz} → Lam iz → iz ≤ jz → Lam jz
# i     ↑ θ = # (i ; θ)
(f $ s) ↑ θ = (f ↑ θ) $ (s ↑ θ)
λ t     ↑ θ = λ (t ↑ θ os)
```

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.

```
record _/_ {K} (T : K̄) (scope : Bwd K) : Set where    -- (T /_) : K̄
  constructor _↑_
  field {support} : Bwd K; thing : T support; thinning : support ≤ scope
```

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

```
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' and 'multiplication'

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

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.

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

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.

```
unit/ : ∀ {K} {T : K̄} → T -:> (T /_)      mult/ : ∀ {K} {T : K̄} → ((T /_) /_) -:> (T /_)
unit/ t = t ↑ oi                          mult/ ((t ↑ θ) ↑ ϕ) = t ↑ (θ ; ϕ)
```

The proof obligations to make (/, unit/, 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.

```
thin/ : ∀ {K T} {iz jz : Bwd K} → T / iz → iz ≤ jz → T / jz
thin/ t θ = mult/ (t ↑ θ)
```

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

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

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

The Kleisli arrows for $/$ are operations with types such as $S \mathrel{-\!\!\!\rightarrow} (T\ /\_)$ 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 $\mathbf{OPE}/\mathit{scope}$.

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:

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

law−oe : ∀ {K} {kz : Bwd K}
         (θ : [] ≤ kz) → θ == oe
```

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

```
oe/ : ∀ {K} {iz kz : Bwd K} (θ : iz ≤ kz) → oe →/ θ
oe/ θ with tri oe θ
... | t rewrite law−oe (oe ; θ) = oe , t
```

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 One_R {K} : ‾K where ⟨⟩ : One_R []   -- 'R' for relevant
⟨⟩_R : ∀ {K} {kz : Bwd K} → One_R / kz
⟨⟩_R = ⟨⟩ ↑ 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 $r \in \mathbb{C}(S, S+T)$ 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!

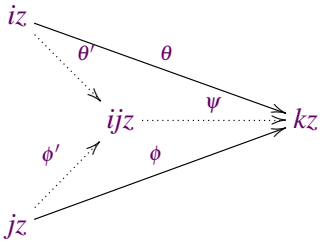
Curiously, though, **OPE** does *not* have coproducts. Taking $K = \mathsf{One}$, let us try to construct the coproduct of two singleton scopes, $S = T = [] \text{-}, \langle\rangle$. We can begin one coproduct diagram by taking $U = [] \text{-}, \langle\rangle$ and $f = g = \mathsf{oi}$: that forces the issue, as our only candidate for $S+T$ is once again the singleton $[] \text{-}, \langle\rangle$, with $l = r = \mathsf{oi}$, making $h = \mathsf{oi}$; 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 $U' = [] \text{-}, \langle\rangle \text{-}, \langle\rangle$ with two variables, allowing us to choose $f' = \mathsf{oz\ os\ o'}$ and $g' = \mathsf{oz\ o'\ os}$, 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 $\mathsf{Tri}\ \theta'\ \psi\ \theta$ and $\mathsf{Tri}\ \phi'\ \psi\ \phi$, 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. Let us define this notion of covering inductively, so we build diagrams and reason by pattern matching.

```
data Cover {K} : {iz jz ijz : Bwd K} → iz ≤ ijz → jz ≤ ijz → Set where
  _c's  : ∀ {iz jz ijz k} {θ : iz ≤ ijz} {ϕ : jz ≤ ijz} →
          Cover θ ϕ → Cover {ijz = _ -, k} (θ o') (ϕ os)
  _cs'  : ∀ {iz jz ijz k} {θ : iz ≤ ijz} {ϕ : jz ≤ ijz} →
          Cover θ ϕ → Cover {ijz = _ -, k} (θ os) (ϕ o')
  _css  : ∀ {iz jz ijz k} {θ : iz ≤ ijz} {ϕ : jz ≤ ijz} →
          Cover θ ϕ → Cover {ijz = _ -, k} (θ os) (ϕ os)
  czz   : Cover 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 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 : ∀ {K} {kz iz jz ijz : Bwd K}
       {θ : iz ≤ kz} {ϕ : jz ≤ kz} {ψ : ijz ≤ kz} {θ' : iz ≤ ijz} {ϕ' : jz ≤ ijz} →
       Tri θ' ψ θ → Cover 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 the edge ψ' and inversion of the coproduct construction. The payoff from this construction is the notion of *relevant pair*:

```
record _×R_ {K} (S T : K̄) (ijz : Bwd K) : Set where    -- S ×R T : K̄
  constructor pair
  field outl : S / ijz; outr : T / ijz; cover : Cover (thinning outl) (thinning outr)
_,R_ : ∀ {K} {S T : K̄} {kz} → 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

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.

```
_++_ : ∀ {K} (kz jz : Bwd K) → Bwd K
kz ++ []      = kz
kz ++ (iz -, j) = (kz ++ iz) -, j
```

```
_++≤_ : ∀ {K} {iz jz iz' jz' : Bwd K} →
  iz ≤ jz → iz' ≤ jz' → (iz ++ iz') ≤ (jz ++ jz')
θ ++≤   oz   = θ
θ ++≤ (ϕ os) = (θ ++≤ ϕ) os
θ ++≤ (ϕ o') = (θ ++≤ ϕ) o'
```

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

```
_⊣_ : ∀ {K} {iz kz} (jz : Bwd K) (ψ : iz ≤ (kz ++ jz)) →
  Σ (Bwd K) λ kz' → Σ (Bwd K) λ jz' → Σ (kz' ≤ kz) λ θ → Σ (jz' ≤ jz) λ ϕ →
  Σ (iz == (kz' ++ jz')) λ {refl → ψ == (θ ++≤ ϕ)}
[]       ⊣ ψ                                = !! ψ , oz , refl , refl
(kz -, k) ⊣ (ψ os)          with kz ⊣ ψ
(kz -, k) ⊣ (. (θ ++≤ ϕ) os) | !! θ , ϕ , refl , refl = !! θ , ϕ os , refl , refl
(kz -, k) ⊣ (ψ o')          with kz ⊣ ψ
(kz -, k) ⊣ (. (θ ++≤ ϕ) o') | !! θ , ϕ , refl , refl = !! θ , ϕ o' , refl , refl
```

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.

```
data _⊢_ {K} (jz : Bwd K) (T : K̄) (kz : Bwd K) : Set where    -- jz ⊢ T : K̄
  _\\_ : ∀ {iz} → iz ≤ jz → T (kz ++ iz) → (jz ⊢ T) kz
_\\R_ : ∀ {K T} {kz} (jz : Bwd K) → T / (kz ++ jz) → (jz ⊢ T) / kz
jz \\R (t ↑ ψ) with jz ⊣ ψ
jz \\R (t ↑ . (θ ++≤ ϕ)) | !! θ , ϕ , refl , refl = (ϕ \\ t) ↑ θ
```

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.

```
data VaR {K} (k : K) : K̄ where only : VaR k ([] -, k)
varR : ∀ {K} {k} {kz : Bwd K} → k ← kz → VaR k / kz
varR x = only ↑ x
```

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.

```
data LamR : One̅ where
  #   : VaR ⟨⟩              -→ LamR
  app : (LamR ×R LamR)      -→ LamR
  λ   : ([] -, ⟨⟩ ⊢ LamR)    -→ LamR

lamR : Lam -→ (LamR /_)
lamR (# x)   = map/ #   (varR x)
lamR (f $ s) = map/ app (lamR f ,R lamR s)
lamR (λ t)   = map/ λ   (_ \\R lamR t)
```

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

```
𝕂 𝕊 : Lam []
𝕂      = λ (λ (# (oe os o')))
lamR 𝕂 = λ (oz os \\ λ (oz o' \\ # only)) ↑ oz
𝕊      = λ (λ (λ (# (oe os o' o') $ # (oe os) $ (# (oe os o') $ # (oe os)))))
lamR 𝕊 = λ (oz os \\ λ (oz os \\ λ (oz os \\
  app (pair (app (pair (# only ↑ oz os o') (# only ↑ oz o' os) (czz cs' c's)) ↑ oz os o' os)
            (app (pair (# only ↑ oz os o') (# only ↑ oz o' os) (czz cs' c's)) ↑ oz o' os os)
            (czz cs' c's css))))) ↑ oz
```

Staring bravely, we can see that 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 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.

```
data Kind (I : Set) : Set where _⇒_ : Bwd (Kind I) → I → Kind 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 $[] \text{-,} ([] \Rightarrow \langle\rangle) \Rightarrow \langle\rangle$.

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

```
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
  yes : X →              Decide X
  no  : (X → Zero) →     Decide X
```

```
record Datoid : Set₁ where
  field
    Data   : Set
    decide : (x y : Data) → Decide (x == y)
open Datoid
```

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

```
data LamTag : Set where app λ : LamTag

LAMTAG : Datoid
Data LAMTAG = LamTag

decide LAMTAG app app = yes refl
decide LAMTAG app λ   = no λ ()
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.

```
⟦_|_⟧ : ∀ {I} → Desc I → (I → Kind I) → Kind I
⟦ Rec_D (jz ⇒ i) | R ⟧ kz = R i (kz ++ jz)
⟦ Σ_D S T        | R ⟧ kz = Σ (Data S) λ s → ⟦ T s | R ⟧ kz
⟦ One_D          | R ⟧ kz = One
⟦ S ×_D T        | R ⟧ kz = ⟦ S | R ⟧ kz × ⟦ T | R ⟧ kz
```

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.

```
Sp_D : ∀ {I} → Bwd (Kind I) → Desc I
Sp_D []      =        One_D
Sp_D (kz -, k) = Sp_D kz ×_D Rec_D k
```

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.

```
data Tm {I} (D : I → Desc I) (i : I) (kz : Bwd (Kind I)) : Set where   -- Tm D i : Kind I
  _#$_ : ∀ {jz} → (jz ⇒ i) ← kz → ⟦ Sp_D jz | Tm D ⟧ kz → Tm D i kz
  [_]  :                           ⟦ D i     | Tm D ⟧ kz → Tm D i kz
infixr 5 _#$_
```

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 → Set where practical.

```
⟦_|_⟧_R : ∀ {I} → Desc I → (I → Kind I) → Kind I
⟦ Rec_D (jz ⇒ i) | R ⟧_R = jz ⊢ R i
⟦ Σ_D S T        | R ⟧_R = λ kz → Σ (Data S) λ s → ⟦ T s | R ⟧_R kz
⟦ One_D          | R ⟧_R = One_R
⟦ S ×_D T        | R ⟧_R = ⟦ S | R ⟧_R ×_R ⟦ T | R ⟧_R

data Tm_R {I} (D : I → Desc I) (i : I) : Kind I where
  #   : ∀ {jz} → (Va_R (jz ⇒ i) ×_R ⟦ Sp_D jz | Tm_R D ⟧_R) -:> Tm_R D i
  [_] :                           ⟦ D i     | Tm_R D ⟧_R  -:> Tm_R D i
```

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

```
code  : ∀ {I} {D : I → Desc I} {i} → Tm D i       -:> (Tm_R D i /_)
codes : ∀ {I} {D : I → Desc I} S  → ⟦ S | Tm D ⟧ -:> (⟦ S | Tm_R D ⟧_R /_)
code          (_#$_ {jz} x ts) = map/ #   (va_R x ,_R codes (Sp_D jz) ts)
code {D = D} {i = i} [ ts ]    = map/ [_] (codes (D i) ts)
codes (Rec_D (jz ⇒ i)) t       = jz \\_R code t
codes (Σ_D S T)        (s , ts) = map/ (s ,_) (codes (T s) ts)
codes One_D            ⟨⟩      = ⟨⟩_R
codes (S ×_D T)        (ss , ts) = codes S ss ,_R codes T 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): *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, 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`.