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`

The key to any nameless representation of syntax is how it indicates the variables we choose to use and thus, implicitly, those we discard. Standard de Bruijn representations delay discarding *maximally* till the *leaves* of terms where one is chosen from the variables in scope at the expense of the rest. Consequently, introducing new but unused variables requires term traversal. This paper introduces a nameless '*co*-de-Bruijn' representation which makes the opposite canonical choice, delaying discarding *minimally*, as near as possible to the *root*. It is literate Agda: dependent types make it a practical joy to express and be driven by strong intrinsic invariants which ensure that scope is aggressively whittled down to just the *support* of each subterm, in which every remaining variable occurs somewhere. The construction is generic, delivering a *universe* of syntaxes with higher-order *meta*variables, for which the appropriate notion of substitution is *hereditary*. The implementation of simultaneous substitution exploits tight scope control to avoid busywork and shift terms without traversal. Surprisingly, it is also intrinsically terminating, by structural recursion alone.

When I was sixteen and too clever by half, I wrote a text editor which cached a plethora of useful but redundant pointers into the buffer, just to shave a handful of nanoseconds off redisplay. Accurately updating these pointers at each keystroke was a challenge which taught me the hard way about the value of simplicity. Now, I am a dependently typed programmer. I do not keep invariants: invariants keep me.

This paper is about scope invariants in nameless representations of syntax. One motivation for such is eliminating redundant name choice to make α-equivalence trivial. Classic de Bruijn syntaxes [9] replace name by number: variable uses count either out from use to binding (*indices*), or in from root to binding (*levels*). Uses are found at the leaves of syntax trees, so any operation which modifies the sequence of variables in scope requires traversal. E.g., consider this β-reduction (under λx) in untyped λ-calculus.

$$\begin{array}{rlcl}
\text{name} & \lambda x.\,(\lambda y.\,y\,x\,(\lambda z.\,z\,(y\,z)))\,\underline{(x\,(\lambda v.\,v))} & \leadsto_\beta & \lambda x.\,\underline{(x\,(\lambda v.\,v))}\,x\,(\lambda z.\,z\,(\underline{(x\,(\lambda v.\,v))}\,z)) \\
\text{index} & \lambda\ .(\lambda\ .0\,1\,(\lambda\ .0\,(1\,0)))\,\underline{(0\,(\lambda\ .0))} & \leadsto_\beta & \lambda\ .\underline{(0\,(\lambda\ .0))}\,0\,(\lambda\ .0\,(\underline{(1\,(\lambda\ .0))}\,0)) \\
\text{level} & \lambda\ .(\lambda\ .1\,0\,(\lambda\ .2\,(1\,2)))\,\underline{(0\,(\lambda\ .1))} & \leadsto_\beta & \lambda\ .\underline{(0\,(\lambda\ .1))}\,0\,(\lambda\ .1\,(\underline{(0\,(\lambda\ .2))}\,1))
\end{array}$$

Underlining shows the movement of the substituted term. In the *index* representation, the free x must be shifted when it goes under the λz. With *levels*, the free x stays 0, but the bound v must be shifted under λz, and the substitution context must be shifted to account for the eliminated λy. Shift happens.

The objective of this paper is not to eliminate shifts altogether, but to ensure that they do not require traversal. The approach is to track exactly which variables are *relevant* at all nodes in the tree and aggressively expel those unused in any given subtree. As we do so, we need and obtain much richer accountancy of variable usage, with much more intricate invariants. Category theory guides the design of these invariants and Agda's dependent types [20] drive their correct implementation.

My explorations follow Sato, Pollack, Schwichtenberg and Sakurai, whose λ-terms make binding sites carry *maps* of use sites [21]. E.g., the $\mathbb{K}$ and $\mathbb{S}$ combinators become (respectively)

$$\begin{array}{rcccc}
\text{names} & \lambda c.\,\lambda e.\,c & \lambda f. & \lambda s. & \lambda e.\ (f\,e)\ \ (s\,e) \\
\text{maps} & \texttt{1\textbackslash\ 0\textbackslash}\square & \texttt{((10)\ (00))\textbackslash} & \texttt{((00)\ (10))\textbackslash} & \texttt{((01)\ (01))\textbackslash}(\square\square)(\square\square)
\end{array}$$

where each abstraction shows with `1`s where in the subsequent tree of applications its variable occurs: leaves, □, are relieved of choice. Of course, the tree under each binder determines which maps are well formed in a highly nonlocal way: these invariants are formalised *extrinsically* both in Isabelle/HOL and in Minlog, over a context-free datatype enforcing neither scope nor shape. Other prior art along similar lines includes the Haskell implementation by Abel and Kraus [2] of a similar representation, recording at each λ which of the variable occurrences free below it are bound by it, in left-to-right order, run-length encoded. Earlier still, the *director strings* representation of Kennedy and Sleep, refined by Sinot, Fernandez and Mackie [16, 22], annotated each node with a mapping from each free variable to the set of indices of the subnodes in which it occurs, and we shall see something similar here.

However, in this paper, we shall obtain an *intrinsically* valid representation, enforced by type, where the map information is localized. Binding sites tell only if the variable is used; the crucial choice points where a term comprises more than one subterm say which variables go where, as in the director strings representation. Not all are used in all subterms, but (as Eccles says to Seagoon) *everybody's got to be somewhere* [19]: variables used nowhere have been discarded already. This property is delivered by a coproduct construction in the slices of the category of order-preserving embeddings, but fear not: we shall revisit all of the category theory required to develop the definition, especially as it strays beyond the familiar (e.g., to Haskellers) territory of types-and-functions.

Intrinsically well scoped de Bruijn terms date back to Bellegarde and Hook [6], using `option` types to grow a type of free variables, but hampered by lack of polymorphic recursion in ML. Substitution (i.e., *monadic* structure) was developed for untyped terms by Bird and Paterson [8] and for simple types by Altenkirch and Reus [5], both dependent either on a *prior* implementation of renumbering shifts (i.e., functorial structure) or a non-structural recursion. My thesis [17] follows McKinna and Goguen [12] in restoring a single structural operation abstracting 'action' on variables, instantiated to renumbering then to substitution, an approach subsequently adopted by Benton, Kennedy and Hur [7] and generalised to semantic actions by Allais et al. [3]. Here, we go directly to substitution: *shifts need no traversal*.

I present not only λ-calculus but a *universe* of syntaxes inspired by Harper, Honsell and Plotkin's Logical Framework [13]. I lift the *sorts* of a syntax to higher *kinds*, acquiring both binding (via subterms at higher kind) and *meta*variables (at higher kind). However, substituting a higher-kinded variable demands substitution of its parameters *hereditarily* [23] and *simultaneously*. Thereby hangs a tale. Abel showed how *sized types* justify this process's apparently non-structural recursion in MSFP 2006 [1]. As editor, I anonymised a discussion with a referee which yielded a structural recursion for hereditary substitution of a *single* variable, instigating Keller and Altenkirch's formalization at MSFP 2010 [15]. Here, at last, simultaneous hereditary substitution becomes structurally recursive.

1 Basic Equipment in Agda

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'.

```
data   Zero : Set where                        │  Tt : Two → Set
record One  : Set where constructor ⟨⟩         │  Tt tt = One
data   Two  : Set where tt ff : Two            │  Tt ff = Zero
```

Dependent pairing is by means of the Σ type, abbreviated by × when non-dependent. The *pattern synonym* !_ allows the first component to be determined by the second: making it a right-associative prefix operator lets us write ! ! *expression* rather than ! (! (*expression*)).

```
record Σ (S : Set) (T : S → Set) : Set where        _×_ : Set → Set → Set
  constructor _,_                                    S × T = Σ S λ _ → T
  field fst : S;   snd : T fst                       pattern !_ t = _ , t
```

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

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

2 Δ_+^K: The (Semi-Simplicial) 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 just the study of compositionality — for everything, not just sets-and-functions. Here, we have an opportunity to develop categorical structure away from the usual apparatus for programming with functions. Let us therefore revisit the basics.

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

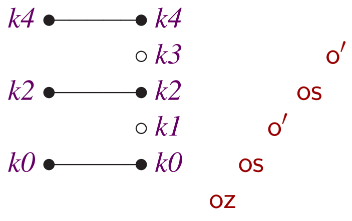
The rest will follow, but let us fix these notions for our example category, Δ_+^K, of *order-preserving embeddings* between variable *scopes*. Objects are given as backward (or 'snoc') lists of the *kinds*, K, of variables. (I habitually suppress K and just write Δ_+ for the category.) Backward lists respect the tradition of writing contexts left of judgements in rules and growing them rightwards. However, I say 'scope' rather than 'context': we track variable availability, but perhaps not all contextual data. Moreover, I take the typesetting liberty of hiding inferable prefixes of implicit quantifiers, to reduce clutter.

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

The morphisms, $iz \sqsubseteq jz$, of Δ_+ embed a source into a target scope. Colloquially, we may call them 'thinnings', as they dilute the variables of the source scope with more. I write step constructors postfix, so thinnings (like scopes) grow on the right. Now, where I give myself away as a type theorist is that I do not consider the notion of 'morphism' to make sense without prior source and target objects. The type $iz \sqsubseteq jz$ (which is a little more mnemonic than $\Delta_+(iz,jz)$) is the type of 'thinnings from iz to jz': there is no type of 'thinnings' *per se*.

Altenkirch, Hofmann and Streicher [4], from whom I learned this notion, take the dual view of morphisms as *selecting* one subcontext from another. When K = One, objects represent numbers and $\sqsubseteq$ generates Pascal's Triangle; excluding the empty scope and allowing *degenerate* (non-injective) maps yields Δ, the *simplex* category beloved of topologists.

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



$\mathsf{os} : [] \text{-,}\ k0\ \text{-,}\ k2\ \text{-,}\ k4 \sqsubseteq [] \text{-,}\ k0\ \text{-,}\ k1\ \text{-,}\ k2\ \text{-,}\ k3\ \text{-,}\ k4$

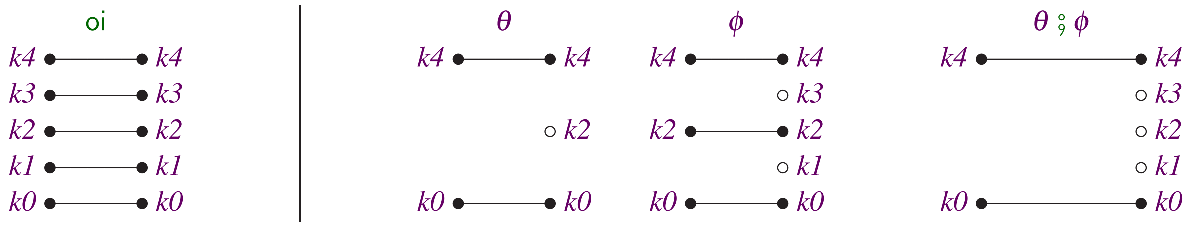
Category (II): Identity and Composition. In any category, certain morphisms must exist. Each object $X \in |\mathbb{C}|$ has an *identity* $\iota_X \in \mathbb{C}(X,X)$, and wherever the target of one morphism meets the source of another, their *composite* makes a direct path: if $f \in \mathbb{C}(R,S)$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

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

```
oi : kz ⊑ kz
oi {kz = iz -, k} = oi os    -- os preserves oi
oi {kz = []}      = oz
```

```
_⨾_ : 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 $\equiv$ will serve. With this definition in place, we may then state the laws. I omit the proofs, which go by functional induction.

```
law-oi⨾ : oi ⨾ θ ≡ θ      law-⨾oi : θ ⨾ oi ≡ θ      law-⨾⨾ : 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 $iz \equiv jz$ even to claim that θ and ϕ are the identity.

```
antisym : (θ : iz ⊑ jz) (ϕ : jz ⊑ iz) → Σ (iz ≡ jz) λ {refl → (θ ≡ oi) × (ϕ ≡ oi)}
```

Example: de Bruijn Syntax via $\Delta_{+}^{\mathsf{One}}$. De Bruijn indices are numbers [9], perhaps a type-enforced bound [6, 8, 5]. Singleton thinnings, k ← kz = [] -, k ⊑ kz, can play this rôle in a syntax.

```
data Lam (iz : Bwd One) : Set where          _↑_ : Lam iz → iz ⊑ jz → Lam jz
  #   : (x : ⟨⟩ ← iz) →         Lam iz        # i     ↑ θ = # (i ⨾ θ)
  _$_ : (f s : Lam iz) →        Lam iz        (f $ s) ↑ θ = (f ↑ θ) $ (s ↑ θ)
  λ   : (t : Lam (iz -, ⟨⟩)) → Lam iz        λ t     ↑ θ = λ (t ↑ θ os)
```

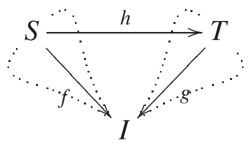
Variables are represented by pointing, eliminating redundant choice of names, but it is only when we point to one variable that we exclude the others. Thus de Bruijn indexing effectively uses thinnings to discard unwanted variables as *late* as possible, in the *leaves* of syntax trees.

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

3 Slices of Thinnings

If we fix the target of thinnings, (_⊑ 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 $(\mathbb{C}/I)((S,f),(T,g))$ is some $h : S \to T$ such that $f = h;g$. (The dotted regions in the diagram show the objects in the slice.)


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

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

```
ψ →/ φ = Σ _ λ θ → (θ ⨾ φ) ≡ ψ    -- beware of ⨾!
```

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

```
data Tri : iz ⊑ jz → jz ⊑ kz → iz ⊑ kz → Set where    tri  : (θ : iz ⊑ jz) (φ : jz ⊑ kz) →
  _t-″  : Tri θ φ ψ → Tri θ      (φ o′) (ψ o′)              Tri θ φ (θ ⨾ φ)
  _t′s′ : Tri θ φ ψ → Tri (θ o′) (φ os) (ψ o′)
  _tsss : Tri θ φ ψ → Tri (θ os) (φ os) (ψ os)        comp : Tri θ φ ψ → ψ ≡ (θ ⨾ φ)
  tzzz  : Tri                 oz     oz     oz
```

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

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

```
egTri : Tri {kz = [] -, k0 -, k1 -, k2 -, k3 -, k4} (oz os o′ os) (oz os o′ os o′ os) (oz os o′ o′ o′ os)
egTri = tzzz tsss t-″ t′s′ t-″ tsss
```

Morphisms in the slice can now be triangles: ψ →/ φ = Σ _ λ θ → Tri θ φ ψ.

A useful Δ_+-specific property is that morphisms in Δ_+/kz are *unique*. It is easy to state this property in terms of triangles with common edges, $\mathsf{triU} : \mathsf{Tri}\ \theta\ \phi\ \psi \to \mathsf{Tri}\ \theta'\ \phi\ \psi \to \theta \equiv \theta'$, and then prove it by induction on the triangles, not edges. It is thus cheap to obtain *universal properties* in the slices of Δ_+, asserting the existence of unique morphisms: uniqueness comes for free!

4 A Proliferation of Functors

Haskell makes merry with `class Functor` and its many subclasses: this scratches but the surface, giving only *endo*functors from types-and-functions to types-and-functions. Once we adopt the general notion, functoriality sprouts everywhere, with the same structures usefully functorial in many ways.

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

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

To see more examples, we need more categories. Let Set's objects be types in Agda's Set universe and $\mathsf{Set}(S,T)$ exactly $S \to T$, with the usual identity and composition. Morphism equality is *pointwise*. Exercises: make $\mathsf{Bwd} : \mathsf{Set} \to \mathsf{Set}$ a functor; check $(\mathsf{Lam}, \uparrow)$ is a functor from Δ_+ to Set.

Let us plough a different furrow, rich in dependent types, constructing new categories by *indexing*. If $I : \mathsf{Set}$, we may then take $I \to \mathsf{Set}$ to be the category whose objects are *families* of objects in Set, $S, T : I \to \mathsf{Set}$ with morphisms (implicitly indexed) families of functions: $S \mathbin{\dot\to} T = \forall \{i\} \to S\, i \to T\, i$. Morphisms are equal if they map each index to pointwise equal functions. In the sequel, it will be convenient to abbreviate $\mathsf{Bwd}\ K \to \mathsf{Set}$ as $\overline{K}$, for types indexed over scopes.

Dependently typed programming thus offers us a richer seam of categorical structure than we see in Haskell. This presents an opportunity to make sense of the categorical taxonomy in terms of concrete programming examples, and at the same time, organising those programs and indicating *what to prove*.

5 Things-with-Thinnings (a Monad)

Let us acquire the habit of packing terms together with an object in the slice of thinnings over their scope, selecting the support of the term and discarding unused variables. Note, $\Uparrow$ is a functor from $\overline{K}$ to itself.

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

In fact, the categorical structure of Δ_+ makes $\Uparrow$ a *monad*. Let us recall the definition.

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

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

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

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

The categorical structure of thinnings makes ⇑ a monad. Here, 'adding a layer' amounts to 'wrapping with a thinning'. The proof obligations to make (⇑, unit⇑, mult⇑) a monad are exactly those required to make Δ_+ 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 Δ_+ to Set.

```
unit⇑ : T →̇ (T ⇑_)     | mult⇑ : ((T ⇑_) ⇑_) →̇ (T ⇑_)       | thin⇑ : iz ⊑ jz → T ⇑ iz → T ⇑ jz
unit⇑ t = t ↑ oi       | mult⇑ ((t ↑ θ) ↑ φ) = t ↑ (θ ⨾ φ)  | thin⇑ θ t = mult⇑ (t ↑ θ)
```

Shortly, we shall learn how to find the variables on which a term syntactically depends. However, merely *allowing* a thinning at the root, Lam ⇑ *iz*, yields a redundant representation, as we may discard variables at either root or leaves. Let us eliminate redundancy by *insisting* that a term's support is *relevant*: a variable retained by the thinning *must* be used in the thing. Everybody's got to be somewhere.

6 The Curious Case of the Coproduct in Slices of Δ_+

The ⇑ construction makes crucial use of objects in the slice category $\Delta_+/$*scope*, which exhibit useful additional structure: they are *bit vectors*, with one bit per variable telling whether it has been selected. Bit vectors inherit Boolean structure, via the 'Naperian' array structure of vectors [11].

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

The *empty type* is famed for this rôle for types-and-functions: empty case analysis gives the vacuously unique morphism. In Δ_+, the empty *scope* plays this rôle, with the 'constant 0' bit vector as unique morphism. By return of post, we get ([], oe) as the initial object in the slice category Δ_+/kz. Hence, we can make *constants* with empty support, i.e., noting that no variable is ($\cdot_R$ for) *relevant*.

```
oe : ∀ {kz : Bwd K} → [] ⊑ kz            | oe/ : (θ : iz ⊑ kz) → oe →/ θ
oe {iz -, k} = oe o′                     | oe/ θ with tri oe θ
oe {[]}      = oz                        | ... | t rewrite law−oe (oe ⨾ θ) = oe , t
law−oe : (θ : [] ⊑ kz) → θ ≡ oe          |

data One_R : K̄ where ⟨⟩ : One_R []       | ⟨⟩_R : One_R ⇑ kz;    ⟨⟩_R = ⟨⟩ ↑ oe
```

We should expect the constant to be the trivial case of some notion of *relevant pairing*, induced by *coproducts* in the slice category. If we have two objects in Δ_+/kz representing two subscopes, (iz,θ) and (jz,ϕ), there should be a smallest subscope which includes both: pairwise disjunction of bit vectors.

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

However, we are not working in Set, but in a slice category. Any category theorist will tell you that slice categories $\mathbb{C}/I$ inherit *colimit* structure (characterized by universal out-arrows) from $\mathbb{C}$, as indeed we just saw with the initial object. If Δ_+ has coproducts, too, we are done! Taking K = One, let us seek the coproduct of two singletons, $S = T =$ [] -, ⟨⟩. Construct one diagram by taking $U =$ [] -, ⟨⟩ and $f = g =$ oi, ensuring that our only candidate for $S+T$ is again the singleton [] -, ⟨⟩, with $l = r =$ oi, making $h =$ oi. Nothing else can sit between S,T and U.

$$\begin{tikzcd}
 & U & \\
S \arrow[ur, "f"] \arrow[r, "l"'] & S+T \arrow[u, "h"] & T \arrow[ul, "g"'] \arrow[l, "r"]
\end{tikzcd}
\qquad
\begin{tikzcd}
 & U' & \\
S \arrow[ur, "f'"] \arrow[r, "l"'] & S+T \arrow[u, dotted, "?"] & T \arrow[ul, "g'"'] \arrow[l, "r"]
\end{tikzcd}$$

Now begin a different diagram, with $U' =$ [] -, ⟨⟩ -, ⟨⟩, allowing $f' =$ oz os o$'$ and $g' =$ oz o$'$ os. No h' post-composes l and r (both oi, making h' itself) to yield f' and g' respectively. We do not get coproducts.

Fortunately, we get what we need: Δ_+ may not have coproducts, but its *slices* do. Examine the data: two subscopes of some kz, $\theta : iz \sqsubseteq kz$ and $\phi : jz \sqsubseteq kz$. Their coproduct must be some $\psi : ijz \sqsubseteq kz$, where our l and r must be triangles Tri $\theta'\ \psi\ \theta$ and Tri $\phi'\ \psi\ \phi$, giving morphisms in $\theta \rightarrow_/ \psi$ and $\phi \rightarrow_/ \psi$. Choose ψ to be pointwise disjunction of θ and ϕ, minimizing ijz: θ' and ϕ' will then *cover* ijz.

```
data Cover (ov : Two) : iz ⊑ ijz → jz ⊑ ijz → Set where
  _c′s :                  Cover ov θ ϕ → Cover ov (θ o′) (ϕ os)
  _cs′ :                  Cover ov θ ϕ → Cover ov (θ os) (ϕ o′)
  _css : {both : Tt ov} → Cover ov θ ϕ → Cover ov (θ os) (ϕ os)
  czz  :                                 Cover ov    oz     oz
```

The flag, ov, determines whether *overlap* is permitted: with tt for coproducts and ff for *partitions*. No constructor allows both θ and ϕ to omit a target variable, so everybody's got to be somewhere. Let us compute the coproduct, ψ then check that any other diagram for some ψ' yields a $\psi \rightarrow_/ \psi'$.

```
cop   : (θ : iz ⊑ kz) (ϕ : jz ⊑ kz) →
        Σ _ λ ijz →            Σ (ijz ⊑ kz) λ ψ →
        Σ (iz ⊑ ijz) λ θ′  → Σ (jz ⊑ ijz) λ ϕ′  →
        Tri θ′ ψ θ × Cover tt θ′ ϕ′ × Tri ϕ′ ψ ϕ
copU  : Tri θ′ ψ θ  → Cover tt θ′ ϕ′  → Tri ϕ′ ψ ϕ  →
        θ →/ ψ′  → ϕ →/ ψ′  →  ψ →/ ψ′
```

$$\begin{tikzcd}
iz \arrow[dr, dashed, "\theta'"'] \arrow[drr, "\theta"] & & \\
 & \langle\, ijz \arrow[r, dashed, "\psi"] & kz \\
jz \arrow[ur, dashed, "\phi'"] \arrow[urr, "\phi"'] & &
\end{tikzcd}$$

where the ⟨ in the diagram indicates that the two incoming arrows form a Cover.

The recursive steps in cop's implementation work explicitly with the two-dimensional triangles and coverings, using ! to hide their boundaries (thinnings) and their boundaries' boundaries (scopes).

```
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
```

The copU proof goes by induction on the triangles which share ψ' and inversion of the coproduct.

A further useful property of coproduct diagrams is that we can selectively refine them by a thinning into the covered scope.

$$\begin{tikzcd} iz \arrow[r, dashed, "\psi_0"] \arrow[dr, dashed, "\theta"] & iz' \arrow[dr, "\theta'"] & \\ & kz \arrow[r, "\psi"] & kz' \\ jz \arrow[ur, dashed, "\phi"] \arrow[r, dashed, "\psi_1"'] & jz' \arrow[ur, "\phi'"'] & \end{tikzcd}$$

```
subCop : (ψ : kz ⊑ kz′) → Cover ov 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 ov θ ϕ
```

The implementation is a straightforward induction on the diagram.

The payoff from coproducts is the type of *relevant pairs* — the co-de-Bruijn touchstone:

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

```
_,R_ : S ⇑ kz → T ⇑ kz → (S ×R T) ⇑ kz
(s ↑ θ) ,R (t ↑ ϕ) =
  let ! ψ , θ′ , ϕ′ , _ , c , _ = cop θ ϕ
  in  pair (s ↑ θ′) (t ↑ ϕ′) c ↑ ψ
```

The corresponding projections are readily definable.

```
outlR : (S ×R T) ⇑ kz → S ⇑ kz
outlR (pair s _ _ ↑ ψ) = thin⇑ ψ s
```

```
outrR : (S ×R T) ⇑ kz → T ⇑ kz
outrR (pair _ t _ ↑ ψ) = thin⇑ ψ t
```

7 Monoidal Structure of Order-Preserving Embeddings

Variable bindings extend scopes. The λ construct does just one ‘snoc’, but binding can be simultaneous, so the monoidal structure on Δ_+ induced by concatenation is what we need.

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

```
_++⊑_ : iz ⊑ jz → iz′ ⊑ jz′ → (iz ++ iz′) ⊑ (jz ++ jz′)
θ ++⊑     oz   =   θ
θ ++⊑ (ϕ os)  = (θ ++⊑ ϕ) os
θ ++⊑ (ϕ o′)  = (θ ++⊑ ϕ) o′
```

Concatenation further extends to Coverings, allowing us to build them in chunks.

```
_++C_ : Cover ov θ ϕ → Cover ov θ′ ϕ′ → Cover ov (θ ++⊑ θ′) (ϕ ++⊑ ϕ′)
c ++C (d c′s)                = (c ++C d) c′s
c ++C (d cs′)                = (c ++C d) cs′
c ++C (_css {both = b} d)  = _css {both = b} (c ++C d)
c ++C czz                    = c
```

One way to build such a chunk is to observe that two scopes cover their concatenation.

```
lrCop : (iz jz : Bwd K) → Σ (iz ⊑ (iz ++ jz)) λ θ → Σ (jz ⊑ (iz ++ jz)) λ ϕ → Cover ov θ ϕ
lrCop iz        (jz -, j) = let ! ! c = lrCop iz jz in ! ! c c′s
lrCop (iz -, i) []        = let ! ! c = lrCop iz [] in ! ! c cs′
lrCop []        []        =                            ! ! czz
```

Now, crucial to the enterprise is that the monoidal structure of scopes lets us not only combine thinnings, but *split* them, into global and local parts.

```
_⊣_ : ∀ jz (ψ : iz ⊑ (kz ++ jz)) →                             Σ _ λ kz′ → Σ _ 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
(jz -, j) ⊣ (ψ os)               with jz ⊣ ψ
(jz -, j) ⊣ (. (θ ++⊑ ϕ) os) | ! ! θ , ϕ , refl , refl = ! ! θ , ϕ os , refl , refl
(jz -, j) ⊣ (ψ o′)               with jz ⊣ ψ
(jz -, j) ⊣ (. (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 _⊢_ jz (T : K̅) kz : Set where
  _⑊_ : iz ⊑ jz → T (kz ++ iz)
                → (jz ⊢ T) kz
```

```
_⑊R_ : ∀ jz → 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) : K̅ where
  only : VaR k ([] -, k)
```

```
vaR : k ← kz → VaR k ⇑ kz
vaR x = only ↑ x
```

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

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

```
lamR : Lam →̇ (LamR ⇑_)
lamR (# x)   = map⇑ #   (vaR x)
lamR (f $ s) = map⇑ app (lamR f ,R lamR s)
lamR (λ t)   = map⇑ λ   (_ ⑊R lamR t)
```

```
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
```

Stare bravely! 𝕂 returns a plainly constant function. Meanwhile, 𝕊 clearly uses all three inputs: the function goes left, the argument goes right, and the environment is shared.

8 A Universe of Metasyntaxes-with-Binding

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

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

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

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

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

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;   Σ_D    : (S : Datoid) → (Data S → Desc I) → Desc I
  One_D  :          Desc I;   _×_D_  : Desc I → Desc I →                  Desc I
```

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

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

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

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

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

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

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

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

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

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

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

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

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

We can compute co-de-Bruijn terms from de Bruijn terms, generically.

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

The reverse translation is left as an (easy) exercise in thinning composition for the reader.

9 Hereditary Substitution for Co-de-Bruijn Metasyntax

Let us develop the appropriate notion of substitution for our metasyntax, *hereditary* in the sense of Watkins et al. [23]. Substituting a higher-kinded variable requires us further to substitute its parameters.

We shall need a type to represent the fate of each variable in some source scope as we construct a term in some target scope. I call this type HSub: let us work through it slowly.

```
record HSub {I} (D       : I → Desc I)      -- the underlying syntax
                (src trg : Bwd (Kind I))    -- source and target scopes
                (act     : Bwd (Kind I))    -- the active subscope
                : Set where
  constructor _⊑[_]≔_
  field    -- to follow
```

While D, src and trg indicate the task at hand, the extra scope parameter, act, serves a more subtle purpose: let us see how, presently. The mixfix constructor is intended to suggest that the partition in the middle splits the source scope into passive and active variables, with different fates, respectively, thinning into the target scope and actual substitution:

```
{pass}    : Bwd (Kind I)
{passive} : pass ⊑ src
{active}  : act  ⊑ src
passTrg   : pass ⊑ trg                   -- passive variables are ‘renamed’
parti     : Cover ff passive active      -- ff forbids overlap
images    : ⟦ →· act | Tm_R D ⟧_R ⇑ trg   -- active variables are substituted
```

It is convenient to store substitution images as a spine, because hereditary substitutions are exactly generated from spines. Key to the design, however, is to index HSub over the *act*ive subscope, as that is what will conspicuously decrease when a recursive substition is triggered, making *termination* obvious — one of my older tricks [18].

Before we see how to perform a substitution, let us think how to *weaken* one: we certainly push under binders, jz, extending source and target scopes, crucially preserving the active subscope.

```
wkHSub : HSub D src trg act → ∀ jz → HSub D (src ++ jz) (trg ++ jz) act
wkHSub (ϕ ⊑[ p ]≔ is) jz = let ! ! p′ = lrCop jz [] in
  (ϕ ++⊑ oi {kz = jz}) ⊑[ p ++_C p′ ]≔ thin⇑ (oi ++⊑ oe {kz = jz}) is
```

We extend the partition to make all the bound variables passive and duly grow the thinning on the left. On the right, co-de-Bruijn representation lets us thin the spine of images at a stroke!

The definition of hereditary substitution is a mutual recursion, terminating because the *act*ive scope is always decreasing: hSub is the main operation on terms; hSubs proceed structurally, following a syntax description; hered handles the variable case, invokes hSub hereditarily as required.

```
hSub  :                 HSub D src trg act → Tm_R D i iz → iz ⊑ src → Tm_R D i ⇑ trg
hSubs : (S : Desc I) →  HSub D src trg act →
          ⟦ S | Tm_R D ⟧_R iz → iz ⊑ src → ⟦ S | Tm_R D ⟧_R ⇑ trg
hered : (jz ⇒ i) ← src → HSub D src trg act → ⟦ →jz | Tm_R D ⟧_R ⇑ trg → Tm_R D i ⇑ trg
```

There is a design choice here: we may either cut the substitution down to fit the support of the term we are processing, or retain the substitution intact and keep the thinning which embeds the term's support in the source scope. The latter makes the termination argument more straightforward, although we are required to curry a $\mathsf{Tm}_R\ D\ i \Uparrow src$ as a $t : \mathsf{Tm}_R\ D\ i\ iz$ with a $\psi : iz \sqsubseteq src$. Our first move is to refine the substitution's partition by ψ to check whether any of the variables in the term's support is actively being substituted. If not, we may simply thin t, with no further traversal.

```
hSub h@ (φ ⊑[ p′ ]≔ is) t ψ with subCop ψ p′
hSub h@ (φ ⊑[ p′ ]≔ is) t ψ | _ , [] , _ , _ , ψ0 , ψ1 , p with allLeft p
hSub h@ (φ ⊑[ p′ ]≔ is) t ψ | _ , [] , _ , _ , ψ0 , ψ1 , p | refl = t ↑ (ψ0 ⨾ φ)
```

The [] pattern matches the active part of the support, with allLeft the lemma that the passive part must be the whole support if the active part is empty. If, on the other hand, there are still active variables to find, we must keep hunting, in the knowledge that we have real work to do.

```
hSub {D = D} {i = i} h@_ [ ts ] ψ | _ = map⇑ [_] (hSubs (D i) h ts ψ)
```

If we find a node from our syntax, we proceed structurally:

```
hSubs (RecD (jz ⇒ i)) h (θ ⑊ t)                  ψ = jz ⑊R hSub (wkHSub h jz) t (ψ ++⊑ θ)
hSubs (ΣD S T)        h (s , ts)                  ψ = map⇑ (s ,_) (hSubs (T s) h ts ψ)
hSubs OneD            h _                         _ = ⟨⟩R
hSubs (S ×D T)        h (pair (s ↑ θ) (t ↑ φ) _) ψ = hSubs S h s (θ ⨾ ψ) ,R hSubs T h t (φ ⨾ ψ)
```

Meanwhile, for a variable with spine attached, we substitute the spine then proceed hereditarily.

```
hSub h@_ (# {jz} (pair (only ↑ x) (ss ↑ θ) c)) ψ | _ = hered (x ⨾ ψ) h (hSubs jz⃗ h ss (θ ⨾ ψ))
```

If the variable we seek is not the top one in the source context, we throw the top variable, passive or active, out of the substitution and keep looking.

```
hered (x o′) (φ ⊑[ p cs′ ]≔ is) ss = hered x (oi o′ ⨾ φ ⊑[ p ]≔ is) ss
hered (x o′) (φ ⊑[ p c′s ]≔ is) ss = hered x (φ ⊑[ p ]≔ outlR is) ss
```

We must rule out the possibility that any variable is *both* active and passive.

```
hered _ (_ ⊑[ _css {both = ()} _ ]≔ _) _
```

Now we have found our variable, and it is either passive (in which case we attach the spine)...

```
hered (x os) (φ ⊑[ p cs′ ]≔ _) ss = map⇑ # (vaR (oe os ⨾ φ) ,R ss)
```

...or active, in which case we substitute hereditarily.

```
hered {trg = trg} {act = (_ -, (jz ⇒ i))} (x os) (_ ⊑[ p c′s ]≔ is) ss
   with outrR is   | lrCop trg jz
... | (ψ ⑊ t) ↑ θ | ! ! p′ = hSub {act = jz} (oi ⊑[ p′ ]≔ ss) t (θ ++⊑ ψ)
```

As you can see, the target scope becomes passive, the bound variables of the substitution image become active, and the spine becomes the substitution for the active variables. The new active scope is visibly a substructure of the old active scope, so hereditary substitution is structurally recursive!

10 Discussion

We have a universe of syntaxes with metavariables and binding, where the Description of a syntax is interpreted as the co-de-Bruijn terms, ensuring intrinsically that unused variables are discarded not at the *latest* opportunity (as in de Bruijn terms), nor at an *arbitrary* opportunity (as in one of Bird and Paterson's variants [8], or with Hendriks and van Oostrom's 'adbmal' operator [14], both of which reduce the labour of shifting at the cost of nontrivial α-equivalence), but at the *earliest* opportunity. Hereditary substitution exploits usage information to stop when there is nothing to substitute, shifts without traversal, and is, moreover, structurally recursive on the *active scope*.

Co-de-Bruijn representation is even less suited to human comprehension than de Bruijn syntax, but its informative precision makes it all the more useful for machines. Dependency checking is direct, so syntactic forms like vacuous functions or η-redexes are easy to spot.

It remains to be seen whether co-de-Bruijn representation will lead to more efficient implementations of normalization and of metavariable instantiation. The technique may be readily combined with representing terms as trees whose top-level leaves are variable uses and top-level nodes are just those (now easily detected) where paths to variables split: edges in the tree are *closed* one-hole contexts, jumped over in constant time [10].

I see two high-level directions emerging from this work. Firstly, the generic treatment of syntax with *meta*variables opens the way to the generic treatment of *metatheory*. Even without moving from scope-safe to type-safe term representations, we can generate the inductive relations we use to define notions such as reduction and type synthesis in a universe, then seek to capture good properties (e.g., stability under substitution, leading to type soundness) by construction. Co-de-Bruijn representations make it easy to capture properties such as variable non-occurrence in the syntax of formulæ, and might also serve as the target term representation for algorithms extracted generically from the rules.

Secondly, more broadly, this work gives further evidence for a way of programming with strong invariants and redundant but convenient information caches without fear of bugs arising from inconsistency. We should put the programmer in charge! Dependent types should let us take control of data representations and optimise them to support key operations, but with the invariants clearly expressed in code and actively supporting program synthesis.

Only a fool would attempt to enforce the co-de-Bruijn invariants without support from a typechecker, so naturally I have done so: using Haskell's `Integer` for bit vectors (making `-1` the identity of the unscoped thinning *monoid*), I implemented a dependent type system, just for fun. It was Hell's delight, even with the Agda version to follow. I was sixteen again.

Acknowledgements. EPSRC project EP/M016951/1 *Homotopy Type Theory: Programming and Verification* funded this work. My Mathematically Structured Programming colleagues at Strathclyde made me get these ideas in shape: Fredrik Nordvall Forsberg offered particularly useful advice about what to omit. Philippa Cowderoy's use of *information effects* for typing contexts increased my sensitivity to the signposting of discards and duplications. An EU TYPES Short Term Scientific Mission brought Andrea Vezzosi to Strathclyde, provoking ideas and action for further work. Invitations to present at *Trends in Functional Programming 2017* (Canterbury) and in Nottingham (with Thorsten Altenkirch in the audience) helped me find the words. Andreas Abel sent some particularly helpful feedback, as did a number of commenters on social media. The advice of anonymous referees has instigated significant improvement. As ever, James McKinna and Randy Pollack remain sources of inspiration.

References

[1] Andreas Abel (2009): *Implementing a normalizer using sized heterogeneous types*. J. Funct. Program. 19(3-4), pp. 287–310, doi:10.1017/S0956796809007266.

[2] Andreas Abel & Nicolai Kraus (2011): *A Lambda Term Representation Inspired by Linear Ordered Logic*. In Herman Geuvers & Gopalan Nadathur, editors: Proceedings Sixth International Workshop on Logical Frameworks and Meta-languages: Theory and Practice, LFMTP 2011, Nijmegen, The Netherlands, August 26, 2011., EPTCS 71, pp. 1–13, doi:10.4204/EPTCS.71.1.

[3] Guillaume Allais, James Chapman, Conor McBride & James McKinna (2017): *Type-and-scope safe programs and their proofs*. In Yves Bertot & Viktor Vafeiadis, editors: Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017, ACM, pp. 195–207, doi:10.1145/3018610.3018613.

[4] Thorsten Altenkirch, Martin Hofmann & Thomas Streicher (1995): *Categorical Reconstruction of a Reduction Free Normalization Proof*. In David H. Pitt, David E. Rydeheard & Peter T. Johnstone, editors: Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7-11, 1995, Proceedings, Lecture Notes in Computer Science 953, Springer, pp. 182–199, doi:10.1007/3-540-60164-3_27.

[5] Thorsten Altenkirch & Bernhard Reus (1999): *Monadic presentations of lambda-terms using generalized inductive types*. In: Computer Science Logic 1999, pp. 453–468, doi:10.1007/3-540-48168-0_32.

[6] Francoise Bellegarde & James Hook (1995): *Substitution: A formal methods case study using monads and transformations*. Science of Computer Programming, doi:10.1016/0167-6423(94)00022-0.

[7] Nick Benton, Chung-Kil Hur, Andrew Kennedy & Conor McBride (2012): *Strongly Typed Term Representations in Coq*. J. Autom. Reasoning 49(2), pp. 141–159, doi:10.1007/s10817-011-9219-0.

[8] Richard Bird & Ross Paterson (1999): *de Bruijn notation as a nested datatype*. Journal of Functional Programming 9(1), pp. 77–92, doi:10.1017/S0956796899003366.

[9] Nicolas G. de Bruijn (1972): *Lambda Calculus notation with nameless dummies: a tool for automatic formula manipulation*. Indagationes Mathematicæ 34, pp. 381–392, doi:10.1016/1385-7258(72)90034-0.

[10] Lucas Dixon, Peter Hancock & Conor McBride (2007): *Why walk when you can take the tube?* Available at http://strictlypositive.org/Holes.pdf. Unpublished draft.

[11] 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.

[12] Healfdene Goguen & James McKinna (1997): *Candidates for Substitution*. Technical Report ECS-LFCS-97-358, University of Edinburgh.

[13] Robert Harper, Furio Honsell & Gordon D. Plotkin (1993): *A Framework for Defining Logics*. J. ACM 40(1), pp. 143–184, doi:10.1145/138027.138060.

[14] Dimitri Hendriks & Vincent van Oostrom (2003): *adbmal*. In Franz Baader, editor: Automated Deduction - CADE-19, 19th International Conference on Automated Deduction Miami Beach, FL, USA, July 28 - August 2, 2003, Proceedings, Lecture Notes in Computer Science 2741, Springer, pp. 136–150, doi:10.1007/978-3-540-45085-6_11.

[15] Chantal Keller & Thorsten Altenkirch (2010): *Hereditary Substitutions for Simple Types, Formalized*. In Venanzio Capretta & James Chapman, editors: Proceedings of the 3rd ACM SIGPLAN Workshop on Mathematically Structured Functional Programming, MSFP@ICFP 2010, Baltimore, MD, USA, September 25, 2010., ACM, pp. 3–10, doi:10.1145/1863597.1863601.

[16] Richard Kennaway & M. Ronan Sleep (1987): *Variable Abstraction in O(n log n) Space*. Inf. Process. Lett. 24(5), pp. 343–349, doi:10.1016/0020-0190(87)90161-X.

[17] Conor McBride (2000): *Dependently typed functional programs and their proofs*. Ph.D. thesis, University of Edinburgh, UK.

[18] Conor McBride (2003): *First-order unification by structural recursion*. *J. Funct. Program.* 13(6), pp. 1061–1075, doi:10.1017/S0956796803004957.

[19] Spike Milligan (1972): *The Last Goon Show of All*. BBC Radio 4.

[20] Ulf Norell (2008): *Dependently Typed Programming in Agda*. In Pieter W. M. Koopman, Rinus Plasmeijer & S. Doaitse Swierstra, editors: *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, *Lecture Notes in Computer Science* 5832, Springer, pp. 230–266, doi:10.1007/978-3-642-04652-0_5.

[21] Masahiko Sato, Randy Pollack, Helmut Schwichtenberg & Takafumi Sakurai (2013): *Viewing λ-terms through Maps*. *Indagationes Mathematicæ* 24(4), doi:10.1016/j.indag.2013.08.003.

[22] François-Régis Sinot, Maribel Fernández & Ian Mackie (2003): *Efficient Reductions with Director Strings*. In Robert Nieuwenhuis, editor: *Rewriting Techniques and Applications, 14th International Conference, RTA 2003, Valencia, Spain, June 9-11, 2003, Proceedings*, *Lecture Notes in Computer Science* 2706, Springer, pp. 46–60, doi:10.1007/3-540-44881-0_5.

[23] Kevin Watkins, Iliano Cervesato, Frank Pfenning & David Walker (2003): *A Concurrent Logical Framework: The Propositional Fragment*. In Stefano Berardi, Mario Coppo & Ferruccio Damiani, editors: *Types for Proofs and Programs, International Workshop, TYPES 2003, Torino, Italy, April 30 - May 4, 2003, Revised Selected Papers*, *Lecture Notes in Computer Science* 3085, Springer, pp. 355–377, doi:10.1007/978-3-540-24849-1_23.