CodeBruijn Programming, Categorically

Conor Mc Bride

Mathematically Structured Programming, Computer & Information Sciences, University of Strathclyde, Scotland

I'll write this later when I know what I'm saying.

CodeBruijn ("co de Bruijn") programming is a methodology for representing and manipulating syntax in a nameless way, like de Bruijn representation [5], but making the opposite canonical choice for exposing the *non*-use of variables. The essence of the idea is to restrict the *scope* (which variables *may* occur free) of a term to its *support* (which variables *do* occur free). Variables are expelled from scope at the roots of maximal subtrees in which they are not used, where de Bruijn representation keeps variables in scope from their binding sites all the way to the leaves, i.e., the minimal subtrees in which they are not used. The codeBruijn approach thus relies on the maintenance of subtle invariants, reminiscent of Kennaway and Sleep's 'director strings' representation [7]. Dependently typed programming languages such as Agda [10] readily taken on the task of minding that business for us. This paper shows how, and hopefully, why.

The key structure at work is the semi-simplicial category on scopes, i.e., the category of order-preserving embeddings (colloquially, 'thinnings') from one scope to another. From Bellegarde and Hook [3], via Bird and Paterson [4], and Altenkirch and Reus [2], it has become a commonplace to index types of terms by their scopes. Such types should really be thought of as 'thinnables' — *functors* from the thinnings — because thinnings act compositionally on terms. The operation of mapping a scope to its identity thinning induces a forgetful functor from thinnables to scope-indexed types. This forgetful functor has a celebrated right adjoint, amounting to abstraction over all scopes into which one's own embeds, which is the basis of Altenkirch, Hofmann and Streicher's

Conor Mc Bride: conor@strictlypositive.org, http://strictlypositive.org

Kripke-model construction which drives normalization by evaluation [1]. But, being a forgetful functor, one should ask after its *left* adjoint. That exists, of course, and is the basis of codeBruijn programming: we define *relevant* terms, indexed by *support*, then make them *freely* thinnable by attaching the thinning from support to scope at the root of the term. Further thinnings act by composition at the root, without traversing the term at all.

This paper is written as a literate Agda implementation of a codeBruijn toolkit, structured categorically. I formalise the active categorical abstractions, given provocation from the task at hand. I have adopted something of a tutorial style, partly because there is some novelty in teaching category theory to functional programmers with examples which are not sets-and-functions, but mostly because I am teaching myself. I hope it is also a useful engagement with dependently typed programming, for category theorists. I shall certainly draw the *diagrams* which drive the constructions. There will also be some transferable lessons about programming 'craft' which I shall seek to draw out.

1 de Bruijn Representation

In 1999, Altenkirch and Reus gave a de Bruijn treatment of simply typed λ-calculus, together with an implementation of simultaneous substitution [2]. Let us review how it goes.

The simple types are given inductively.

```
infixr 30 _⊃_

data Ty : Set where
  base : Ty
  _⊃_ : Ty → Ty → Ty
```

In Agda, infix operators are named with _ in their argument places. Types are arranged in backwards ('*snoc*-') lists to form *contexts*.

```
infixl 20 _-,_

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

Cx = Bwd Ty
```

Craft 1 (Backwards Lists) *Forwards lists are much more commonplace in functional programming, but I have learned the hard way to use a separate type for lists which grow on the right. The cognitive cost of interpreting lists in reverse is higher, at times, than I can pay: I make mistakes. I also choose symbols for 'snoc' and 'cons' which avoid misleading reflectional symmetry and have modest pictographic value.*

Typed de Bruijn indices select *one* entry from a context.

```
infix 10 _←_
data _←_ (τ : Ty) : Cx → Set where
  ze : τ ← Γ -, τ
  su : τ ← Γ
     → τ ← Γ -, σ
```

Craft 2 (Parameters, Uniform Indices, Restrictable Indices) *In the* **data** *declaration of an Agda type former, some things are declared left of* : *and scope over the whole declaration. They must be used* uniformly *in the* return *types of value constructors. They are, however, free to vary in the types of recursive substructures. If they do so vary, we call them* uniform *indices. Only if they remain constant throughout should we refer to them as* parameters. *So,* τ*, above is a parameter, but* Γ*, below is a uniform index. The distinction impacts the category in which an initial object is being constructed.* $(\tau <\!\!- \_)$ *is constructed in* Cx → Set*, while* $\_ \vdash \_$ *is constructed in* Cx → Ty → Set*. Meanwhile, right of* : *come those things which may be restricted to particular patterns of value in the return types of value constructors, e.g.,* nonempty *contexts above, and* function *types below.*

The type of terms reflect the typing rules, indexed by a context and the type being inhabited.

```
infix 10 _⊢_
data _⊢_ (Γ : Cx) : Ty → Set where
  va : τ ← Γ
     → Γ ⊢ τ
  ap : Γ ⊢ σ ⊃ τ
     → Γ ⊢ σ
     → Γ ⊢ τ
```

```
la : Γ –, σ ⊢ τ
  → Γ ⊢ σ ⊃ τ
```

Observe that the context we are handed at the root of a term only ever gets larger, each time we use a lambda. Only when we reach a variable do we choose one thing from the context and disregard the rest.

Now, a *simultaneous substitution* is a type-respecting mapping from variables in some source context Γ to terms over target context Δ — from $(\_ \leftarrow \Gamma)$ to $(\Delta \vdash \_)$, if you will. When we push such a thing under la, we need instead a mapping from $(\_ \leftarrow \Gamma\ –, \sigma)$ to $(\Delta\ –, \sigma \vdash \_)$. We can map the newly bound ze to va ze, but the trouble is that all of Γ's variables are mapped to terms over Δ, not $\Delta\ –, \sigma$. It is thus necessary to traverse all those terms and adjust their leaves, because it is only at the leaves that we document *non*-usage of variables. Shift happens.

Worse, if we attempt to carry out the shift by simultaneous substitution, we leave the comfortable territory of structural recursion and have some explaining to do. It is useful to observe that shifts are merely simultaneous (order-preserving, injective) renumberings which may readily act on terms. Once we have simultaneous renumbering available, simultaneous substitution is easy. Moreover, they are very similar, so we may readily abstract the common structure, as I learned from Goguen and McKinna and demonstrated in my thesis [6, 8].

Shifts — simultaneous renumberings — are the problem, but they are also the key to the solution.

2 Thinnings

Definition 3 (Thinnings) *We may define the thinnings, i.e., the order-preserving embeddings, our simultaneous renumberings, as follows:*

```
module _ {X : Set} where    -- fix a set X of sorts, e.g., Ty
  infix 10 _≤_
  infixl 20 _-^-_
  data _≤_ : Bwd X → Bwd X → Set where
    _-^-_ :              γ      ≤ δ
          → ∀ x →        γ      ≤ δ –, x
```

```
_-,_ :                 γ       ≤ δ
     → ∀ x →           γ -, x  ≤ δ -, x
[]   :                 []      ≤ []
```

I am careful to speak of our backward lists as *scopes*, rather than *contexts*, as it is not necessary for them to document the *types* of the variables for this machinery to work.

We lift the constructors from lists to represent the situation where parts of the source scope are copied to the target, but we also introduce _-^ to insert an extra element in the target.

Electronic engineers will notice that a thinning is more or less a vector of bits, with _-^ for 0 and _-, for 1, but it is indexed by its *population* — the entries marked 1. Expect Boolean operations.

Definition 4 (identity thinnings, ι) *Let us observe that identity thinnings exist, copying their scope.*

```
ι : ∀ {γ} → γ ≤ γ
ι {[]}      = []
ι {γ -, x} = ι {γ} -, x
```

Craft 5 (Implicits) *Agda uses curly braces to mark arguments which are normally suppressed. In general, it is sensible to adopt the suppression convention appropriate for the expected* use sites. *Here, the fact that ≤ is a type constructor means that γ will be determined if the type is given in advance. Often, we then have to use an explicit override at definition sites. This sort of thing never happens in Hindley-Milner languages because any information for which inference is permitted is guaranteed to be operationally useless. The inference of operationally useful information represents progress.*

Further, thinnings compose. I write composition diagrammatically.

Definition 6 (thinning composition, ⨟) *Thinnings fore and aft compose thus:*

```
infixl 20 _⨟_
_⨟_ : γ ≤ δ → δ ≤ ζ → γ ≤ ζ
θ       ⨟ (ϕ -^ x) = θ ⨟ ϕ -^ x
```

```
(θ -< x) ;; (φ -, x) = θ ;; φ -< x
(θ -, x) ;; (φ -, x) = θ ;; φ -, x
[]       ;; []       = []
```

Craft 7 (Operator Priority and Association) *My habit is to arrange priority and association so that computation results in net decrease of parentheses.*

Craft 8 (Laziness and Definitional Equality) *Working in intensional type theories like Agda involves a certain amount of care with the equational properties of programs — the typechecker will run these programs, as defined, on open terms. So the order of the lines in the above program matters. If the aft-thinning inserts, there is no call to inspect the fore-thinning. Only if the aft thinning copies need we ask what the fore-thinning gives us. If the first line is moved later, the function becomes unnecessarily strict in the fore-thinning, and definitional equality loses power.*

The reader should note that I will shortly substitute a subtly different definition of composition for this one. Rest assured that its replacement will satisfy the above equations.

3 When 'Green Slime' is Bad, Avoid It

We are accustomed to reasoning by *equation*.

Definition 9 (Inductive Equality, $\sim$) *In Agda, we may give an inductive definition of equality, as follows:*

```
infix 5 _∼_
data _∼_ {l} {X : Set l} (x : X) : X → Set l where
  r∼ : x ∼ x
```

The l parameter is an arbitrary level *in the Russell-style hierarchy:* Set *abbreviates* Set 0; Set 0 : Set 1, *and so on.*

The above definition is *intensional*, in that we can give canonical evidence that $x \sim y$ only if x and y have the same *implementation*, up to definitional equality. We shall have trouble because of that, in what is to follow, but that is not the trouble I mean to discuss in this section.

Suppose we have an hypothesis $q : \theta \;;; \phi \sim \psi \;-, x$. We ought to be able to deduce that θ and ϕ are both made by $(\_ -, x)$. However, pattern matching

on the equality proof q will fail, because it is not clear how to unify $\theta \fatsemi \phi$ with $\psi \mathbin{-\!\!,} x$, unless we are gifted with the power to run functions backwards, which Agda is not. Our only option is to *remember* how $\fatsemi$ computes, then match on ϕ and θ, eliminating the impossible cases which arise by refuting q.

'Green slime' is a colloquialism for expressions involving recursively defined functions which do not compute to canonical form. It is toxic to unification, as it unifies only with variables (purple things). Fortunately, there are strategies to avoid it.

Craft 10 (Inductive Relations are Invertible) *It is often useful to replace equations* `f s ∼ t` *with relations* `F s t`*, where* `F` *is defined inductively to be the* graph *of* f*.*

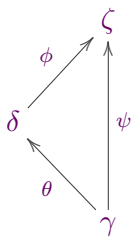
Let us put this into practice at once!

Definition 11 (composition triangle) *Reading from the definition of* $\fatsemi$*, construct its graph, thus:*

```
infixl 20 _-,-_

data ⌜ _ ⨟ _ ⌝∼ _ : (θ : γ ≤ δ) (ϕ : δ ≤ ζ) (ψ : γ ≤ ζ) → Set where
  _-,-_ :                    ⌜ θ       ⨟ ϕ       ⌝∼ ψ
        → ∀ x →              ⌜ θ       ⨟ ϕ -, x  ⌝∼ ψ -, x

  _-,^_ :                    ⌜ θ       ⨟ ϕ       ⌝∼ ψ
        → ∀ x →              ⌜ θ -, x  ⨟ ϕ -, x  ⌝∼ ψ -, x

  _^,_ :                     ⌜ θ       ⨟ ϕ       ⌝∼ ψ
        → ∀ x →              ⌜ θ -, x  ⨟ ϕ -, x  ⌝∼ ψ -, x

  []    :                    ⌜ []      ⨟ []      ⌝∼ []
```

And there was nothing green to be seen! The only green things on the right in the definition of $\fatsemi$ were recursive calls to $\fatsemi$, and these we have replaced by variables.

I call inhabitants of $\ulcorner \theta \fatsemi \phi \urcorner\sim \psi$ *composition triangles* exactly because they witness the commutation of the diagram



Now let us get our hands on them. You might think that the thing to do is to prove

$$(\theta : \gamma \leq \delta)\,(\phi : \delta \leq \zeta) \to \ulcorner\, \theta \fatsemi \phi \,\urcorner \sim \theta \fatsemi \phi$$

but that amounts to implementing composition a *third* time, as well as being pragmatically suboptimal, as I shall explain later. A better move is to reimplement $\fatsemi$ by proving, morally,

$$(\theta : \gamma \leq \delta)\,(\phi : \delta \leq \zeta) \to \text{`}\exists\psi.\text{'}\ \ulcorner\, \theta \fatsemi \phi \,\urcorner \sim \psi$$

We shall have need of existential quantification!

Definition 12 (Dependent Pair Types) *Dependent pair types (Σ-types, in the jargon) may be introduced as* records*, where the type of the second projection depends on the value of the first. The definition is polymorphic in the type theoretic hierarchy.*

```
infixr 5 _,_

record Σ {k l} (S : Set k) (T : S → Set l) : Set (l ⊔ k) where
  constructor _,_
  field fst : S
        snd : T fst
open Σ public     -- brings the projections into scope

syntax Σ S (λ x → T) = (x : S) × T
```

It is convenient to sugar dependent pair types as a binding form, after the fashion of dependent function types[1]*.*

Two commonly occurring degenerate forms merit abbreviation.

```
_×_ : ∀ {k l} → Set k → Set l → Set (l ⊔ k)
S × T = (_ : S) × T
⟨_⟩ : ∀ {k l} {S : Set k} (T : S → Set l) → Set (l ⊔ k)
⟨ T ⟩ = (x : _) × T x
```

The first of these is ordinary non-dependent pairing. The second, prounounced 'possibly T', asserts that a 'predicate' (i.e., a function to Set l) is somehow satisfied: the domain of the predicate is elided.

[1] Agda does not let you do precisely this, but LaTeX does the rest.

Craft 13 (Mixfix Operator Sections) *The relation* $\ulcorner\ \_\ ⨟\ \_\ \urcorner\sim\ \_$ *is given in mixfix form exactly because Agda supports operator sections —* $(\ulcorner\ \_ ⨟ \phi \urcorner\sim \psi)$, $(\ulcorner\ \theta ⨟ \_ \urcorner\sim \psi)$ *and* $(\ulcorner\ \theta ⨟ \phi \urcorner\sim \_)$ *— which are predicates designed to be used with* $\langle\_\rangle$.

Lemma 14 (constructing composition triangles, $\langle ⨟ \rangle$**)** *If* θ *and* ϕ *are thinnings which meet in the middle, it is possible to construct their composition triangle. Accordingly, we may redefine* $⨟$ *to project the existential witness.*

```
infix 10 _⟨⨟⟩_

_⟨⨟⟩_ : (θ : γ ≤ δ) (ϕ : δ ≤ ζ) → ⟨ (⌜ θ ⨟ ϕ ⌝∼ _) ⟩
θ       ⟨⨟⟩ ϕ -≤ x with _ , v ← θ ⟨⨟⟩ ϕ = _ , v -≤ x
θ -≤ x  ⟨⨟⟩ ϕ -, x with _ , v ← θ ⟨⨟⟩ ϕ = _ , v -≤ x
θ -, x  ⟨⨟⟩ ϕ -, x with _ , v ← θ ⟨⨟⟩ ϕ = _ , v -, x
[]      ⟨⨟⟩ []                           = _ , []

_⨟_ :  γ ≤ δ → δ ≤ ζ → γ ≤ ζ
θ ⨟ ϕ = fst (θ ⟨⨟⟩ ϕ)
```

Craft 15 (with Programs) *The* **with** *programming notation [9]*

```
f p⃗  with e
f p⃗₁ |    p₁ = e₁
     ⋮
f p⃗ₙ |    pₙ = eₙ
```

allows us to extend the left-hand side of a program with an extra column for the value of e*, so we may match patterns anywhere, refining the original patterns* $\vec{p}$ *as well as asking about* e*. Usefully,* e *is abstracted from any types where it occurs, so matching on its value refines types, too.*

By defining $⨟$ *as a projection of* $\langle ⨟ \rangle$*, we make* **with** $\theta\ \langle ⨟ \rangle\ \phi$ *abstract all occurrences of* $⨟$ *at the same time as it yields up the composition triangle. The same is true of any program defined as the existential witness to the possibility of satisfying a specification.*

The recently added notation, **with** $p \leftarrow e$*, allows us to avoid introducing a new line to the pattern match if there is only* one *case, given by* p*.*

Craft 16 (Invisible Programs) *Agda allows the 'don't care'* _ *to be used both in patterns, where it neglects to ask a question, and in expressions, where it neglects to give an answer. In the latter case, the missing term must be inferable by unification. In the*

case of function graphs, the program already given in the relation determines by its construction the missing existential witnesses. The composition function *has all but disappeared!*

Now, composition triangles give the graph of a function, and functions are deterministic, accordingly, we should be able to recover the fact that the inputs determine both the output and the witness.

Lemma 17 (Uniqueness of Composition Triangles) *For any given inputs θ and ϕ, there is at most one ψ and at most one v such that $v : \ulcorner \theta \fatsemi \phi \urcorner \sim \psi$.*

```
infix 10 _⁓⨟_
_⁓⨟_ :  (v₀ : ⌜ θ ⨟ φ ⌝∼ ψ₀) (v₁ : ⌜ θ ⨟ φ ⌝∼ ψ₁)
     → ((⋆) : ψ₀ ∼ ψ₁) × v₀ ∼ v₁
v₀ -^ x ⁓⨟ v₁ -^ x with ⋆ ← v₀ ⁓⨟ v₁ = ⋆
v₀ -^; x ⁓⨟ v₁ -^; x with ⋆ ← v₀ ⁓⨟ v₁ = ⋆
v₀ -, x ⁓⨟ v₁ -, x with ⋆ ← v₀ ⁓⨟ v₁ = ⋆
[]      ⁓⨟ []                        = ⋆
```

Craft 18 (Dependent Equations) *The equation $v_0 \sim v_1$ may look ill typed, but it is not. The inlined pattern match in the dependent pair type causes ψ_0 to unify with ψ_1. This can be quite a convenient way to specify 'telescopic' equations.*

Craft 19 (Contraction Pattern, $\star$) *In typeset code, I write $\star$ instead of unique patterns which can be constructed by record expansion followed by at most one step of case analysis for each field. Matching with $\star$ collapses spaces to a point. Agda does not yet support this feature. Here, it abbreviates* r∼ , r∼*, but it can scale to much larger uninteresting types.*

We have a notion of thinning, closed under identity and composition. I like to visualize thinnings as two horizontal sequences of dots. Each dot on the bottom is joined to a dot on top by a vertical chord, but there may be dots on top with no chord incident.

The identity thinning has all chords present. Composition is vertical pasting, followed by the contraction of chords which do not reach the bottom.

Spatial intuition makes it clear, informally, that identitied are absorbed and that composition is associative. We should thus be able to construct a category. But what does it mean *in type theory* to construct a category? That is when our troubles really begin.

4 Type Theorists Worry About Equality

An ingenue (or very sophisticated troll) once wrote to some mathematical mailing list asking whether category theory and type theory were the same. Some category theorists answered vaguely in the positive, at which point the type theorists accused them of insufficiently interrogating the meaning of 'the same'. The title of this section is tantamount to a definition of the discipline, especially if you come from the school which takes the classification of a thing to be the diagonal of the classified partial equivalence relation which says when two things are the same.

Informally, a category is given by

1. some notion of *objects*;
2. for every pair of objects, *source* and *target*, some notion of *arrows* from source to target;
3. for each object, an *identity* arrow from that object to itself;
4. for each pair of arrows which meet in the middle, a *composite* arrow from the source of the first to the target of the second;
5. ensuring that composition absorbs identity and associates, i.e., that some equations between arrows hold.

For thinnings, our objects are scopes and $\leq$ tells us what the arrows are. We have candidates for identity and composition. We can take the same view of types and functions: Set gives our objects, $\to$ our arrows, and we have the identity function and function composition.

But any type theorist will ask, or rather will *be asked* by their equipment, 'What is the status of equations between arrows?'. For thinnings, which are first order inductive data structures (indeed, bit vectors), the intensional $\sim$ should suffice; for functions, $\sim$ is dangerously restrictive, identifying only

functions with the same *implementation*, up to definitional equality. Any workable notion of category within type theory has to negotiate this distinction, which is waved away in everyday mathematical practice.

We have three options:

1. *Worry About Equality.* Work to replace intensional $\sim$ by something which better reflects mathematical intuition. That is the work of many lifetimes, mine included, and it is beginning to pay off. Observational Type Theory gave a good answer to when values are equal, but not such a good answer to when types are equal. (It was never the basis of a usable implementation, a fact for which I bear some blame.) Homotopy Type Theory gives a better answer, and in its Cubical variant, is beginning to materialise. This is the best option, if you have patience.

2. *Tell Lies.* Postulate that $\sim$ has the properties we wish it had, e.g. that pointwise equal functions are equal. Get on with exploring the important ideas. Unfortunately, the computational properties of postulates frustrate the execution of actual, if sophisticated, programs when proofs of equations are used to transport actual values between merely provably equal types. None the less, this is the best option, if you have undergraduates.

3. *Tell Weaker Truths.* Arrange to work up to $\sim$ when you can (e.g., with thinnings) and to weaker notions (e.g., pointwise equality for functions) when you cannot. This is the best option, if you are in a hurry.

My head is with option 1, my heart is with option 2, but my entire digestive system is with option 3, so that is how I shall proceed in this paper.

The plan, which is far from original, is to work with setoids of arrows, carefully managing the appropriate notion of equivalence on a case-by-case basis. A *setoid* is a set equipped with an *arbitrary* equivalence relation.

Definition 20 (Setoid) *Every level of the hierarchy is equipped with a notion of* Setoid*.*

```
record Setoid l : Set (lsuc l) where
  field El : Set l                                  -- Elements
        Eq : El → El → Set l                        -- Equivalence
        Rf : (x : El)       → Eq x x                -- Reflexivity
        Sy : (x y : El)     → Eq x y → Eq y x       -- Symmetry
```

```
    Tr : (x y z : El) → Eq x y → Eq y z → Eq x z   -- Transitivity
open Setoid
```

Definition 21 (Intensional Setoid) *Every Set gives rise to a Setoid whose equivalence is given by $\sim$.*

```
IN : Set l → Setoid l
El (IN X) = X
Eq (IN X) = _∼_
Rf (IN X) x         = ⋆
Sy (IN X) x y ⋆     = ⋆
Tr (IN X) x y z ⋆ ⋆ = ⋆
```

Craft 22 (Green Things in Blue Packaging) *I anticipate that we shall need to construct explanations which look like equational proofs, but are constructed within the equivalence of a known Setoid. I therefore introduce a type constructor whose entire purpose is to fix the Setoid at work. There is no general way to infer the setoid X from a type which is known to be Eq X x y, so the craft lies in ensuring that we never forget which Setoid we work in.*

```
record _∋_≈_ {l} (X : Setoid l) (x y : El X) : Set l where
  constructor eq
  field       qe : Eq X x y
open _∋_≈_ public
```

When we formulate categorical laws, we shall use this wrapped version.

At last, we are ready to say what a category might be.

5 Categories, Type Theoretically

What follows is far from perfect. The best that can be said is that it is an effective pragmatic compromise. Neither is it an unusual recipe. I labour the point only to teach the craft of the cooking.

A category will have a Set of objects and, indexed by source and target objects, a Setoid of arrows.

But there's another catch: type theoretic level. There is no good reason to believe that the level objects live on is in any way related to the the level that

arrows live on. Agda is particularly bad at supporting *cumulativity* — implicit upward flow between levels — and by 'bad', I mean it just does not. (Coq by contrast, is rather good at it.) Agda forces one to use level polymorphism instead of cumulativity. The two are poor stablemates, but they have backed the wrong horse. In the now, the pragmatic policy is to keep the levels of objects and arrows separate.

Definition 23 (Category) *Fix k, the level of objects, and l, the level of arrows. We may then define a notion of* Category.

```
record Cat : Set (lsuc (k ⊔ l)) where
  -- We have a Set of Objects, and a family of Setoids of Arrows.
  field Obj  : Set k
        Arr  : Obj → Obj → Setoid l
          -- Agda allows one to pause between fields to make definitions...
  _▷_ : Obj → Obj → Set l
  S ▷ T = El (Arr S T)
          -- ...and then resume requesting fields.
  -- We have identity and composition.
  field ι    : T ▷ T
        _⨟_  : R ▷ S → S ▷ T → R ▷ T
  -- Locally define equality of arrows...
  _≈_ : {S T : Obj} (f g : S ▷ T) → Set l
  _≈_ {S} {T} f g = Arr S T ∋ f ≈ g
  -- ...then require the laws.
  field coex : f ≈ f' → g ≈ g' → (f ⨟ g) ≈ (f' ⨟ g')
        idco : (f : S ▷ T) →          (ι ⨟ f)       ≈ f
        coid : (f : S ▷ T) →          (f ⨟ ι)       ≈ f
        coco : (f : R ▷ S) (g : S ▷ T) (h : T ▷ U) → (f ⨟ (g ⨟ h)) ≈ ((f ⨟ g) ⨟ h)
```

Note the inevitable necessity of coex*, the explicit witness that composition respects the weak notion of equivalence given by $\approx$: let us ensure that this proof is always trivial.*

```
module _ {l} {X : Setoid l} where
  private RfX = Rf X; SyX = Sy X; TrX = Tr X
  infixr 5 _≈⟨_⟩≈_ _≈⟨_⟩≈_
  infixr 6 _□
```

```
_≋⟨_⟩≋_ : ∀ x → X ∋ x ≈ y → X ∋ y ≈ z → X ∋ x ≈ z
x ≋⟨ eq q ⟩≋ eq q′ = eq (TrX _ _ _ q q′)

_≋⟨_⟩≋_ : ∀ x → X ∋ y ≈ x → X ∋ y ≈ z → X ∋ x ≈ z
x ≋⟨ eq q ⟩≋ eq q′ = eq (TrX _ _ _ (SyX _ _ q) q′)

_□ : (x : El X) → X ∋ x ≈ x
x □ = eq (RfX x)

r≈ : X ∋ x ≈ x
r≈ {x} = eq (RfX x)

qprf : (X : Setoid l) {x y : El X} → X ∋ x ≈ y → Eq X x y
qprf X = qe
```

References

[1] Thorsten Altenkirch, Martin Hofmann, and Thomas Streicher. Categorical reconstruction of a reduction free normalization proof. In David H. Pitt, David E. Rydeheard, and Peter T. Johnstone, editors, *Category Theory and Computer Science, 6th International Conference, CTCS '95, Cambridge, UK, August 7–11, 1995, Proceedings*, volume 953 of *Lecture Notes in Computer Science*, pages 182–199. Springer, 1995.

[2] Thorsten Altenkirch and Bernhard Reus. Monadic presentations of lambda-terms using generalized inductive types. In *Computer Science Logic 1999*, pages 453–468, 1999.

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

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

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

[6] Healfdene Goguen and James McKinna. Candidates for substitution. Technical Report ECS-LFCS-97-358, University of Edinburgh, 1997.

[7] Richard Kennaway and M. Ronan Sleep. Variable abstraction in o(n log n) space. *Inf. Process. Lett.*, 24(5):343–349, 1987.

[8] Conor McBride. *Dependently typed functional programs and their proofs*. PhD thesis, University of Edinburgh, UK, 2000.

[9] Conor McBride and James McKinna. The view from the left. *J. Funct. Program.*, 14(1):69–111, 2004.

[10] Ulf Norell. Dependently typed programming in Agda. In Pieter W. M. Koopman, Rinus Plasmeijer, and S. Doaitse Swierstra, editors, *Advanced Functional Programming, 6th International School, AFP 2008, Heijen, The Netherlands, May 2008, Revised Lectures*, volume 5832 of *Lecture Notes in Computer Science*, pages 230–266. Springer, 2008.