3-for-1 Monadic Notation: Do-it-yourself module types

—ICFP Deadline: March 3, 2020—

Folklore has held that any 'semantic unit' is essentially a type-theoretic context —this includes, for example, records and algebraic datatypes. We provide foundation for such an observation.

We show that languages with a sufficiently powerful type system and reflection mechanism permit a *single declaration interface* for functions, records, type classes, type constructors, and algebraic data types. Moreover, the interface is monadic and thus actually practical to use.

Along the way, we solve the bundling problem: Record fields can be lifted to parameters as needed. Traditionally, unbundling a record requires the use of transport along propositional equalities, with trivial `refl`-exivity proofs. The `:waist` approach presented here removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

A third contribution of this paper is to provide a semantics for the PackageFormer editor extension, which realises the folklore observation by providing users with meta-primitives for making modules to allow arbitrary grouping mechanisms to be derived, such as obtaining the homomorphism type of a given record.

ACM Reference Format:

. 2020. 3-for-1 Monadic Notation: Do-it-yourself module types —ICFP Deadline: March 3, 2020—. 1, 1 (February 2020), 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The humblest notion of a grouping mechanism is described by a pair type $\texttt{A} \times \texttt{B} \times \texttt{C}$, usually later values depend on earlier values and so we have the dependent-pair type $\Sigma\ \texttt{a} : \texttt{A} \bullet \Sigma\ \texttt{b} : \texttt{B}\ \texttt{a} \bullet \Sigma\ \texttt{C}\ \texttt{a}\ \texttt{b}$. The kind of these types is $\texttt{Set}_1$, the type of small types. If we wish to speak of groupings where $\texttt{a} : \texttt{A}$ is *fixed*, then we must lift it from being a *field* component to being a *parameter*, thereby arriving at the *function* $\lambda\ \texttt{a} : \texttt{A} \bullet \texttt{b} : \texttt{B}\ \texttt{a} \bullet \texttt{C}\ \texttt{a}\ \texttt{b}$ which has *type* $\Pi\ \texttt{a} : \texttt{A} \bullet \texttt{Set}$. Similarly, we may expose `b` as a parameter to further indicate the possible grouping structure.

Author's address:

Manuscript submitted to ACM

Grouping Description	Kind
`Σ a : A • Σ b : B a • Σ C a b`	`Set`
`λ a : A • Σ b : B a • Σ C a b`	`Π a : A • Set`
`λ a : A • λ b : B a • Σ C a b`	`Π a : A • Π b : B a • Set`

At each step, we "pull out" more information at the kind level; at first we have a `Set`, an opaque grouping mechanism, then we obtain a `Π a : A • Set` which is a grouping mechanism that somehow makes use of an `A`-value.

(1) **Type constructor reification Π→λ:** Function *types* like `Π a : A • Set` cannot be applied since they are not functions, so how do we get to `λ a : A • Set`?
- λ-terms are values of Π-types, but in general there is no natural construction to transform a type into one of its values.
- Given τ `= Π (X : Set) • ⋯ : Set`$_1$, we want Π→λ τ `= λ (X : Set) • ⋯ : Π (X : Set) • Set`; the former's type states it to be a `Set`$_1$, a grouping mechanism of which we know nothing, whereas the latter's type indicates it to be a parameterised grouping mechanism. Since Π→λ τ can be applied and is thus more concrete, we call Π→λ a reification combinator.

(2) **Unbundling**: How do we go from `Set` to `Π a : A • Set`?
A function from function-types to functions-on-types necessarily requires a way to pattern match on the possible type constructions in a language.
Perhaps an example will clarify the issue. The ubiquitous graph structure is contravariant in its collection of vertices. Recall that a multi-graph, or quiver, is a collection of vertices along with a collection of edges between any two vertices; here's the traditional record form:

```
Graph₀ : Context ℓ₁
Graph₀ = do Vertex ← Set
            Edges  ← (Vertex → Vertex → Set)
            End {ℓ₀}
```

Using the record form, it is akward to phrase contravariance, which simply "relabels the vertices". Even worse, the awkward phrasing only serves to ensure certain constraints hold —which are reified at the value level via the uninsightful `refl`-exivity proof.

```
comap₀ : ∀ {A B : Set}
       → (f : A → B)
       → Σ G : Graph₀ :kind ‘record • Field 0 G ≡ B
       → Σ G : Graph₀ :kind ‘record • Field 0 G ≡ A
comap₀ {A} {B} f (⟨ .B , edgs ⟩ , refl) = (A , (λ a₁ a₂ → edgs (f a₁) (f a₂)) , tt) , refl
```

Without redefining graphs, we can phrase the definition at the typeclass level —i.e., records parameterised by the vertices. This form is not only clearer and easier to implement at the value-level, it also

makes it clear that we are "pulling back" the vertex type and so have also shown graphs are closed under reducts.

```
-- Way better and less akward!
comap : ∀ {A B : Set}
     → (f : A → B)
     → (Graph₀ :kind ‘typeclass) B
     → (Graph₀ :kind ‘typeclass) A
comap f ⟨ edgs ⟩₁ = ⟨ (λ a₁ a₂ → edgs (f a₁) (f a₂)) ⟩₁
```

Later we show how to form `Context`, its do-notation, and the `:kind` mechanism which shifts between records, typeclasses, and algebraic datatypes.

We shall outline how this can be achieved in dependently-typed languages which have support for reflection. Our target language will be Agda, but the ideas easily transfer to other languages. In particular, the resulting in-language syntax we obtain is rather close to the existing Agda record syntax for declarations and Agda constructor tuples for instances. In the next section, we begin by way of a more concrete example of a grouping mechanism, then we take a goal-driven approach to building the necessarily apparatus for a clean imperative-like declaration notation, then we conclude with a brief discussion on how the resulting framework can act as a simple theory for the Agda PackageFormer editor extension.

In order to be language-agnostic and underscore the ideas, we shall present the core definitions along with Agda-checked examples. Details are left to an appendix(?) or can be read below (MA: Haven't decided yet):

https://github.com/alhassy/next-700-module-systems/tree/master/prototype

2 A GROUPING MECHANISM: AUTOMATA

We begin by way of example.

The humblest of automata is a dynamical system —a collection of states, a designated start state, and a transition function. For example, a machine with an initial display and only one button that changes the display. In dependently-typed languages, there are at least three ways to encode such a structure —via records possibly with parameters, which are sometimes called typeclasses.

```
record DynamicSystem₀ : Set₁ where
  field
    States : Set
    start  : States
    next   : States → States

record DynamicSystem₁ (States : Set) : Set where
  field
```

```
    start : States
    next  : States → States

record DynamicSystem₂ (States : Set) (start : States) : Set where
  field
    next : States → States
```

The kinds of these types provide insight into the sort of data they contain:

Type	Kind
`DynamicSystem`$_0$	`Set`$_1$
`DynamicSystem`$_1$	`Π X : Set • Set`
`DynamicSystem`$_2$	`Π X : Set • Π x : X • Set`

Observe that `DynamicSystem`$_i$ is a type constructor of `i`-many arguments.

Yet another way to encode dynamical systems would be by their syntax, as it would be desirable when serialising them —i.e., to obtain first-class descriptions of dynamical system values.

```
data DSTerms₀ : Set where
  start : DSTerms₀
  next  : DSTerms₀ → DSTerms₀

data DSTerms₁ (States : Set) : Set where
  start : States → DSTerms₁ States
  next  : DSTerms₁ States → DSTerms₁ States

data DSTerms₂ (States : Set) (start : States) : Set where
  next : DSTerms₂ States start → DSTerms₂ States start
```

Notice that the first algebraic data type is isomorphic to $\mathbb{N}$, whereas the remaining two are isomorphic to `States` $\times$ $\mathbb{N}$ which keeps track of how many `next` steps are necessary until a `State` value is reached —this may be called `Eventually States`. The `DSTerms`$_i$ share the same pattern of kind exposure as the `DynamicSystem`$_i$ types.

Using monadic notation, we can obtain all of these notions from a single user-friendly context declaration.

3 FROM DO-NOTATION TO (PARAMETERISED) RECORD TYPES

Traditionally a context is a list of name-type pairs, for us it will be a set —namely the product of the types, since the names "do not matter". Moreover, contexts will be have a numeric 'waist' argument that indicates which of the first entries are 'parameters', leaving the remaining elements as 'fields'.

```
-- Contexts are waist-indexed types
```

Table 1. Contexts embody all kinds of grouping mechanisms

Concept	Concrete Syntax	Description
Context	`do S ← Set; s ← X; n ← (X → X); End`	"name-type pairs"
Record Type	`Σ S : Set • Σ s : S • Σ n : S → S • ⊤`	"bundled-up data"
Function Type	`Π S • Σ s : S • Σ n : S → S • ⊤`	"a type of functions"
Type constructor	`λ S • Σ s : S • Σ n : S → S • ⊤`	"a function on types"
Algebraic datatype	`data D : Set where s : D; n : D → D`	"a descriptive syntax"

```
Context = λ ℓ → ℕ → Set ℓ

-- The "empty context" is the unit type
End : ∀ {ℓ} → Context ℓ
End = ‘ ⊤

-- Every type is a context
‘_ : ∀ {ℓ} → Set ℓ → Context ℓ
‘ S = λ _ → S
```

In order to use do-notation we must provide a definition of a bind operator `_>>=_`.

```
do X ← Set
   z ← X
   s ← (X → X)
   End

⟶⟨ Removing syntactic sugar ⟩

  ‘ Set >>= λ X → ‘ X >>= λ z → ‘ (X → X) >>= End
```

Notice the quote method is forced due to the typing of bind: `_>>= : ∀ {X Y} → m X → (X → m Y) → m Y`. The definition of the bind operator accounts for the current waist: If zero, we have records, otherwise functions.

```
_>>=_ : ∀ {a b}
      → (Γ : Context a)
      → (∀ {n} → Γ n → Context b)
      → Context (a ⊎ b)
(Γ >>= f) ℕ.zero  = Σ γ : Γ 0 • f γ 0
(Γ >>= f) (suc n) = (γ : Γ n) → f γ n
```

Unfortunately, this would require too many calls to quote; e.g.,

```
do X ← ‘ Set
   z ← ‘ X
   s ← ‘ (X → X)
   End
```

So let's "build it into the definition" of _>>=_:

```
_>>=_ : ∀ {a b}
      → (Γ : Set a)  -- Main difference
      → (Γ → Context b)
      → Context (a ⊎ b)
(Γ >>= f) ℕ.zero  = Σ γ : Γ • f γ 0
(Γ >>= f) (suc n) = (γ : Γ) → f γ n
```

Let's see this in action, and for variety let's encode monoids.

```
Monoid : ∀ ℓ → Context (ℓsuc ℓ)
Monoid ℓ = do Carrier ← Set ℓ
              Id      ← Carrier
              _⊕_     ← (Carrier → Carrier → Carrier)
              leftId  ← ∀ {x : Carrier} → x ⊕ Id ≡ x
              rightId ← ∀ {x : Carrier} → Id ⊕ x ≡ x
              assoc   ← ∀ {x y z} → (x ⊕ y) ⊕ z  ≡  x ⊕ (y ⊕ z)
              End {ℓ}
```

Likewise, we encode a context `DynamicSystem`, [**?**], which we tabulate its elaboration at particular waists:

Waist	Elaboration
0	`Σ X : Set • Σ z : X • Σ s : (X → X) • ⊤`
1	`Π X : Set • Σ z : X • Σ s : (X → X) • ⊤`
2	`Π X : Set • Π z : X • Σ s : (X → X) • ⊤`
3	`Π X : Set • Π z : X • Π s : (X → X) • ⊤`

Notice that the elaborations are function types, but we want functions *on* types —as is the case with the `DynamicSystem`$_i$ from the introduction.

4 UNBUNDLING: FROM FUNCTION TYPES TO FUNCTIONS *ON* TYPES

Evaluation transforms functions to values and currying reorganises functions, but we want a combinator, call it $\Pi{\to}\lambda$, that takes a type and results in a value of that type. In general, this is not feasible when the type is empty nor is it naturally canonical when there are multiple possible values to choose from.

One could use a universe, an algebraic type of codes denoting types, to define $\Pi{\to}\lambda$. However, one can no longer then easily use existing types since they are not formed from the universe's constructors, thereby resulting in duplication of existing types via the universe encoding. This is not practical nor pragmatic.

As such, we are left with pattern matching on the language's type formation primitives as the only reasonable approach. The method Π→λ is thus a macro that acts on the syntactic term representations of types.

```
Π→λ (Π a : A • Ba) = (λ a : A • Ba)
Π→λ τ              = τ  {- otherwise -}
```

Similarly, `_:waist_` is a macro acting on contexts that repeats Π→λ a number of times in order to lift a number of field components to the parameter level.

```
τ :waist n     = Π→λ^n n (τ n)

Π→λ^n 0       τ = τ
Π→λ^n (n + 1) τ = Π→λ^n n (Π→λ τ)
```

Let's see this in action. Here are our dynamical systems.

```
DynamicSystem : Context (ℓsuc Level.zero)
DynamicSystem = do X ← Set
                   s ← X
                   n ← (X → X)
                   End {Level.zero}
```

Then using our macros:

```
DynamicSystem 1        ≡  Π X : Set • Σ s : X  • Σ n : X → X  • ⊤
DynamicSystem :waist 1 ≡  λ X : Set • Σ s : X  • Σ n : X → X  • ⊤
```

Each type exposes more and more information about what kind of grouping structure we have at hand. The definitions could not be simpler.

```
A' : Set₁
B' : Π X : Set • Set
C' : Π X : Set • Π x : X • Set
D' : Π X : Set • Π x : X • Π s : (X → X) • Set

A' = DynamicSystem :waist 0
B' = DynamicSystem :waist 1
C' = DynamicSystem :waist 2
D' = DynamicSystem :waist 3
```

If the language allows mixfix unicode identifiers, then one declares grouping mechanisms with `do ⋯ End` then forms instances using, say, ⟨⋯⟩.

```
-- Helpful syntactic sugar
⟨ : ∀ {ℓ} {S : Set ℓ} → S → S
```

```
⟨ s = s

_⟩ : ∀ {ℓ} {S : Set ℓ} → S → S × ⊤ {ℓ}
s ⟩ = s , tt

⟨⟩ : ∀ {ℓ} → ⊤ {ℓ}
⟨⟩ = tt
```

The following *instances* of these grouping types demonstrate how *information moves from the body level to the parameter level.*

```
𝒩⁰ : A'
𝒩⁰ = ⟨ ℕ , 0 , suc ⟩

𝒩¹ : B' ℕ
𝒩¹ = ⟨ 0 , suc ⟩

𝒩² : C' ℕ 0
𝒩² = ⟨ suc ⟩

𝒩³ : D' ℕ 0 suc
𝒩³ = ⟨⟩
```

It is interesting to note, that if a context C has only n-many fields, then there are only n-many interesting unbundled forms, after which there are no new ones: `C (n + k) ≡ C n`.

With `:waist` we can fix parameters ahead of time. For example, above the type `B' ℕ` is the type of "dynamic systems over carrier ℕ" whereas `C' ℕ 0` is the type of "dynamic systems over carrier ℕ and start state 0". Without the unbundling mechanism we would have had to resort to awkward trivial constraints, as below, which are tolerable for one-off uses but clearly do not scale at all as indicated by the need to use equals-for-equals `subst`-itutions of propositional equalities.

```
C'' : Π X : Set • Π x : X • Set₁
C'' X x = Σ 𝒟 : DynamicSystem 0
        • Σ Carrier-is-X : proj₁ 𝒟 ≡ X
        • proj₁ (proj₂ 𝒟) ≡ subst id (sym Carrier-is-X) x

𝒩²eek : C'' ℕ 0
𝒩²eek = (ℕ , 0 , suc , tt) , refl , refl
```

Traditionally, unbundling a record requires the use of transport along propositional equalities, with trivial `refl`-exivity proofs. The `:waist` approach presented here removes the boilerplate necessary at the type specialisation location as well as at the instance declaration location.

5 SEMANTICS FOR PACKAGEFORMER

The PackageFormer editor extension reads contexts —in nearly the same notation as above— enclosed in dedicated comments, then generates and imports Agda code from them seamlessly in the background whenever typechecking transpires. The framework provides a fixed number of meta-primitives for producing arbitrary notions of grouping mechanisms, and allows arbitrary Emacs Lisp to be invoked in the construction of complex grouping mechanisms.

One of PackageFormer's primitives is called `:waist` and behaves exactly as ours above. As such, our current setup provides a formalisation of PackageFormer limited to only the `:waist` meta-primitive. Moreover, it is nearly as readable and is a library method, rather than an editor extension.

PackageFormer is extensible via Emacs Lisp and one of its possible uses is to obtain algebraic data types from a context. With Agda's current reflection mechanism, even this is possible! For example, we may obtain a type D from `DynamicSystem` with user-defined constructors `zeroD` and `sucD` as if it where defined:

```
data D : Set where
     zeroD : D
     sucD  : D → D
```

Here are the core pieces necessary to form `termtype`; obtained by viewing an algebraic data-type as a fixed-point of the functor obtained from union of the sources of its constructors. For example, the above D is the fixpoint of $\lambda\ D \to 1 \uplus D$, where the summands are the sources of D's constructors.

```
⇊ τ = "reduce all de brujin indices by 1"

Σ→⊎ (Σ a : A • Ba) = A ⊎ Σ→⊎ (⇊ Ba)
{- Extend 'Σ→⊎' homomorphicly to other syntactic constructs -}

sources (λ x : (Π a : A • Ba) • ⋯) = (λ x : A • ⋯)
sources (λ x : A              • ⋯) = (λ x : ⊤ • ⋯)
{- Extend 'sources' homomorphicly to other syntactic constructs -}

data Fix (F : Set → Set) : Set where
  μ : F (Fix F) → Fix F

termtype τ = Fix (Σ→⊎ (sources τ))
```

One can then declare D `= termtype (DynamicSystem :waist 1)`.

With `termtype` in hand, we have provided a theoretical basis for yet another meta-primitive of PackageFormer, the `_:kind_` primitive which dictates how an abstract grouping mechanism should be viewed in terms of existing Agda syntax. However, unlike PackageFormer all of our syntax is legitimate Agda syntax.

```
data Kind : Set where
  ‘record    : Kind
  ‘typeclass : Kind
  ‘data      : Kind
```

Since syntax is being manipulated, we have yet another macro:

```
C :kind ‘record    = C 0
C :kind ‘typeclass = C :waist 1
C :kind ‘typeclass = termtype (C :waist 1)
```

Interestingly, useful programming datatypes arise from termtypes of theories (contexts). That is, if C : `Set` $\to$ `Context` ℓ_0 then C`’ = λ X →` C `X :kind ‘data` can be used to form ‘free, lawless, C-instances’. For example,

Theory	Termtype
Dynamical Systems	$\mathbb{N}$
Pointed Structures	Maybe
Monoids	Binary Trees

The final correspondence in the table is a claim mentioned briefly in the first PackageFormer paper. With our setup we can not only formally express the relationship but also prove it true. We present the setup and leave it as a tremendously easy exercise to the reader to present a bijective pair of functions between M and `TreeSkeleton`. Hint: Interactively case-split on values of M until the declared patterns appear, then replace them with the constructors of `TreeSkeleton`.

```
M : Set
M = termtype (Monoid ℓ₀ :waist 1)

-- Pattern synonyms for more compact presentation
pattern emptyM      = μ (inj₁ tt)                       -- : M
pattern branchM l r = μ (inj₂ (inj₁ (l , r , tt)))      -- : M → M → M
pattern absurdM a   = μ (inj₂ (inj₂ (inj₂ (inj₂ a))))  -- absurd values of 0

data TreeSkeleton : Set where
  empty  : TreeSkeleton
  branch : TreeSkeleton → TreeSkeleton → TreeSkeleton
```

To obtain trees over some 'value type' Ξ, one must start at the theory of "monoids containing a given set Ξ". Similarly, by starting at "theories of pointed sets over a given set Ξ", the resulting termtype is the Maybe type constructor —another simple exercise to the reader: Show $P \cong$ Maybe.

```
PointedOver  : Set → Context (ℓsuc ℓ₀)
PointedOver Ξ    = do Carrier ← Set ℓ₀
                      point   ← Carrier
                      embed   ← (Ξ → Carrier)
                      End

P : Set → Set
P X = termtype (PointedOver X :waist 1)

-- Pattern synonyms for more compact presentation
pattern nothingP = μ (inj₁ tt)       -- : P
pattern justP e  = μ (inj₂ (inj₁ e)) -- : P → P
```

For PackageFormer, we have implemented its primitives `:waist` and `:kind`, the other core meta-primitives are `_⊕→_` and `:alter-elements`. The former is a syntactic form of function application, `x ⊕→ f ≈ f x`, which we already have by juxtaposition in Agda. The latter, however, is a "hammer" that alters the constituents of a grouping mechanism in an arbitrary fashion using the entire power of Emacs Lisp —which includes a large portion of Common Lisp. We have currently presented a partial semantics of PackageFormer's syntactic entities by presenting them here as semantic functions on contexts.

6 WHAT ABOUT THE META-LANGUAGE'S PARAMETERS?

Besides `:waist`, another way to introduce parameters into a context grouping mechanism is to use the language's existing utility of parameterising a context by another type —as was done earlier in `PointedOver`.

For example, a pointed set needn't necessarily be termined with `End`.

```
PointedSet : Context ℓ₁
PointedSet = do Carrier ← Set
                point   ← Carrier
                End {ℓ₁}
```

We instead form a grouping consisting of a single type and a value of that type, along with an instance of the parameter type Ξ.

```
PointedPF : (Ξ : Set₁) → Context ℓ₁
PointedPF Ξ = do Carrier ← Set
                 point   ← Carrier
                 ‘ Ξ
```

Clearly `PointedPF ⊤ ≈ PointedSet`, so we have a more generic grouping mechanism. The natural next step is to consider other parameters such as `PointedSet` in-place of `Ξ`.

```
-- Convenience names
PointedSetᵣ = PointedSet        :kind ‘record
PointedPFᵣ  = λ Ξ → PointedPF Ξ :kind ‘record

-- An extended record type: Two types with a point of each.
TwoPointedSets = PointedPFᵣ PointedSetᵣ

_ :   TwoPointedSets
    ≡ ( Σ Carrier₁ : Set • Σ point₁ : Carrier₁
      • Σ Carrier₂ : Set • Σ point₂ : Carrier₂ • ⊤)
_ = refl

-- Here's an instance
one : PointedSet :kind ‘record
one = 𝔹 , false , tt

-- Another; a pointed natural extended by a pointed bool,
-- with particular choices for both.
two : TwoPointedSets
two = ℕ , 0 , one
```

More generally, *record **structure** can be dependent on values:*

```
_PointedSets : ℕ → Set₁
zero  PointedSets = ⊤
suc n PointedSets = PointedPFᵣ (n PointedSets)

_ :   4 PointedSets
    ≡ (Σ Carrier₁ : Set • Σ point₁ : Carrier₁
      • Σ Carrier₂ : Set • Σ point₂ : Carrier₂
      • Σ Carrier₃ : Set • Σ point₃ : Carrier₃
      • Σ Carrier₄ : Set • Σ point₄ : Carrier₄ • ⊤)
_ = refl
```

Using traditional grouping mechanisms, it is difficult to create the family of types `n PointedSets` since the number of fields, $2 \times n$, depends on n.

It is interesting to note that the termtype of `PointedPF` is the same as the termtype of `PointedOver`, the `Maybe` type constructor!

```
PointedD : (X : Set) → Set₁
PointedD X = termtype (PointedPF (Lift _ X) :waist 1)

-- Pattern synonyms for more compact presentation
pattern nothingP = μ (inj₁ tt)
pattern justP x  = μ (inj₂ (lift x))

casingP : ∀ {X} (e : PointedD X)
        → (e ≡ nothingP) ⊎ (Σ x : X • e ≡ justP x)
casingP nothingP  = inj₁ refl
casingP (justP x) = inj₂ (x , refl)
```

7 NEXT STEPS

We have shown how a bit of reflection allows us to have a compact, yet practical, one-stop-shop notation for records, typeclasses, and algebraic data types. There are a number of interesting directions to pursue:

- How to write a function working homogeneously over one variation and having it lift to other variations.
  - Recall the `comap` from the introductory section was written over `Graph :kind ‘typeclass`; how could that particular implementation be massaged to work over `Graph :kind` k for any k.
- The current implementation for deriving termtypes presupposes only one carrier set positioned as the first entity in the grouping mechanism.
  - How do we handle multiple carriers or choose a carrier from an arbitrary position or by name? `PackageFormer` handles this by comparing names.
- How do we lift properties or invariants, simple ≡-types that ‘define’ a previous entity to be top-level functions in their own right?

Lots to do, so little time.