

# 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.

An application of our setup will be to provide a semantics for the `PackageFormer` editor extension, which realises the aforementioned 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), 15 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

## 1 INTRODUCTION

In dependently-typed programming languages, such as Agda [3, 9], there is a tendency to define concepts repeatedly along syntactic constructs provided by the language. In particular, one bundles up related data into a record structure, then considers the need to expose some of the fields as parameters and so provides a parameterised record construction, then for the need to have a description language for terms of these record types, one forms an associated algebraic datatype. For example, we may form a type  $\text{Monoid}_0$  of monoids, which consists of a type along with an operation and some laws, but may want  $\text{Monoid}_1 \ M \oplus$  to speak of monoids over *particular* types  $M$  and particular operations  $\oplus$  —the latter is handled, say in the Haskell standard library, by having isomorphic copies of types for each binary operation, such as `Sum`

---

Author’s address:

---

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from [permissions@acm.org](mailto:permissions@acm.org).

© 2020 Association for Computing Machinery.

Manuscript submitted to ACM

Manuscript submitted to ACM

$\cong \text{Prod} \cong \text{Int}$  for the classical additive and multiplicative monoidal structures on integers. This is the problem we are solving: *How can parameterised records and their associated algebraic datatypes be obtained from a core declaration?*

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

Grouping Description	Kind
$\Sigma a : A \bullet \Sigma b : B a \bullet \Sigma C a b$	$\text{Set}$
$\lambda a : A \bullet \Sigma b : B a \bullet \Sigma C a b$	$\Pi a : A \bullet \text{Set}$
$\lambda a : A \bullet \lambda b : B a \bullet \Sigma C a b$	$\Pi a : A \bullet \Pi b : B a \bullet \text{Set}$

At each step, we “pull out” more information at the kind level; at first we have a  $\text{Set}$ , an opaque grouping mechanism, then we obtain a  $\Pi a : A \bullet \text{Set}$  which is a grouping mechanism that somehow makes use of an  $A$ -value.

- (1) **Type constructor reification**  $\Pi \rightarrow \lambda$ : Function *types* like  $\Pi a : A \bullet \text{Set}$  cannot be applied since they are not functions, so how do we get to  $\lambda a : A \bullet \text{Set}$ ?
  - $\lambda$ -terms are values of  $\Pi$ -types, but in general there is no natural construction to transform a type into one of its values.
  - Given  $\tau = \Pi (X : \text{Set}) \bullet \dots : \text{Set}_1$ , we want  $\Pi \rightarrow \lambda \tau = \lambda (X : \text{Set}) \bullet \dots : \Pi (X : \text{Set}) \bullet \text{Set}$ ; the former’s type states it to be a  $\text{Set}_1$ , a grouping mechanism of which we know nothing, whereas the latter’s type indicates it to be a parameterised grouping mechanism. Since  $\Pi \rightarrow \lambda \tau$  can be applied and is thus more concrete, we call  $\Pi \rightarrow \lambda$  a reification combinator.
- (2) **Unbundling** [4]: How do we go from  $\text{Set}$  to  $\Pi a : A \bullet \text{Set}$ ?

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:

```

Graph  : Context  $\ell_1$ 
Graph  = do Vertex  $\leftarrow \text{Set}$ 
          Edges  $\leftarrow (\text{Vertex} \rightarrow \text{Vertex} \rightarrow \text{Set})$ 
          End  $\{\ell_0\}$ 

```

Using the record form, it is awkward 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 un insightful `refl`-exivity proof.

```
comap0 : ∀ {A B : Set}
  → (f : A → B)
  → ∑ G : Graph : kind 'record • Field 0 G ≡ B
  → ∑ G : Graph : kind 'record • Field 0 G ≡ A
```

```
comap0 {A} {B} f (⟨ .B , eds ⟩ , refl) = (A , (λ a1 a2 → eds (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 awkward!
```

```
comap : ∀ {A B : Set}
  → (f : A → B)
  → (Graph : kind 'typeclass) B
  → (Graph : kind 'typeclass) A
comap f ⟨ eds ⟩1 = ⟨ (λ a1 a2 → eds (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.

It is important to note that we are using the word ‘typeclass’ as an abbreviation for “parameterised record”. In particular, we have no support for the traditional unification algorithm that makes typeclasses and canonical structures [7] useful for ad-hoc polymorphism.

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 [2] —which solves the `Monoidi` problem mentioned earlier.

In order to be language-agnostic and underscore the ideas, we shall present the core definitions along with Agda-checked examples. Details can be read at the following URL in a literate and reproducible fashion [10].

<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 <sub>0</sub>	Set <sub>1</sub>
DynamicSystem <sub>1</sub>	$\prod X : \text{Set} \bullet \text{Set}$
DynamicSystem <sub>2</sub>	$\prod X : \text{Set} \bullet \prod x : X \bullet \text{Set}$

Observe that DynamicSystem<sub>i</sub> 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 DTerms0 : Set where
```

```
  start : DTerms0
```

```
  next   : DTerms0 → DTerms0
```

```
data DTerms1 (States : Set) : Set where
```

```
  start : States → DTerms1 States
```

```
  next  : DTerms1 States → DTerms1 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  $\text{States} \times \mathbb{N}$  which keeps track of how many next steps are necessary until a State value is reached —this may be called `Eventually States`. The  $\text{DSTerms}_i$  share the same pattern of kind exposure as the  $\text{DynamicSystem}_i$  types.

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

Table 1. Contexts embody all kinds of grouping mechanisms

Concept	Concrete Syntax	Description
Context	<code>do S ← Set; s ← X; n ← (X → X); End</code>	“name-type pairs”
Record Type	$\Sigma S : \text{Set} \bullet \Sigma s : S \bullet \Sigma n : S \rightarrow S \bullet \top$	“bundled-up data”
Function Type	$\Pi S \bullet \Sigma s : S \bullet \Sigma n : S \rightarrow S \bullet \top$	“a type of functions”
Type constructor	$\lambda S \bullet \Sigma s : S \bullet \Sigma n : S \rightarrow S \bullet \top$	“a function on types”
Algebraic datatype	<code>data D : Set where s : D; n : D → D</code>	“a descriptive syntax”

### 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 have a numeric ‘waist’ argument that indicates which of the first entries are ‘parameters’, leaving the remaining elements as ‘fields’. The subtlety of what is a ‘parameter’ —exposed at the type level— and what is a ‘field’ —a component value— has led to awkward formulations and the duplication of existing types for the sole purpose of different uses. We shall aim toward a monadic [8] interface to declare such grouping mechanisms.

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

```
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:  $\_>>= : \forall \{X\ Y\} \rightarrow m\ X \rightarrow (X \rightarrow m\ Y) \rightarrow m\ Y$ . 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	$\Sigma X : \text{Set} \bullet \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \top$
1	$\Pi X : \text{Set} \bullet \Sigma z : X \bullet \Sigma s : (X \rightarrow X) \bullet \top$
2	$\Pi X : \text{Set} \bullet \Pi z : X \bullet \Sigma s : (X \rightarrow X) \bullet \top$
3	$\Pi X : \text{Set} \bullet \Pi z : X \bullet \Pi s : (X \rightarrow X) \bullet \top$

Notice that the elaborations are function types, but we want functions *on* types —as is the case with the `DynamicSystemi` 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  $\Pi \rightarrow \lambda$ , 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  $\Pi \rightarrow \lambda$ . 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  $\Pi \rightarrow \lambda$  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  $\Pi \rightarrow \lambda$  a number of times in order to lift a number of field components to the parameter level.

```

τ :waist n      = Π → λn n (τ n)

```

```

Π → λn 0      τ = τ
Π → λn (n + 1) τ = Π → λn 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 • T
DynamicSystem :waist 1 ≡ λ X : Set • Σ s : X • Σ n : X → X • T

```

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 × T {ℓ}
s > = s , tt

<> : ∀ {ℓ} → T {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 >

```



$$\mathcal{N}^2 : C' \ \mathbb{N} \ 0$$

$$\mathcal{N}^2 = \langle \text{suc} \rangle$$

$$\mathcal{N}^3 : D' \ \mathbb{N} \ 0 \ \text{suc}$$

$$\mathcal{N}^3 = \langle \rangle$$

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:  $C \ (n + k) \equiv C \ n$ .

With `:waist` we can fix parameters ahead of time. For example, above the type  $B' \ \mathbb{N}$  is the type of “dynamic systems over carrier  $\mathbb{N}$ ” whereas  $C' \ \mathbb{N} \ 0$  is the type of “dynamic systems over carrier  $\mathbb{N}$  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-ituitions of propositional equalities.

$$C'' : \Pi X : \mathbf{Set} \bullet \Pi x : X \bullet \mathbf{Set}_1$$

$$C'' \ X \ x = \Sigma \ \mathcal{D} : \mathbf{DynamicSystem} \ 0$$

- $\Sigma \text{Carrier-is-}X : \text{proj}_1 \ \mathcal{D} \equiv X$
- $\text{proj}_1 \ (\text{proj}_2 \ \mathcal{D}) \equiv \text{subst id (sym Carrier-is-}X) \ x$

$$\mathcal{N}^{\text{seek}} : C'' \ \mathbb{N} \ 0$$

$$\mathcal{N}^{\text{seek}} = (\mathbb{N} , \ 0 , \ \text{suc} , \ \text{tt}) , \ \text{refl} , \ \text{refl}$$

Traditionally [6], 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 [1, 2] 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 [5] 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 were 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  $\lambda D \rightarrow 1 \uplus D$ , where the summands are the sources of  $D$ 's constructors.

$\Downarrow \tau$  = “reduce all de brujin indices by 1”

```

Σ→⊔ (Σ a : A • Ba) = A ⊔ Σ→⊔ (⊔ Ba)
{- Extend 'Σ→⊔' homomorphically to other syntactic constructs -}

```

```

sources (λ x : (Π a : A • Ba) • ...) = (λ x : A • ...)
sources (λ x : A • ...) = (λ x : T • ...)
{- Extend 'sources' homomorphically 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  $D = \text{termtype } (\text{DynamicSystem} : \text{waist } 1)$ .

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  $C : \text{Set} \rightarrow \text{Context } \ell_0$  then  $C' = \lambda X \rightarrow C X : \text{kind 'data}$  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  $\text{TreeSkeleton}$ . Hint: Interactively case-split on values of  $M$  until the declared patterns appear, then replace them with the constructors of  $\text{TreeSkeleton}$ .

$M : \text{Set}$

$M = \text{termtype } (\text{Monoid } \ell_0 : \text{waist } 1)$

-- Pattern synonyms for more compact presentation

```

pattern emptyM      =  $\mu$  (inj1 tt)                -- : M
pattern branchM l r =  $\mu$  (inj2 (inj1 (l , r , tt))) -- : M → M → M
pattern absurdM a   =  $\mu$  (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’  $\Xi$ , one must start at the theory of “monoids containing a given set  $\Xi$ ”. Similarly, by starting at “theories of pointed sets over a given set  $\Xi$ ”, the resulting termtype is the Maybe type constructor —another simple exercise to the reader: Show  $P \cong \text{Maybe}$ .

**PointedOver** : Set → Context ( $\ell \text{ suc } \ell_0$ )

```

PointedOver  $\Xi$  = do Carrier ← Set  $\ell_0$ 
               point   ← Carrier
               embed    ← ( $\Xi \rightarrow \text{Carrier}$ )
               End

```

$P : \text{Set} \rightarrow \text{Set}$

$P X = \text{termtype } (\text{PointedOver } X : \text{waist } 1)$

-- Pattern synonyms for more compact presentation

```

pattern nothingP =  $\mu$  (inj1 tt)                -- : P
pattern justP e   =  $\mu$  (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,  $x \oplus f \approx f\ x$ , 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 terminated 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  $\Xi$ .

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

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

```
-- 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 • T )
_ = refl

-- Here's an instance
Manuscript submitted to ACM
```

```
one : PointedSet :kind 'record
one =  $\mathbb{B}$  , false , tt
```

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

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

```
_PointedSets :  $\mathbb{N} \rightarrow \text{Set}_1$ 
zero PointedSets = T
suc n PointedSets = PointedPFr (n PointedSets)

_ : 4 PointedSets
≡ (  $\sum$  Carrier1 : Set •  $\sum$  point1 : Carrier1
    •  $\sum$  Carrier2 : Set •  $\sum$  point2 : Carrier2
    •  $\sum$  Carrier3 : Set •  $\sum$  point3 : Carrier3
    •  $\sum$  Carrier4 : Set •  $\sum$  point4 : Carrier4 • T )
_ = 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)  $\rightarrow$  Set1
PointedD X = termtype (PointedPF (Lift _ X) :waist 1)
```

```
-- Pattern synonyms for more compact presentation
```

```
pattern nothingP =  $\mu$  (inj1 tt)
pattern justP x =  $\mu$  (inj2 (lift x))
```

```
casingP :  $\forall \{X\}$  (e : PointedD X)
 $\rightarrow$  (e  $\equiv$  nothingP)  $\uplus$  ( $\sum$  x : X • e  $\equiv$  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  $\equiv$ -types that ‘define’ a previous entity to be top-level functions in their own right?

Lots to do, so little time.

## REFERENCES

- [1] Musa Al-hassy. The next 700 module systems: Extending dependently-typed languages to implement module system features in the core language, 2019. URL <https://alhassy.github.io/next-700-module-systems-proposal/thesis-proposal.pdf>.
- [2] Musa Al-hassy, Jacques Carette, and Wolfram Kahl. A language feature to unbundle data at will (short paper). In Ina Schaefer, Christoph Reichenbach, and Tijs van der Storm, editors, *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences, GPCE 2019, Athens, Greece, October 21-22, 2019*, pages 14–19. ACM, 2019. ISBN 978-1-4503-6980-0. doi: 10.1145/3357765.3359523. URL <https://doi.org/10.1145/3357765.3359523>.
- [3] Ana Bove, Peter Dybjer, and Ulf Norell. A brief overview of Agda — A functional language with dependent types. In *Theorem Proving in Higher Order Logics, 22nd International Conference, TPHOLs 2009, Munich, Germany, August 17–20, 2009. Proceedings*, pages 73–78, 2009. doi: 10.1007/978-3-642-03359-9\_6.
- [4] François Garillot, Georges Gonthier, Assia Mahboubi, and Laurence Rideau. Packaging Mathematical Structures. In Tobias Nipkow and Christian Urban, editors, *Theorem Proving in Higher Order Logics*, volume 5674 of *Lecture Notes in Computer Science*, Munich, Germany, 2009. Springer. URL <https://hal.inria.fr/inria-00368403>.
- [5] Paul Graham. *ANSI Common Lisp*. Prentice Hall Press, USA, 1995. ISBN 0133708756.
- [6] Jason Gross, Adam Chlipala, and David I. Spivak. Experience Implementing a Performant Category-Theory Library in Coq, 2014.
- [7] Assia Mahboubi and Enrico Tassi. Canonical Structures for the working Coq user. In Sandrine Blazy, Christine Paulin, and David Pichardie, editors, *ITP 2013, 4th Conference on Interactive Theorem Proving*, volume 7998 of *LNCS*, pages 19–34, Rennes, France, July 2013. Springer. doi: 10.1007/978-3-642-39634-2\_5. URL <https://hal.inria.fr/hal-00816703>.
- [8] Eugenio Moggi. Notions of computation and monads. *Inf. Comput.*, 93(1):55–92, 1991. doi: 10.1016/0890-5401(91)90052-4. URL [https://doi.org/10.1016/0890-5401\(91\)90052-4](https://doi.org/10.1016/0890-5401(91)90052-4).
- [9] Ulf Norell. *Towards a Practical Programming Language Based on Dependent Type Theory*. PhD thesis, Dept. Comp. Sci. and Eng., Chalmers Univ. of Technology, September 2007.
- [10] Luka Stanisic and Arnaud Legrand. Effective reproducible research with org-mode and git. In Luís M. B. Lopes, Julius Zilinskas, Alexandru Costan, Roberto G. Cascella, Gabor Kecsckemeti, Emmanuel Jeannot, Mario Cannataro, Laura Ricci, Siegfried Benkner, Salvador Petit, Vittorio Scarano, José Gracia, Sascha Hunold, Stephen L. Scott, Stefan Lankes, Christian Lengauer, Jesús Carretero,

Manuscript submitted to ACM

Jens Breitbart, and Michael Alexander, editors, *Euro-Par 2014: Parallel Processing Workshops - Euro-Par 2014 International Workshops, Porto, Portugal, August 25-26, 2014, Revised Selected Papers, Part I*, volume 8805 of *Lecture Notes in Computer Science*, pages 475–486. Springer, 2014. ISBN 978-3-319-14324-8. doi: 10.1007/978-3-319-14325-5\_41. URL [https://doi.org/10.1007/978-3-319-14325-5\\_41](https://doi.org/10.1007/978-3-319-14325-5_41).