A Language Feature to Unbundle Data at Will (Short Paper) [1]

Musa Al-hassy, Jacques Carette, Wolfram Kahl

McMaster University, Hamilton, Ontario, Canada
{alhassy|carette|kahl}@mcmaster.ca

GPCE 2019, Athens, Greece
21st October 2019

[1] This research is supported by NSERC (National Science and Engineering Research Council of Canada).

Which Category Should I use?

"A category consists of a collection of *objects*, a collection of *morphisms*, an operation ...":

```
record Category (i j k : Level) : Set (suc (i ⊍ j ⊍ k)) where
  field Obj : Set i
        Hom : Obj → Obj → Setoid j k
  Mor = (λ A B → Setoid.Carrier (Hom A B)) : Obj → Obj → Set j
  field _⨾_ : {A B C : Obj} → Mor A B → Mor B C → Mor A C
        Id  : {A : Obj} → Mor A A
```

"A category over a given collection Obj of *objects*, with Hom providing *morphisms*, is given by defining an operation ...":

```
record Category′ {i j k : Level} {Obj : Set i} (Hom : Obj → Obj → Setoid j k) : Set (i ⊍ j ⊍ k) where
  Mor = (λ A B → Setoid.Carrier (Hom A B)) : Obj → Obj → Set j
  field _⨾_ : {A B C : Obj} → Mor A B → Mor B C → Mor A C
        Id  : {A : Obj} → Mor A A
```

Tom Hales (of Kepler conjecture / Flyspeck fame) about Lean:

"Structures are meaninglessly parameterized from a mathematical perspective. [...] I think of the parametric versus bundled variants as analogous to currying or not; are the arguments to a function presented in succession or as a single ordered tuple? However, there is a big difference between currying functions and currying structures. Switching between curried and uncurried functions is cheap, but it is nearly impossible in Lean to curry a structure. That is, what is bundled cannot be later opened up as a parameter. (Going the other direction towards increased bundling of structures is easily achieved with sigma types.) This means that library designers are forced to take a conservative approach and expose as a parameter anything that any user might reasonably want exposed, because once it is bundled, it is not coming back."

Tom Hales, 2018-09-18 blog post

This is the problem we are solving!

Library Design

- **Goals:**
  - Reusability
  - Generality
  - (Mathematical) "Naturality"
- **Result: Conflict of Interests:**

  For a record type bundling up items that "naturally" belong together:

  - **Which parts of that record should be parameters?**
  - **Which parts should be fields?**

Candidate Types for Monoids

An arbitrary monoid:

```
record Monoid₀
  : Set₁ where
  field
    Carrier : Set
    _⨾_    : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: The category of monoids.

A monoid **over** type Carrier:

```
record Monoid₁
    (Carrier : Set)
  : Set where
  field
    _⨾_    : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: Sharing the carrier type.

Candidate Types for Monoids (2)

An arbitrary monoid:

```
record Monoid₀
  : Set₁ where
  field
    Carrier : Set
    _⨾_    : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: The category of monoids.

A monoid over Carrier with operation ⨾:

```
record Monoid₂
    (Carrier : Set)
    (_⨾_ : Carrier → Carrier → Carrier)
  : Set where
  field
    Id      : Carrier
    assoc : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: Additive monoid of integers

Related Problem: Control over Parameter Instantiation

Instances of Haskell typeclasses

- are indexed by **types** only
- so that there can be only one Monoid instance for Bool

Crude solution: Isomorphic copies with different type **name**:

```
data Bool = False | True

newtype All = All {getAll :: Bool}    -- for Monoid instance based on conjunction

newtype Any = Any {getAny :: Bool}    -- for Monoid instance based on disjunction
```

Which Items should be Fields? Which Items should be Parameters?

- Monoid_0, Monoid_1, and Monoid_2 showed some combinations of items selected as parameters.
- There are other combinations of what is to be exposed and hidden, for applications that we might never think of.
- Providing always the most-general parameterisation produces **awkward** library interfaces!

Proposed Solution:

- Commit to no particular formulation and allow on-the-fly "unbundling"

  — This is the **converse** of instantiation
- **New language feature:** PackageFormer

The Definition of a Monoid, and Recreating Monoid_0

```
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x
```

- We regain the different candidates by applying Variationals

```
Monoid₀′ = MonoidP record
```

An arbitrary monoid:

```
record Monoid₀
  : Set₁ where
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc   : ∀ {x y z}
            → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: The category of monoids.

The Definition of a Monoid, and Recreating Monoid_1

```
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x
```

- We regain the different candidates by applying Variationals

```
Monoid₁′  = MonoidP record ⟴ unbundled 1
Monoid₁″  = Monoid₀′ exposing (Carrier)
```

A monoid **over** type Carrier:

```
record Monoid₁
    (Carrier : Set)
  : Set where
  field
    _⨾_     : Carrier → Carrier → Carrier
    Id      : Carrier
    assoc   : ∀ {x y z}
            → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: Sharing the carrier type.

The Definition of a Monoid, and Recreating Monoid_2

```
PackageFormer MonoidP
  : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x
```

- We regain the different versions by applying Variationals

```
Monoid₂′  = MonoidP record ⟴ unbundled 2
Monoid₂′  = MonoidP record ⟴ exposing (Carrier; _⨾_)
Monoid₂″  = Monoid₀′ exposing (Carrier; _⨾_)
```

A monoid over type Carrier with operation ⨾:

```
record Monoid₂
    (Carrier : Set)
    (_⨾_ : Carrier → Carrier → Carrier)
  : Set where
  field
    Id      : Carrier
    assoc   : ∀ {x y z}
            → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
    leftId  : ∀ {x} → Id ⨾ x ≡ x
    rightId : ∀ {x} → x ⨾ Id ≡ x
```

Use-case: Additive monoid of integers

The Definition of a Monoid, and Instantiations

```
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x
```

- We regain the different candidates by applying Variationals
- **Linear** effort in number of variations

```
Monoid₀′  = MonoidP record
Monoid₁′  = MonoidP record ⟴ unbundled 1
Monoid₂″  = Monoid₀′ exposing (Carrier; _⨾_)
```

Monoid Syntax

```
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x
```

- . . . and we can do more

```
Monoid₃′ = MonoidP termtype "Carrier"
```

```
data Monoid₃ : Set where
  _⨾_ : Monoid₃ → Monoid₃ → Monoid₃
  Id : Monoid₃
```

```
Monoid₄ = MonoidP
  termtype-with-variables "Carrier"
```

```
data Monoid₄ (Var : Set) : Set where
  inj : Var → Monoid₄ Var
  _⨾_ : Monoid₄ Var
      → Monoid₄ Var → Monoid₄ Var
  Id  : Monoid₄ Var
```

The Language of Variationals

```
Variational ≅ (PackageFormer → PackageFormer)
```

```
id                       : Variational
_⟴_                      : Variational → Variational → Variational
record                   : Variational
termtype                 : String → Variational
termtype-with-variables  : String → Variational
unbundled                : ℕ → Variational
exposing                 : List Name → Variational
```

Variational Polymorphism

```
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z}
          → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x} → Id ⨾ x ≡ x
  rightId : ∀ {x} → x ⨾ Id ≡ x

  concat : List Carrier → Carrier
  concat = foldr _⨾_ Id
```

- Items with default definitions get adapted types

```
Monoid₀′ = MonoidP record
Monoid₁′ = MonoidP record ⟴ unbundled 1
Monoid₂″ = Monoid₀′ exposing (Carrier; _⨾_)
Monoid₃′ = MonoidP termtype "Carrier"
```

```
concat₀ : {M : Monoid₀}
  → let C = Monoid₀.Carrier M
    in List C → C

concat₁ : {C : Set} {M : Monoid₁ C}
  → List C → C

concat₂ : {C : Set} {_⨾_ : C → C→ C}
  {M : Monoid₂ C _⨾_}
  → List C → C

concat₃ : let C = Monoid₃
  in List C → C
```

How Does This Work?

- Currently implemented as an "editor tactic" meta-program
- Using the "default IDE" of Agda: Emacs
- Implementation is an **extensible** library built on top of 5 meta-primitives
- Generated Agda file is automatically imported into the current file
- Special-purpose IDE support

Generated Code Displayed on Hover

```
{-700
PackageFormer M-Set : Set₁ where
   Scalar  : Set
   Vector  : Set
   _·_     : Scalar → Vector → Vector
   1       : Scalar
   _×_     : Scalar → Scalar → Scalar
   leftId  : {𝓋 : Vector}  →  1 · 𝓋  ≡  𝓋
   assoc   : ∀ {a b 𝓋} → (a × b) · 𝓋  ≡  a · (b · 𝓋)

NearRing = M-Set record ⟴ single-sorted "Scalar"
-}
```

```
{- NearRing = M-Set record ⟴ single-sorted "Scalar" -}
record NearRing : Set₁ where
  field Scalar        : Set
  field _·_       : Scalar → Scalar → Scalar
  field 1         : Scalar
  field _×_       : Scalar → Scalar → Scalar
  field leftId        : {𝓋 : Scalar} → 1 · 𝓋 ≡ 𝓋
  field assoc         : ∀ {a b 𝓋} → (a × b) · 𝓋 ≡ a · (b · 𝓋)
```

Future Work

- Explicit (elaboration) semantics for PackageFormers and Variationals within a minimal type theory
  - Refactor meta-primitives from LISP flavour to Agda flavour
  - Integrate with a reflection interface for Agda
- Explore multiple default definitions
- Explore inheritance, coercion, and transport along canonical isomorphisms
- Generate mutually-recursive definitions for certain instances of many-sorted PackageFormers?

Conclusion

- Naming, terminology, concrete syntax, combinator interfaces are all still in flux!
- The present system already allows to replace hand-written instances of structuring schemes with invocations of (generative) library methods
- We already influenced the naming conventions of the Agda "standard library"
- Our approach based on PackageFormers and Variationals makes it possible
  - to codify, name, and document "design patterns" of uses of structuring mechanisms
  - to enable and encourage re-use at a high level of abstraction
  - to drastically reduce the interface size of "interface libraries"

  and therewith has the potential to

  drastically change how we provide and use structures via libraries