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}@cas.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))
  field Obj : Set i
        Hom : Obj → Obj → Setoid j k
  Mor : Obj → Obj → Set j
  Mor = λ A B → Setoid.Carrier (Hom A B)
  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 : Obj → Obj → Set j
  Mor = λ A B → Setoid.Carrier (Hom A B)
  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:
  When creating a record to bundle up certain information that “naturally” belongs together, **what parts of that record should be parameters and what 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 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

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 Parameters?

- There are other combinations of what is to be exposed and hidden, for applications that we might never think of.
- What to do?
- 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

```
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

$$
\begin{aligned}
&\textbf{PackageFormer}\ \mathsf{MonoidP} : \mathsf{Set}_1\ \textbf{where}\\
&\quad \mathsf{Carrier} : \mathsf{Set}\\
&\quad \_⨾\_ \ : \mathsf{Carrier} \to \mathsf{Carrier} \to \mathsf{Carrier}\\
&\quad \mathsf{Id} \ : \mathsf{Carrier}\\
&\quad \mathsf{assoc} : \forall \{x\ y\ z\}\\
&\qquad \to (x ⨾ y) ⨾ z \equiv x ⨾ (y ⨾ z)\\
&\quad \mathsf{leftId} : \forall \{x\} \to \mathsf{Id} ⨾ x \equiv x\\
&\quad \mathsf{rightId} : \forall \{x\} \to x ⨾ \mathsf{Id} \equiv x
\end{aligned}
$$

- We regain the different candidates by applying Variationals

$$
\begin{aligned}
\mathsf{Monoid}_1{}' &= \mathsf{MonoidP}\ \textbf{record} \mathbin{\oplus\!\!\to} \mathsf{unbundled}\ 1\\
\mathsf{Monoid}_1{}'' &= \mathsf{Monoid}_0{}'\ \mathsf{exposing}\ (\mathsf{Carrier})
\end{aligned}
$$

A monoid **over** type Carrier:

$$
\begin{aligned}
&\textbf{record}\ \mathsf{Monoid}_1\\
&\qquad (\mathsf{Carrier} : \mathsf{Set})\\
&\quad : \mathsf{Set}\ \textbf{where}\\
&\quad \textbf{field}\\
&\qquad \_⨾\_ \ : \mathsf{Carrier} \to \mathsf{Carrier} \to \mathsf{Carrier}\\
&\qquad \mathsf{Id} \ : \mathsf{Carrier}\\
&\qquad \mathsf{assoc} : \forall \{x\ y\ z\}\\
&\qquad\quad \to (x ⨾ y) ⨾ z \equiv x ⨾ (y ⨾ z)\\
&\qquad \mathsf{leftId} : \forall \{x\} \to \mathsf{Id} ⨾ x \equiv x\\
&\qquad \mathsf{rightId} : \forall \{x\} \to x ⨾ \mathsf{Id} \equiv x
\end{aligned}
$$

Use-case: Sharing the carrier type.

The Definition of a Monoid

```
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

```
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

PackageFormer MonoidP : Set_1 **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

$\mathsf{Monoid}_0{}'$ = MonoidP **record**
$\mathsf{Monoid}_1{}'$ = MonoidP **record** ⟴ unbundled 1
$\mathsf{Monoid}_2{}''$ = $\mathsf{Monoid}_0{}'$ exposing (Carrier; _⨾_)

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

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

The Language of Variationals

$$\mathsf{Variational} \quad \cong \quad (\mathsf{PackageFormer} \to \mathsf{PackageFormer})$$

id	:	Variational
_⟴_	:	Variational → Variational → Variational
record	:	Variational
termtype	:	String → Variational
termtype-with-variables	:	String → Variational
unbundled	:	$\mathbb{N}$ → 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?

- Implemented our system 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 Visualised 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

- Provide explicit (elaboration) semantics for PackageFormer within a minimal type theory.
- Explain how generative modules are supported by this scheme.
- . How do multiple default, or optional, clauses for a constituent fit into this language feature.
- Explore inheritance, coercion, and transport along canonical isomorphisms.

Conclusion

- Our resulting system has turned hand-written instances of structuring schemes from a design pattern into full-fledged library methods
- textsfPackageFormers and Variationals have the potential to dramatically change the way we write instances of structuring mechanisms: Giving names and documentation to recurring patterns and reusing them where needed.
- Naming/terminology, concrete syntax, and combinator interfaces are still tentative!