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

Musa Al-hassy
alhassy@gmail.com
McMaster University
Hamilton, Canada

Jacques Carette
carette@mcmaster.ca
McMaster University
Hamilton, Canada

Wolfram Kahl
kahl@cas.mcmaster.ca
McMaster University
Hamilton, Canada

Abstract

Programming languages with sufficiently expressive type systems provide users with different means of data 'bundling'. Specifically, in dependently-typed languages such as Agda, Coq, Lean and Idris, one can choose to encode information in a record either as a parameter or a field. For example, we can speak of graphs *over* a particular vertex set, or speak of arbitrary graphs where the vertex set is a component. These create isomorphic types, but differ with respect to intended use. Traditionally, a library designer would make this choice (between parameters and fields); if a user wants a different variant, they are forced to build conversion utilities, as well as duplicate functionality. For a graph data type, if a library only provides a Haskell-like typeclass view of graphs *over* a vertex set, yet a user wishes to work with the category of graphs, they must now package a vertex set as a component in a record along with a graph over that set.

We design and implement a language feature that allows both the library designer and the user to make the choice of information exposure only when necessary, and otherwise leave the distinguishing line between parameters and fields unspecified. Our language feature is currently implemented as a prototype meta-program incorporated into Agda's Emacs ecosystem, in a way that is unobtrusive to Agda users.

CCS Concepts • **Software and its engineering → Extensible languages**; **Modules / packages**; *Functional languages*; *Polymorphism*; *Source code generation*; *Integrated and visual development environments.*

Keywords Agda, meta-program, extensible, Emacs, packages, modules, dependent-types

GPCE '19, October 21–22, 2019, Athens, Greece

ACM ISBN 978-1-4503-6980-0/19/10…$15.00
https://doi.org/10.1145/3357765.3359523

ACM Reference Format:
Musa Al-hassy, Jacques Carette, and Wolfram Kahl. 2019. A Language Feature to Unbundle Data at Will (Short Paper). In *Proceedings of the 18th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE '19), October 21–22, 2019, Athens, Greece.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3357765.3359523

1 Introduction — Selecting the 'Right' Perspective

Library designers want to produce software components that are useful for the perceived needs of a variety of users and usage scenarios. It is therefore natural for designers to aim for substantial generality, in the hopes of increased reusability. One such particular "choice" will occupy us here: 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*? This is analogous to whether functions are curried and so arguments may be provided partially, or otherwise must be provided all-together in one tuple.

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. Tom Hales [5] is quite eloquent in his critique of 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.*

This is the problem we are solving.

For example, each Haskell typeclass can have only one instance per datatype; since there are several monoids with the datatype `Bool` as carrier, in particular those induced by conjunction and disjunction, the de-facto-standard libraries for Haskell define two isomorphic copies `All` and `Any` of `Bool`, only for the purpose of being able to attach the respective monoid instances to them.

But perhaps Haskell's type system does not give the programmer sufficient tools to adequately express such ideas. As such, for the rest of this paper we will illustrate our ideas in Agda [2, 7]. For the monoid example, it seems that there are three contenders for the monoid interface:

```
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

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

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
```

In $\texttt{Monoid}_0$, we will call `Carrier` "bundled up", while we call it "exposed" in $\texttt{Monoid}_1$ and $\texttt{Monoid}_2$. The bundled-up version allows us to speak of *a* monoid, rather than *a monoid on a given type* which is captured by $\texttt{Monoid}_1$. While $\texttt{Monoid}_2$ exposes both the carrier and the composition operation, we might in some situation be interested in exposing the identity element instead — e.g., the discrepancy '≢' and indistinguishability '≡' operations on the Booleans have the same identities as conjunction and disjunction, respectively. Moreover, there are other combinations of what is to be exposed and hidden, for applications that we might never think of.

Rather than code with *interface formulations we think people will likely use*, we can instead try to *commit to no particular formulation* and allow the user to select the form most convenient for their use-cases. This desire for reusability motivates a new language feature: The PackageFormer.

Moreover, it is often the case that one begins working with a record of useful semantic data, but then, say, for proof automation, may want to use the associated datatype for syntax. For example, the syntax of closed monoid terms can be expressed, using trees, as follows.

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

We can see that this can be obtained from $\texttt{Monoid}_0$ by discarding the fields denoting equations, then turning the remaining fields into constructors.

We show how these different presentations can be derived from a *single* PackageFormer declaration via a generative meta-program integrated into the most widely-used Agda "IDE", the Emacs mode for Agda. In particular, if one were to explicitly write M different bundlings of a package with N constants then one would write nearly $N \times M$ lines of code, yet this quadratic count becomes linear $N + M$ by having a single package declaration of N constituents with M subsequent instantiations. We hope that reducing such duplication of effort, and of potential maintenance burden, will be beneficial to the software engineering of large libraries of formal code — and consider it the main contribution of our work.

2 PackageFormers — Being Non-committal as Much as Possible

We claim that the above monoid-related pieces of Agda code can be unified as a single declaration which does not distinguish between parameters and fields, where PackageFormer is a keyword with similar syntax as `record`:

```
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
```

(For clarity, this and other non-native Agda syntax is left uncoloured.)

Then, with various directives that let one declare what should be parameters and what should be fields, we can reproduce the above presentations. The directives can be built from the following grammar:

```
id          :  Variational
record      :  Variational
```

```
typeclass   :  Variational
termtype    :  String → Variational
unbundled   :  ℕ → Variational
exposing    :  List Name → Variational
_⊕>_        :  Variational
            →  Variational → Variational
```

A package former is used via *instantiations*, written as low-precedence juxtapositions of a package former name and expression of type `Variational`. Notice that some variationals have arguments. For example, `exposing` explicitly lists the names that should be turned into parameters, in that sequence, "`unbundled` n" exposes the first n names declared in the package former.

An *instantiation* juxtaposition is written $\mathsf{PF}\ v$ to indicate that the PackageFormer named PF is to be restructred according to scheme v. A *composition* of variationals is denoted using the symbol '$\oplus\!\!>$'; for example,

$$\mathsf{PF}\ v_1 \oplus\!\!> v_2 \oplus\!\!> \cdots \oplus\!\!> v_n$$

denotes the forward-composition of iterated instantiations, namely $(((\mathsf{PF}\ v_1)\ v_2)\ \cdots)\ v_n$, since we take prefix instantiation application to have higher precedence than variational composition. In particular, an empty composition is the identity scheme, which performs no alteration, and has the explicit name `id`. Since $\mathsf{PF}\ \mathsf{id} \approx \mathsf{PF}$ and `id` is the identity of composition, we may write any *instantiation* as a sequence of $\oplus\!\!>$-separated clauses:

$$\mathsf{PF} \oplus\!\!> v_1 \oplus\!\!> v_2 \oplus\!\!> \cdots \oplus\!\!> v_n$$

The previous presentations can be obtained as follows.

0. To make `Monoid₀'` the type of *arbitrary monoids* (that is, with arbitrary carrier), we declare:

   ```
   Monoid₀'  = MonoidP record
   ```

1. We may obtain the previous formulation of `Monoid₁` in two different equivalent ways:

   ```
   Monoid₁'  = MonoidP record ⊕> unbundled 1
   Monoid₁'' = Monoid₀' exposing (Carrier)
   ```

   It is interesting to note that *PackageFormer* `MonoidP` is treated on the same footing as *record* `Monoid₀'`: Both may be subjected to variationals.

2. As with `Monoid₁`, there are also different ways to obtain `Monoid₂`.

   ```
   Monoid₂'  = MonoidP record ⊕> unbundled 2
   Monoid₂'' = Monoid₀' exposing (Carrier; _⨾_)
   ```

3. Metaprogramming is clearly needed to produce the term language:

   ```
   Monoid₃' = MonoidP termtype "Carrier"
   ```

   Our running example uses the theory of monoids, which is a single-sorted theory. In general, a PackageFormer may have multiple sorts — as is the case with graphs — and so one of the possibly many sorts needs to be designated as the universe of discourse, or carrier, of the resulting inductively defined term type. Such a purpose is served by the single argument to `termtype`.

   We may also want to have terms *over* a particular variable set, and so declare the following after extending the system with a user defined variational `termtype-with-variables`.

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

   Since a parameter's name does not matter, due to α-equivalence, an arbitrary, albeit unique, name for the variable set is introduced along with an embedding function from it to the resulting term type. For brevity, the embedding function's name is `inj` and the user must ensure there is no name clash. The resulting elaboration is as follows.

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

   Note that these instantiations implicitly drop equations, such as associativity from `MonoidP`. This is what is commonly done in Universal Algebra. If we were instead doing n-category theory, these would be kept, but will be the subject of future work.

We also have elaborations into nested dependent-sums, which is useful when looking at coherent substructures. Alongside `unbundled`, we also have infix combinators for extending an instantiation with additional fields or constructors, and the renaming of constituents according to a user provided String-to-String function. Moreover, just as syntactic datatype declarations may be derived, we also allow support for the derivation of induction principles and structure-preserving homomorphism types. Our envisioned system would be able to derive simple, tedious, uninteresting concepts; leaving difficult, interesting ones for humans to solve.

> **Quadratic to Linear:** Notice that the previous 5 monoid presentations, `Monoid₀` to `Monoid₄`, spanned 32 lines (8 for the original, 24 for the variants). Using `MonoidP` and our operators, this can be done in 7 + 6 = 13 lines. This corresponds to using a 2-part code, with the initial lines being a model, and then 1-2 lines to specify variants.

3 Variational Polymorphism

Suppose we want to produce the function `concat`, which folds over the elements of a list according to a compositionality scheme — examples of this include summing over a list, multiplication over a list, checking all items in a list are true, or at least one item in the list is true. Depending on the selected instantiation, the resulting function may have types such as the following:

```
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
```

Given our previous work, and providing that the variationals are already defined, we add a new declaration which, unlike the rest, comes equipped with a *definition.*

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

This is known as a *definitional extension* (of a theory), which is known to be conservative (i.e. has the same models).

The variationals is where this power comes from. Furthermore, we have alluded to the fact that the type of variationals is extensible; this is achieved by having

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

Indeed, our implementation relies on 5 meta-primitives to form arbitrarily complex schemes that transform abstract PackageFormers into other grouping mechanisms. The meta-primitives were arrived at by codifying a number of structuring mechanisms directly then carefully extracting the minimal ingredients that enable them to be well-defined.

4 How Does This Work?

We have implemented our system as an "editor tactic" meta-program.

In actual use, an Agda programmer declares what they want using the combinators above (inside special Agda code comments). The comments are read by Emacs Lisp and legitimate Agda is produced in a generated file, which is then automatically imported into the current file — examples are provided in an appendix. The generated file never needs to be consulted, as the declared names are furnished with tooltips rendering the elaborated Agda form, see Figure 1. Moreover, we also provide a feature to extract a 'bare bones' version of a file that strips out all PackageFormer annotations, leaving only Agda as well as the import to the generated file. Finally, since the elaborations are just Agda, one only needs to use the system once and future users are not forced to know about it.

The existing prototype already has the following nice properties:

Extensible Users may extend the collection of variationals by providing the intended elaboration scheme. We have provided a number of auxiliary, derived, combinators that can be used to construct complex and common schemes. Furthermore, the user has full and direct access to the entirety of Emacs Lisp as a programming language for restructuring PackageFormers into any desired shape — the well-formedness of which is a matter the user must then worry about.

Practical The user manual demonstrates how boilerplate code for renamings, hidings, decorations, and generations of hierarchical structures can be formed; [3].

Pragmatic The prototype comes equipped with a number of menus to display the abstract PackageFormer's defined, as well as the variationals defined, and one may enable highlighting for these syntactical items, have them folded away, or simply extract an Agda file that does not mention them at all.

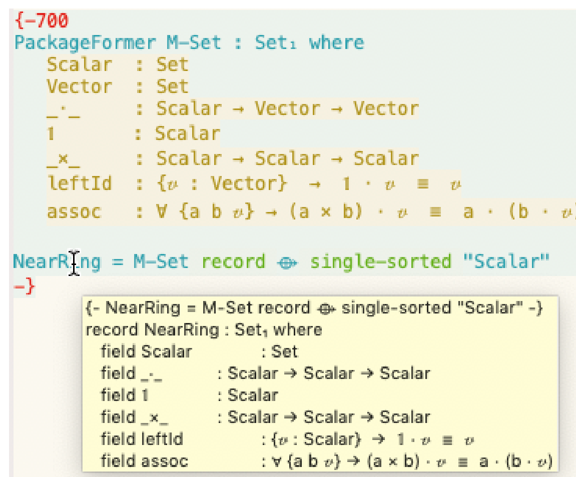

Figure 1. Hovering to show details. Notice special syntax has default colouring: Red for PackageFormer delimiters, yellow for elements, and green for variationals.

The details of the implementation and numerous common structuring mechanisms derived from the meta-primitives can be found on the prototype's homepage:

https://alhassy.github.io/next-700-module-systems/prototype/package-former.html

5 Conclusion and Next Steps

We have outlined a new language feature that is intended to reduce duplicated effort involved in taking different perspectives on structures — and to solve Hales' problem of premature commitment to a particular encoding. Moreover, on the road to making this tractable, we have unearthed a novel form of polymorphism and demonstrated its usefulness with some examples.

We have presented our work indirectly by using examples, which we hope are sufficiently clear to indicate our intent. We next intend to provide explicit (elaboration) semantics for PackageFormer within a minimal type theory; [4].

Furthermore, there are additional pieces of future work, including:

1. Explain how generative modules [6] are supported by this scheme.
2. How do multiple default, or optional, clauses for a constituent fit into this language feature.
3. Explore inheritance, coercion, and transport along canonical isomorphisms.

Finally, the careful reader will have noticed that our abstract mentions graphs, yet there was no further discussion of that example. We have avoided it for simplicity; the prototype accommodates multi-sorted structures where sorts may *depend* on one another, as edge-sets depend on the vertex-set chosen. Examples can be found on the prototype's webpage.

This short paper proposes a language feature that enables users to selectively choose how information is to be organised, such as which parts are exposed as parameters, thereby reducing effort when taking different perspectives on structures. To demonstrate that this feature seems useful in practice, we have implemented a meta-program to generate Agda using special code comments that specify how package elements are to be organised, such as their selective exposure as parameters which is a common issue with libraries in dependently-typed languages.

Our variationals cannot yet be directly defined in Agda. Instead, we are making use of Emacs Lisp, a language close to the Agda ecosystem. Going forward, one of the aims of our work is to have variationals definable directly within Agda — rather than having our users learn yet another language. Our exploratory efforts suggest that we may be able to realise PackageFormers as Agda records of 'elements' —a tuple of qualifier, name, type, and definitional clauses— and, so, the result is a conservative extension to Agda's underlying type theory. However, from a practical standpoint, it is highly likely that we will extend Agda to support the new syntax.

Our resulting system has turned hand-written instances of structuring schemes from a design pattern into full-fledged library methods. In turn, the system addresses the following extremely unsatisfactory points of hand-written instances, mentioned by the "Deriving Via" [1] group:

1. *It is not obvious that we are instantiating a general principle.*
2. *Because the general principle is not written down in code with a name and documentation, it has to be communicated through folklore or in comments and is difficult to discover and search for. Our code has lost a connection to its origin.*
3. *There are many such rules, some quite obvious, but others more surprising and easy to overlook.*
4. *While the work required to define instances manually for* `Monoid` *—which only has 6 constituents— is perhaps acceptable, it quickly becomes extremely tedious and error-prone for packages with many constituents.*

Paraphrasing [1], we believe that PackageFormers have the potential to dramatically change the way we write instances of structuring mechanisms, as it encourages giving names and documentation to recurring patterns and reusing them where needed.

A Appendices

Full code scripts may be found on the prototype's repository; below are snippets for the presented fragments.

A.1 Module Header

```
open import Data.List hiding (concat)
open import Relation.Binary.PropositionalEquality
            using (_≡_)

module gpce19 where

open import gpce19-generated
```

The import of the generated file is automatically produced and inserted by the system, if need be.

A.2 Plain `MonoidP` PackageFormer

```
{-700
PackageFormer MonoidP : Set₁ where
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)
  leftId  : ∀ {x : Carrier} → Id ⨾ x ≡ x
  rightId : ∀ {x : Carrier} → x ⨾ Id ≡ x
-}
```

A.3 Variational `record` and 3 Instantiations

In the paper proper we mentioned "unbundled", which in the prototype takes the form of the meta-primitive `:waist`.

```
{-lisp
(𝒱 record
  = "Reify a variational as an Agda “record”.

     Elements with equations are construed as
     derivatives of fields  ---the elements
     without any equations.
    "
  :kind record
  :alter-elements
     (λ es → (--map (map-qualifier
```

```
        (λ _ → (unless (element-equations it)
           "field")) it) es)))
-}
```

Notice that the organisational mechanism not only has a name and documentation, but also an unambiguous implementation. We may use it as follows.

```
{-700
Monoid₀'  = MonoidP record
Monoid₁'' = MonoidP record ⟴ :waist 1
Monoid₂'' = MonoidP record ⟴ :waist 2
-}
```

A.4 termtype Variationals

We may also have shorter variational definitions directly in `700`-blocks.

```
{-700
𝒱-termtype                carrier  =  ⋯
𝒱-termtype-with-variables carrier  =  ⋯

Monoid₃' = MonoidP termtype "Carrier"
Monoid₄  = MonoidP termtype-with-variables "Carrier"
-}
```

A.5 PackageFormers with Equations

```
{-700
PackageFormer MonoidPE : Set₁ where
  -- A few declarations
  Carrier : Set
  _⨾_     : Carrier → Carrier → Carrier
  Id      : Carrier
  assoc   : ∀ {x y z} → (x ⨾ y) ⨾ z ≡ x ⨾ (y ⨾ z)

  -- A few declarations with equations
  Rid : Carrier → Carrier
  Rid x = x ⨾ Id
  concat : List Carrier → Carrier
  concat = foldr _⨾_ Id

  -- More declarations
  leftId  : ∀ {x : Carrier} → Id ⨾ x ≡ x
  rightId : ∀ {x : Carrier} → Rid x ≡ x
-}
```

A.6 $\texttt{concat}_0$ and $\texttt{concat}_3$

```
{-700
𝒱-decorated by = ⋯
Monoid⁰ = MonoidPE decorated "⁰" ⟴ record
Monoid³ = MonoidPE ⟴ decorated "³"
                   ⟴ termtype "Carrier³"
-}
```

Then, concatenation over an arbitrary monoid:

```
concat₀ : {M : Monoid⁰}
        → let C = Monoid⁰.Carrier⁰ M
          in List C → C
concat₀ {M} = Monoid⁰.concat⁰ M
```

As well as, concatenation over an arbitrary **closed** monoid term:

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

References

[1] Baldur Blöndal, Andres Löh, and Ryan Scott. Deriving via: or, how to turn hand-written instances into an anti-pattern. In *Proceedings of the 11th ACM SIGPLAN International Symposium on Haskell, Haskell@ICFP 2018, St. Louis, MO, USA, September 27-17, 2018*, pages 55–67, 2018. doi: 10.1145/3242744.3242746. URL https://doi.org/10.1145/3242744.3242746.

[2] 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.

[3] Jacques Carette and Russell O'Connor. Theory presentation combinators. *Intelligent Computer Mathematics*, page 202–215, 2012. doi: 10.1007/978-3-642-31374-5_14.

[4] Derek Dreyer, Karl Crary, and Robert Harper. A type system for higher-order modules. In *Conference Record of POPL 2003: The 30th SIGPLAN-SIGACT Symposium on Principles of Programming Languages, New Orleans, Louisisana, USA, January 15-17, 2003*, pages 236–249, 2003. doi: 10.1145/640128.604151. URL https://doi.org/10.1145/640128.604151.

[5] Tom Hales. A review of the lean theorem prover, 2018. URL https://jiggerwit.wordpress.com/2018/09/18/a-review-of-the-lean-theorem-prover/.

[6] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3): 269–303, 2000. doi: 10.1017/S0956796800003683.

[7] 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.