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

Musa Al-hassy, Jacques Carette, Wolfram Kahl

Abstract

Programming languages with sufficiently expressive type theories provide users with different means of data 'bundling'. Specifically one can choose to encode information in a record either as a parameter or a field, in dependently-typed languages such as Agda, Coq, Lean and Idris. 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.

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 a high-level of 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.

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 [Norell(2007), Bove et al.(2009)Bove, Dybjer, and Norell]. 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*, it is far more general 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 is formalised, using trees, as follows.

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

We can see that this version can also be mechanically obtained from $\texttt{Monoid}_0$ by discarding 'non-simple' fields then turning the remaining fields into constructors.

Conference'17, Washington, DC, USA
2019. 978-x-xxxx-xxxx-x/YY/MM…$15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

We show how all these different presentations can be derived from a *single* PackageFormer declaration via a generative metaprogram integrated into the most widely used Agda "IDE", the Emacs mode for Agda. In particular, a package of N constituents with M presentations of bundling results in 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. It is this massive reduction in duplicated efforts and maintenance that we view as the main contribution of our work.

2 PackageFormers — Being non-committal as much as possible

We claim that the previous monoid-related pieces of Agda code can all 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
```

Coupled with various directives that let one declare what should be parameters and what should be fields, we can reproduce the above presentations. A package former is used via *instantiations*, written as low-precedence juxtapositions of a package former name and expression of type `Variational`. The latter can be built in particular via the following:

```
id          :  Variational
record      :  Variational
typeclass   :  Variational
termtype    :  Variational
unbundled   :  ℕ → Variational
exposing    :  List Name → Variational
_-⊕→_       :  Variational → Variational → Variational
```

The variationals `unbundled` and `exposing` have arguments. While `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 `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 '$\text{-}\oplus\!\!\rightarrow$'; for example, `PF` v_1 $\text{-}\oplus\!\!\rightarrow$ v_2 $\text{-}\oplus\!\!\rightarrow$ $\cdots$ $\text{-}\oplus\!\!\rightarrow$ v_n denotes the forward-composition of iterated instantiations, namely $(((\texttt{PF}\ v_1)\ v_2)\ \cdots)\ v_n$, since we take prefix instantiation application to have lower precedence that variational composition. In particular, an empty composition is the identity scheme, which performs no alteration, and has the explicit name `id`. Since `PF id` $\approx$ `PF` and `id` is the identity of composition, we may write any *instantiation* as a sequence of $\text{-}\oplus\!\!\rightarrow$-separated clauses: `PF` $\text{-}\oplus\!\!\rightarrow$ v_1 $\text{-}\oplus\!\!\rightarrow$ v_2 $\text{-}\oplus\!\!\rightarrow$ $\cdots$ $\text{-}\oplus\!\!\rightarrow$ v_n —which is equivalent to `PF` $((((\texttt{id}\ \text{-}\oplus\!\!\rightarrow\ v_1)\ \text{-}\oplus\!\!\rightarrow\ v_2)\ \cdots))\ v_n$.

The previous presentations can be obtained as follows.

0. To make $\texttt{Monoid}_0$`'` the type of *arbitrary monoids* (that is, with arbitrary carrier), we declare:

   ```
   Monoid₀'  = MonoidP record
   ```

1. We may obtain the previous formulation of $\texttt{Monoid}_1$ as follows:

   ```
   Monoid₁'  = MonoidP record -⊕→ exposing (Carrier)
   Monoid₁'' = MonoidP record -⊕→ unbundled 1
   ```

2. As for $\texttt{Monoid}_1$, there are also different ways to regain the previous formulation of $\texttt{Monoid}_2$.

   ```
   Monoid₂'  = MonoidP record -⊕→ unbundled 2
   Monoid₂'' = MonoidP
                 record -⊕→ exposing (Carrier; _⨟_)
   ```

3. Finally, we mentioned metaprogramming's need to work with terms:

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

   Our main example is the theory of monoids, which are single-sorted. However, a general 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. This is accomplished with the `:carrier` argument.

   Of course we may want to have terms *over* a particular variable set, and so declare:

   ```
   Monoid₄ = MonoidP termtype-with-variables
               :carrier "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 essentially is as follows.

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

   Note that only 'functional' symbols have been exposed in these instantiations; no 'proof-matter' such as the associativity consistent declared in `MonoidP`.

For brevity, we have only discussed product representations of packages, however the language feature also supports elaborations into nested dependent-sums as in the case where we may have a coherent substructure. 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 4 monoid presentations, $\texttt{Monoid}_i$, had at least 2 constituents each totalling more than $4 \times 2 = 16$ lines of user input. However, using `MonoidP`, above we have obtained the same presentations, $\texttt{Monoid}_i$`'`, using

one definition having at least 2 lines and 4 single-line declarations, for a total of at least 4 + 2 = 8 lines of user input.

The PackageFormer declarations are not legal Agda syntax and as such appear in dedicated 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. Currently, the user enters commands into special comments from which legitimate Agda is produced in an auxiliary 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, legitimate, Agda form. Moreover, we also provide a feature to extract a 'bare bones' version of a file that strips out all PackageFormer annotations, leaving only legitimate Agda as well as the import to the generated file. Finally, since the elaborations are legitimate Agda, one only needs to utilise the system once and future users are not forced to know about it.

The PackageFormer language feature unifies disparate representations of the same concept under a single banner. How does one actually *do* anything with these entities? Are we forced to code along particular instantiations? No; unless we desire to do so.

3 Variational Polymorphism

Suppose we want to produce the function `concat`, which composes 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 : List Monoid₃ → Monoid₃
```

An attempt to unify these declarations is trivial —provided the variationals are already defined— as one merely appends the aforementioned PackageFormer definition with a new declaration that, unlike the rest, comes equipped with an *equation*.

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

The real magic is the variationals. We have alluded that the type of variationals is extensible and this is achieved by having `Variational ≅ (PackageFormer → PackageFormer)`. Indeed, our implementation relies on 5 meta-primitives to form arbitrary and complex schemes to transforming 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. This approach is reminiscent to that of Haskell's typeclasses.

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-proposal/prototype/PackageFormer.html

It is important at this juncture to observe that the type of `concatP` depends crucially on the variational that is invoked.

4 Next Steps

We have outlined a new unifying language feature that is intended to massively reduce duplicated efforts involving different perspectives of datatypes. Moreover, to make this tractable we have also provided a novel form of polymorphism and demonstrated it with minimal examples.

We have implemented this 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), and these are then elaborated into Agda code.

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; [Dreyer et al.(2003)Dreyer, Crary, and Harper].

Moreover there are a number of auxiliary goals, including:

1. Explain how generative modules [Leroy(2000)] 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.

However, the features of the existing prototype more than make up for the system's shortcoming. The features list currently include:

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. In doubt, 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; [Carette and O'Connor(2012)].

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 folded away, or simply extract an Agda file that does not mention them at all.

Moreover, it can be tedious to consult generated code for high-level PackageFormer instantiations and so every variational and PackageFormer is tagged with tooltips providing relevant information.

Finally, the careful reader will have noticed that our abstract mentions graphs, yet there was no further discussion on that example.

We have avoided it for simplicity only. 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 has proposed a unifying language feature that enables users to selectively choose how information is to be organised, such as which parts are exposed as parameters, thereby reducing duplicated efforts involving different perspectives of datatypes. To demonstrate that the proposed feature is promising, 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.

As a prototype witnessing that a generic packaging formalism is possible, 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 PackageFormer's 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.

> *Structuring schemes tend to be easy to explain, yet the benefit of our system is that it transports them from design patterns to full-fledged library methods.*

References

[Bove et al.(2009)Bove, Dybjer, and Norell] 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.

[Carette and O'Connor(2012)] Jacques Carette and Russell O'Connor. Theory presentation combinators. *Intelligent Computer Mathematics*, page 202–215, 2012. ISSN 1611-3349. doi: 10.1007/978-3-642-31374-5_14. URL http://dx.doi.org/10.1007/978-3-642-31374-5_14.

[Dreyer et al.(2003)Dreyer, Crary, and Harper] 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.

[Leroy(2000)] Xavier Leroy. A modular module system. *J. Funct. Program.*, 10(3):269–303, 2000. URL http://journals.cambridge.org/action/displayAbstract?aid=54525.

[Norell(2007)] 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. See also http://wiki.portal.chalmers.se/agda/pmwiki.php.

5 Appendix: Source code

Below is a nearly self-contained source sample for the presented fragments. We have omitted some variational definitions, using $\cdots$, since they offer little insight but their definitions may be involved.

Module Header

```
open import Relation.Binary.PropositionalEquality using (_≡_)
open import Data.List hiding (concat)
module Paper0 where
{- Automatically generated & inserted by the prototype -}
open import Paper0_Generated
```

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
-}
```

The record variational and three instantiations

```
{-700
𝒱-record = :kind record :waist-strings ("field")

Monoid₀'  = MonoidP record
Monoid₁'' = MonoidP record ⊕ :waist 1
Monoid₂'' = MonoidP record ⊕ :waist 2
-}
```

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

Complex variationals in `lisp` blocks

```
{-lisp
(𝒱 termtype carrier
  = "Reify as parameterless Agda "data" type.

     CARRIER refers to the sort that is designated as the
     domain of discourse of the resulting single-sorted
     inductive term data type.
    "
    :kind data
    :level dec
    :alter-elements (lambda (fs)
      (thread-last fs
        (--filter (s-contains? carrier (target (get-type it))))
        (--map (map-type (s-replace carrier $name type) it)))))

(𝒱 termtype-with-variables carrier = ⋯) -}

{-700
Monoid₃' = MonoidP termtype :carrier "Carrier"
Monoid₄  = MonoidP termtype-with-variables :carrier "Carrier"
-}
```

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
-}
```

concat_0

```
{-lisp
(𝒱 record_e
  = "Record variation with support for equations."
    ⋯)

(𝒱 decorated  by  = ⋯) -}

{-700
Monoid⁰ = MonoidPE decorated :by "⁰" ⊕ record_e
-}

{- "Concatenation over an arbitrary monoid" -}
concat₀ : {M : Monoid⁰}
        → let C = Monoid⁰.Carrier⁰ M
          in List C → C
concat₀ {M} = Monoid⁰.concat⁰ M
```

concat_3

```
{-lisp
(𝒱 termtype_e carrier = ⋯) -}

{-700
Monoid³ = MonoidPE ⊕ decorated :by "³"
                   ⊕ termtype_e :carrier "Carrier³"
-}

{- Concatenation over an arbitrary *closed* monoid term -}
concat₃ : let C = Monoid³
          in List C → C
concat₃ = concat³
```