A Language Feature to Unbundle Data at Will

MUSA AL-HASSY, JACQUES CARETTE, WOLFRAM KAHL

ACM Reference format:
Musa Al-hassy, Jacques Carette, Wolfram Kahl. YYYY. A Language Feature to Unbundle Data at Will. VV, NNN, Article AA (MM YYYY), 5 pages.
DOI: 10.1145/nnnnnnn.nnnnnnn

Abstract

Programming languages with sufficiently expressive type theories provide users with essentially two levels of data 'bundling'. This applies in particular to dependently-typed languages such as Agda, Coq, and Idris, where one can choose to make certain constituents of a record either parameters or fields. For example, we can speak of graphs *over* a particular vertex set, or speak of arbitrary graphs wherein 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. In our earlier example about graph datatypes, 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 prototypically implemented as a meta-program that is not only easily incorporated into Agda's Emacs ecosystem, but is also unobtrusive to Agda users.

1 INTRODUCTION —SELECTING THE 'RIGHT' PERSPECTIVE

Library designers want to produce software components that are not only useful to their immediate needs but also useful to the needs of others, such as themselves for a later project. As such, designers aim for a high-level of generality for increased reusability. The dimension we tackle in this paper is how much of a structures constituents are exposed at the type level and how many are left inside, to be projected when needed.

The subtlety of what is a 'parameter' —exposed at the type level— and what is a 'field' —hidden as a component value— has led to awkward formulations and the duplication of existing types for the sole purpose of different uses. For example, the ever-ubiquitous monoid, a prime model of compositionality, in traditional Haskell, is only allowed one instance per datatype. The Booleans, however, have multiple monoids such as sequential and parallel monoids —the former being conjunction with identity *true* and the latter being disjunction with identity /false.

The solution, in Haskell, is to have two isomorphic copies `All` and `Any` for which the former is given the sequential monoid and the latter the parallel monoid. An alternative solution would be to parameterise the monoid interface not only by its carrier type —the Booleans in our example— but also by the underlying compositionality scheme —the conjunction and disjunction operators in our scheme.

It seems that there are two contenders for the monoid interface:

YYYY. XXXX-XX/YYYY/MM-ARTAA $15.00
DOI: 10.1145/nnnnnnn.nnnnnnn

```
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
    Id     : Carrier
    assoc  : ∀ {x y z} → (x ⨟ y) ⨟ z ≡ x ⨟ (y ⨟ z)
    leftId : ∀ {x} → Id ⨟ x ≡ x
    rightId : ∀ {x} → x ⨟ Id ≡ x
```

Of-course there are also the alternative formulations of where we bundle up the `Carrier` rather than have it exposed —e.g., when we wish to speak of *a* monoid rather than *a monoid on a given type*— or where we are interested in the identity element rather than in the compositionality scheme —e.g., discrepancy '≢' and indistinguishability '≡' 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, what if the user wanted the syntax to form monoid terms as in metaprogramming. That would necessitate yet another nearly identical data-structure —having constructors rather than field projections. We show how all these different presentations can be derived from a *single* PackageFormer declaration. 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

It is notoriously difficult to reconstruct the possible inputs to a function that yielded a certain output. That is, unless you are using Prolog of-course, where the distinctions between input and output are an illusion that is otherwise made real only by how Prolog users treat arguments to a relation. Dependently-typed programming at its core is the adamant hygienic blurring of concepts and so the previous presentations of monoids are unified in the following single declaration which does not distinguish between parameters and fields.

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

Superficially, the parameters and fields have been flattened into a single location and the name `Carrier` has been dispensed with in-favour of `MonoidP`, which also happens to be name of this newly declared entity. We commend the astute reader who has noticed a hint of predicitivity here, but it is an issue we shall not address in the current work.

One uses a PacakageFormer *by instantiating the particular presentation that is desired.*

We conceive of an extensible type `Variations` which includes `datatype` and `record` as two keywords. Moreover, this type is equipped with a number of combinators, one of which is the infix operator `_unbundled_ : Variation → ℕ → Variation` which modifies a particular presentation by also lifting the first `n` constituents from the field level to the parameter level. In particular, `typeclass = record unbundled 1`. We also allow the named version of this combinator, namely `_exposing_ : Variation → List Name → Variation`. Let's demonstrate these concepts.

(0) We may obtain the previous formulations of $\texttt{Monoid}_1$ in two different ways:

```
Monoid₁'  = MonoidP typeclass
Monoid₁'' = MonoidP record exposing Carrier
```

(1) Likewise, there are number of ways to regain the previous formulation of $\texttt{Monoid}_2$.

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

(2) To speak of *a monoid over an arbitrary carrier*, we declare:

```
Monoid₃ = MonoidP record
```

It behaves as if it were declared thusly:

```
record Monoid₃ : Set₁ where
  field
    Carrier : Set
    _⨾_     : Carrier → Carrier → Carrier
    Id      : Carrier
    ...
```

The name `Carrier` is a default and could be renamed; likewise for `Vars` below.

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

```
Monoid₄ = MonoidP datatype
```

It behaves as if it were declared thusly:

```
data Monoid₄ : Set where
    _⨾_ : Monoid₄ → Monoid₄ → Monoid₄
    Id  : Monoid₄
```

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

```
Monoid = MonoidP datatype exposing (Vars)
```

It behaves as if it were declared thusly:

```
data Monoid (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 elaborations; no 'proof-matter'.

There are of-course a number of variation on how a package is to be presented, we have only mentioned a two for brevity. The thesis proposal mentions more and provides examples as well.

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 A NOVEL 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 interface presentation selected, the typing of this function could be elegant or awkward, as follows.

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

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

concat₃ : {M : Monoid₃} → let C = Monoid₃.Carrier M  in  List C → C

concat₄ : List Monoid₄ → Monoid₄
```

An immediate attempt to unify these declarations requires pinpointing exactly *which type is referred to semantically by the phrase MonoidP.* For the `datatype` variation, it could only refer to the resulting algebraic data-type; whereas for the `record` variation, it could refer to the result record type *or* to the `Carrier` projection of such record types. Consequently, we use monad-like notation `do τ ← MonoidP; ⋯τ⋯` whenever we wish to refer to *values* of the underlying carrier of a particular instantiaiton, rather than referring to the type *of* such values. In particular:

```
do τ ← MonoidP record; ℬ τ    ≈  λ {τ : MonoidP record} → ℬ (MonoidP.Carrier τ)
do τ ← MonoidP datatype; ℬ τ  ≈  ℬ (MonoidP datatype)
```

With this understanding in-hand, we may write *variation polymorphic* programs:

```
concatP : {v : Variation}  →  do τ ← MonoidP v;  List τ → τ
concatP []       = MonoidP.Id
concatP (x :: xs) = x ⨾ concatP xs where _⨾_ = MonoidP._⨾_
```

It is important at this juncture to observe that the type of `concatP` depends crucially on the variation `v` that is supplied, or inferred. This is a prime reason for using a dependently-typed language as the setting for the PackageFormer feature.

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 a meta-program that realises these elaborations in an unobtrusive fashion: An Agda programmer simply declares them in special comments. The resulting 'editor tactic' demonstrates that this language feature is promising.

Thus far we have relied on the reader's understanding of functional programming and algebraic data types to provide an informal and indirect semantics by means of elaborations into existing notions. An immediate next step would be to provide explicit semantics for PackageFormer's within a minimal type theory. Moreover there are a number of auxiliary goals, including:

(1) How do users extend the built-in `Variations` type along with the intended elaboration scheme.

    One possible route is for a user to 'install' a new variation by specifying where the separation line between parameters and fields happens; e.g., by providing a function such as `List Constituent → Pair (List Constituent)`, which may introduce new names, such as the aforementioned `Carrier` and `Vars`.

(2) Explain how generative modules are supported by this scheme, and they indeed are.
(3) Demonstrate how tedious boilerplate code for renamings, hidings, extensions, and the flattening of hierarchical structures can be formed.
(4) How do multiple default, or optional, clauses for a constituent fit into this language feature. This may necessitate a form of limited subtyping.
(5) Discuss inheritance, coercion, and transport along canonical isomorphisms.
(6) Flexible polymorphic definitions: One should be able to construct a program according to the most convenient presentation, but be able to have it *automatically* applicable to other instantiations.

    For example, the `concat` function was purely syntactic and the easiet formulation uses the algebraic data-type rendeition, whence one would write
    `concat : List MonoidP datatype → MonoidP datatype`
    and the variation is found then systematically generalised to obtain
    `concatP : {v : Variation} → do τ ← MonoidP v; List τ → τ.`
    When there are multiple variations mentioned, the problem becomes less clear cut and the simplest solution may be to simply indicate which variation or occurrences thereof is intended to be generalised.

Finally, the astute reader will have remembered that our abstract mentions graphs yet there was no further discussion on that example. Indeed, one of the next goals is to accommodate multi-sorted structures where sorts may *depend* on one another, as edge-sets depend on the vertex-set chosen.

There are many routes to progress on this fruitful endeavour.

We look forward to this feature reducing the length of our code and alleviating us of tedious boilerplate constructions.