A Language Feature to Unbundle Data at Will

Anonymous Author(s)

1 LaTeX setup IGNORE

2 Abstract IGNORE

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 depdently-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.

3 Introduction — Selecting the 'right' perspective

Library designers want to produce software components that are useful to 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 [BDN09]. For the monoid example, it seems that there are three contenders for the monoid interface:

2019. XXXX-XXXX/2019/6-ART $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

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

In Monoid$_0$, we will call Carrier “bundled up”, while we call it “exposed” in Monoid$_1$ and 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 Monoid$_1$. While 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, what if the user wanted the syntax to form monoid terms as in metaprogramming. **[JC:** *{* **]** This is too imprecise — please put specific Agda code for MonoidTerm (you have the room) instead. Then you can say something like "We can see that this version can also be mechanically obtained from Monoid$_1$ by turning each field into a constructor". This will make the idea much clearer.} 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. **[JC:** *{* **]** “massive reduction in duplicated efforts and maintenance” begs the question of doing a quantitative study to measure this — which you have neither done, nor intend to do.}

4 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 : 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
```

Coupled with various ‘directives’ that let one declare what should be parameters and what should be fields, we can reproduce the above. Notice that here Carrier has been removed in favour of the name MonoidP, which also is the name of the newly declared entity. This does indeed mean that so far our facility is single sorted. ## 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.

One uses a PackageFormer *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`. Instantiation syntax is of the form “`package-former-name variation`”, as such, `_unbundled_` and `_exposing_` have higher precedence. Let us demonstrate these concepts.

0. We may obtain the previous formulations of 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 Monoid$_2$.

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

Our precedence rules indicate that `MonoidP ⋯` parenthesises as if it were `MonoidP (⋯)`. Moreover, notice that the infix combinators for unbundling and exposing, behave similar to the curry functional $(A \times B \to C) \;\to\; (A \to B \to C)$.

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 two for brevity. The interested reader may consult the 'next 700 module systems' proposal [Alh19]; which discusses more variations and examples in detail.

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.

5 A New Kind of 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

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

6 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 [Ler00] 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; [CO12].
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; [DCH03].
   For example, the `concat` function was purely syntactic and the easiet formulation uses the algebraic data-type rendition, 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. However, a prototype capable of supporting the examples mentioned can be found at https://alhassy.github.io/next-700-module-systems-proposal/.

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

7 Bib **IGNORE**

References

[Alh19] Musa Al-hassy. *The Next 700 Module Systems: Extending Dependently-Typed Languages to Implement Module System Features In The Core Language*. 2019. URL: https://alhassy.github.io/next-700-module-systems-proposal/thesis-proposal.pdf.

[BDN09] 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*. 2009, pp. 73–78. DOI: 10.1007/978-3-642-03359-9_6. URL: https://doi.org/10.1007/978-3-642-03359-9%5C_6.

[CO12] Jacques Carette and Russell O'Connor. "Theory Presentation Combinators". In: *Intelligent Computer Mathematics* (2012), pp. 202–215. 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.

[DCH03] 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*. 2003, pp. 236–249. DOI: 10.1145/640128.604151. URL: https://doi.org/10.1145/640128.604151.

[Ler00] Xavier Leroy. "A modular module system". In: *J. Funct. Program.* 10.3 (2000), pp. 269–303. URL: http://journals.cambridge.org/action/displayAbstract?aid=54525.