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:

```
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

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

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.

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

A package former is used via *instantiations*, written as low-precedence juxtapositions of a package former name and and expression of type `Variation`.

The latter can be built in particular via the following:

```
record      : Variation
typeclass   : Variation
termtype    : Variation
_unbundled_ : Variation → ℕ → Variation
_exposing_  : Variation → List Name → Variation
                          -- Syntax as for ~using~
```

The operators _unbundled_ and _exposing_ have higher precedence than the instantiation juxtaposition. While _exposing_ explicitly lists the names that should be turned into parameters, in that sequence, "$_{\text{unbundled}}$ *n/~" exposes the first /n* names declared in the package former.

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

   ```
   Monoid₀'  = MonoidP record
   ```

The package former name is currently by default replaced by the name "~Carrier~" in instantiations.

1. We may obtain the previous formulation of `Monoid`$_1$ as follows:

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

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

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

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

   ```
   Monoid₄ = MonoidP termtype
   ```

   This behaves as if it were declared as follows:

   ```
   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 termtype exposing (Vars)
   ```

   Since the name `Vars` does not occur in the `MonoidP` package former, and is different from the special name `Carrier`, it is added as a `Set` parameter together with an injection into the created datatype `Monoid`, which could equivalently be defined as follows:

   ```
   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 instantiations; no 'proof-matter'.

[**JC:** *I would instead, especially as you have the room, insert a paragraph naming the additional things that can be done.*]

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 Variation 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:

```
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 attempt to unify these declarations requires understanding exactly *which type is referred to by the phrase MonoidP.* For the `termtype` variation, it can 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` $\tau \leftarrow$ `MonoidP;` $\cdots\tau\cdots$ whenever we wish to refer to *values* of the underlying carrier of a particular instantiation, rather than referring to the type *of* such values. For example:

[**WK:** *Is the following "Agda code" meant to be a two-column layout? I have no idea what it might mean!*]

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

do τ ← MonoidP termtype; ℬ τ ≈ ℬ (MonoidP termtype)
```

This lets us write the following *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 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.

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.
3. Demonstrate how 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.

1. 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 easiest formulation uses the algebraic data-type rendition, whence one would write
   `concat : List MonoidP termtype → MonoidP termtype`
   and the variation is found then systematically generalised to obtain
   `concatP : {v : Variation} → do` $\tau \leftarrow$ `MonoidP v; List` $\tau \rightarrow \tau$.
   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 careful reader will have noticed 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 have a complete prototype available on github, which we will link to once the paper is no longer double-blind.

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.