Inheritance and Overloading in Agda

Paolo Capriotti

3 June 2013

Abstract

One of the challenges of the formalization of mathematics in a proof assistant is defining things in such a way that the syntax resembles the usual informal mathematical notation as much as possible.

I present a collection of techniques that make it possible to obtain a reasonably compact notation in an Agda implementation of basic algebra and category theory.

Although the solution is not completely satisfactory by itself, it shows how the current feature set of a language like Agda could be enhanced in order to solve the problem completely, and that the extensions needed would be minimal.

Introduction

Informal mathematical notation is full of abuses and clever syntactical conventions. We write $\circ$ for the composition in any category, we apply things like functors and natural transformations to their arguments, even though they are not strictly functions, and we use all the notation and results for monoids when we are talking about groups.

When formalizing mathematics in a proof checker like Agda, however, we need to be a little more careful. As in a given scope, there can only be one definition for a given name of symbol, we need to employ some cleverness if we want to replicate the flexibility that the informal notation allows.

The crudest solution is to just use different names for different things. Although this doesn't sound like a bad principle at first, it becomes unweildy pretty quickly. We would need different symbols for operations on natural numbers, integers, rationals. For every new category or group that we define, we would need to come up with names for its operations.

Modules

Fortunately, Agda comes with a remarkably powerful module system. By dividing groups of related definitions into modules, we can then reuse the same names for different definitions. Since every record is also a module, we can, for

example, define a record `Category` with all the structure that defines what a category is:

```
record Category : Set₁ where
  field
    obj : Set
    hom : obj → obj → Set
    id : (x : obj) → hom x x
    _∘_ : {x y z : obj} → hom y z → hom x y → hom x z

    - laws, etc...
```

and then just open it when we need it:

```
open Category C
id-id : (x : obj) → id x ∘ id x ≡ id x
```

Unfortunately, sometimes we need to deal with more than one category at a time. For example, to define the notion of functor, we need at least two. We cannot simply open the module twice, of course, so we are forced to choose: either we qualify everything explicitly, or we use the renaming feature of Agda, and pick new names for every definition that would appear twice.

Explicit qualification is really awkward and makes for completely unreadable code. For example:

```
(f : Category.hom C x y) → Category._∘_ C f (Category.id C x) ≡ f
```

is the type of the left identity law for a category. This is not really a viable solution.

So the only choice we have left is to rename duplicate definitions. Here is how a definition of `Functor` would look like:

```
record Functor (C D : Category) : Set where
  open Category C renaming
    ( obj to objC -
    ; hom to homC )
    - etc...
  open Category D renaming
    ( obj to objD -
    ; hom to homD )
    - etc...
  field
    apply : objC → objD
    map : {x y : objC} → homC x y → homD (apply x) (apply y)
    - etc...
```

This is not too bad, and in fact some existing category theory libraries for Agda, like for example [1] do indeed use this approach.

Unfortunately, this is still very far from a satisfactory overloading mechanism, as it requires enormous amounts of boilerplate code on every usage of overloaded definitions, it's error prone, and still much more noisy than the corresponding informal notation, even ignoring the renaming boilerplate.

Instance arguments

Since version 2.3.0, Agda provides a feature which is specifically designed to make real overloading of names possible: *instance arguments* ([4]).

Instance arguments are similar to implicit arguments: when an argument of a function is marked as "instance", it doesn't need to be given, and is going to be automatically inferred at the function call site.

The strategy for inference of instance arguments, however, differs from the one used for implicit arguments. While the latter is based on unification, the former searches for possible candidates *in scope*, and succeeds if it finds exactly one.

Syntactically, instance arguments are enclosed in double curly braces.

With instance arguments in hand, we can now solve the overloading problem more effectively. If we mark the `Category` argument of each field of our record above as an instance argument, we can open the record only once, and Agda will automatically fill in the correct category from the scope, without us having to qualify it explicitly.

There is even a special syntax for that:

```
open Category {{ ... }} hiding (obj)
open Category using (obj)
```

will simultaneously open the `Category` module and mark the `Category` argument of each of the fields as an instance argument. We refrain from using instance arguments for the `obj` field, as without the `Category` argument, it's likely to be often ambiguous, and `obj C` is an acceptable notation already.

Now we can write, for example:

```
id-id : (x : obj C)(y : obj D) → id x ∘ id x ≡ id x
```

where the object `y` of the category `D` doesn't play any role, but it's included to show that this also works when multiple categories are in scope.

Implementing hierarchies

Using instance arguments directly as in the previous section is a good enough solution for simple cases, but in practice, things are a bit more involved.

In particular, in a formalization of algebra or category theory, concepts are organised into a hierarchy: sets, monoids, groups, abelian groups, rings, etc. are not completely independent entities, but each is, in some sense, a subtype of the previous one.

Agda doesn't have built-in support for subtyping, but we can encode the first three levels, for example, with something like:

```
record IsMonoid (X : Set) : Set where
  field
    unit : X
    _*_ : X → X → X
    - laws...

Monoid : Set₁
Monoid = Σ Set IsMonoid

mon-carrier : Monoid → Set
mon-carrier = proj₁

open IsMonoid {{ ... }} public

record IsGroup (M : Monoid) : Set where
  private
    X = mon-carrier M
    is-mon = proj₂ M   - this must be in scope for
                       - the following to type-check
  field
    inv : X → X
    left-inv : (x : X) → inv x * x ≡ unit

Group : Set₁
Group = Σ Monoid IsGroup

grp-carrier : Group → Set
grp-carrier G = mon-carrier (proj₁ G)

open IsGroup {{ ... }} public
```

We could inline the definition of `IsMonoid` into the `Monoid` record (and similarly `IsGroup` into `Group`), but keeping them separate has many benefits, as I will show later.

Enabler interfaces

Whenever records containing overloaded definitions (which I will refer to as *instance records*) are organised into a hierarchy, they are usually not readily available in a scope, so they need to be explicitly extracted, as in the definition of `IsGroup` above.

The situation is even worse than that, when multiple levels are involved. For example, to use the full structure of a group `G` we need to write something like:

```
is-grp = proj₂ G
is-mon = proj₂ (proj₁ G)
```

and in general, we need a number of statements equal to how many levels deep we need to dig into the hierarchy to find all the definitions that we need.

One solution is to package all these statements into a single module per type, which I refer to as the *enabler* for that type:

```
module mon-enabler (M : Monoid) where
  mon-instance = proj₂ M

module grp-enabler (G : Group) where
  open mon-enabler (proj₁ G) public
  grp-instance = proj₂ G
```

This still requires some boilerplate, but now it is almost entirely on the definition side. The client code can just open the top-level enabler and immediately gain access to the full interface, including definitions in super-types.

Furthermore, the code size of a single enabler is now constant, rather than linear, because we can define new enablers "recursively" over previously defined ones, for example:

```
record IsCommutative (M : Monoid) : Set where
  open mon-enabler M
  field
    comm : (x y : mon-carrier M) → x * y ≡ y * x

CommMonoid : Set₁
CommMonoid = Σ Monoid IsCommutative

AbGroup : Set₁
AbGroup = Σ Group (λ G → IsCommutative (proj₁ G))

module cmon-enabler (M : CommMonoid) where
  open mon-enabler (proj₁ M) public
  cmon-instance = proj₂ M
```

```
module abg-enabler (A : AbGroup) where
  open grp-enabler (proj₁ A) public
  abg-instance = proj₂ A
```

Incidentally, this code also shows why it's beneficial to separate the definition for a new concept X into `IsX` and `X` proper: we can easily reuse the `IsX` records to create parallel hierarchies with minimal code duplication.

Coercions and static methods

Although definitions contained in all the instance records of an inheritance chain can now be accessed very easily just by opening the appropriate enabler module, there is still no way to create *new* definitions in such a way that they can be shared by all super-types.

If we prove a theorem about monoids, like for example:

```
left-right-unit : {M : Monoid}(x : mon-carrier M)
                → let open mon-enabler M
                  in unit * x ≡ x * unit
```

we might want to apply it to groups, abelian groups, etc.

So we define all possible *coercions* to `Monoid`:

```
mon-is-mon : Monoid → Monoid
mon-is-mon M = M   - trivial coercion

grp-is-mon : Group → Monoid
grp-is-mon = proj₁

cmon-is-mon : CommMonoid → Monoid
cmon-is-mon = proj₁

abg-is-mon : AbGroup → Monoid
abg-is-mon A = proj₁ (proj₁ A)
```

then, using instance methods again, we can easily define functions that work for any subtype of `Monoid`:

```
module monoid-static
  {Source : Set₁}
  {{ c : Source → Monoid }}
  (source : Source)
  where
    private M = c source
    open mon-enabler M
```

```
    carrier : Set
    carrier = mon-carrier M

    left-right-unit : (x : carrier) → unit * x ≡ x * unit
    - etc...


open monoid-static public
```

I call such definitions *static methods*, because they behave similarly to static methods in object oriented languages. By contrast, we refer to definitions appearing in some instance record as *instance methods*.

We can even turn enabler modules into static methods:

```
module as-monoid
  {Source : Set₁}
  {{ c : Source → Monoid }}
  (source : Source)
  where
    private M = c source
    mon-instance = proj₂ M
```

Now we can enable `Monoid` methods for any superclass of it:

```
module example (A : AbGroup) where
  open as-monoid A
  - we can use _*_ and unit for A here
```

Unfortunately, coercions to all super-types need to be defined manually for each type in the hierarchy. There does not seem to be way to alleviate this problem with instance arguments, as the instance search is limited to the current scope, and cannot combine instances in any way.

Potentially, a code generator or some kind of macro system could be used to generate coercions automatically. The transitive closure of coercions from `Σ X isY` to `X` could be generated this way, and any other desired coercion could be added manually.

Implementation

The `agda-base` library ([2]) contains an implementation of the inheritance and overloading patterns described in this paper.

The library code employs some extra tricks, like wrapping instance records and coercions into specialized data types.

The data type for instance records is called `Styled` and has a phantom parameter `style`, which can be used to implement alternative notations for instance methods. For example, besides the usual monoid enabler for the `default` style, one can define a secondary enabler for monoids, exposing an instance record with an `additive` style parameter.

Therefore, the user can select the notation to use by opening the corresponding enabler, without having to define additional monoid instances, or perform any renaming of definitions.

Unfortunately, the implementation presents some performance issues during type-checking, probably related to the interaction between instance search and unification of universe levels, as some of the instance definitions (like the one for composition) contain a large number of level meta-variables.

Conclusion and future work

I showed how Agda's instance records can be used to implement hierarchies of data types with overloaded methods.

Although I was mainly focused on the formalization of hierarchies of algebraic or categorical structures, the techniques exemplified here should be also applicable to the domains where object oriented design is usually employed.

The solution is not completely satisfactory, as it requires relatively large amounts of boilerplate code, and it makes type-checking quite slow when combined with universe polymorphism.

However, boilerplate code is limited to data type definitions, whereas the client code looks clean and very close to informal mathematical notation. Furthermore, it is relatively easy to see how the boilerplate generation could be automated or integrated in the language.

I have only dealt with Agda, here, but similar considerations should be true for other implementations of type theory.

The Coq proof assistant, for example, provides built-in support for coercions ([3]), and provides a *type class* construct ([5]), which subsumes instance records and enablers. It is likely that those features would make it possible to achieve comparable (or even better) expressiveness in Coq with minimal amounts of boilerplate.

However, Coq's features feel much less minimalistic. In particular, the instance search employs not very well-specified heuristics, and thus is not as predictable as Agda's.

I feel the ideas presented in this paper show there is a sweet spot in the design space of instance search and implicit coercion features that has not yet been implemented, and that it lies not very far from the current capabilities of the Agda system.

Investigating in more detail and possibly implementing this sweet spot is the subject of future work.

References

[1] `https://github.com/copumpkin/categories`.

[2] `https://github.com/pcapriotti/agda-base`.

[3] `http://coq.inria.fr/distrib/current/refman/Reference-Manual021.html#Coercions-full`.

[4] Dominique Devriese and Frank Piessens. On the bright side of type classes: instance arguments in agda. In *ICFP*, pages 143–155, 2011.

[5] Matthieu Sozeau and Oury Nicolas. First-Class Type Classes.