Inheritance and Overloading in Agda

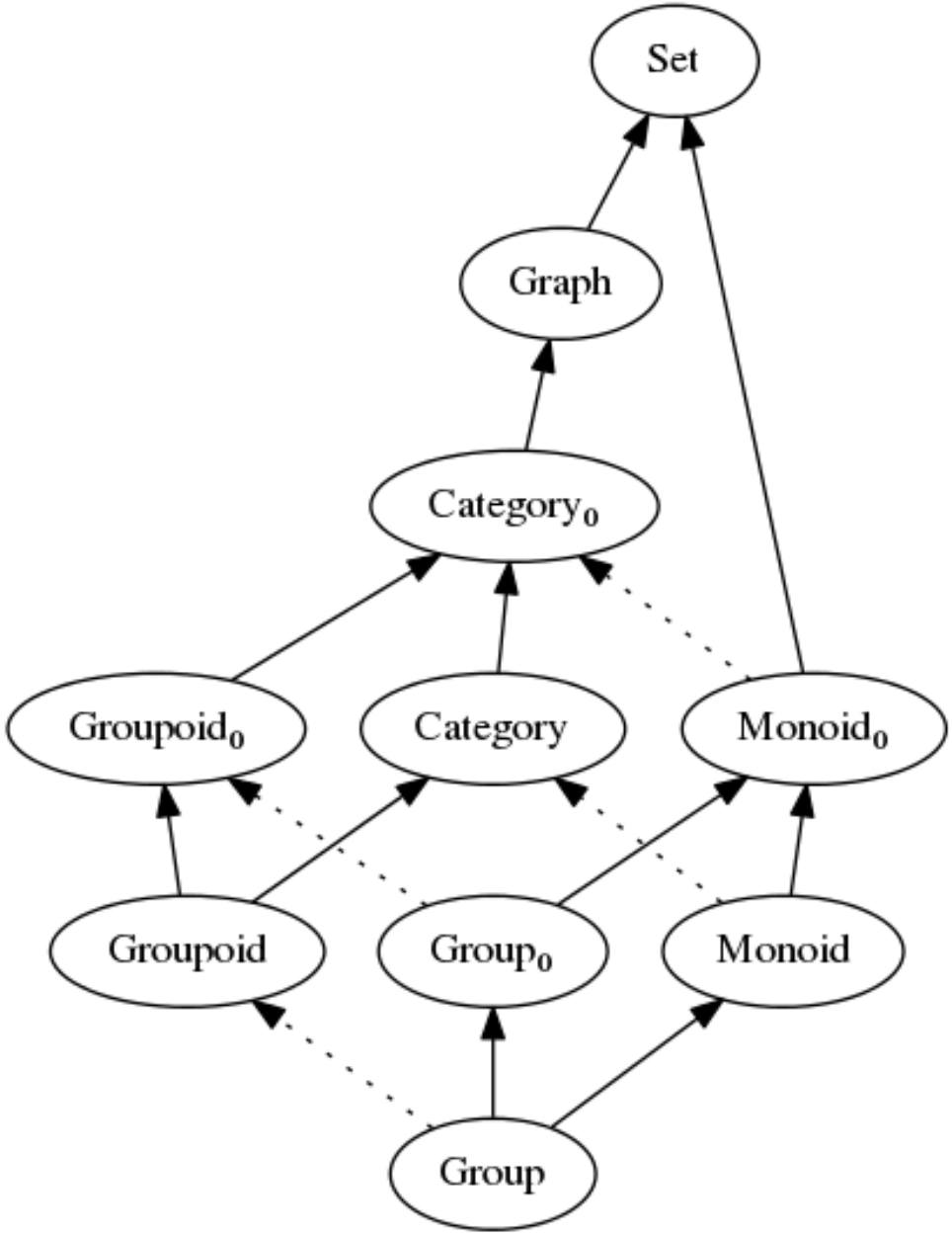
Paolo Capriotti

June 17, 2013

Notation

- Abuses of notation are very common in mathematics
- Ambiguities are used to make formulas more expressive
- We want to do the same in Agda!

Example: algebra and category theory



Using modules I

```
record Monoid : Set₁ where
  field
    carrier : Set
    unit : carrier
    _*_ : carrier → carrier → carrier

record Group : Set₁ where
  field
    carrier : Set
    unit : carrier
    _*_ : carrier → carrier → carrier
    inv : carrier → carrier
```

Using modules II

Good enough with *one* `Monoid` or `Group` in scope:

```
M : Monoid

open Monoid M
```

Using modules III

What if we have more than one?

```
M : Monoid
G : Group

open Monoid M renaming
  ( carrier to |M| -
  ; unit to unit-M -
  ; _*_ to _*M_ )
open Group G renaming
  ( carrier to |G| -
  ; unit to unit-G -
  ; _*_ to _*G_ )
```

Yuck...

"Real world" example

```
module C = Category C
module D = Category D
module E = Category E
module F = Functor F
module G = Functor G renaming (F₀ to G₀; F₁ to G₁; F-resp-≡ to G-resp-≡)
module H = Functor H renaming (F₀ to H₀; F₁ to H₁; F-resp-≡ to H-resp-≡)
module I = Functor I renaming (F₀ to I₀; F₁ to I₁; F-resp-≡ to I-resp-≡)
module X' = Adjunction X renaming (unit to Xη'; counit to Xε')
module Y' = Adjunction Y renaming (unit to Yη'; counit to Yε')
module Xη = NaturalTransformation (Adjunction.unit X) renaming (η to ηX)
module Yη = NaturalTransformation (Adjunction.unit Y) renaming (η to ηY)
module Xε = NaturalTransformation (Adjunction.counit X) renaming (η to εX)
module Yε = NaturalTransformation (Adjunction.counit Y) renaming (η to εY)
open C
open D
open E
open F
open G
open H
open I
open X'
open Y'
open Xη
open Yη
open Xε
open Yε
```

Can we do better?

Instance arguments I

- Enclosed in double curly braces: `{{ a : A }}`
- They are automatically inferred
- The search is limited to the current scope
- Inference only works if there is *exactly* one match
- Special syntax to set a record as implicit parameter for all its fields:

```
record X : Set where
  - ...

open X {{ ... }}
```

Instance arguments II

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

Monoid = Σ Set IsMonoid

open IsMonoid {{ ... }}
```

Instance arguments III

```
record IsGroup (M : Monoid) : Set where
  private X = proj₁ M
  field
    inv : X → X

Group = Σ Monoid IsGroup

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

Enabler modules

- Deeply nested instances of `IsX` records can be awkward to bring into scope
- Enabling overloaded definitions for all super-types requires boilerplate at every invocation

Enabler modules

- Deeply nested instances of `IsX` records can be awkward to bring into scope
- Enabling overloaded definitions for all super-types requires boilerplate at every invocation
- Solution: write boilerplate only once per type

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

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

Coercions

- We want to define things for `Monoid` and apply them to any subtype
- We want to be able to "apply" things that are not strictly functions

Coercions

- We want to define things for `Monoid` and apply them to any subtype
- We want to be able to "apply" things that are not strictly functions
- Solution: define *coercions* manually, and use them as instance arguments

Coercions

- We want to define things for `Monoid` and apply them to any subtype
- We want to be able to "apply" things that are not strictly functions
- Solution: define *coercions* manually, and use them as instance arguments
- A lot of boilerplate required for each new type, but client code looks nice

Conclusion

- Code on github:
  `http://github.com/pcapriotti/agda-base`
- Typechecking is *really* slow
- How to get rid of the boilerplate?