

Typeclasses and reification in ATBR, AAC_tactics, and ring

Thomas Braibant, Assia Mahboubi, and Damien Pous

GT Coq, 31/03/2011

Typeclasses

What for?

- ▶ Overloaded notations
- ▶ Inference of mathematical structures
- ▶ Reification
- ▶ Possibly a lot more

(I won't talk about canonical structures here)

Contents

- ▶ What we did with Thomas in ATBR and AAC_tactics
 - ▶ and how it is related to Bas and Eelis work
- ▶ What we started with Assia and Thomas for ring
 - ▶ and how it is related to Loïc's work
- ▶ Lessons learned

Outline

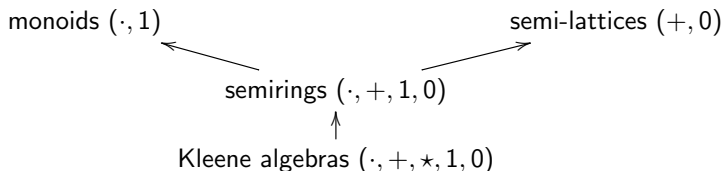
Typeclasses for notations and algebraic hierarchies

Typeclasses for reification
with notational grip
without notational grip

Lessons learned

ATBR [Coq W.'09, ITP'10]

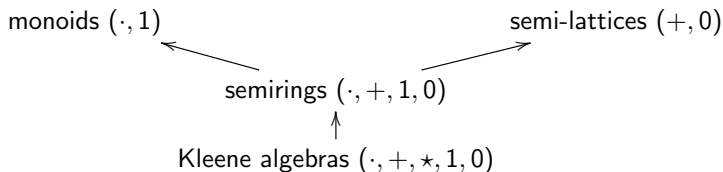
- ▶ A small hierarchy, for Kleene Algebras:



- ▶ A few constraints:
 - ▶ need for matrices
 - ▶ need for modularity (theorems and tactics)
- ▶ Typeclasses do the trick!

ATBR [Coq W.'09, ITP'10]

- ▶ A small hierarchy, for Kleene Algebras:



- ▶ A few constraints:
 - ▶ need for matrices
 - ▶ need for modularity (theorems and tactics)
- ▶ Typeclasses do the trick!

Algebraic hierarchy with Typeclasses, summary

▶ Good points

- ▶ simple, elegant, modular
- ▶ reification for free
- ▶ it worked for us (Coq W.'09, ITP'10)
- ▶ it seems to work for Bas and Eelis too (ITP'10, MSCS'11 ?)

Algebraic hierarchy with Typeclasses, summary

▶ Good points

- ▶ simple, elegant, modular
- ▶ reification for free
- ▶ it worked for us (Coq W.'09, ITP'10)
- ▶ it seems to work for Bas and Eelis too (ITP'10, MSCS'11 ?)

▶ Bad points

- ▶ rigid end-user interface
- ▶ it does not scale! (for us, for now)
- ▶ naive reification might result in large proofs and long Qeds

Outline

Typeclasses for notations and algebraic hierarchies

Typeclasses for reification
with notational grip
without notational grip

Lessons learned

Reification, with notational grip

- ▶ In a settings like the previous one, goals are almost already reified:

$$x * (y * x) + y$$

is actually

$$\text{plus } P \text{ (mult } M \text{ x (mult } M \text{ y x)) y}$$

- ▶ `plus` and `mult` are typeclass projections (constants),
 - ▶ `P` and `M` are instances depending on the current model.
-
- ▶ All we have to do is to find the environment (here, something like `[x; y]`).

Reification, with notational grip

- ▶ In ATBR, we proposed a trick based on an inductive predicate together with `eauto` hints for building the environment.
 - ▶ Drawbacks:
 - ▶ quadratic time reification (lookups in the environment),
 - ▶ unnecessarily large proofs (not a simple conversion step).
 - ▶ We re-implemented reification using OCaml in the last version.

Reification, with notational grip

- ▶ In ATBR, we proposed a trick based on an inductive predicate together with `eauto` hints for building the environment.
 - ▶ Drawbacks:
 - ▶ quadratic time reification (lookups in the environment),
 - ▶ unnecessarily large proofs (not a simple conversion step).
 - ▶ We re-implemented reification using OCaml in the last version.
- ▶ Bas and Eelis replay the same trick with typeclasses:
 - ▶ in a nicer way: they use typeclass resolution rather than our inductive predicate → **no Ltac at all!**
 - ▶ working with 'heaps' rather than lists;
 - ▶ with the **same drawbacks** (from what I understand).

Reification, with notational grip

- ▶ In ATBR, we proposed a trick based on an inductive predicate together with `eauto` hints for building the environment.
 - ▶ Drawbacks:
 - ▶ quadratic time reification (lookups in the environment),
 - ▶ unnecessarily large proofs (not a simple conversion step).
 - ▶ We re-implemented reification using OCaml in the last version.
- ▶ Bas and Eelis replay the same trick with typeclasses:
 - ▶ in a nicer way: they use typeclass resolution rather than our inductive predicate → **no Ltac at all!**
 - ▶ working with 'heaps' rather than lists;
 - ▶ with the **same drawbacks** (from what I understand).
- ▶ Loïc also, in an even nicer way!
 - ▶ still quadratic,
 - ▶ but the reification step is a **conversion step**.

Outline

Typeclasses for notations and algebraic hierarchies

Typeclasses for reification

with notational grip

without notational grip

Lessons learned

Rewriting modulo AC: AAC_tactics

aac1.v

Rewriting modulo AC: AAC_tactics

aac1.v

- ▶ The approach with typeclass projections is not an option:
 - ▶ a lot of operations are A or AC,
→ they cannot share the same notation
 - ▶ we reify several A/AC operations at the same time,
 - ▶ we reify free function symbols.
→ the environment is much more involved

Rewriting modulo AC: AAC_tactics

aac1.v

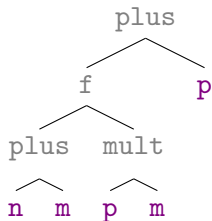
- ▶ The approach with typeclass projections is not an option:
 - ▶ a lot of operations are A or AC,
→ they cannot share the same notation
 - ▶ we reify several A/AC operations at the same time,
 - ▶ we reify free function symbols.
→ the environment is much more involved
- ▶ But we can take advantage of typeclasses!

aac2.v

Reification without notational grip

- ▶ In a goal, nothing is explicit about A/AC operations:

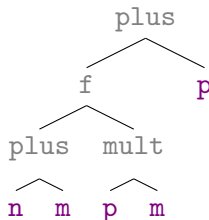
$f(n+m)(p*m) + p$ is just



Reification without notational grip

- ▶ In a goal, nothing is explicit about A/AC operations:

$f(n+m)(p*m) + p$ is just



- ▶ We have to guess that `plus` and `mult` are AC, while `f` is just a binary free function symbol.

Reification without notational grip

- ▶ Brute force approach: at each application node $[f \ a \ b \ c]$,
 1. test whether $[f \ a]$ is A/AC,
 2. if not, test whether $[f]$ is a proper morphism,
 3. if not, test whether $[f \ a]$ is a proper morphism,
 4. if not, test whether $[f \ a \ b]$ is a proper morphism,
 5. if not, consider the node as a constant.

Reification without notational grip

- ▶ Brute force approach: at each application node $[f\ a\ b\ c]$,
 1. test whether $[f\ a]$ is A/AC,
 2. if not, test whether $[f]$ is a proper morphism,
 3. if not, test whether $[f\ a]$ is a proper morphism,
 4. if not, test whether $[f\ a\ b]$ is a proper morphism,
 5. if not, consider the node as a constant.
- ▶ Each test is performed by a call to typeclass resolution.
- ▶ These calls are memoised to keep it reasonable.

Reification without notational grip

- ▶ Brute force approach: at each application node $[f\ a\ b\ c]$,
 1. test whether $[f\ a]$ is A/AC,
 2. if not, test whether $[f]$ is a proper morphism,
 3. if not, test whether $[f\ a]$ is a proper morphism,
 4. if not, test whether $[f\ a\ b]$ is a proper morphism,
 5. if not, consider the node as a constant.
- ▶ Each test is performed by a call to typeclass resolution.
- ▶ These calls are memoised to keep it reasonable.
- ▶ Currently implemented within an OCaml plugin:
 - ▶ this allows us to use efficient data-structures (e.g., Hashtables),
 - ▶ we need OCaml for pattern matching modulo AC, anyway.
 - makes it heavy to work modulo convertibility

Achieving notation-independent reification for ring

- ▶ **Note:** this is the current behaviour!
 - ▶ but it currently requires an OCaml plugin,
 - ▶ relying on an ad-hoc registration mechanism ([Add Ring ...](#))

Achieving notation-independent reification for ring

- ▶ **Note:** this is the current behaviour!
 - ▶ but it currently requires an OCaml plugin,
 - ▶ relying on an ad-hoc registration mechanism (`Add Ring ...`)
- ▶ **Goal:** use typeclasses for declaring ring structures:

```
Class Ring X (eqX: relation X) := {  
  mult, plus: X → X → X;  
  opp: X → X;  
  one, zero: X;  
  mult_assoc: ∀x y z, eqX (mult x (mult y z)) (mult (mult x y) z);  
  ... }.
```

```
Instance Ring_nat: Ring nat eq := ...
```

```
Instance Ring_Q: Ring Q Qeq := ...
```

Achieving notation-independent reification for ring

- ▶ Given a goal $x, y: \text{nat} \vdash x * y = y * x + x * 0$, one can
 1. guess a Ring instance R (here, Ring_nat);
 2. at each application node [f a b c],
 - 2.1 test whether [f a] equals [mult R] or [mult R],
 - 2.2 if not, test whether [f a b] equals [opp R],
 - 2.3 if not, test the node equals [one R] or [zero R],
 - 2.4 if not, declare the node as a constant.

Achieving notation-independent reification for ring

- ▶ Given a goal $x, y: \text{nat} \vdash x * y = y * x + x * 0$, one can
 1. guess a Ring instance R (here, Ring_nat);
 2. at each application node [f a b c],
 - 2.1 test whether [f a] equals [mult R] or [mult R],
 - 2.2 if not, test whether [f a b] equals [opp R],
 - 2.3 if not, test the node equals [one R] or [zero R],
 - 2.4 if not, declare the node as a constant.
- ▶ We quickly implemented this idea with Assia and Thomas, as an OCaml plugin:
 - ▶ it seems to work pretty well,
 - ▶ we would have to finish it.

Achieving notation-independent reification for ring

- ▶ Given a goal $x, y: \text{nat} \vdash x * y = y * x + x * 0$, one can
 1. guess a Ring instance R (here, Ring_nat);
 2. at each application node [f a b c],
 - 2.1 test whether [f a] equals [mult R] or [mult R],
 - 2.2 if not, test whether [f a b] equals [opp R],
 - 2.3 if not, test the node equals [one R] or [zero R],
 - 2.4 if not, declare the node as a constant.
- ▶ We quickly implemented this idea with Assia and Thomas, as an OCaml plugin:
 - ▶ it seems to work pretty well,
 - ▶ we would have to finish it.
- ▶ I suspect this could also be done using Loïc's idea, without going through OCaml.

Outline

Typeclasses for notations and algebraic hierarchies

Typeclasses for reification
with notational grip
without notational grip

Lessons learned

Typeclasses for mathematical hierarchies

- ▶ It works pretty well. . .

Typeclasses for mathematical hierarchies

- ▶ It works pretty well...

... until it breaks!

Typeclasses for mathematical hierarchies

- ▶ It works pretty well. . .

. . . until it breaks!

- ▶ Need for more control on typeclass resolution:
 - ▶ `eauto` “backward” proof search is fine in simple cases,
 - ▶ one often needs “forward” search,
 - ▶ very hard to understand which instances (not) to declare.

- ▶ *Technical remark*: there is something to be done about the scope of `(e)auto` hints (the `typeclass_instances` db is shared).

Typeclasses for reification

- ▶ Good solutions when relying on annotations
 - + lightweight, simple
 - = be careful with generated proof terms
 - efficiency
 - atoms indices

- ▶ Work to be done for reifying unannotated goals

Typeclasses for reification

- ▶ Good solutions when relying on annotations
 - + lightweight, simple
 - = be careful with generated proof terms
 - efficiency
 - atoms indices

- ▶ Work to be done for reifying unannotated goals

Thanks!