A Notion of Class for Theory Development in Algebra (in a Predicative Type Theory)

Peter Aczel
Manchester

partially in collaboration with
Gilles Barthe

Båstad
June, 1994

Preliminaries for the Galois Formalisation

I. Preformalisation of Constructive Galois Theory.

II. Development of mathematical abstractions for the formalisation of algebra in a type theory

III. Make suggestions for a computer environment in which formalisation is attractive to do and in which the product can be attractive to read by humans.

To formalise abstract algebra we will need one or more notions of 'system' of algebras

e.g. '<u>system</u>' of abelian groups

- a system should determine the <u>type</u> of algebras of the system; i.e. algebras are 1st. class objects.
- systems can be defined incrementally;

  e.g. a <u>ring</u> is an <u>abelian group</u> with ... such that ...
- notation introduced for a given system should be inherited by systems defined from it.

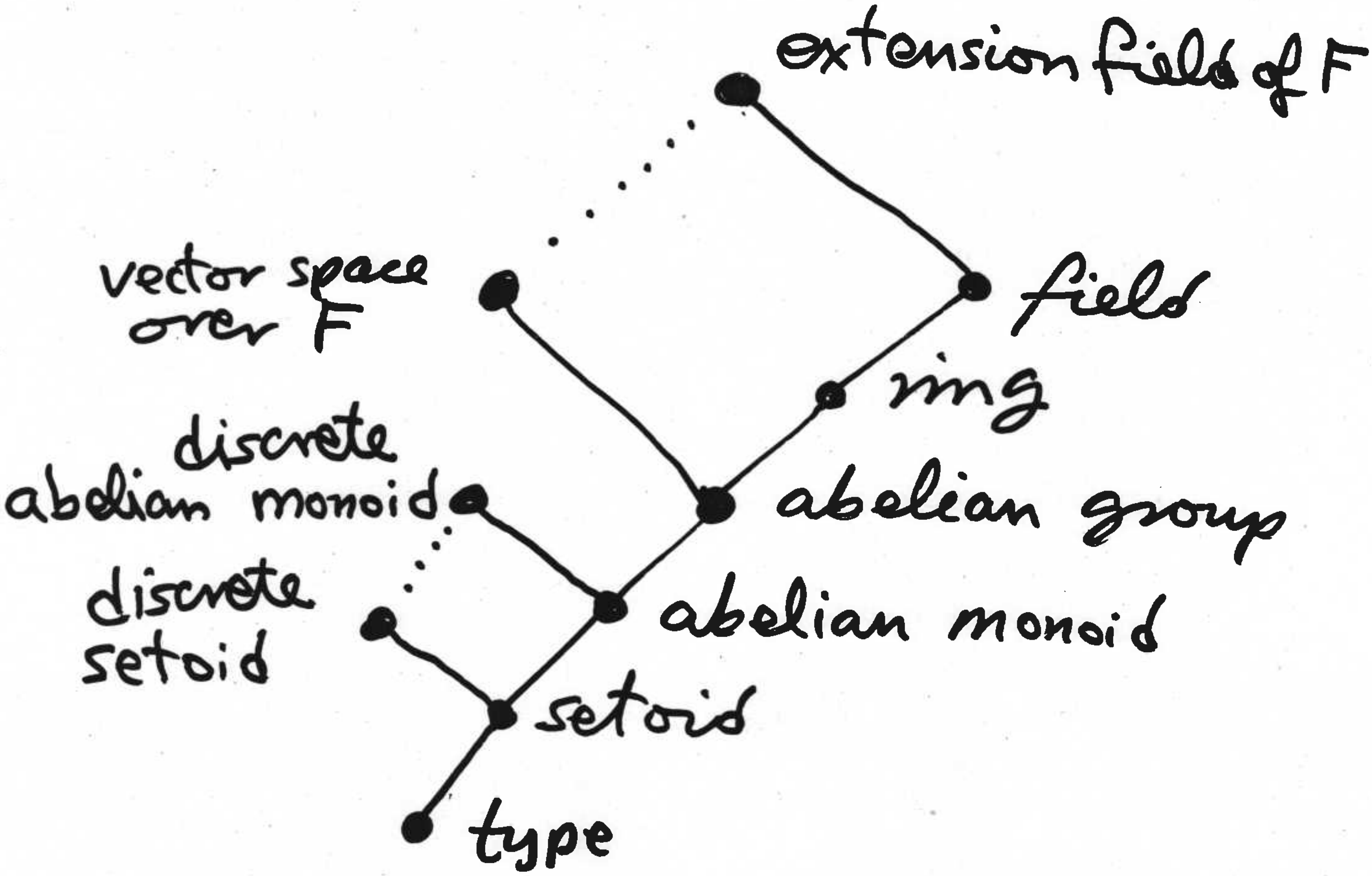
? system = algebraic theory

= category of algebras

⋮

Some kinds of algebraic structure

- A setoid is a type with a relation on it such that the relation is an equivalence relation.
- An abelian monoid is a setoid with a binary operation on it and a distinguished element such that the binary operation
  preserves equality
  is associative
  is commutative
  has the distinguished element as unit
- An abelian group is an abelian monoid with ... such that ...
- A ring is an abelian group with ... such that ...
- A field is a ring ...
- A vector space over a field F
  is an abelian group ...
- A field extension of a field F
  is a

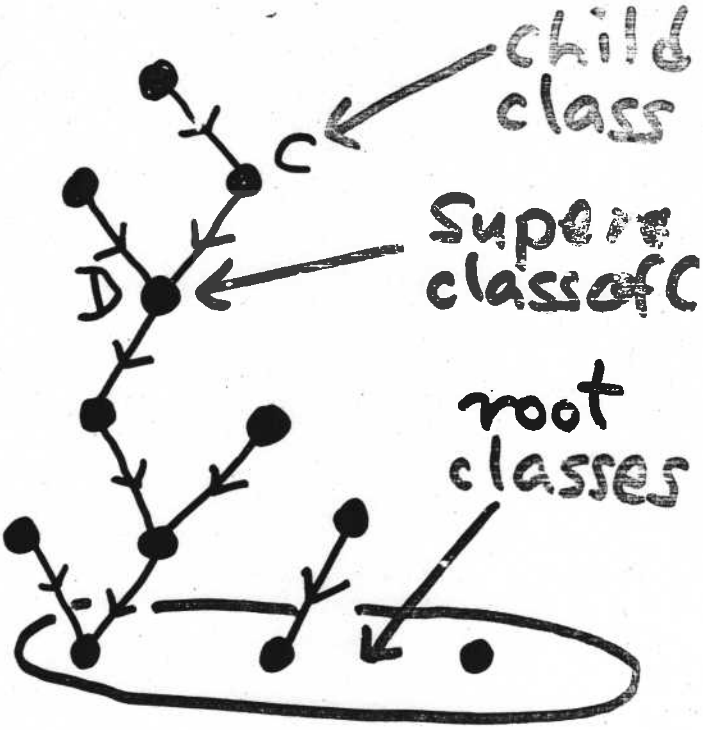

extension field of F
vector space over F
field
ring
discrete abelian monoid
abelian group
discrete setoid
abelian monoid
setoid
type

SOME POSSIBLE APPROACHES

- Use some trick that exploits argument synthesis.
- implement ML style overloading.
- Introduce subtyping in the style of Cardelli and others.
- Use type classes as in Haskell and Isabelle.
- Use another notion of class with method inheritance

CLASS FORESTS

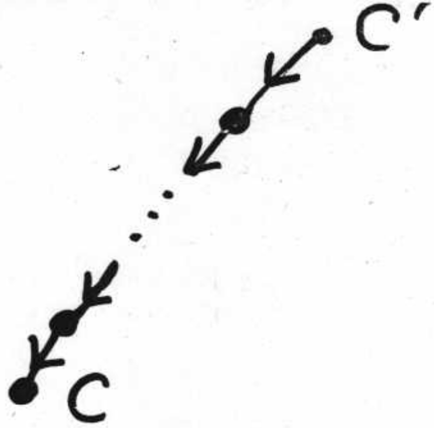
- finite sets of finite upward growing trees
  - nodes called classes
  - non-root nodes called child classes
  - each child class C has a unique edge $C \to D$ to its superclass D



- Each class C has the type ^C of its instances
- Each edge $C \to D$ has the coercion function

$$!^C : {}^\wedge C \to {}^\wedge D$$

Subclasses



- C' is a subclass of C, written $C' \subseteq C$, if there is a path
- By composing the coercion maps of the edges of the path we get a coercion map $!^{C'}_{C} : \hat{C'} \longrightarrow \hat{C}$

Methods

- Each method on a class C can be applied to any object of C; i.e. any instance of any subclass of C
- Each <u>function</u> f on $\hat{C}$ determines a <u>method</u> m on C, with the contraction rule

$$m\,a \Longrightarrow f(!^{C'}_{C}\,a)$$

for $C' \subseteq C$, $a : \hat{C'}$.

Class and Method Definitions

- In implemented type theories a context is made up of
  declarations and definitions
- Extend a type theory by allowing also class and method definitions

  C = rootclass ...

  C = childclass ... D ...

  m = method ... C ...

  - Classes and methods will always be identifiers
  - The classes defined in a context will form a class forest.

Root Class Definitions

$$C = \text{rootclass } T$$

can be added to a context provided that, relative to the context,

C is a new identifier

T type

In any context containing the def.

$$\hat{}C = T$$

Child Class Definitions

$$C = \text{childclass } D \ (x{:}T)a$$

can be added to a context provided that, relative to the context,

C is a new identifier

D is a defined class

$x{:}T \vdash a : \hat{}D$

In any context containing the def.

$$\hat{}C = T \qquad \frac{b : \hat{}C}{\begin{cases} !^C b : \hat{}D \\ !^C b = a[b/x] : \hat{}D \end{cases}}$$

METHOD DEFINITIONS

$$m = \text{method } (x : {}^\wedge C)\, b : B$$

can be added to a context provided that, relative to the context,

m is a new identifier

C is a defined class

$x : {}^\wedge C \vdash b : B$

In any context containing the def., if C' is a defined class and $C' \subseteq C$,

$$\frac{a : C'}{\begin{cases} m\, a : B[!^{C'}_{C}\, a/x] \\ m\, a = b[!^{C'}_{C}\, a/x] : B[!^{C'}_{C}\, a/x] \end{cases}}$$

STANDARD CLASS FORESTS

These are class forests where, for each edge $C \to D$ there is a family of types P over $\hat{}D$ such that

$$\begin{cases} \hat{}C = (\Sigma y : \hat{}D)\, P\, y \\ !^C_{\cdot} = (\lambda x : \hat{}C)\, x.1 \end{cases}$$

- Then, if $C' \subseteq C$,

$$!^{C'}_{\cdot C} = (\lambda x : \hat{}C')\, x.\overbrace{1.\cdots.1}$$

- Any object of class C has the form $(\cdots((e, ?), ?), \ldots)$ with $e : C$
- If m is the method on C defined from the function f on $\hat{}C$ then

$$m\, a \Longrightarrow f(a.\overbrace{1.\cdots.1})$$

if $C' \subseteq C$, $a : C'$.

Standard Childclass Definitions

$$C = \text{st-childclass}\,(y : {}^\wedge D)\,P$$

can be added to a context provided that, relative to the context,

C is a new identifier

D is a defined class

$y : {}^\wedge D \vdash P$ type

This definition is alternative syntax for

$$C = \text{childclass}\ D\ (x : (\Sigma y : {}^\wedge D))\,x.1$$

So, in any context containing the definition

$${}^\wedge C = (\Sigma y : {}^\wedge D)P \qquad \frac{b : {}^\wedge C}{\begin{cases} !^C b : {}^\wedge D \\ !^C b = b.1 : {}^\wedge D \end{cases}}$$

```
[Typeclass = rootclass Type];
[el        = method [A:^Typeclass] A];
[Setoid    = st-childclass [T:^Typeclass]
                <R:Rel T> Equiv R];
[eq        = method [A|^Setoid] A.2.1];
[AbMon     = st-childclass [A:^Setoid]
      <p:A.el → A.el → A.el> <ze:A.el>
 and4
     {x,y:A.el} (eq x y) → (eq (p x) (p y))
     {x,y,z:A.el} eq (p (p x y) z) (p x (p y z))
     {x,y:A.el} eq (p x y) (p y x)
     {x:A.el} eq (p x ze) x ];
[plus      = method [A|^AbMon] A.2.1];
[zero      = method [A|^AbMon] A.2.2.1];
              ⋮
```