The type system of Axiom

Erik Poll

Radboud University of Nijmegen

- joint work with Simon Thompson at University of Kent at Canterbury (UKC)
- work done last century, so probably outdated in parts

Mathoinformatics

Doing mathematics with a computer:

- **Computer Algebra**
  - **eg. Mathematica, Maple**
  - **millions of users (practitioners)**
- **Computer Logic (theorem proving)**
  - **eg. Automath, Mizar, Coq, PVS, HOL, . . .**
    Simplify, SAT & SMT solvers. . .
  - **hundreds of users (researchers, esp. computer scientists)**

Computer Algebra vs Logic

- Computer Algebra
  - usually untyped
  - bad at doing logic
  - unsound, due to unchecked side-conditions (eg. continuity of function), silent switching from domain (eg. from real to complex numbers), etc
- Computer logic
  - usually typed
  - bad at doing algebra

A combination of computer logic and computer algebra would be great . . .

Axiom/Aldor

Axiom is a computer algebra system, that is unusual in having a rich strong type system.

The type system is (almost) expressive enough to encode a logic using the Curry-Howard-de Bruijn isomorphism, which offers a way to combine computer algebra with computer logic . . .

History of Axiom/Aldor

- started life as Scratchpad by IBM, with a language Spad in 1971.
- renamed to Axiom
- new compiler Aldor (aka $A^{\#}$ and Axiom-XL) built by Stephen Watt et.al. 1985-94
- sold to NAG (Numerical Algorithms Group) in mid 90's
- open source since 2002:
  see `www.aldor.org` and `www.nongnu.org/axiom`
- there were rumours about linking Maple and Axiom, and using Aldor for Maple libraries; I don't know what happened to that.

Aldor

- interpreted or compiled to Common Lisp or C, via intermediate language FOAM (First Order Abstract Machine)
- includes a complete functional programming language, with higher order functions etc.
- also has references, overloading, inheritance, subtyping, courtesy conversions, macros, multiple values . . .
- the type system is very expressive and complex
- to understand the type system we implemented a tool that maps Aldor terms to type-annotated terms in HTML (using the type checker in the compiler)

The type system of Axiom

Types as values

Aldor provides explicit parametric polymorphism

```
polyid (T:Type, t:T) : T = t;
```

and treats types as first class citizens

```
idType (T:Type) : Type = T;
```

(Aldor allows overloading, so we could give both functions the same name, but that would get confusing.)

Types as values

Alternatively, using Aldor's notation for λ,

```
polyid : (T:Type)->T->T
          == (T:Type)(t:T):T +-> t;

idType : Type -> Type
          == (T:Type):Type +-> T;
```

In Aldor, $(\lambda$`x:A.b`$)$ is written as `(x:A):B +-> b`

Impredicativity and Type:Type

The type of the polymorphic identity is a type

```
PolyidType : Type ==  (T:Type)->T -> T;
```

In fact, Type is a type

```
MyType : Type == Type;
MyTypeArrowType : Type ==  Type -> Type;
MyType2 : Type == (polyid Type) Type ;
```

Warning: application associates to the right!

Categories

Aldor provides a powerful notion of abstract datatype

```
Monoid : Category == BasicType with {
    1 :  %;
    * : (%,%) -> %
  }
```

Intuitively, this is the type of all monoids.

In type theory, Σ `X:Type.Record(1:X,*:X`$\times$`X->X)`

Domains

Elements of categories are domains

```
IntegerAdditiveMonoid : Monoid == add {
  Rep == Integer;
  import from Integer;
  1               : % == per 0;
  (x:%)*(y:%)     : % == per((rep x) + (rep y))
}
```

Here `per:%->Rep` and `rep:Rep->%` are conversion functions between the abstract carrier % and the concrete representation `Integer`.

NB. no guarantee that elements of `Monoid` are monoids!

Inheritance

Categories can extend other categories, eg.

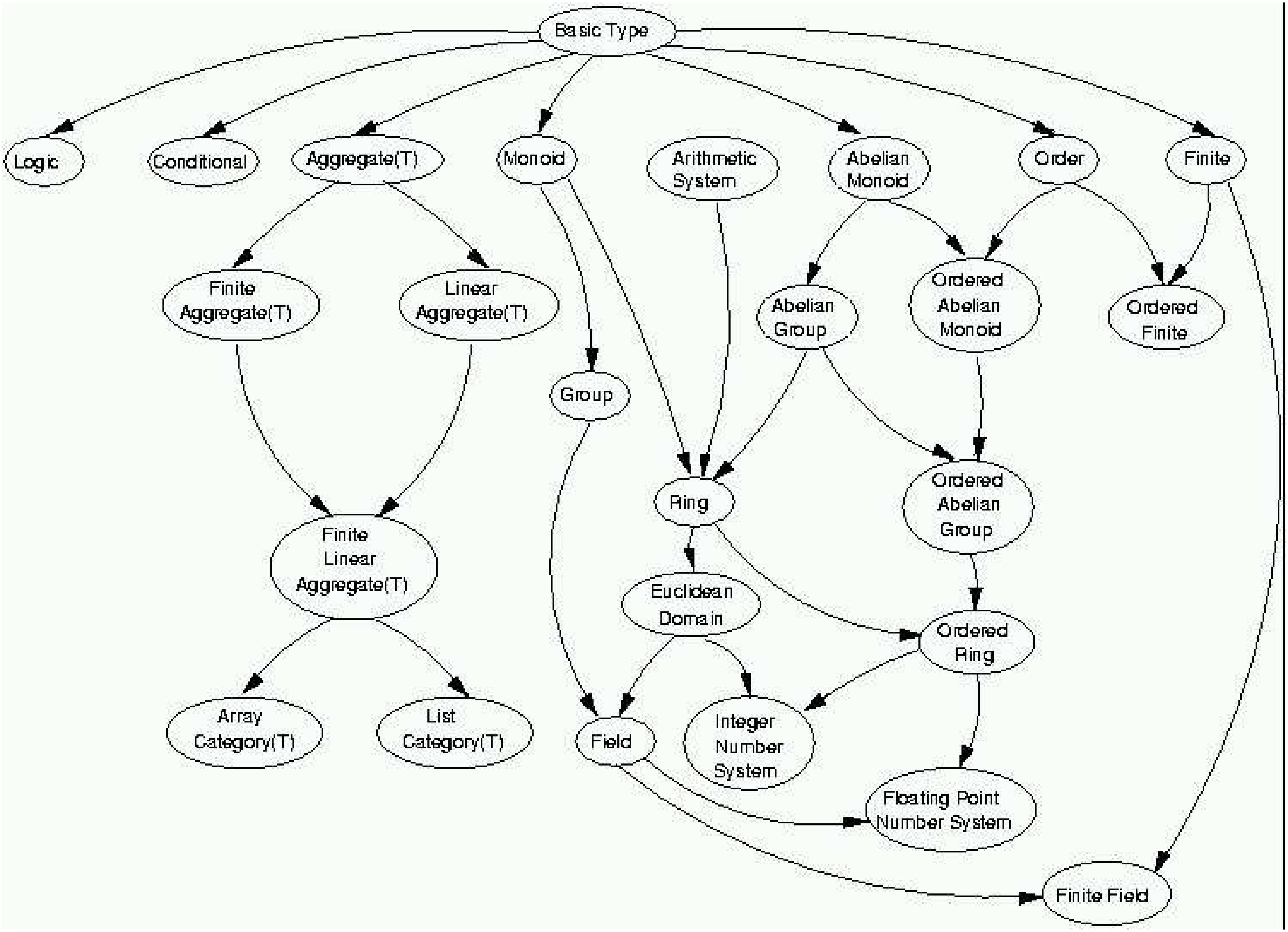
```
Monoid : Category == BasicType with {
    1 :  %;
    * : (%,%) -> %
  }
```

extends

```
BasicType : Category == with {
    = : (%,%) -> %;
  }
```

This provides a rich subtyping hierarchy

Aldor’s category hierarchy



Category as values

Categories are first-class citizens, eg.

```
FancyOutput(c:Category) : Category
  == c with { prettyPrint : % -> BoundingBox };
```

In fact, `Category` is a type

```
MyCategory : Type == Category;
```

Dependent types

Aldor supports dependent types, eg. we can define

```
Vector: (n:Integer) Type;

vectorSum: (n:Integer) -> Vector(n) -> Integer;

append: (n:Integer,m:Integer,Vector(n),Vector(m))
            -> Vector(n+m);
```

This suggests Aldor is powerful enough to code up a logic, using the Curry-Howard-de Bruijn isomorphism.

Axioms in Aldor

Eg using dependent types we could include the monoid axioms in the `Monoid` category, as follows

```
Monoid : Category == BasicType with {
  1 :  %;
  * : (%,%) -> %

  leftUnit(x:%)  : (1*x=x);
  rightUnit(x:%) : (x*1=x);
  assoc(x:%,y:%,z:%) : (x*(y*z)=(x*y)*z);
}
```

However, . . .

Limits of Aldor: type conversion

Aldor performs no computation in types during type checking.

So

```
append (2,3,vec2,vec3) : Vector(2+3)
```

but *not*

```
append (2,3,vec2,vec3) : Vector(5)
```

As Aldor is *not* strongly normalising, this shouldn't really surprise us.

Limits of Aldor: type conversion

Another example

```
eight : Integer == 8;
```

but *not*

```
idType (T:Type) : Type = T;
seven : idType(Integer) == 7;
```

Logic with Aldor (1)

We could use Aldor as a logic if we

- **extended Aldor to allow with type conversion,**
- **imposed restrictions to avoid inconsistencies by eg. Girard’s paradox or nonterminating functions.**
- **Simon Thompson and Leonid Timochouck defined Aldor- -, a purely functional language, a subset of Aldor, that does support evaluation in types.**

pretend

Aldor is not type safe, as it has the following `pretend` construct:

$$\frac{\texttt{t:T} \quad \texttt{S:Type}}{\texttt{(t pretend S) : S}}$$

We can use `pretend` in those places where the type checker fails to compute/convert types.

pretend

We can use `pretend` to fix the problematic examples earlier

```
append (2,3,vec2,vec3) pretend Vector(5)
      : Vector(5)

seven : idType(Integer)
      == 7 pretend idType(Integer);
```

Every use of `pretend` induces a proof obligation

Logic with Aldor (2)

We could use Aldor as a logic if we

- emit a proof obligation for every use of `pretend`
- imposed restrictions to avoid inconsistencies by eg. Girard's paradox or nonterminating functions.

- We could use `pretend` not just for computations in types, but also to conjure up proofs for say that some structure is a monoid, some function is continuous, etc.
- We could choose not to prove the obligations, but simply use `pretend` as a lightweight formal method to keep track of assumptions that are made.

Conclusions

- **Several ways of combining computer algebra and theorem proving have been proposed; exploiting the type system of Aldor is another.**
- **Reasoning could be supported by extending (a subset of) Aldor to compute with types when typechecking, or by exporting proof obligations for every use of `pretend`.**
- **Restrictions would be needed to avoid inconsistencies by eg. Girard's paradox or nonterminating functions.**
- **Different levels of rigour are possible, eg. one could simply use `pretend` to document assumptions that are made.**

Links and references

- Aldor : `www.aldor.org`
- Axiom : `www.nongnu.org/axiom`

Papers:

- The type system of Aldor. Erik Poll and Simon Thompson, 1999.
- Adding the axioms to Axiom. Erik Poll and Simon Thompson, 1999.
- Logic and dependent types in the Aldor Computer Algebra System. Simon Thompson, 2000.
- The Aldor- - language. Simon Thompson and Leonid Timochouck, 2001.

Computation in types

Computation in types is not without drawbacks!

- **decidability**
  Typing effectively becomes semi-decidable:
  eg. deciding `Vector(Ack(100,100)+1) = Vector(1+Ack(100,100))`, where `Ack` is Ackerman function, takes ages.
  (This does not appear to be a problem in practice?)
- **abstraction**
  Whether `Vector(x+0) = Vector(x)` depends on definition of +.
  So definition of a function like + (and the intensional equality its provides) affects all theories that use it.