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• work done last century, so probably outdated in parts
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 (e.g. continuity of function), silent switching from domain (e.g. 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 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](http://www.aldor.org) and [www.nongnu.org/axiom](http://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
Aldor provides explicit parametric polymorphism

\[
polyid \ (T:\text{Type}, \ t:T) : T = t;
\]

and treats types as first class citizens

\[
idType \ (T:\text{Type}) : \text{Type} = T;
\]

(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 $\lambda$, 

\[
\text{polyid} : \ (T:\text{Type}) \rightarrow T \rightarrow T \\
\quad == \ (T:\text{Type}) \ (t:T) : T \ +\rightarrow \ t;
\]

\[
\text{idType} : \ Type \rightarrow \ Type \\
\quad == \ (T:\text{Type}) : Type \ +\rightarrow \ T;
\]

In Aldor, $(\lambda x:A. \ b)$ is written as $(x:A) : B \ +\rightarrow \ b$
Impredicativity and Type:Type

The type of the polymorphic identity is a type

\[
\text{PolyidType} : \text{Type} == (T:\text{Type}) \to T \to T;
\]

In fact, Type is a type

\[
\text{MyType} : \text{Type} == \text{Type};
\]
\[
\text{MyTypeArrowType} : \text{Type} == \text{Type} \to \text{Type};
\]
\[
\text{MyType2} : \text{Type} == (\text{polyid Type}) \text{Type};
\]

Warning: application associates to the right!
Aldor provides a powerful notion of abstract datatype

\[
\text{Monoid} : \text{Category} == \text{BasicType} \text{ with } \{ \\
1 : \% ; \\
* : (\%, \%) \to \% \\
\}
\]

Intuitively, this is the type of all monoids.

In type theory, \[\sum X : \text{Type} . \text{Record} (1 : X, * : X \times X \to X)\]
Elements of categories are **domains**

\[
\text{IntegerAdditiveMonoid} : \text{Monoid} == \text{add} \begin{cases} 
\text{Rep} == \text{Integer}; \\
\text{import from Integer}; \\
1 : \% == \text{per } 0; \\
(x:\%)*(y:\%) : \% == \text{per}((\text{rep } x) + (\text{rep } y)) 
\end{cases}
\]

Here \text{per} : \%\rightarrow\text{Rep} and \text{rep} : \text{Rep}\rightarrow\% are conversion functions between the abstract carrier \% and the concrete representation Integer.

**NB.** no guarantee that elements of \text{Monoid} are monoids!
Inheritance

Categories can extend other categories, eg.

\[
\text{Monoid : Category == BasicType with } \{ \\
1 : \%; \\
* : (\%,\%) \to \% \\
\}
\]

extends

\[
\text{BasicType : Category == with } \{ \\
= : (\%,\%) \to \%; \\
\}
\]

This provides a rich subtyping hierarchy
Aldor’s category hierarchy
Categories are first-class citizens, eg.

\[
\text{FancyOutput}(c:\text{Category}) : \text{Category} \\
\quad = c \text{ with } \{ \text{prettyPrint} : \% \rightarrow \text{BoundingBox} \};
\]

In fact, \text{Category} is a type

\[
\text{MyCategory} : \text{Type} = \text{Category};
\]
Dependent types

Aldor supports dependent types, eg. we can define

\texttt{Vector: (n:Integer) Type;}

\texttt{vectorSum: (n:Integer) \rightarrow Vector(n) \rightarrow Integer;}

\texttt{append: (n:Integer,m:Integer,Vector(n),Vector(m)}
\hspace{1em} \rightarrow \texttt{Vector(n+m);} \hspace{1em}

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

\[\text{eight : Integer == 8;}\]

but \textit{not}

\[\text{idType (T:Type) : Type = T;}\]
\[\text{seven : idType(Integer) == 7;}\]
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.
Aldor is not type safe, as it has the following `pretend` construct:

\[
\begin{array}{c}
\text{t: T} \\
\text{S: Type} \\
\text{(t pretend S) : S}
\end{array}
\]

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

```plaintext
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.
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](http://www.aldor.org)
- Axiom: [www.nongnu.org/axiom](http://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.
Computation in types

Computation in types is not without drawbacks!

- **decidability**
  Typing effectively becomes semi-decidable:
  eg. deciding $\text{Vector}(\text{Ack}(100,100)+1) = \text{Vector}(1+\text{Ack}(100,100))$, where $\text{Ack}$ is Ackerman function, takes ages.  
  *(This does not appear to be a problem in practice?)*

- **abstraction**
  Whether $\text{Vector}(x+0) = \text{Vector}(x)$ depends on definition of $+$.  
  So definition of a function like $+$ (and the intensional equality its provides) affects all theories that use it.