Aldor: An Introduction to the Language

Stephen M. Watt

University of Western Ontario

Edinburgh, September 26 2000

Aldor — Motivation

- Original motivation computer algebra,
  as extension language for the AXIOM system.
- Need to model rich relationships among mathematical structures.
- More emphasis on uniform handling of values independent of their type;
  less emphasis on a particular object model.
- Primary considerations:
  - generality
  - composibility
  - efficiency
  - interoperability

Aldor — Motivation II

- Be able to express the *requirements* and the rich *relationships* among inputs.
- Be albe to express *guarantees* on the results.

- Then have a language in which to explore how to
  weaken the requirements or
  strengthen the guarantees.

Context

- Scratchpad II (IBM) 1984-1990.
- $A\sharp$ as extension language to Axiom 1990-1994
  Generate code to run in Lisp envorinment or linked into C applications.
- Available with Axiom 2 from NAG
- FRISCO 1996-1999 (NAG, INRIA, CNRS, U Cantabria, U Pisa)
  C++ and Fortran interfaces, algebra libraries, etc

Aldor — Characterization

- Imperative language, statically typed, strict.
- Natural blend of functional and OO styles.

  Both types an functions are *first class*:
  can be constructed during execution and used as any other value.

  Functional features: closures, currying, etc.

  Pervasive use of *dependent types* – allows static checking
  of dynamic objects.
  OO features via dependent types
- Based on ADT, separate inheritance of interface and implementation.

  Types values belong to type categories.
  Parametric polymorphism.

  Post facto type extensions allow complex libraries to be separated
  into decoupled components.

First Examples I

```
#include "axllib"

double(n: Integer): Integer == n + n
```

First Examples II

```
#include "axllib"

-- Compute a square root by six steps of Newton's method.
-- This gives 17 correct digits for numbers between 1 and 10.

DF ==> DoubleFloat;

miniSqrt(x: DF): DF == {
        r := x;
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r := (r*r + x)/(2.0*r);
        r
}
```

First Examples III

```
#include "axllib"

factorial(n: Integer): Integer == {
        p := 1;
        for i in 1..n repeat p := p * i;
        p
}

import from Integer;

print << "factorial 10 = " << factorial 10 << newline
```

First Examples IV

```
#include "axllib"

MiniList(S: BasicType): LinearAggregate(S) == add {
        Rep == Union(nil: Pointer, rec: Record(first: S, rest: %));

        import from Rep, SingleInteger;

        local cons (s:S,l:%):% == per(union [s, l]);
        local first(l: %): S   == rep(l).rec.first;
        local rest (l: %): %   == rep(l).rec.rest;

        empty (): %            == per(union nil);
        empty?(l: %):Boolean   == rep(l) case nil;
        sample: %              == empty();

        [t: Tuple S]: % == {
                l := empty();
                for i in length t..1 by -1 repeat
                        l := cons(element(t, i), l);
                l
        }
        [g: Generator S]: % == {
                r := empty(); for s in g repeat r := cons(s, r);
                l := empty(); for s in r repeat l := cons(s, l);
```

```
        l
}
generator(l: %): Generator S == generate {
        while not empty? l repeat {
                yield first l; l := rest l
        }
}
apply(l: %, i: SingleInteger): S == {
        while not empty? l and i > 1 repeat
                (l, i) := (rest l, i-1);
        empty? l or i ~= 1 => error "No such element";
        first l
}
(l1: %) = (l2: %): Boolean == {
        while not empty? l1 and not empty? l2 repeat {
                if first l1 ~= first l2 then return false;
                (l1, l2) := (rest l1, rest l2)
        }
        empty? l1 and empty? l2
}
(out: TextWriter) << (l: %): TextWriter == {
        empty? l => out << "[]";
        out << "[" << first l;
        for s in rest l repeat out << ", " << s;
        out << "]"
}
```

Language Elements: Outline

- Abstraction and application
- Names
- Variables and Constants
- Sequencing
- Dependent types, Types as values
- Domains
- Categories
- Categorical Generic Programming, Categories vs ABC
- Examples

Abstraction and Application

- Lambda abstraction:

```
(a: S): R +-> e

(a1: A1, a2: A2, ...): (R1, R2, ...) +-> e

(): () +-> e
```

- Application:

`f g x`	associates as `f (g x)`
`f.g.x`	associates as `(f g) x`
`f(a)(b)`	associates as `(f(a)) (b)`
`e * f`	application of infix name `*`.

Application is generic and does not necessarily imply the operator is a function:

E.g. `f x`, `M v`, `t.left`, ...

Names

- Syntax:

Alphanumeric, !, ?:	`freddy12`, `prime?`, `update!`
Arbitrary:	`_.entry`, `_if`, `_12`
Specific others:	`0`, `1`, `<`, `+`, `=`, etc.

- What are special operators in other languages are simply *names* with an infix syntax property, but no special semantics:

```
a + b     <=>    (+)(a,b)

+ == (a: R, b: R): R +-> ...

map(+, l)
```

- Certain syntaxes correspond to particular applications:

```
[a,b,c]    --  bracket(a,b,c)
a i := v   --  set!(a,i,v)
```

Variables and Constants

- Variable assignment:

```
v : T := e          v := e          (a, b, c) := e
```

- Constant definition:

```
c: T == e           c == e
```

- Variables and constants are introduced to a scope by an assignment or definition, and are visible over the whole scope.

  They may be explicitly declared, if desired.

- Constant names may be overloaded; variable names cannot.

- Application on LHS is a syntactic sugar:

  Assignment

```
a(i1,i2,...) := e      <=>    set!(a, i1, i2, ..., e)

a.i.j := e             <=>    set!(a.i, j, e)
```

  Definition

```
f(a: A): R == e        <=>    f: (a: A) -> R == (a: A): R +-> e


f(a: A)(b: B)(c: C): R == e    <=>

f: (a: A) -> (b: B) -> (c: C) -> R ==

   (a: A): (b: B) -> (c: C) -> R  +->  (b: B): (c: C) -> R  +->  (c: C):  R  +->  e
```

- The *type* of each variable or constant is fixed, and visible to all that can see it.
- The *value* of constants may also be made visible, using `define`.

```
define n: Integer == 3;
```

Sequencing

Compound expression:	`{e1; e2; ...; eN}`

Conditional:	`if C then T` *`[`*`else E`*`]`*
	`e1 and e2`
	`e1 or e2`

	`{a; b; c => d; e; f => g; h}`
Loop:	*Iterator** `repeat E`
Collection:	`E` *Iterator**
Iterator:	`for lhs in E` *`[`*`| E`*`]`* `while E`
Generator:	`generate E`
Exception handling:	`try E` *`[`*`but X in ...`*`]`* *`[`*`always F`*`]`*

`Exits:`	`iterate L` `break L`
	`return E` `yield E`
	`except E` `goto L`

`Not reached:`	`never`

Sequencing Examples

```
(a: Integer)..(b: Integer): Generator Integer == generate {
    while a <= b repeat { yield a; a := a + 1 }
}


for i1 in a1..b1  for i2 in a2..b2 repeat
    for j1 in c1..d1  for j2 in c2..d2 repeat {
        ...
    }

l := [i^2 for i in 1..10 | even? i];

y := {
    a < 0 => sm;
    a = 0 => sz;
    a > 0 => sp;
    never
}
```

Dependent Types: Products

- Usual records and tuples can be modelled with "Cartesian product" types, composed of components which can be independently selected.
- Generalized to "dependent products" where the *type* of one component depends on the *value* of another, $(a : A) \times B(a)$. E.g

```
rna:       Record(n: Integer, a: Array(n, Window))

eigenvals: Matrix(n,n,K) ->
           (p: UnivPoly(K), Array(n, AlgExtension(p, K))
```

  Must destructure via parallel binding;
  cannot select all components independently.

- These correspond to existentially quantified types in some literature.

Dependent Types: Mappings

- Functions may return results whose *type* depends on the *value* of their arguments. E.g.

```
72 mod 5                          ==> 2 (mod 5)
```

  This relation can be expressed with "dependent mapping" types.

```
mod:      (Integer, m: Integer) -> IntegerMod(m)
```

- These correspond to universally quantified types in some literature.

Types as Values

- If types can be used as values, then dependent types become very natural for generic programming.

```
identity: (n: Integer, R: Ring) -> Matrix(n, n, R)

identity(2, Float)           ==> [1.0  0.0]
                                 [0.0  1.0]
```

- Parametric polymorphism:

```
commutator(R: Ring)(p: R, q: R): R == p*q - q*p;
```

  This easily addresses the argument-coherence problem without dynamic type tests.

- Mutually dependent products are useful in expressing relationships among types.

Domains

- A "domain" exports a collection of related constants.

```
add {
    gcd(n: Integer, m: Integer): Integer == ...;
    lcm(n: Integer, m: Integer): Integer == ...;
}
```

- All constants defined within a domain are exported by default unless declared local.

```
add {
    local gcd(n: Integer, m: Integer): Integer == ...;

    lcm(n: Integer, m: Integer): Integer == ...;
}
```

- Within a domain-valued expression, the name "%" refers to the domain being computed (and is fix-pointed).
- "%" may be used as a type name.
- Example

```
FunkyType == add {
    coerce(n: Integer): % == ...;
    gcd(a: %, b: %): % == ...;
    lcm(a: %, b: %): % == ...;
}
```

- Importing from domains:
  Constants exported by domains may be imported into a program's scope.

```
-- Import some constants

import {
    nil:     %;
    empty?:  % -> Boolean;
    cons:    (Float, %) -> %;
    first:   % -> Float;
    rest:    % -> %
} from List(Float);


-- Import all exported constants.

import from List(Float);
```

- A type `Rep` is defined, to give a representation for `%`.
- `rep` and `per` are type conversions:

```
rep: % -> Rep
per: Rep -> %
```

- Example

```
Complex == add {
    R  ==> DoubleFloat;

    Rep == Record(real: R, imag: R);

    import from Rep, R;

    real(u: %): R == rep(u).real;
    imag(u: %): R == rep(u).imag;
    complex(a: R, b: R): % == per [a, b];

    (u: %) + (v: %): % == complex(real u + real v, imag u + imag v);
    (k: R) * (u: %): % == complex(k * real u, k * imag u);
    abs(u: %): R       == sqrt(real(u)^2 + imag(u)^2);
    ...
}
```

- The `add` operator may be used to build on an existing domain.
- Example

```
Polygon == add {
    Rep == List Complex;
    new(l: List Complex): % == per l;
    vertex(p: %, i: Integer): Complex == rep(p).i;
}

Square == Polygon add {
    area(s: %): DoubleFloat == {
        s1 := vertex(s,1) - vertex(s,0);
        s2 := vertex(s,2) - vertex(s,0);
        abs(s1 * s2)
    }
}
```

Categories

- Every domain belongs to the type `Type`, but it is useful to be able to assert more.
- *Categories* provide *subtype* information about domains values, indicating what exports must be present.
- A basic category-valued expression gives a list of exports:

```
with { coerce: Integer -> %;   lcm: (%, %) -> %; }
```

- Categories may be used in declarations:

```
FunkyType: with {
    coerce: Integer -> %;
    lcm:    (%, %)  -> %;
}
== add {
    coerce: Integer -> % == ...
    lcm: (%, %) -> %     == ...
    gcd: (%, %) -> %     == ...
}
```

Note the type of the rhs is

```
with { coerce: Integer->%; lcm: (%,%)->%; gcd: (%,%)->% }
```

which is a subtype of the declared type of the lhs

```
with { coerce: Integer->%; lcm: (%,%)->% }
```

- We may create category–valued constants.
- It is usually necessary to know facts about the category value (e.g. the export list), these constants are typically `defined`.

  (Recall this makes the constant's value public knowledge.)

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

define Finite: Category == with {
    cardinality: Integer
}
```

- The `Join` operator combines catgories, providing multiple inheritance:

```
define FiniteMonoid: Category == Join(Monoid, Finite)
```

  The expression "`C with {...}`" is equivalent to "`Join(C, with {...})`"

- One may use functions to compute categories:

```
define Module(R: Ring): Category == Ring with {
    *: (R, %) -> %
}

define ComplexCategory(R: Ring): Category ==  Module(R) with {
    complex: (R, R) -> R;
    real:    % -> R;
    imag:    % -> R;
}
```

Categorical Generic Programming

- In generic programming we may **use categories to specify the requirements on parameters and to state properties of the results.**
- Example: `Polynomial(R: Ring): Module(R) == add { ... }`

  `Polynomial` has the dependent mapping type `(R: Ring) -> Module(R)`.

  It takes one parameter R, which is a domain satisfying the `Ring` category, and produces another ring, which is also an R-module.

  Static analysis can use the fact that R provides all the operations required by Ring.
- This allows static resolution of names and separate compilation of parameterized modules.

Categories vs Abstract Base Classes

- Want a base type `X`, which provides an internal multiplication, `X * X -> X`.
- Suppose we define an abstract base class, and derive

```
Integer                  3 * 3                          9

IntegerMod(5)            6 * 7                          2

DoubleFloat              3.2 * 6.0                      19.2

Polynomial(x,Integer)    (x^2 + 1)*(x-1)                x^3-x^2+x-1

SqMatrix(2,DoubleFloat) [1.0 1.0] * [1.0 2.0]           [3.1  3.0]
                        [0.0 1.0]   [2.1 1.0]           [2.1  1.0]
```

Categories vs Abstract Base Classes

- Want a base type `X`, which provides an internal multiplication, `X * X -> X`.
- Suppose we define an abstract base class, and derive

```
Integer                  3 * 3                          9

IntegerMod(5)            6 * 7                          2

DoubleFloat              3.2 * 6.0                      19.2

Polynomial(x,Integer)    (x^2 + 1)*(x-1)                x^3-x^2+x-1

SqMatrix(2,DoubleFloat) [1.0 1.0] * [1.0 2.0]           [3.1  3.0]
                        [0.0 1.0]   [2.1 1.0]           [2.1  1.0]


IntegerMod(5) * SqMatrix(2, DoubleFloat)  ??????????
```

Categories vs Abstract Base Classes

- The difference is that with Categories, we have *e.g.*

$$2 \in \texttt{IntegerMod(5)} \in X$$

$$6.2 \in \texttt{DoubleFloat} \in X$$

  whereas with derived classes we have

$$2 \in \texttt{IntegerMod(5)} \subset X$$

$$6.2 \in \texttt{DoubleFloat} \subset X$$

Example: Prime Number Sieve

```
# include "axllib.as"

import from Boolean, SingleInteger;

sieve(n: SingleInteger): SingleInteger  == {
    prime?: PrimitiveArray Boolean := new(n, true);

    np := 0;

    for p in 2..n | prime? p repeat {
        np := np + 1;
        for i in 2*p..n by p repeat prime? i := false;
    }
    np
}

for i in 1..6 repeat {
    n := 10^i;
    print << "There are " << sieve n << " primes <= " << n;
    print << newline;
}
```

Example: Multiple Values

```
#include "axllib.as"
import from Integer;

I      ==> Integer;
MapIII ==> (I,I,I) -> (I,I,I);

(f: MapIII) * (g: MapIII): MapIII ==
        (i:I, j:I, k: I): (I,I,I) +-> f g (i,j,k);

id: MapIII ==
        (i:I, j:I, k: I): (I,I,I) +-> (i,j,k);

(f: MapIII) ^ (p: Integer): MapIII == {
        p < 1  => id;
        p = 1  => f;
        odd? p => f*(f*f)^(p quo 2);
        (f*f)^(p quo 2);
}

cycle(a: I, b: I, c: I): (I,I,I) == (c, a, b);

cycle(1,2,3);
cycle cycle  (1,2,3);
(cycle*cycle)(1,2,3);
(cycle^10)   (1,2,3);
```

Example: Constructing an alternate view

```
-++ This constructor creates the
-++ MonogenicLinearOperator domain which is
-++ opposite to P in the ring sense.
-++ That is, as sets P = %
-++ but a * b in P is equal to b * a in %.

OppositeMonogenicLinearOperator(P: MonogenicLinearOperator R, R: Ring) :

  MonogenicLinearOperator(R) with {
       if P has DifferentialRing then DifferentialRing;
       op: P -> %;
       po: % -> P;
  }

== P add {
       Rep == P;
       import from Rep;

       op(a: P): % == per a;
       po(x: %): P == rep x;
       (x: %) * (y: %): % ==  op(po y * po x);
```

```
++ This domain defines a ring of differential operators which act
++ upon an A-module, where A is a differential ring.
++ Multiplication of operators corresponds to functional composition:
++   (L1 * L2).(f) = L1 L2 f
NI ==> NonNegativeInteger;
UP ==> SparseUnivariatePolynomial;

inearOrdinaryDifferentialOperator(
    A: DifferentialRing,
    M:  LeftModule(A) with differentiate: % -> %
  : MonogenicLinearOperator(A) with {
      D: %;
      apply: (%, M) -> M;
      ...
      if A has Field then {
          leftDivide:   (%, %) -> Record(quotient: %, remainder: %);
              ++ [q,r] = leftDivide(a,b)  means a=b*q+r, deg r < deg b

          rightDivide:  (%, %) -> Record(quotient: %, remainder: %);
              ...
      }
    }
```

```
== SUP(A) add {

      ...

      if A has Field then {
          Op    == OppositeMonogenicLinearOperator(%, A);

          DOdiv == NonCommutativeOperatorDivision(%, A);
          OPdiv == NonCommutativeOperatorDivision(Op,A);

          leftDivide(a, b) == leftDivide(a, b)$DOdiv;
          rightDivide(a,b) == {
              qr := leftDivide(op a, op b)$OPdiv;
              [po qr.quotient, po qr.remainder]
          }
          ...
      }
```

Example: Constructors as Functional Arguments

```
#include "axllib"

I  ==> SingleInteger;
Ag ==> (S: BasicType) -> LinearAggregate S;

-- This function takes two type constructors as arguments and
-- produces a new function to swap aggregate data structure layers.

swap(X:Ag,Y:Ag)(S:BasicType)(x:X Y S):Y X S == [[s for s in y] for y in x];

-- Form an array of lists:

al: Array List I := array(list(i+j-1 for i in 1..3) for j in 1..3);

print << "This is an array of lists: " << newline;
print << al << newline << newline;

-- Swap the structure layers:

la: List Array I := swap(Array,List)(I)(al);

print << "This is a list of arrays:  " << newline;
print << la << newline
```