Aldor: The language and recent directions

Stephen M. Watt

University of Western Ontario

Sophia Antipolis, June 15 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 — 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.

Language Elements: Outline

- Abstraction and application
- Names
- Variables and Constants
- Sequencing
- Dependent types, Types as values
- Domains
- Categories
- Categorical Generic Programming, Categories vs ABC
- Post Facto Extensions
- The Object Constructor
- 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];

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

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

- Often certain operations are equivalent to certain combination of others.

  Aldor therefore allows default values to be specified:

```
AbelianGroup: Category == with {
    0: %;
    +: (%, %) -> %;
    -: % -> %;
    -: (%, %) -> %;

    default (a: %) - (b: %): % == a + (-b);
}
```

- Defaults are late binding. A domain satisfying `AbelianGroup` will provide its own version of an export in preference to a default.

- Typically, additional assertions can be made when more properties are known to be satisfied.

  Aldor therefore provides conditional construction of domains and categories.

```
UnivariatePolynomialCategory(R: Ring): Category == Module R with {
    monomial: (R, Integer) -> %;

    if R has Field then  EuclideanDomain
}

SparseUnivariatePolynomial(R: Ring): UnivariatePolynomialCategory(R) == {
    ...
    if R has Field then {
        gcd(p: %, q: %): % == ...
    }
}
```

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

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

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

Post Facto Extensions

- If T is a type-valued constant, then its meaning may be extended with a definition of the form:

  ```
  extend T: C == D
  ```

- C is a new category to which T will now belong
  D is a domain providing the newly required exports.

- Example: `Integer` can made to satisfy the new categories `FancyOutput` and `DifferentialRing` by providing appropriate extensions:

  ```
  extend Integer: FancyOutput == add {
          box(n: %): BoundingBox == [1, ndigits n, 0, 0]
  }

  extend Integer: DifferentialRing == add {
          differentiate(n: %): % == 0
  }
  ```

- This generalizes to the extension of functions which compute types.

The Object Constructor

- From the Aldor library:

```
Object(C: Category): with {
        object:         (T: C, T) -> %;
        avail:          % -> (T: C, T);
}
== add {
        Rep == Record(T: C, val: T);
        import from Rep;

        object(T: C, t: T) : %    == per [T, t];
        avail (ob: %) : (T: C, T) == explode rep ob;
}
```

- E.g.

```
import from String, Integer, List Integer;

boblist: List Object BasicType := [
        object (String,       "Ahem!"),
        object (Integer,      42),
        object (List Integer, [1,2,3,4])
];

bobfun(T: BasicType, t: T) : () ==
    print << "This prints itself as: " << t << newline;

for bob in boblist repeat bobfun(avail bob)
```

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: Using Type Constructors

```
+++  CliffordAlgebra(n, K, Q) defines a vector space
+++  of dimension 2**n over K, given a quadratic form
+++  Q on K**n.
+++
+++  If e[i], 1<=i<=n is a basis for K**n then a basis
+++  for the Clifford Algebra is:
+++     1,
+++     e[i]         (1 <= i <= n),
+++     e[i1]*e[i2]  (1 <= i1 < i2 <= n)
+++     ...
+++     e[1]*e[2]*..*e[n]
+++
+++  The algebra is defined by the relations
+++     e[i]*e[j] = -e[j]*e[i]  (i &lnot. j),
+++     e[i]*e[i] = Q(e[i])
+++
+++  Examples of Clifford Algebras are:
+++   gaussians, quaternions, exterior algebras and
+++  spin algebras.

PI ==> PositiveInteger;
NNI==> NonNegativeInteger;
```

```
CliffordAlgebra(n: PositiveInteger, K: Field, Q: QuadraticForm(n,K)):

    Join(Ring, Algebra(K), VectorSpace(K)) with {

        e: PI -> %;

        monomial: (K, List PI) -> %;
          ++ monomial(c,[i1,i2,...,iN])
          ++   produces the value given by c*e(i1)*e(i2)*...*e(iN).

        coefficient:  (%, List PI) -> K;
          ++ coefficient(x,[i1,i2,...,iN])
          ++   extracts the coefficient of e(i1)*e(i2)*...*e(iN) in x.
    }

== add {
        Qeelist ==  [Q unitVector i for i in 1..n];
        dim     ==  2**n;
        Rep     == PrimitiveArray K;

        New     ==> new(dim, 0$K)$Rep;

        default x, y, z: %;
        default c: K;
        default m: Integer;
```

```
characteristic(): NNI       == characteristic()$K;
dimension(): CardinalNumber == dim::CardinalNumber;

x = y: Boolean == {
    for i in 0..dim-1 repeat
          x.i &lnot.= y.i => return false;
    true
}

x + y: % == [ x.i + y.i  for i in 0..dim-1];
x - y: % == [ x.i - y.i  for i in 0..dim-1];
- x: %   == [ - x.i      for i in 0..dim-1];
m * x: % == [ m*x.i      for i in 0..dim-1];
c * x: % == [ c*x.i      for i in 0..dim-1];

0: %         == [0 for i in 0..dim-1];
1: %         == {z := New; z.0 := 1; z}
coerce(m): % == {z := New; z.0 := m::K; z}
coerce(c): % == {z := New; z.0 := c; z}

e b: % == {
    b:NNI > n => error "No such basis element";
    iz := 2**((b-1):NNI);
    z := New; z.iz := 1; z
}
```

```
        x * y: % == {
            z := New;
            for ix in 0..dim-1 repeat
                if x.ix &lnot.= 0 then for iy in 0..dim-1 repeat
                    if y.iy &lnot.= 0 then addMonomProd(x.ix,ix,y.iy,iy,z);
            z
        }
        ...
}
```

```
-- The complex numbers as a Clifford Algebra

K := FRAC POLY INT

   (1)  Fraction Polynomial Integer

qf: QFORM(1, K) := quadraticForm(matrix([[-1]])$(SQMATRIX(1,K)))

   (2)  [- 1]

C := CLIF(1, K, qf)

   (3)  CliffordAlgebra(1,Fraction Polynomial Integer,MATRIX)

i := e(1)$C

   (4)  e
         1

x := a + b * i

   (5)  a + b e
               1

y := c + d * i
```

```
   (6)  c + d e
               1

x * y

   (7)  - b d + a c + (a d + b c)e
                                  1
```

```
-- The quaternions as a Clifford Algebra

qf:QFORM(2, K) :=quadraticForm matrix([[-1, 0], [0, -1]])$(SQMATRIX(2,K))

        |- 1   0 |
   (8)  |        |
        | 0   - 1|

H  := CLIF(2, K, qf)


   (9)  CliffordAlgebra(2,Fraction Polynomial Integer,MATRIX)

i  := e(1)$H

   (10)  e
          1

j  := e(2)$H

   (11)  e
          2

k  := i * j

   (12)  e e
```

```
          1 2

x := a + b * i + c * j + d * k

   (13)  a + b e  + c e  + d e e
                1      2      1 2

y := e + f * i + g * j + h * k

   (14)  e + f e  + g e  + h e e
                1      2      1 2

x * y

   (15)
     - d h - c g - b f + a e + (c h - d g + a f + b e)e
                                                       1
   +
     (- b h + a g + d f + c e)e  + (a h + b g - c f + d e)e e
                               2                           1 2
```

```
-- The exterior algebra on a 3 space.

qf: QFORM(3, K) := quadraticForm(0::SQMATRIX(3,K))


         |0  0  0|
         |       |
   (18)  |0  0  0|
         |       |
         |0  0  0|

Ext := CLIF(3,K,qf)

   (19)  CliffordAlgebra(3,Fraction Polynomial Integer,MATRIX)

i := e(1)$Ext

   (20)  e
          1

j := e(2)$Ext

   (21)  e
          2

k := e(3)$Ext
```

```
   (22)  e
          3

x := x1*i + x2*j + x3*k

   (23)  x1 e  + x2 e  + x3 e
              1       2       3

y := y1*i + y2*j + y3*k

   (24)  y1 e  + y2 e  + y3 e
              1       2       3

x + y

   (25)  (y1 + x1)e  + (y2 + x2)e  + (y3 + x3)e
                    1             2             3

x * y + y * x

   (26)  0

-- In n space, a grade p form has a dual n-p form.
-- In particular, in 3 space the dual of a grade 2 element identifies
--   e1*e2->e3, e2*e3->e1, e3*e1->e2.
```

```
dual2 a ==
    coefficient(a,[2,3])$Ext * i + _
    coefficient(a,[3,1])$Ext * j + _
    coefficient(a,[1,2])$Ext * k

-- The vector cross product is then given by
dual2(x*y)

   (28)  (x2 y3 - x3 y2)e  + (- x1 y3 + x3 y1)e  + (x1 y2 - x2 y1)e
                         1                     2                   3
```

```
-- The Dirac Algebra used in Quantum Field Theory.

K := FRAC INT


   (29)  Fraction Integer

g: SQMATRIX(4, K) := [[1,0,0,0],[0,-1,0,0],[0,0,-1,0],[0,0,0,-1]]

         |1   0    0    0 |
         |                |
         |0  - 1   0    0 |
   (30)  |                |
         |0   0   - 1   0 |
         |                |
         |0   0    0   - 1|

qf: QFORM(4, K) := quadraticForm g

         |1   0    0    0 |
         |                |
         |0  - 1   0    0 |
   (31)  |                |
         |0   0   - 1   0 |
         |                |
         |0   0    0   - 1|
D := CLIF(4,K,qf)

   (32)  CliffordAlgebra(4,Fraction Integer,MATRIX)

-- The usual notation is gamma sup i.
gam := [e(i)$D for i in 1..4]

   (33)  [e ,e ,e ,e ]
           1  2  3  4

-- There are various contraction identities of the form
-- g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) =
--     2*(gam(s)gam(m)gam(n)gam(r) + gam(r)*gam(n)*gam(m)*gam(s))
-- where the sum over l and t is implied.

-- Verify this identity for m=1,n=2,r=3,s=4
m := 1; n:= 2; r := 3; s := 4;

lhs := reduce(+,[reduce(+, _
    [g(l,t)*gam(l)*gam(m)*gam(n)*gam(r)*gam(s)*gam(t) _
                 for l in 1..4]) for t in 1..4])

   (35)  - 4e e e e
             1 2 3 4
```

```
rhs := 2*(gam s * gam m*gam n*gam r + gam r*gam n*gam m*gam s)

   (36)  - 4e e e e
             1 2 3 4
```

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
NNI ==> NonNegativeInteger;
SUP ==> SparseUnivariatePolynomial;

LinearOrdinaryDifferentialOperator(
     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 Funtional 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
```

Aldor — Implementation

- Optimizing compiler
- Interpreted interactive environment for the same language
- Generates
  - Stand-alone executable programs
  - Object libraries in native OS formats
  - Portable byte code libraries
  - C or Lisp source

Optimizations

- Procedural integration.
- Data structure elimination
- Constant folding, etc.

Example

```
#include "axllib.as"

SI ==> SingleInteger;
F  ==> DoubleFloat;
CF ==> Complex F;

import from CF;
inline from CF;

default xa, xb, ya, yb: F;
default xn, yn, MAX:   SI;
default draw: (x: SI, y: SI, n: SI) -> ();

drawMand(xa, xb, xn, ya, yb, yn, draw, MAX): () == {

  mandel(c: CF): SI == {
       z: CF := 0;
       n: SI := 0;
       while norm z < 4.0 for free n in 1..MAX repeat z  := z*z + c;
       n
   }
   for y in step(yn)(ya, yb) for yi in 1..yn repeat
       for x in step(xn)(xa, xb) for xi in 1..xn repeat
           draw(xi, yi, mandel complex(x, y));
}
```

FOAM for the Inner Loop

```
   ...
(Label 4)
  (CCall NOp (Loc 14))
  (If (Cast Bool (CCall Word (Loc 13))) 5)
  (Set (Loc 12 x) (CCall Word (Loc 15)))
  (CCall NOp (Loc 19))
  (If (Cast Bool (CCall Word (Loc 18))) 5)
  (Set (Loc 11 xi) (CCall Word (Loc 20)))
  (CCall NOp (Par 6 draw) (Loc 11 xi) (Loc 0 yi)
    (OCall Word (Const 2 mandel) (Env 0)
      (CCall Word (Lex 1 9 complex) (Loc 12 x) (Loc 1 y))))
  (Goto 4)
    ...
```

Optimized FOAM for the Inner Loop

```
  ...
  (Set (Loc 10 double) (DFlo 4.000000000000000e0))
  (If (BCall DFloLE (Loc 10 double) (Loc 13 double)) 9)
  (Set (Loc 9 a) (SInt 1))
(Label 1)
  (If (BCall SIntLT (Cast SInt (Par 7 MAX)) (Loc 9 a)) 2)
  (Set (Loc 0) (Loc 9 a))
  (Set (Loc 20 double) (BCall DFloMinus (Loc 27) (Loc 28)))
  (Set (Loc 15 double) (BCall DFloTimes (Loc 17 double) (Loc 18 double)))
  (Set (Loc 14 double) (BCall DFloTimes (Loc 18 double) (Loc 17 double)))
  (Set (Loc 19 double) (BCall DFloPlus (Loc 15 double) (Loc 14 double)))
  (Set (Loc 17 double) (BCall DFloPlus (Loc 20 double) (Loc 22 double)))
  (Set (Loc 18 double) (BCall DFloPlus (Loc 19 double) (Loc 24 double)))
  (Set (Loc 27) (BCall DFloTimes (Loc 17 double) (Loc 17 double)))
  (Set (Loc 28) (BCall DFloTimes (Loc 18 double) (Loc 18 double)))
  (Set (Loc 13 double) (BCall DFloPlus (Loc 27) (Loc 28)))
  (Set (Loc 10 double) (DFlo 4.000000000000000e0))
  (If (BCall DFloLE (Loc 10 double) (Loc 13 double)) 9)
  (Set (Loc 9 a) (BCall SIntNext (Loc 9 a)))
  (Goto 1)
(Label 9)
  (CCall NOp (Par 6 draw) (Cast Word (Loc 8 a)) (Cast Word (Loc 16 a)) (Cast Word (Loc 0)))
  ...
```

Generated C for the Inner Loop

```
        ...
        T27 = T17_double * T17_double;
        T28 = T18_double * T18_double;
        T13_double = T27 + T28;
        T10_double = 4.000000000000000;
        if (T10_double <= T13_double) goto L9;
        T9_a = 1;
L1: if ((FiSInt) P7_MAX < T9_a) goto L2;
        T0 = T9_a;
        T20_double = T27 - T28;
        T15_double = T17_double * T18_double;
        T14_double = T18_double * T17_double;
        T19_double = T15_double + T14_double;
        T17_double = T20_double + T22_double;
        T18_double = T19_double + T24_double;
        T27 = T17_double * T17_double;
        T28 = T18_double * T18_double;
        T13_double = T27 + T28;
        T10_double = 4.000000000000000;
        if (T10_double <= T13_double) goto L9;
        T9_a = T9_a + 1;
        goto L1;
L9: fiCCall3(void, P6_draw,
                 (FiWord) T8_a, (FiWord) T16_a, (FiWord) T0);
        ...
```

Generated Common Lisp for the Inner Loop

```
  ...
  (setq t27 (|DFloTimes| |T17-double| |T17-double|))
  (setq t28 (|DFloTimes| |T18-double| |T18-double|))
  (setq |T13-double| (|DFloPlus| t27 t28))
  (setq |T10-double| (the |DFlo| 4.000000000000000e0))
  (when (|DFloLE| |T10-double| |T13-double|) (go |Lab9|))
  (setq |T9-a| (the |SInt| 1))
|Lab1|
  (when (|SIntLT| p7-max |T9-a|) (go |Lab2|))
  (setq t0 |T9-a|)
  (setq |T20-double| (|DFloMinus| t27 t28))
  (setq |T15-double| (|DFloTimes| |T17-double| |T18-double|))
  (setq |T14-double| (|DFloTimes| |T18-double| |T17-double|))
  (setq |T19-double| (|DFloPlus| |T15-double| |T14-double|))
  (setq |T17-double| (|DFloPlus| |T20-double| |T22-double|))
  (setq |T18-double| (|DFloPlus| |T19-double| |T24-double|))
  (setq t27 (|DFloTimes| |T17-double| |T17-double|))
  (setq t28 (|DFloTimes| |T18-double| |T18-double|))
  (setq |T13-double| (|DFloPlus| t27 t28))
  (setq |T10-double| (the |DFlo| 4.000000000000000e0))
  (when (|DFloLE| |T10-double| |T13-double|) (go |Lab9|))
  (setq |T9-a| (|SIntNext| |T9-a|))
  (go |Lab1|)
|Lab9|
  (|CCall| |P6-draw| |T8-a| |T16-a| t0)
  ...
```

Current and Future Directions

- Tools for Aldor: Aldordoc and XML.
- Incremental improvements:
  More specific integer types. Immediate arrays and structures.
- Aldor and parallel computation:
  Categories for supercomputer computation.
  Distributed GC. Thread-safe basic library.
- Theoretical work:
  Generic dependency, e.g. HashTable(k: K, E(k))
  Optimization of type towers.
- Language Convergence:
  Common ground between C++, GJ, F95, Aldor.
  Extension of CORBA IDL for parameterized types.
  Language Kit.