# Aldor: Interfaces

*Stephen M. Watt*

*University of Western Ontario*

Edinburgh, September 27 2000

# Outline

- Inter-language communication
- Additional category concepts
- Object oriented programming
- Ex post facto extensions
- Specializing generic constructors

# Interlanguage Communication

Call *to* and be called *by* programs in other languages/systems.

- Axiom
- C
- C++
- Fortran
- Maple

# Importing a function from C

```
--
-- arigato.as: A main Axiomxl program calling the C function ‘nputs’.
--
#include "axllib.as"
SI ==> SingleInteger;

import { nputs: (SI, String) -> SI } from Foreign C;
import from SI;

nputs(3, "Arigato gozaimasu!");
```

# Exporting a function to C

```
--
-- aside.as: An Aldor function made available to C.
--
#include "axllib.as"

SI ==> SingleInteger;

export { lcm: (SI, SI) -> SI } to Foreign C;

lcm(n: SI, m: SI): SI == (n quo gcd(n,m)) * m;
```

```
/*
 * cside.c:  A main C program calling the Aldor function 'lcm'.
 */
#include "foam_c.h"

extern FiSInt   lcm     (FiSInt, FiSInt);

int
main()
{
        printf("The lcm of 6 and 4 is %d\n", lcm(6,4));
        return 0;
}
```

# Category Defaults

- Often certain operations are equivalent to certain combination of others. Aldor therefore allows default values to be specified:

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

    default {
        (a: %) - (b: %): % == a + (-b);
        (n: Integer) * (a: %): % ==
            power(a, n)$BinaryPowering(%, 0, +);
    }
}
```

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

```
define OrderedSet: Category == Set with {
    <:  (%, %) -> Boolean;
    >:  (%, %) -> Boolean;
    <=: (%, %) -> Boolean;
    >=: (%, %) -> Boolean;
    default {
        (a: %) > (b: %): Boolean  == b < a;
        (a: %) <= (b: %): Boolean == a < b or a = b;
        (a: %) >= (b: %): Boolean == a > b or a = b;
    }
    <:  (%, %) -> (Boolean, %);
    <:  (Boolean, %, %) -> (Boolean, %);
    <:  (Boolean, %, %) -> Boolean;
    ...
    default {
        (a: %) < (b: %): (Boolean, %) == (a < b, b);
        (t: Boolean, a: %) < (b: %): (Boolean, %) == ...
        ...
    }
-

T: OrderedSet == add {
    s = t == ...
    t < t == ...
    -- Rest given by cat defaults
-
if  t0 < t1 <= t2 = t3 < t4 then ...
```

# Conditional Categories

- 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: %): % == ...
     }
}
```

```
define DirectProductCategory(S: Set): Category == Set with {

    apply: (%, SingleInteger) -> S;

    if S has Semigroup then  Semigroup;
    if S has Monoid    then Monoid;
    if S has Group     then Group;
    if S has Ring      then Ring;

    if S has Finite    then Finite;
    if S has Ordered   then Ordered;
}
```

# Categories as Views

```
MyDom(K: Field): Join(Ring, Algebra K, VectorSpace K)  == add ...


FM(M: Monoid) == ...
FR(R: Ring) == ...
FA(K: Field, A: Algebra K) == ...


FM(MyDom Rational)
FR(MyDom Z7)

FA(IntegerMod 7, MyDom IntegerMod 7)
```

# 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)
```

# 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.

# Specializing Generic Constructors

```
SI==> SingleInteger;

QuadraticForm(n: Integer, R: Ring):

    Join(Algebra R, MatrixCategory R) with {
        apply: (%, Vector(n, R)) -> R;
    }

== SquareMatrix(n, R) add {
    Rep == SquareMatrix(n, R);
    import from Rep;

    apply(q: %, v: Vector(n, R)): R == {
        r: R := 0;
        for i in 1..n::SI repeat {
            s: R := 0;
            for j in 1..n::SI repeat s := s + rep(q)(i,j)*v(j);
            r := r + v(i) * s;
        }
        r;
    }
```

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

    Join(Ring, Algebra K, VectorSpace K) with {
        e:     Integer -> %;
        term:  (K, List Integer) -> %;
        coef:  (%, List Integer) -> K;
    }

== add {
    Rep == Array(K);

    import from Rep;
    import from SI;
    import from Vector(n, K);

    local nsi: SI == n::SI;
    local nel: SI == 2^nsi;
    local Qee: Array K := [ Q e i for i in 1..n ];

    local new(): Rep == new(nel, 0);
    local apply(a: %, i: SI): K == rep(a).(i);

    dimension: Integer == 2^n;
```

```
0: % == per new();
1: % == { local z := new(); z.1 := 1; per z }

(x: %) + (y: %): % == per [ x.i + y.i for i in 1..nel ];
(x: %) - (y: %): % == per [ x.i - y.i for i in 1..nel ];
(k: K) * (x: %): % == per [ k * x.i for i in 1..nel ];
(x: %) * (k: K): % == per [ x.i * k for i in 1..nel ];
- (x: %): %        == per [ -x.i for i in 1..nel ];
(x: %) / (k: K): % == per [ x.i / k for i in 1..nel ];

coerce(m: SI): %      == { local z := new(); z.1 := m::K; per z }
coerce(m: Integer): % == { local z := new(); z.1 := m::K; per z }
coerce(k: K): %       == { local z := new(); z.1 := k; per z }

(x: %) = (y: %): Boolean == {
    for i in 0..nel repeat if x.i ~= y.i then return false;
    true
}
```

```
(x: %) ^ (n: Integer): % == {
    n < 0 => error "Cannot take inverse";
    n = 0 => 1;
    n = 1 => x;
    n = 2 => x*x;
    odd? n => x*x^(n-1);
    x^(n quo 2)^2;
}

-- Basis elements are e1..en.
e(i: Integer): % ==  {
    i >= n => error "No such basis element";
    iz: SI := 2^i;
    local z := new(); z.iz :=1; per z
}
```

```
-- Returns coef and index after swapping el into correct order.
-- Indices k in the array are 1..2^n.
--  ej    =>  k-1 = 2^(j-1).   Or together.

local canonIndex(el: List Integer): (K, SI) == {
    -- 1. Check input
    for e in el repeat
        e >= n => error "No such basis element";

    -- 2. Apply identity ei ej = -ej ei, i ~= j.
    -- Bubble sort keeps the swaps obvious.  OK because n is small.
    ea: Array SI := [ e::SI for e in el ];
    ne := #ea;

    exchanges: SI := 0;
    changed  := true;
    while changed repeat {
        changed := false;
        for i in 1..ne-1 repeat
            if ea(i) > ea(i+1) then {
                t := ea(i); ea(i) := ea(i+1); ea(i+1) := t;
                exchanges := exchanges + 1;
                changed   := true;
            }
    }
    coef: K := if odd? exchanges then -1 else 1;
```

```
    -- 3. Apply identity ei ei = Q(ei), and compute index.
    kminus1: SI := 0;
    for e in ea repeat {
        jminus1 := e::SI - 1;
        if bit(kminus1, jminus1) then {
            coef := coef * Qee(jminus1+1);
            kminus1 := kminus1 - 2^jminus1;
        }
        else
            kminus1 := kminus1 + 2^jminus1;
    }
    (coef, kminus1 + 1)
}
```

```
local canonIndexProd(ix: SI, iy: SI): (K, SI) == {
    c: K := 1;
    ixminus1: SI := ix - 1;
    izminus1: SI := iy - 1;
    for i in 0..nsi-1 | bit(izminus1, i) repeat {
        -- Apply rule  ei*ej = -ej*ei for i^=j
        k: SI := 0;
        for j in i+1..nsi-1 | bit(ixminus1, j) repeat k := k + 1;
        for j in 0..i-1     | bit(izminus1, j) repeat k := k + 1;
        if odd? k then c := -c;

        if bit(izminus1, i) then {
            c  := c * Qee(i+1);
            izminus1 := izminus1 - 2^i
        }
        else
            izminus1 := izminus1 + 2^i;
    }
    (c, izminus1 + 1)
}
```

```
(x: %)*(y: %): % == {
    local z := new();
    for ix in 1..nel | x.ix ~= 0 repeat
        for iy in 1..nel | y.ix ~= 0 repeat {
            (c, iz) := canonIndexProd(ix, iy);
            if c ~= 0 then
                z.iz := z.iz + x.ix * y.iy / c
        }
    per z
}

term(c: K, el: List Integer): % ==  {
    (ic, ix) := canonIndex(el);
    local z := new(); z.ix := ic * c; per z
}

coef(z: %, el: List Integer): K == {
    (ic, ix) := canonIndex(el);
    ic = 0 => error "Asking for coefficient of []";
    z.ix/ic
}
```

```
local writeTerm(o: TextWriter, coef: K, ix: SI): () == {
    ix = 1 =>  o << coef;
    o << coef;
    o << " ";
    for i in 1..n::SI repeat {
        if bit(ix, i-1) then o << "e__" << i
    }
}
(o: TextWriter) << (x: %): TextWriter == {
    nout: SI := 0;
    for i in 1..nel repeat {
        c := x.i;
        if c ~= 0 then {
            if nout > 0 then o << " + ";
            nout := nout + 1;
            writeTerm(o, c, i);
        }
    }
    if nout = 0 then o << "0";
    o
}
```

# Specializing CliffordAlgebra (Axiom)

```
-- 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

z := a + b * i

   (5)  a + b e
                1
```

```
y := c + d * i

   (6)  c + d e
                1

t * 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

   := 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)

 := e(1)$Ext

   (20)  e
          1

  := e(2)$Ext

   (21)  e
          2
```

```
x := 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

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

C := 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
```