Aldor III

Stephen M. Watt

University of Western Ontario

Edinburgh, September 28 2000

Outline

- Generic tie-ins
- Convenience features
- A detailed example
- Fortran-like code
- Using Aldor interactively
- Summary

Generic Tie-Ins

- Want user-defined types to enjoy all the benefits of system defined types.
  - efficiency
  - convenience
  - notation
- *ALL* of the "pre-defined" types seen so far have been defined by user-level libraries.
- Certain syntactical constructs call functions of specific names, allowing user types to behave as built-in.

Generic Tie-Ins: Literals

- Define meanings for literal constants by defining any of

```
integer: Literal -> X
string:  Literal -> X
float:   Literal -> X
```

- The type `Literal` represents source text for constants in a program.
- Packages exist to help to convert Literals to the required types

```
float(l: Literal): % == {
    import from NumberScanPackage(%);
    scanNumber string l
}
```

- If the exact type is visible at compile time then the literals can be inlined/evaluated to give constant data.
- Note also that `0` and `1` are *identifiers* and *not* integer literals.

Generic Tie-Ins: Program-defined tests

- There are certain expressiosn in which a condition controls evaluation:

```
if <condition> then ...
while <condition> repeat ...
<condition> or <condition>
not <condition>
```

```
<condition> => ...
for ... in ... | <condition> repeat ...
<condition> and <condition>
```

- In many situations a value can be treated as a condition even though it is not of type `Boolean`
- In these conditional contexts, Aldor applies the function

```
test: T -> Boolean
```

- E.g.

```
l: List Integer := ...;
while l repeat { f l; l := rest l }
```

Generic Tie-Ins: Generators

- Expressions traversed by iterators automatically have the appropriate generator function applied:

  ```
  generator: X -> Generator Y
  ```

- E.g. When `List(Integer)` is in imported, we have in scope

  ```
  generator: List Integer -> Generator Integer
  ```

  so the following is possible

  ```
  #include "axllib.as"
  import from List Integer;
  ls := [1, 2, 3, 4];
  for e in ls repeat { print << e << newline }
  ```

Generic Tie-Ins: Apply

- In an application, e.g. `f(a)`, if `f` does not evaluate to a function, then the form is treated as a call to `apply`.
- E.g. With the following in scope

```
m:     Mat
apply: (Mat, Integer, Integer) -> R
```

then the expression

```
m(i,j)
```

is evaluated as

```
apply(m, i, j)
```

Generic Tie-Ins: Set!, Bracket

- If the left-hand side of an assignment (`:=`) is an application, then the assignment is taken to be a call to `set!`.

```
f(a,b) := E          -- set!(f,a,b,E)
```

- The form

```
[ expr ]
```

  is a call to the function `bracket`

- E.g.

```
[1,2,3]              -- bracket(1,2,3)
```

Generic Tie-Ins: Coerce

- Type conversions are typically implemented by functions named `coerce`.
- These are not applied automatically.
- An expression of the form

  ```
  Expr :: T
  ```

  is equivalent to the operation `coerce` applied to *Expr* and restricted to be of type *T*.

  ```
  coerce(Expr) @ T
  ```

- E.g.

  ```
  n::DoubleFloat
  ```

Convenience Features: Keyword arguments and default argument values

- Argument names and default values may be given as part of the function type:

```
DF ==> DoubleFloat

line: (DF, slope: DF == 1, intercept: DF == 0) -> DF

line(3)

line(7, intercept == 4.8)

line(7, slope == -8.3)
```

- Note that `name: Type == val` is a type-valued expression in this context, and can be used elsewhere as such, e.g.

```
Record(A,B)
```

exports `bracket: (A,B) ->%`, so

```
Record(a: A == a0, b: B == b0)
```

exports `bracket: (a: A == a0, b: B == b0) -> %` and may be called as

```
[exa, exb]

[a == exa]

[b == exb, a == exa]

...
```

Convenience Features: Courtesy Conversions

- While in general coercions must be indicated explicitly in a program, a very conservative set of inexpensive isomorphisms are performed as needed.
- These "courtesy conversions" are applied to change between items represented as muliple values, cross products and tuples:

```
Cross(T,...,T) -> Tuple T
Cross(T) -> T
Cross(T1,...,Tn) -> (T1,...,Tn)
(T,...,T) -> Tuple T
(T1,...,Tn) -> Cross(T1,...,Tn)
T -> Tuple T
T -> Cross T
```

A Detailed Example

```
-------------------------------------------------------------------
---
--- table.as: Generic hash tables.
---
-------------------------------------------------------------------
#include "basiclib"

Hash ==> SingleInteger;

++ ‘HashTable(Key,Val)’ provides a parameterised hash-table data type.

HashTable(Key: SetCategory, Value: BasicType): BasicType with {

    export from Key;
    export from Value;

    table: () -> %;
        ++ ‘table()’ creates a new table using the equality test ‘=’
        ++ and the hash function ‘hash’ from the ‘Key’ type.

    eqtable: () -> %;
        ++ ‘eqtable()’ creates a new table using instance equality.

    table: ((Key, Key) -> Boolean, Key->Hash) -> %;
```

```
    ++ ‘table(=, hash)’ creates a new hash table using the
    ++ equality test ‘=’ and the hash function ‘hash’.

copy: % -> %;
    ++ ‘copy t’ creates a copy of the table ‘t’.

#: % -> SingleInteger;
    ++ ‘#t’ returns the number of elements in ‘t’.

search: (%, Key, Value) -> (Boolean, Value);
    ++ ‘(b,v) := search(t,k,d)’ searches table ‘t’ for the value
    ++ associated with key ‘k’. If there is such a value, ‘vk’,
    ++ then ‘b’ is set to ‘true’ and ‘v’ is set to ‘vk’.
    ++ Otherwise ‘b’ is ‘false’ and ‘v’ is set to ‘d’.

apply: (%, Key) -> Value;
    ++ ‘t.k’ searches the table ‘t’ for the value associated with
    ++ the key ‘k’.  It is an error if there is no value for ‘k’.

set!: (%, Key, Value) -> Value;
    ++ ‘t.k := val’ associates ‘val’ with ‘k’ in ‘t’.

drop!: (%, Key) -> Value;
    ++ ‘drop!(t, k)’ removes the entry for ‘k’ in ‘t’.

dispose!: % -> ();
```

```
        ++ ‘dispose! t’ indicates a table will no longer be used.

    generator: % -> Generator Cross(Key, Value);
        ++ ‘generator t’ is a generator which produces all the
        ++ ‘(key, value)’ pairs from ‘t’.
-
== add {
    -- Parameters to tune table performance.
    InitBuckC ==> primes.3;
    MaxLoad   ==> 5.0;
    MinLoad   ==> 0.5;

    -- primes.i is the largest prime <= 2^i.
    local primes: Array SingleInteger == [
        2,     3,      7,      13,
        31,    61,     127,    251,
        509,     1021,     2039,     4093,
        8191,    16381,    32749,    65521,
        131071,    262139,     524287,     1048573,
        2097143,   4194301,    8388593,    16777213,
        33554393,  67108859,   134217689,  268435399,
        536870909, 1073741789, 2147483647, 4294967291
    ];
    local lg(n: SingleInteger): SingleInteger == {
        p := 1;
        for i in 0.. repeat { if n <= p then return i; p := p + p; }
```

```
    never
}

-- Representation
Entry ==> Record(key: Key, value: Value, hash: Hash);

Rep   ==> Record(isEq?: Boolean,
                 equal:  (Key, Key) -> Boolean,
                 hash:   (Key) -> Hash,
                 count:  SingleInteger,
                 buckv:  Array List Entry);

-- Local representation operations
import from Rep;

local new(isEq?: Boolean, e: (Key,Key)->Boolean, h: Key->Hash):% ==
    per [isEq?, e, h, 0, new(InitBuckC, nil())];

local isEq? (t: %): Boolean              == rep(t).isEq?;
local hash  (t: %): (Key) -> Hash        == rep(t).hash;
local equal (t: %): (Key,Key) -> Boolean == rep(t).equal;
local buckv (t: %): Array List Entry     == rep(t).buckv;
local buckc (t: %): SingleInteger        == #rep(t).buckv;

local inc!(t: %): () == {
    import from SingleFloat;
```

```
    rep(t).count := rep(t).count + 1;
    if #t::SingleFloat/buckc(t)::SingleFloat > MaxLoad then
        enlarge! t;
}
local dec!(t: %): () == {
    import from SingleFloat;
    rep(t).count := rep(t).count - 1;
    if #t::SingleFloat/buckc(t)::SingleFloat < MinLoad then
        shrink! t;
}

local peq(k1: Key, k2: Key): Boolean == {
    import from Pointer;
    k1 pretend Pointer = k2 pretend Pointer
}
local phash(k1: Key): Hash == {
    k1 pretend Pointer pretend Hash
}

-- Find the chain for k, moving the link to the front on success.
local findChain(t: %, k: Key): SingleInteger == {
    h := hash(t)(k);
    n := h mod buckc(t) + 1;
    b := buckv(t).n;
    p := nil();     -- Previous link or nil.
```

```
    while b repeat {
        e := first b;
        if h = e.hash then {
            if isEq? t or equal(t)(e.key, k) then {
                -- Move to front
                if p then {
                    p.rest     := b.rest;
                    b.rest     := buckv(t).n;
                    buckv(t).n := b;
                }
                return n;
            }
        }
        p := b;
        b := rest b;
    }
    return 0;
}

-- Resize the table, larger or smaller.
local enlarge!(t: %): % == resize!(t, lg buckc(t) + 1);
local shrink! (t: %): % == resize!(t, lg buckc(t) - 1);

local resize!(t: %, sizeix: SingleInteger): % == {
    sizeix < 1 or sizeix > #primes => t;
```

```
    nbuckc := primes sizeix;
    nbuckv := new(nbuckc, nil());

    for b0 in buckv t repeat {
        b := b0;
        while b repeat {
            hd := b;
            b  := b.rest;

            n  := (hd.first.hash mod nbuckc) + 1;
            hd.rest  := nbuckv.n;
            nbuckv.n := b0;
        }
    }
    dispose! rep(t).buckv;
    rep(t).buckv := nbuckv;
    t;
}

-- Exported operations
sample: % == table();
(t1: %) = (t2: %): Boolean == {
    import from Pointer;
    t1 pretend Pointer = t2 pretend Pointer
}
(out: TextWriter) << (t: %): TextWriter == {
```

```
    out << "table(";
    any? := false;
    for b in buckv(t) repeat
        for e in b repeat {
            if any? then out << ", " else any? := true;
            out << e.key << " = " << e.value;
        }
    out << ")"
}

#(t: %): SingleInteger == rep(t).count;

eqtable(): % == new(true, peq, phash);
table(): %   == new(false, =$Key, hash$Key);
table(eq:(Key,Key)->Boolean, hash:Key->Hash): % ==
    new(false,eq,hash);

copy(t: %): % ==
    per [isEq? t, equal t, hash  t, #t,
       [[[e.key, e.value, e.hash] for e in b] for b in buckv t]];

search(t: %, k: Key, def: Value): (Boolean, Value) == {
    n := findChain(t, k);
    if n = 0 then
        (false, def)
    else
```

```
        (true,  buckv(t).n.first.value)
}
apply(t: %, k: Key) : Value == {
    n := findChain(t, k);
    n = 0 => error "Element missing from table.";
    buckv(t).n.first.value;
}
set!(t: %, k: Key, v: Value) : Value == {
    n := findChain(t, k);
    n > 0 => buckv(t).n.first.value := v;
    h := hash(t)(k);
    n := (h mod buckc(t)) + 1;
    buckv(t).n := cons([k,v,h], buckv(t).n);
    inc! t;
    v;
}
drop!(t: %, k: Key): Value == {
    n := findChain(t, k);
    n = 0 => error "Element missing from table.";
    e := buckv(t).n.first;
    v := e.value;
    buckv(t).n := disposeHead! buckv(t).n; -- Dispose of the link.
    dispose! e;                -- Dispose of the record.
    dec! t;
    v;
}
```

```
dispose!(t: %): () == {
    for b in buckv(t) repeat dispose! b;
    dispose! buckv(t);
    dispose! rep(t);
}

generator(t: %): Generator Cross(Key, Value) == generate {
    for b in buckv t repeat
        for e in b repeat {
            c: Cross(Key, Value) := (e.key, e.value);
            yield c
        }
}
```

Fortran-like code

```
#include "basiclib"

R ==> DoubleFloat;
I ==> SingleInteger;

++ quanc8: Quadrature, Newton-Cotes 8-panel
++
++ (This is a literal translation of the Fortran program given
++  in ``Computer Methods for Mathematical Computations'' by Forsythe,
++  Malcolm and Moler, Prentice-Hall 1977.)
++
++ Estimate the integral of fun(x) from a to b to a given tolerance.
++ An automatic adaptive routine based on the 8-panel Newton-Cotes
++ rule.
++
++ Input:
++   fun     The name of the integrand function subprogram f(x).
++   a       The lower limit of integration.
++   b       The upper limit of integration. (b may be less than a.)
++   relerr  A relative error tolerance. (Should be non-negative)
++   abserr  An absolute error tolerance. (Should be non-negative)
++
++ Output:
++   result  An approximation to the integral hopefully satisfying
```

```
++          the least stringent of the two error tolerances.
++   errest  An estimate of the magnitute of the actual error.
++   nofun   The number of function values used in the calculation of
++          the result.
++   flag    A reliability indicator.  If flag is zero, then result
++          probably satisfies the error tolerance.  If flag is
++          xxx.yyy then xxx = the number of intervals which have
++          not converged and 0.yyy = the fraction of the interval
++          left to do when the limit on nofun was approached.

quanc8(fun: R -> R, a: R, b: R, abserr: R, relerr: R):
     (Xresult: R, Xerrest: R, Xnofun: I, Xflag: R)
== {
       local result, errest, flag: R;
       local nofun: I;
       RETURN ==> return (result, errest, nofun, flag);

       local w0, w1, w2, w3, w4, area, x0, f0, stone, step, cor11: R;
       local qprev, qnow, qdiff, qleft, esterr, tolerr, temp: R;
       default i, j : I;

       qright: Array R       := new(31, 0.0);
       f:      Array R       := new(16, 0.0);
       x:      Array R       := new(16, 0.0);
       fsave:  Array Array R := [new(30, 0.0) for i in 1..8];
       xsave:  Array Array R := [new(30, 0.0) for i in 1..8];
```

```
local levmin, levmax, levout, nomax, nofin, lev, nim: I;

--
-- *** Stage 1 ***    General initializations
-- Set constants
--
levmin := 1;
levmax := 30;
levout := 6;
nomax  := 5000;
nofin  := nomax - 8 * (levmax - levout + 2 ^ (levout + 1));
--
-- Trouble when nofun reaches nofin
--
w0 :=   3956.0 / 14175.0;
w1 :=  23552.0 / 14175.0;
w2 :=  -3712.0 / 14175.0;
w3 :=  41984.0 / 14175.0;
w4 := -18160.0 / 14175.0;
--
-- Initialize running sums to zero.
--
flag := 0.0;
result := 0.0;
cor11 := 0.0;
```

```
errest := 0.0;
area := 0.0;
nofun := 0;
if a = b then RETURN;
--
-- *** Stage 2 ***   Iniitalization for first interval
--
lev := 0;
nim := 1;
x0  := a;
x(16) := b;
qprev := 0.0;
f0 := fun(x0);
stone := (b - a)/16.0;
x(8)  := (x0   + x(16)) / 2.0;
x(4)  := (x0   + x(8) ) / 2.0;
x(12) := (x(8) + x(16)) / 2.0;
x(2)  := (x0   + x(4) ) / 2.0;
x(6)  := (x(4) + x(8) ) / 2.0;
x(10) := (x(8) + x(12)) / 2.0;
x(14) := (x(12)+ x(16)) / 2.0;
for j in 2..16 by 2 repeat
        f(j) := fun(x(j));
nofun := 9;
--
-- *** Stage 3 ***   Central calculation
```

```
      -- Requires qprev,x0,x1,...,x16,f0,f2,f4,...,f16.
      -- Calculates x1,x3,...x15, f1,f3,...f15,qleft,qright,
--                qnow,qdiff,area.
      --
QL30  x(1) := (x0 + x(2)) / 2.0;
      f(1) := fun(x(1));
      for j in 3..15 by 2 repeat {
              x(j) := (x(j-1) + x(j+1)) / 2.0;
              f(j) := fun(x(j));
      }
      nofun := nofun + 8;
      step := (x(16) - x0) / 16.0;
      qleft := (w0*(f0+f(8)) + w1*(f(1)+f(7)) + w2*(f(2)+f(6)) +
                w3*(f(3)+f(5)) + w4*f(4)) * step;
      qright(lev+1) := (w0*(f(8) + f(16))+w1*(f(9)+f(15))+w2*(f(10)+
                f(14)) + w3*(f(11)+f(13)) + w4*f(12)) * step;
      qnow := qleft + qright(lev+1);
      qdiff := qnow - qprev;
      area := area + qdiff;
      --
      -- *** Stage 4 ***   Interval convergence test
      --
      esterr := abs(qdiff) / 1023.0;
      tolerr := max(abserr, relerr*abs(area)) * (step/stone);
      if lev < levmin then goto L50;
      if lev >=levmax then goto L62;
```

```
      if nofun > nofin then goto L60;
      if esterr <= tolerr then goto L70;
      --
      -- *** Stage 5 ***   No convergence
      -- Locate next interval
      --
0L50  nim := 2*nim;
      lev := lev + 1;
      --
      -- Store right hand elements for future use.
      --
      for i in 1..8 repeat {
              fsave(i)(lev) := f(i+8);
              xsave(i)(lev) := x(i+8);
      }
      --
      -- Assemble left hand elements for immediate use.
      --
      qprev := qleft;
      for i in 1..8 repeat {
              j := -i;
              f(2*j+18) := f(j+9);
              x(2*j+18) := x(j+9);
      }
      goto L30;
      --
```

```
      -- *** Stage 6 ***   Trouble section
      -- Number of function values is about to exceed limit.
      --
@L60  nofin := 2*nofin;
      levmax := levout;
      flag := flag + (b - x0)/(b - a);
      goto L70;
      --
      -- Current level is levmax.
      --
@L62  flag := flag + 1;
      --
      -- *** Stage 7 ***   Interval converged
      -- Add contributions into running sums.
      --
@L70  result := result + qnow;
      errest := errest + esterr;
      cor11 := cor11 + qdiff / 1023.0;
      --
      -- Locate next interval.
      --
@L72  if nim = 2*(nim quo 2) then goto L75;
      nim := nim quo 2;
      lev := lev - 1;
      goto L72;
```

```
L75     nim := nim + 1;
        if lev <= 0 then goto L80;
        --
        -- Assemble elements required for the next interval.
        --
        qprev := qright(lev);
        x0 := x(16);
        f0 := f(16);
        for i in 1..8 repeat {
                f(2*i) := fsave(i)(lev);
                x(2*i) := xsave(i)(lev);
        }
        goto L30;
        --
        -- *** Stage 8 ***   Finalize and return
        --
L80     result := result + cor11;
        --
        -- Make sure errest not less than roundoff level.
        --
        if errest = 0.0 then RETURN;
L82     temp := abs(result) + errest;
        if temp ~= abs(result) then RETURN;
        errest := 2.0 * errest;
        goto L82;
```

```
-- Test with a well-known example: the result should be Pi.

import from R, I;

f(x:R):R == 4.0/(x*x+1);

(result, errest, nofun, flag) := quanc8(f,0.0,1.0,0.00001,0.00001);

if zero? flag then {
  print << "result = " << result << newline;
  print << "error = " << errest << newline;
  print << "(after " << nofun << " function evaluations)" << newline;
}
else
  print << "error flag is " << flag << newline;
```

Using Aldor Interactively

- Use `-g loop` to have Aldor execute one command at a time.
- Provides interactive environment for experimentation, much like Matlab.

```
[3.1] watt@bungee:~ >> axiomxl -g loop
%1 >> #include "axllib.as"
                                          Comp: 390 msec, Interp: 30 msec
%2 >> I ==> Integer
                                          Comp: 0 msec, Interp: 0 msec
%3 >> import from I
                                          Comp: 100 msec, Interp: 0 msec
%4 >> 123*456
56088 @ Integer
                                          Comp: 20 msec, Interp: 140 msec
%5 >> fact(n: I): I == if n = 0 then 1 else n*fact(n-1)
Defined fact @ (n: Integer) -> Integer
                                          Comp: 10 msec, Interp: 10 msec
%6 >> l: List I := [fact n for n in 0..40]
list(1, 1, 2, 6, 24, 120, 720, 5040, 40320, 362880, 3628800, 39916800,
     479001600, 6227020800, 87178291200, 1307674368000, 20922789888000,
     355687428096000, 6402373705728000, 121645100408832000, 2432902008176640000,
     51090942171709440000, 1124000727777607680000, 25852016738884976640000,
```

```
    620448401733239439360000, 155112100433309859840000000,
    403291461126605635584000000, 10888869450418352160768000000,
    304888344611713860501504000000, 884176199373970195454361600000,
    265252859812191058636308480000000, 82228386541779228177255628800000000,
    263130836933693530167218012160000000,
    868331761881188649551819440128000000,
    295232799039604140847618609643520000000,
    10333147966386144929666651337523200000000,
    37199332678990121746799944815083520000000,
    13763753091226345046315979581580902400000000,
    523022617466601111760007224100074291200000000,
    203978820811974433586402817399028973568000000000,
    8159152832478977343456112695961158942720000000000) @ List(Integer)
                                                Comp: 120 msec, Interp: 150 msec

%7 >> red(f:(I,I)->I, f0: I)(l: List I): I == {
          if empty? l then f0 else f(first l, red(f,f0)(rest l))
      }
Defined red @ (f: (Integer, Integer) -> Integer, f0: Integer)
              -> (l: List(Integer)) -> Integer
                                                Comp: 30 msec, Interp: 0 msec

%8 >> summ := red(+, 0)
  () @ (l: List(Integer)) -> Integer
                                                Comp: 10 msec, Interp: 0 msec
```

```
%9 >> sum l
836850334330315506193242641144055892504420940314 @ Integer

                                                  Comp: 10 msec, Interp: 20 msec
%10 >> #quit
[3.2] watt@bungee:~ >>
```

Summary

- Language allows functions and types as first class values
- Pervasive use of dependent types
- Simple, elegant formulation of OO and functional styles
- Efficient were needed
- Easy interlanguage communication with other important languages, e.g. `C++`, `Fortran`
- Interactive or compiled environment
- Expressive, natural language for expressing relationships among objects

For Further Information

- User Guide — see `http://www.csd.uwo.ca/~watt/aldor/UserGuide`
  Hardcopies from NAG (ISBN 1-85206-106-5).
- Under construction
  `aldor.org`
  `aldor.org.uk`