CS342: Organization of Prog. Languages

Topic 21: The Aldor Programming Language

- What is it?
- What are people doing with it?
What is Aldor?

- Initially conceived as extension language for Axiom.
- Required very expressive type system to model rich relations among mathematical types.
- Higher order: types and functions first class values.
- Full support for dependent types, use of type categories.
- Optimizing compiler.
- Has users. Some libraries 200-400 Kloc.
Context

- Scratchpad II (IBM) 1984-1990.
- A# as extension language to Axiom 1990-1994
  Generate code to run in Lisp environment 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
- www.aldor.org 2002
- Workshop on Categorical Programming Languages
  (2001 London ON, 2002 Lille, 2004 Santander)
- Recent papers, e.g. Domain Specific Aspect Languages 2006,
  Workshop on Generic Programming 2006.
Aldor Language Characterization

- Imperative language, statically typed, strict.

- Blend of functional, OO and AO styles.

  Both types and 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* – provide static information about dynamic objects. Basis for OO features.

  *Ex post facto* extensions of types.

  Based on ADT.

- Does *not* support continuation passing, to interoperate with C, Fortran, Java, etc.
First Examples I

#include "axlllib"

double(n: Integer): Integer == n + n
First Examples II

#include "axlllib"

-- 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 "axlllib"

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
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); 
   }
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:
  
  \((a: S): R \rightarrow+ e\)
  
  \((a1: A1, a2: A2, \ldots): (R1, R2, \ldots) \rightarrow+ e\)
  
  \((\_): () \rightarrow+ e\)

- 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. \(f \ x, M \ v, t.left, \ldots\)
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:

\[ v : T := e \quad v := e \quad (a, b, c) := e \]

• Constant definition:

\[ c : T == e \quad c == e \]

• 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**

\[
a(i_1, i_2, \ldots) := e \quad \iff \quad \text{set!}(a, i_1, i_2, \ldots, e)
\]

\[
a.i.j := e \quad \iff \quad \text{set!}(a.i, j, e)
\]

**Definition**

\[
f(a: A): R == e \quad \iff \quad f: (a: A) \rightarrow R == (a: A): R \rightarrow e
\]

\[
f(a:A)(b:B)(c:C): R == e \quad \iff \quad f: (a:A)\rightarrow(b:B)\rightarrow(c:C)\rightarrow R ==
\]

\[
(a:A): (b:B)\rightarrow(c:C)\rightarrow R \rightarrow (b:B): (c:C)\rightarrow R \rightarrow (c:C): R \rightarrow e
\]
• 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`.

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

Compound expression: \{e_1; e_2; \ldots; e_N\}

Conditional: if C then T [else E]
e_1 and e_2
e_1 or e_2

\{a; b; c \Rightarrow d; e; f \Rightarrow g; h\}

Loop: \textit{Iterator}^* \text{ repeat } E

Collection: E \textit{Iterator}^*

\textit{Iterator}: for \textit{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

\[
\begin{align*}
\text{rna:} & \quad \text{Record}(n: \text{Integer}, a: \text{Array}(n, \text{Window})) \\
\text{eigenvals:} & \quad \text{Matrix}(n,n,K) \rightarrow \\
& \quad (p: \text{UnivPoly}(K), \text{Array}(n, \text{AlgExtension}(p, K))
\end{align*}
\]

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 \textit{type} depends on the \textit{value} of their arguments. E.g.

\[
72 \mod 5 \quad \Rightarrow \quad 2 \mod 5
\]

This relation can be expressed with “dependent mapping” types.

\[
\text{mod: } (\text{Integer}, m: \text{Integer}) \rightarrow \text{IntegerMod}(m)
\]

- 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.

\[ \text{identity: (n: Integer, R: Ring) -> Matrix(n, n, R)} \]

\[ \text{identity(2, Float) ==> } \begin{bmatrix} 1.0 & 0.0 \\ 0.0 & 1.0 \end{bmatrix} \]

- Parametric polymorphism:

\[ \text{commutator(R: Ring)(p: R, q: R): R == p*q - q*p;} \]

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.

```plaintext
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.

```plaintext
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 categories, 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:

```lisp
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, \( X \times X \to X \).

- Suppose we define an abstract base class, and derive

<table>
<thead>
<tr>
<th>Class</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>3 \times 3</td>
<td>9</td>
</tr>
<tr>
<td>IntegerMod(5)</td>
<td>6 \times 7</td>
<td>2</td>
</tr>
<tr>
<td>DoubleFloat</td>
<td>3.2 \times 6.0</td>
<td>19.2</td>
</tr>
<tr>
<td>Polynomial(x,Integer)</td>
<td>( (x^2 + 1)*(x-1) )</td>
<td>( x^3-x^2+x-1 )</td>
</tr>
<tr>
<td>SqMatrix(2,DoubleFloat)</td>
<td>[1.0 1.0] \times [1.0 2.0]</td>
<td>[3.1 3.0]</td>
</tr>
<tr>
<td></td>
<td>[0.0 1.0] \times [2.1 1.0]</td>
<td>[2.1 1.0]</td>
</tr>
</tbody>
</table>
### Categories vs Abstract Base Classes

- Want a base type $x$, which provides an internal multiplication, $x \times x \rightarrow x$.
- Suppose we define an abstract base class, and derive

<table>
<thead>
<tr>
<th>Category</th>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer</td>
<td>$3 \times 3$</td>
<td>9</td>
</tr>
<tr>
<td>IntegerMod(5)</td>
<td>$6 \times 7$</td>
<td>2</td>
</tr>
<tr>
<td>DoubleFloat</td>
<td>$3.2 \times 6.0$</td>
<td>19.2</td>
</tr>
<tr>
<td>Polynomial(x, Integer)</td>
<td>$(x^2 + 1)(x-1)$</td>
<td>$x^3-x^2+x-1$</td>
</tr>
<tr>
<td>SqMatrix(2, DoubleFloat)</td>
<td>$\begin{bmatrix} 1.0 &amp; 1.0 \ 0.0 &amp; 1.0 \end{bmatrix} \times \begin{bmatrix} 1.0 &amp; 2.0 \ 2.1 &amp; 1.0 \end{bmatrix}$</td>
<td>$\begin{bmatrix} 3.1 &amp; 3.0 \ 2.1 &amp; 1.0 \end{bmatrix}$</td>
</tr>
</tbody>
</table>

$\text{IntegerMod}(5) \times \text{SqMatrix}(2, \text{DoubleFloat})$  ?????????????
Categories vs Abstract Base Classes

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

\[
2 \in \text{IntegerMod}(5) \subseteq X
\]

\[
6.2 \in \text{DoubleFloat} \subseteq X
\]

whereas with derived classes we have

\[
2 \in \text{IntegerMod}(5) \subseteq X
\]

\[
6.2 \in \text{DoubleFloat} \subseteq X
\]
Example: Prime Number Sieve

```plaintext
#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:

\[(L1 \ast L2).(f) = L1 \, L2 \, f\]

NNI \(\Rightarrow\) NonNegativeInteger;
SUP \(\Rightarrow\) SparseUnivariatePolynomial;

\[
\text{LinearOrdinaryDifferentialOperator(}
    \begin{array}{ll}
    \text{A: DifferentialRing,} \\
    \text{M: LeftModule(A) with differentiate: } \% \to \%
    \end{array}
\) : MonogenicLinearOperator(A) with {
    \begin{array}{ll}
    \text{D: } \%;} \\
    \text{apply: } (\%, \text{ M}) \to \text{ M;} \\
    \end{array}
\)

if A has Field then {
    \begin{array}{ll}
    \text{leftDivide: } (\%, \%) \to \text{ Record(quotient: } \%, \text{ remainder: } \%);} \\
    \quad \quad \text{++ [q,r] = leftDivide(a,b) means a=b*q+r, deg r < deg b} \\
    \text{rightDivide: } (\%, \%) \to \text{ Record(quotient: } \%, \text{ remainder: } \%); \\
    \quad \quad \text{...} \\
    \end{array}
}
== 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 Args

#include "axlllib"

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
Dependent types are fully supported

- Gives dynamic typing. E.g. with

  \[ f: (n: \text{Integer}, m: \text{SquareMatrix}(n, \text{Integer})) \rightarrow \text{IntegerMod}(n) \]

  If \( n = 3 \), then \( m \) has type \( \text{SquareMatrix}(3, \text{Integer}) \) and \( f(n, m) \) has type \( \text{IntegerMod}(3) \).

- Recovers OO through dependent products. E.g.

  \[
  \text{prodl}: \text{List} \text{ Record}(S: \text{Semigroup}, s: S) = [ \\
  \quad [\text{DoubleFloat}, x], \\
  \quad [\text{Permutation}, p], \\
  \quad [\text{DoubleFloat}, y] \\
  \]

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

- Type producing expressions may be conditional

```plaintext
UnivariatePolynomial(R: Ring): Module(R) with {
    coeff: (% , Integer) -> R;
    monomial: (R, Integer) -> %;

    if R has Field then EuclideanDomain;
    ...
}
```
Post facto extensions

- View existing domains in additional categories.
- Provides “aspect oriented” programming, or “separation of concerns”

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

extend Integer: DifferentialRing == add {
    differentiate(n: Integer): Integer == 0;
    constant?(n: Integer): Boolean == true;
}
```

- Allows well-structured libraries on the same types to be developed independently.
Extending Constructions

- Categorical properties can be quite complex.

```plaintext
DirectProduct(n: Integer, S: Set): Set with {
    component: (Integer, %) -> S;
    new: Tuple S -> %;
    if S has Semigroup then Semigroup;
    if S has Monoid then Monoid;
    if S has Group then Group;
    ...
    if S has Ring then Join(Ring, Module(S));
    if S has Field then Join(Ring, VectorField(S));
    ...
    if S has DifferentialRing then DifferentialRing;
    if S has Ordered then Ordered;
    ...
} == add { ... }
```

- Certain constructors are open-ended in their conditionalization requirements.
Post Facto Extensions

• A better direct product:

\[
\text{DirectProduct}(n: \text{Integer}, S: \text{Set}): \text{Set} \text{ with } \{
    \text{component}: (\text{Integer}, \%) \to S;
    \text{new}: \text{Tuple S} \to \%;
\} \text{ == add } \{ \ldots \}
\]

extend \text{DirectProduct}(n: \text{Integer}, S: \text{Semigroup}): \text{Semigroup} == \ldots
extend \text{DirectProduct}(n: \text{Integer}, S: \text{Monoid}): \text{Monoid} == \ldots
extend \text{DirectProduct}(n: \text{Integer}, S: \text{Group}): \text{Group} == \ldots

extend \text{DirectProduct}(n: \text{Integer}, S: \text{Ring}): \text{Join(Ring, Module(S))} == \ldots
extend \text{DirectProduct}(n: \text{Integer}, S: \text{Field}): \text{Join(Ring, VectorField(S))} == \ldots

extend \text{DirectProduct}(n: \text{Integer}, S: \text{Field}): \text{Join(Ring, VectorField(S))} == \ldots
extend \text{DirectProduct}(n: \text{Integer}, S: \text{DifferentialRing}): \text{DifferentialRing} == \ldots
extend \text{DirectProduct}(n: \text{Integer}, S: \text{Ordered}): \text{Ordered} == \ldots

• Normally these extensions would all be in separate files.
Use in Library Design

• Staged Building of Libraries

1. *Basic raw types plus loose operations.*

   Boolean: Type == add { };
   Integer: Type == add { };
   String: Type == add { };

   +$Machine: (Integer, Integer) -> Integer, ...

2. *Add the relevant primitive operations.*

   extend Boolean: with {
      =: (%, %) -> Boolean; convert: % -> String; ...
   } == ...
   extend Integer: with {
      =, <: (%, %) -> Boolean; convert: % -> String; ...
   } == ...
   extend String: with {
      =: (%, %) -> Boolean; #: % -> Integer; ...
   } == ...
Use in Library Design

• Staged Building of Libraries (cont’d)

3. Define data structure domains


5. Define mathematical categories.

6. Extend basic domains.

7. Define mathematical domains.
Use in Library Design

- Adding callback algorithms to parameters
  Old style: Fixed conditionalization.

```plaintext
LinearAlgebra(R:CommutativeRing, M:MatrixCategory R):
with {...} == add {

  local Elim: LinearEliminationCategory(R, M) == {
    R has Field =>
      OrdinaryGaussElimination(R, M);
    R has IntegralDomain =>
      TwoStepFractionFreeGaussElimination(R,M);
    DivisionFreeGaussElimination(R, M);
  }

  determinant(m:M):R == determinant(m)$Elim;
}
```
Use in Library Design

- **Adding callback algorithms to parameters**
  
  New style: Open ended.

  ```lisp
  LinearAlgebraRing: Category == with {
    determinant: (M:MatrixCategory %) -> M -> %;
    rank: (M:MatrixCategory %) -> M -> Integer;
    ...
  }
  
  Modify LinearAlgebra package to use algorithms carried in on parameter. Replace the determinant function with

  ```lisp
  determinant(m:M):R == {
    if R has LinearAlgebraRing then
      determinant(M)(a)$R;
    else
      determinant(m)$Elim;
  }
  
  Now we can extend rings, e.g. Integer, IntegerMod(p), before passing them to the LinearAlgebra package.
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
Foam: Intermediate Code

- First order: functions and types now explicit
- Target level:
  Maps simply to register-based or stack-based
  Maps simply to Lisp, C, or assembly level
- Primitive types:
  
  Nil  Char  Bool  Byte  HInt  SInt  SFlo  DFlo  Word
  Int8  Int16  Int32  Int64  Flo32  Flo64
  Ptr  Env  Arr  Rec  Prog  Clos

- Low-level operations, e.g. DFloPlus.
Optimization

- The most important ones:
  - Procedural integration (inlining).
  - Data structure elimination (including lexical environments).
  - Constant propagation, common sub-expression elimination.

- Certain easy optimizations delegated to concrete code back end.
Optimization of Generators

generator(seg: Segment Int): Generator Int == generate {
  i := a;
  while a <= b repeat { yield a; a := a + 1 }
}
generator(l: List Int): Generator Int == generate {
  while not null? l repeat { yield first l; l := rest l }
}

client() == {
  ar := array(...);
  li := list(...);
  s := 0;
  for i in 1..#ar for e in l repeat { s := s + ar.i + e }
  stdout << s
}
Aldor vs C (Part I)

Non-floating-point
(Aldor = Red, C = Green)

<table>
<thead>
<tr>
<th>Optimisation Level</th>
<th>Q2/00</th>
<th>Q3/01</th>
<th>Q4/02</th>
<th>Q5/03</th>
</tr>
</thead>
<tbody>
<tr>
<td>Geometric Mean</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>11</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>12</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aldor vs C (Part II)
Example: Prime Number Sieve

```
# include "axlllib.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

```plaintext
#include "axlllib.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 operator domain with the opposite ring multiplication. That is, as sets P == %, but a * b in P is b * a in %.

OppositeLinearOperator(P: LinearOperator R, R: Ring): LinearOperator(R) with {
  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);
}

extend OppositeLinearOperator(P: DifferentialRing, R: Ring): DifferentialRing == add {
  deriv(x: %): % == op(deriv 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:

\[(L_1 \ast L_2).(f) = L_1 L_2 f\]

\[\text{NNI} ==> \text{NonNegativeInteger};\]
\[\text{SUP} ==> \text{SparseUnivariatePolynomial};\]

LinearOrdinaryDifferentialOperator(
    A: DifferentialRing,
    M: LeftModule(A) with differentiate: % -> %
): LinearOperator(A) with {
    D: %;
    apply: (%, M) -> M;
    ...
    if A has Field then {
        leftDivide: (%, %) -> Record(quotient: %, remainder: %);
        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]
        }
    }
    ...
}
Working in Hom: Morphisms as Objects

• View, e.g., Poly(x), SqMat(n), Complex, etc as elements of Hom(Ring).

• Wish to compute on these, construct compositions, conversions.

• E.g. have many isomorphisms,

Poly(x) Complex R === Complex Poly(x) R
Poly(x) Poly(y) R === Poly(y) Poly(x) R
SqMat(n) Complex R === Complex SqMat(n) R
SqMat(n) SqMat(m) R === SqMat(m) SqMat(n) R

Wish to generically re-organize towers of functors.
E.g. If F,G: (R: Ring)→Module R, generically compute F G R → G F R.

• Construct and optimize compositions, e.g.

Pxy == Poly(x) Poly(y);

p: Pxy Integer := ... 
f: Pxy IntegerMod(7) := ...

Optimization complicated by presence of post-facto extensions.
Recent and On-going Work

• Advanced libraries for polynomial and differential systems (triangular decomposition, generic solution of $\text{ODE/\partial E/OD}_qE$, ...)

• Maple/Aldor interface

• Parallel Aldor for QCD: Diff ops in cat of fiber bundles. Code gen via Todd-Coxeter exploits problem and computer symmetry.

• Categorical framework to link C++ and Java templates with Aldor functors (OOPSLA).

• Segue between concrete values and symbolic expression trees in a general way. Relate concrete types to trees with adjoints.

• Extended construction: Construction in an extended computation. Mutable during construction, then afterwards they are immutable.

• Distinguish coercions: embeddings, retractions, liftings.

• Support more kinds of arrows naturally and efficiently.

• Optimizaiton of generics.
Conclusions

- It is possible to write mathematical algorithms at a high level of abstraction and to compile them to efficient code.

- Quantifying over categories solves a number of practical problems in software specification, library construction and code optimization.

- Experience shows this approach leads programmers to try to write code as generally as reasonable, minimizing assumptions.

Aldor Availability

- www.aldor.org

- Freely available by download

- Standard base and advanced math libraries