Categories in the Design of Aldor

Stephen M. Watt
`Stephen.Watt@uwo.ca`

Ontario Research Centre for Computer Algebra
University of Western Ontario

Applied and Computational Category Theory (ETAPS ACCAT2007)

Univesidade do Minho, Braga, Portugal. March 25, 2007.

What is Computer Algebra?

- The study of algorithms and software so computers to do mathematics, producing equations and expressions rather than just numbers.
- E.g., polynomial factorization, trig simplification, integration.
- Successful commercial systems, e.g. Maple, Mathematica.
- Many special-purpose systems for research.
- We study this at ORCCA,
  the Ontario Research Centre for Computer Algebra,
  a joint laboratory involving Western, Waterloo and Maplesoft.

- Personal role: as an author of Maple, Axiom, Aldor, MathML.

What is Axiom?

- **Axiom** was a CAS designed in the 80s and early 90s at IBM research.
- Based on concepts of abstract algebra, e.g. the library is built on such things as `AbelianMonoid`, `Ring`, `Field`, `Module(R)`, etc.
- Initially disseminated by the Numerical Algorithms Group, Oxford. Now open source.

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.

Aldor Motivation

- Originally an extension language for the AXIOM system.
- Need to model rich relationships among mathematical structures.
- Emphasis on uniform handling of values independent of their type; less emphasis on a particular object model.
- Primary considerations:
  generality, composibility, efficiency, interoperability
- Express the *requirements* and the rich *relationships* among inputs.
  Express *guarantees* on the results.

- Then have a language encouranging one to
  weaken the requirements and *strengthen* the guarantees.

Context

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

Why Math in Prog Language Research?

- Rich relationships among non-trivial concepts.
- Well-defined domain.
- Many programming langauge problems have had early use here: algebraic expressions, arrays, big integers, garbage collection, pattern matching, parametric polymorphism, ...

Why Prog Language Research in Math?

- Large libraries, requiring efficient code.
- Complex interfaces.
- Simple programming language ideas insufficient.

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

A Problem in Computer Algebra Software

- Systems usually have several implementations of the same algorithm for different structures.
  E.g. Gaussian elimination over $\mathbf{Q}$, $\mathbf{Z}/p\mathbf{Z}$, $\mathbf{Z}(x)$,...
- Sometimes in alternative views
  E.g. repeated squaring (f^n) *vs* repeated doubling ($n * p$).
- Difficult to implement improvements where needed.
- Difficult to extend system to work with new objects.

- Want to be able to:
  - define algorithms for some specific category of objects
  - impelement them efficiently, and
  - compose constructions flexibly.

Aldor and Its Type System

- **Types and functions are first class values**
  - May be created dynamically.
  - Provide representations mathematical sets and functions.
- **The type system has two levels**
  - Each value belongs to a unique *domain* that can be declared statically.
  - Domains belong to the domain Type, and may additionally belong to a number of type *categories*, which are subtypes of Type.
  - Categories specify what exports (e.g. operations) a domain must provide.
  - Categories fill the role of interfaces or abstract base classes.

Types as Values

- When types can be used as values, 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;
```

Type Categories vs OO

- Suppose we have

```
Semigroup:   Category == with { *: (%, %) -> % }
DoubleFloat: Join(Semigroup, ...) == ...
Permutation: Join(Semigroup, ...) == ...
```

- In OOP we can multiply a `DoubleFloat` by a `Permutation`.

$$\left.\begin{array}{lllll} x, y & \in & DoubleFloat & \subset & Semigroup \\ p, q & \in & Permutation & \subset & Semigroup \end{array}\right\}\text{OOP}$$

  Liskov recognized this problem with binary operations already with CLU.

- In Aldor, the two levels allow `x*y` but prevent `x * p`.

$$\left.\begin{array}{lllll} x, y & \in & DoubleFloat & \in & Semigroup \\ p, q & \in & Permutation & \in & Semigroup \end{array}\right\}\text{Aldor}$$

Dependent types are fully supported

- Gives dynamic typing. E.g. with

  ```
  f: (n: Integer, m: SquareMatrix(n, Integer)) -> IntegerMod(n)
  ```

  If $n = 3$, then m has type SquareMatrix(3, Integer) and $f(n, m)$ has type IntegerMod(3).

- Recovers OO through dependent products. E.g.

  ```
  prodl: List Record(S: Semigroup, s: S) == [
      [DoubleFloat, x],
      [Permutation, p],
      [DoubleFloat, y]
  ]
  ```

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

Categories and Parametric Polymorphism

- Category- and domain-producing functions use the same language as first-order functions.

```
-- A function returning an integer.
factorial(n: Integer): Integer == {
    if n = 0 then 1 else n*factorial(n-1)
}

-- Functions returning a category and a domain.
define Module(R: Ring): Category == Ring with {
    *: (R, %) -> %
}
Complex(R: Ring): Module(R) with {
    complex: (%, %) -> R;
    real: % -> R;
    imag: % -> R;
    conjugate: % -> %; ...
} == add {
    Rep == Record(real: R, imag: R);
    real(z:%): R == rep(z).real;
    (w: %) + (z: %): % == ...
}
```

Conditional Types

- Type producing expressions may be conditional

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

    if R has Field then EuclideanDomain;
    ...
} == add {
    ...
}
```

Post facto extensions

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

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


extend Integer: DifferentialRing == add {
        differentiate(n: Integeger): 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.

```
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 Extension of Functors

- Extending names bound to domain-producing functions.

```
F(a1: T01,...,ak: T0k): R0 == A0
extend F(a1: T11,...,ak: T1k): R1 == A1
...
extend F(a1: Tn1,...,ak: Tnk): Rn == An
```

gives

```
F(a1:Meet(T01...Tn1),...,an:Meet(T0k...Tnk)): with {
    if a1 ∈ T01 and ... and ak ∈ T0k then R0;
    if a1 ∈ T11 and ... and ak ∈ T1k then R1;
    ...
    if a1 ∈ Tn1 and ... and ak ∈ Tnk then Rn;
} == add {
    if a1 ∈ T01 and ... and ak ∈ T0k then A0;
    if a1 ∈ T11 and ... and ak ∈ T1k then A1;
    ...
    if a1 ∈ Tn1 and ... and ak ∈ Tnk then An;
}
```

Post Facto Extensions

- A better direct product:

```
DirectProduct(n: Integer, S: Set): Set with {
    component: (Integer, %) -> S;
    new:       Tuple S -> %;
} == add { ... }

extend DirectProduct(n: Integer, S: Semigroup): Semigroup == ...
extend DirectProduct(n: Integer, S: Monoid): Monoid == ...
extend DirectProduct(n: Integer, S: Group): Group == ...
...
extend DirectProduct(n: Integer, S: Ring): Join(Ring, Module(S)) == ...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
extend DirectProduct(n: Integer, S: DifferentialRing): DifferentialRing == ...
extend DirectProduct(n: Integer, S: Ordered): Ordered == ...
...
```

- Normally these extensions would all be in separate files.

Use in Library Design

- **Staged Building of Libraries**

  1. *Basic raw types without operations*, as above.
  2. *Add the relevant primitive operations.*

```
extend Boolean: with {
    =: (%, %) -> Booelean;
    convert: % -> String; ...
} == ...
extend Integer: with {
    =: (%, %) -> Boolean;
    <: (%, %) -> Boolean;
    convert: % -> String; ...
} == ...
extend String: with {
    =: (%, %) -> Boolean;
    #: % -> Integer;      ...
} == ...
```

Use in Library Design

- **Staged Building of Libraries (cont'd)**

  3. *Define data structure domains*
  4. *Define data structure categories. Extend 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.

```
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.

  ```
  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

  ```
  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

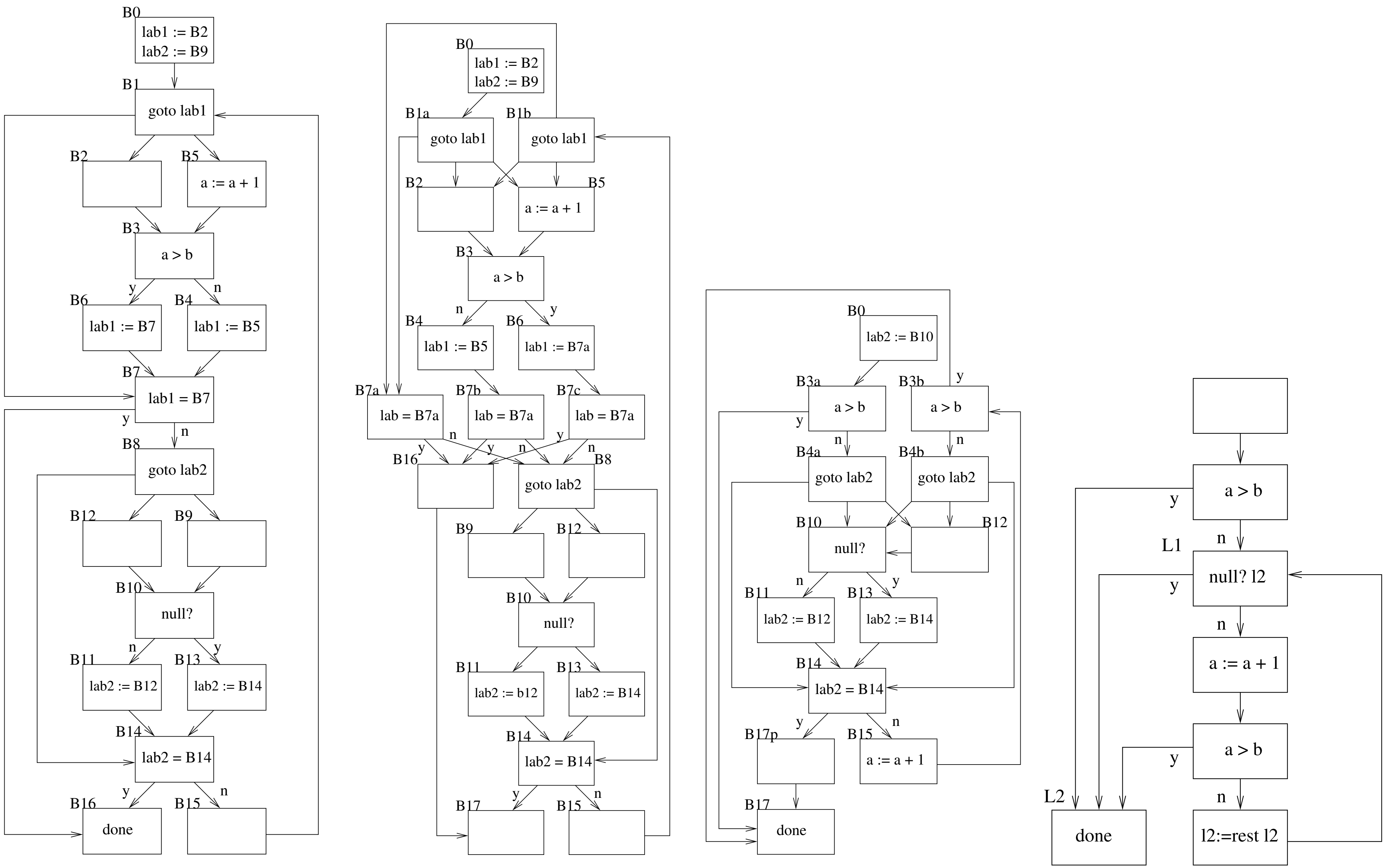
- The most important ones:
  - Procedural integration (inlining).
  - Data structure elimination (including lexical environments).
  - Constant propagation, common sub-expression elimination.

- Certain easy optimizations delegated to concrete code back end.

Optimization of Generators

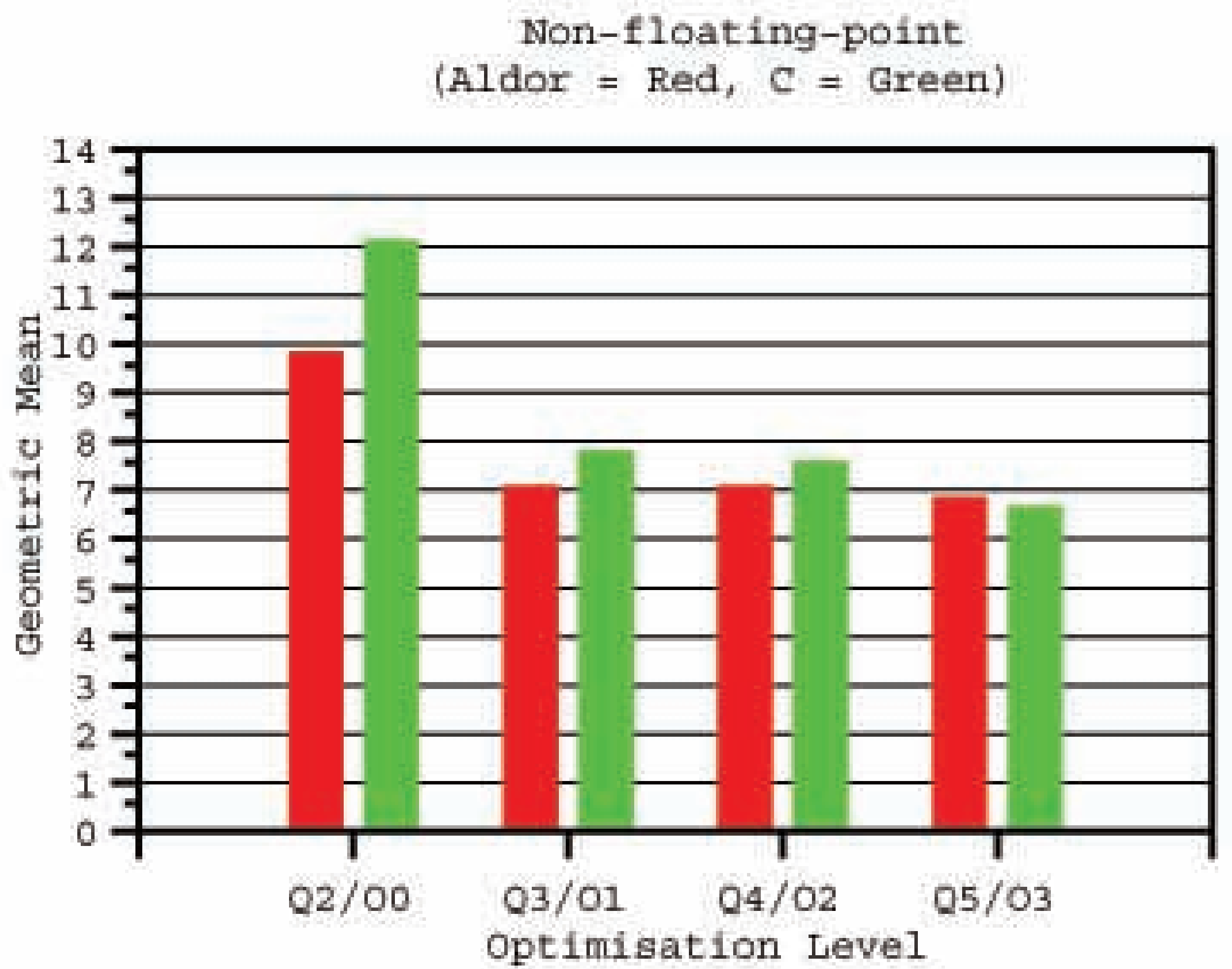
```
generator(seg:Segment Int):Generator Int == generate {
     i := a;
     while a <= b repeat { yield a; a := a + 1 }
}
generator(l: List Int): Generator Int == generate {
     while not null? l repeat { yield first l; l := rest l }
}

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

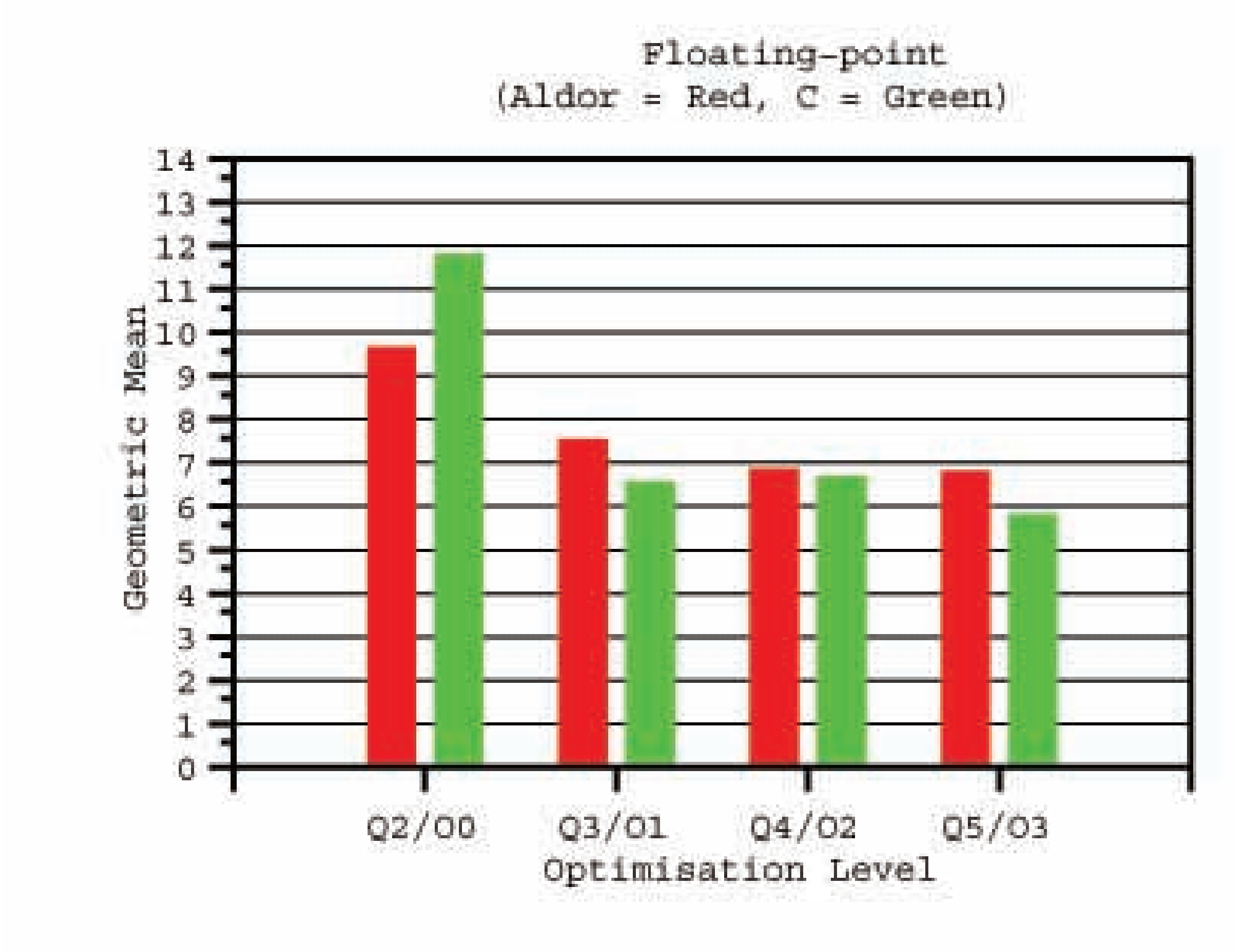


B0
lab1 := B2
lab2 := B9
B1
goto lab1
B2
B5
a := a + 1
B3
a > b
y
n
B6
lab1 := B7
B4
lab1 := B5
B7
lab1 = B7
y
n
B8
goto lab2
B12
B9
B10
null?
n
y
B11
lab2 := B12
B13
lab2 := B14
B14
lab2 = B14
y
n
B16
done
B15
B0
lab1 := B2
lab2 := B9
B1a
goto lab1
B1b
goto lab1
B2
B5
a := a + 1
B3
a > b
n
y
B4
lab1 := B5
B6
lab1 := B7a
B7a
lab = B7a
B7b
lab = B7a
B7c
lab = B7a
y
n
y
n
y
n
B16
B8
goto lab2
B9
B12
B10
null?
B11
lab2 := b12
B13
lab2 := B14
B14
lab2 = B14
y
n
B17
B15
B0
lab2 := B10
B3a
a > b
B3b
a > b
y
y
n
n
B4a
goto lab2
B4b
goto lab2
B10
null?
B12
n
y
B11
lab2 := B12
B13
lab2 := B14
B14
lab2 = B14
y
n
B17p
B15
a := a + 1
B17
done
a > b
y
n
L1
null? l2
y
n
a := a + 1
a > b
y
n
L2
done
l2:=rest l2

Aldor vs C (Part I)



Aldor vs C (Part II)



Example: Prime Number Sieve

```
# include "axllib.as"

import from Boolean, SingleInteger;

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

    np := 0;

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

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

Example: Multiple Values

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

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

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

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

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

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

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

Example: Constructing an alternate view

```
+++ This constructor creates the 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:
+++   (L1 * L2).(f) = L1 L2 f
NNI ==> NonNegativeInteger;
SUP ==> 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)`$\rightarrow$`Module R`, generically compute `F G R` $\rightarrow$ `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.

Example: Re-organizing Data Structures

```
#include "axllib"

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 Integer := 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 Integer := swap(Array,List)(Integer)(al);

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

Recent and On-going Work

- Advanced libraries for polynomial and differential systems (triangular decomposition, generic solution of ODE/OΔE/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 efficienlty.
- Optimizaiton of generics.

Conclusions

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