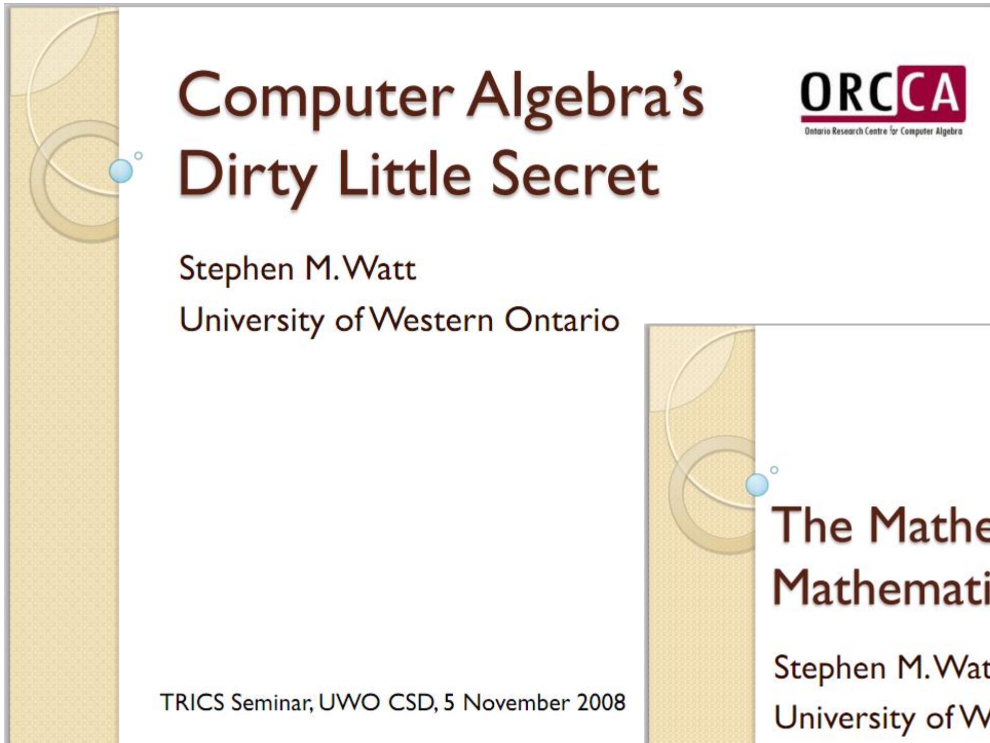
Dependent Types and Categorical Programming

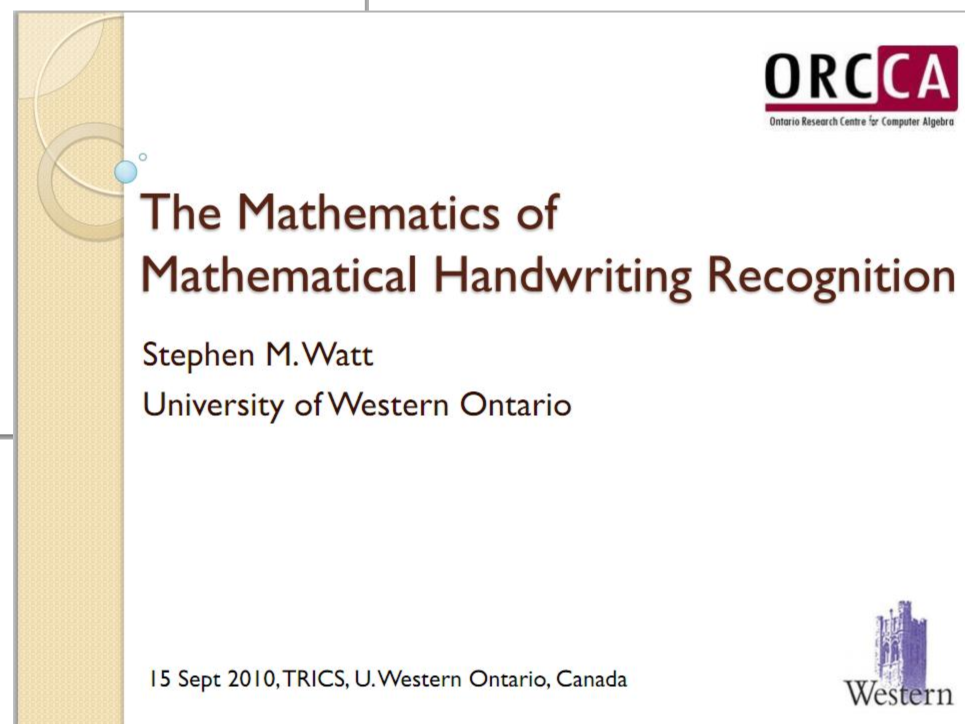
or What can we learn from Aldor?

Stephen M. Watt
University of Western Ontario

TRICS, University of Western Onatrio, 18 January 2012
based on a talk given at École Polytechnique, Palaiseau, 23 September 2011

Previous TRICS!





And now for something completely different….

Types in Programming Languages

- Built from some basic types, e.g.
  - int, double, char, …
- Composed with some built-in constructors, e.g
  - records, unions, functions, enumerations, objects, …
- Different type systems have different properties, e.g.
  - Dynamic vs static, opaque vs explicit, …

The Basic Building Blocks

- Usually the built in constructors are based on simple ideas from mathematics.
- Mapping types:

  $$A \to B$$

- Cartesian product types:

  $$A \times B$$

- … and combinations: $A1 \times A2 \times A3 \to R1 \times R2$

Properties

- Using these basic mathematical ideas allows one to reason about types.

  For example the projections on Cartesian products model record selection.

  first: $A \times B \to A$
  second: $A \times B \to B$

Types and execution time values

- A type system may allow type expressions to have symbols bound at execution time, e.g. arrays of size n.
- In this situation, we may want a type to *depend* on a run time value, such as an argument to a function, e.g.

$$(n\text{:}\ \mathrm{int}) \rightarrow \mathrm{SquareMatrix}(n, \mathrm{double})$$
$$(n\text{:}\ \mathrm{int}) \times \mathrm{SquareMatrix}(n, \mathrm{double})$$

Types and execution time values

$$(n{:}\,\mathrm{int}) \to \mathrm{SquareMatrix}(n, \mathrm{double})$$
$$(n{:}\,\mathrm{int}) \times \mathrm{SquareMatrix}(n, \mathrm{double})$$

- These are dependent types, which can be very powerful and useful.
- We have given up some useful properties of cartesian products though.
- Things get really interesting when the variables can themselves be types…..

What would a language based on these ideas look like…

Aldor

- What is Aldor?
- Aspects of the Aldor language.
- Some lessons learned.

What is Aldor?

IBM Research

- Initially, extension language for Scratchpad II (➔ Axiom).
- First experiments 1984-1990.
- New implementation 1990-1995.
- Various early names, first described as $\boldsymbol{A}^{\#}$ (ISSAC 1994).
- Several new features back-ported to Scratchpad II/Axiom.

Numerical Algorithms Group

- Distributed by NAG with Axiom 2 1995-2000
- Used in FRISCO 1996-1999.

Aldor.org

- Available open source 2002-date

What is Aldor?

- Want a programming language for library development.
- Programming in the small *vs* programming in the large.
- Want abstraction, flexibility, safety and efficiency.
- Want to model rich relations among mathematical objects.

What is Aldor?

- A higher order language for natural expression of mathematical programs.
- Functional, Object-Oriented, Aspect-Oriented characteristics.
- Types and functions first class values.
- Efficiency/flexibility tradeoff achieved through categories and dependent typing.
- Optimizing compiler generates intermediate code, then LISP or C.

Why Math in Prog Lang Research?

- Rich relationships among sophisticated abstractions.
- Well-defined domain.
- Many programming language ideas originated here: algebraic expressions, arrays, big integers, garbage collection, pattern matching, parametric polymorphism, …

Aldor Example I

```
#include "aldor"
double(n: Integer): Integer == n + n
```

Aldor Example II

```
#include "aldor"
#include "aldorio"

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
```

Aldor Example III

```
#include "aldor"
MiniList(S: BasicType): LinearAggregate(S) == add {
        Rep == Union(nil: Pointer,
                     rec: Record(first: S, rest: Rep));
        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);
            l
        }
        generator(l: %): Generator S == generate {
            while not empty? l repeat {
                  yield first l; l := rest 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
        }
        ...
}
```

Aldor and Its Type System

- Types and functions are values
  - May be created dynamically
  - Provide representations of mathematical sets and functions
- The type system has two levels
  - Each value belongs to a unique type, its *domain,* known statically.
  - This is an abstract data type that gives the representation.
  - The domains are values with domain Domain.
  - Each value may belong to any number of subtypes of its domain.
  - Subtypes of Domain are called *categories*.
- Categories
  - specify what exports (operations, constants) a domain provides.
  - fill the role of OO interfaces or abstract base classes.

Why Two Levels?

- OO inheritance pb with multi-argument fns:

```
class SG { “*”: (SG, SG) -> SG; }
DoubleFloat extends SG ...
Permutation extends SG ...

x, y ∈ DoubleFloat ⊂ SG
p, q ∈ Permutation ⊂ SG

x * y   ✓
p * q   ✓
p * y   ✓   !!!
```

Why Two Levels?

- OO inheritance pb with multi-argument fns:

```
SG == ... { “*”: (%, %) -> %; }
DoubleFloat: SG ...
Permutation: SG ...

x, y ∈ DoubleFloat ∈ SG
p, q ∈ Permutation ∈ SG

x * y   ✓
p * q   ✓
p * y   ✗
```

Parametric Polymorphism

- PP is via category- and domain-producing 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; conj: % -> %; ...
} == add {
    Rep  == Record(real: R, imag: R);
       0: % == …
       1: % == …
       (x: %) + (y: %): % == ...
}
```

Dependent Types

- Give dynamic typing, e.g.

```
f: (n: Integer, R: Ring, m: IntegerMod(n)) -> SqMatrix(n, R)
```

- Recover OO through dependent products:

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

- With categories, guarantee required operations available:

```
f(R: Ring)(a: R, b: R): R == a*b + b*a
```

Multi-sorted Algebras

- Category signature as a dependent product type.

```
ArithmeticModel: Category == with {
    Nat: IntegralDomain;
    Rat: Field;
    /:   (Nat, Nat) -> Rat;
}
```

Aldor and Its Type System

- 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 allow domains to belong to new categories *after* they have been initially defined.

Post Facto Extension for Structuring Libraries

```
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 { ... }
```

Post Facto Extension for Structuring Libraries

```
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.

Higher Order Operations

- E.g. Reorganizing constructions

```
Polynomial(x) Matrix(n) Complex R ≈
Complex Matrix(n) Polynomial(x) R
```

- Slightly simpler example

```
List Array String R ≈ String Array List R
```

Higher Order Operations

```
Ag ==> (S: BasicType) -> LinearAggregate S;

swap(X:Ag, Y:Ag)(S:BasicType)(x:X Y S):Y X S ==
  [[s for s in y] for y in x];

al: Array List Integer :=
  array(list(i+j-1 for i in 1..3) for j in 1..3);

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

Phew!

Using Genericity

```
LinearOrdinaryDifferentialOperator(
     A: DifferentialRing,
     M: LeftModule(A) with differentiate: % -> %
) : MonogenicLinearOperator(A) with {
     D: %;
     apply: (%, M) -> M;
     ...
     if A has Field then {
         leftDivide: (%, %) -> (quotient: %, remainder: %);
         rightDivide:(%, %) -> (quotient: %, remainder: %);
          … // rgcd, lgcd
        }
} == ...
```

Using Genericity

```
LinearOrdinaryDifferentialOperator(
     A: DifferentialRing,
     M: LeftModule(A) with differentiate: % -> %
) : ...
== SUP(A) add {
        ...
        if A has Field then {
            Op    == OppositeOperator(%, A);
            DOdiv == NonCommutativeOperatorDivision(%, A);
            OPdiv == NonCommutativeOperatorDivision(Op,A);
            leftDivide (a,b) == leftDivide(a, b)$DOdiv;
            rightDivide(a,b) == leftDivide(a, b)$OPdiv;
        }
        ...
}
```

Design Principles I

- No compromises on flexibility
- No compromises on efficiency
- Use optimization to bridge the gap.
- Compilation. Separate compilation.
- Generated intermediate code is platform independent, even though word-sizes, *etc*, vary.
- Libraries can be distributed, if desired, as binary only.
- Be a good citizen in a multi-language framework.
  - Call and be called by C/C++/Fortran/Lisp/Maple
  - Functional arguments
  - Cooperating memory management

Design Principles II

- Language-defined types should have no privilege whatsoever over application-defined types.
  - Syntax, semantics (e.g. in type exprs), optimization (e.g. constant folding)
- Language semantics should be independent of type.
  - E.g. named *constants* overloaded, not functions
- Combining libraries should be easy, $O(n)$, not $O(n^2)$.
  - Should be able to extend existing things with new concepts without touching old files or recompiling.
- Safety through optimization removing run-time checks, not by leaving off the checks in the first place.

The Compiler as an Artefact

- Written primarily in C (C++ too immature in 1990)
- 1550 files, 295 K loc C + 65 K loc Aldor
- Intermediate code (FOAM):
  - Primitive types: booleans, bytes, chars, numeric, arrays, closures
  - Primitive operations: data access, control, data operations
- Runtime system:
  - Memory management
  - Big integers
  - Stack unwinding
  - Export lookup from domains
  - Dynamic linking
  - Written in C and Aldor

Example of Optimization

From the domain Segment(E: OrderedAbelianMonoid)

```
generator(seg:Segment E):Generator E == generate {
    (a, b) := (low seg, hi seg);
    while a <= b repeat { yield a; a := a + 1 }
}
```

From the domain List(S: Set)

```
generator(l: List S): Generator S == generate {
    while not null? l repeat { yield first l; l := rest l }
}
```

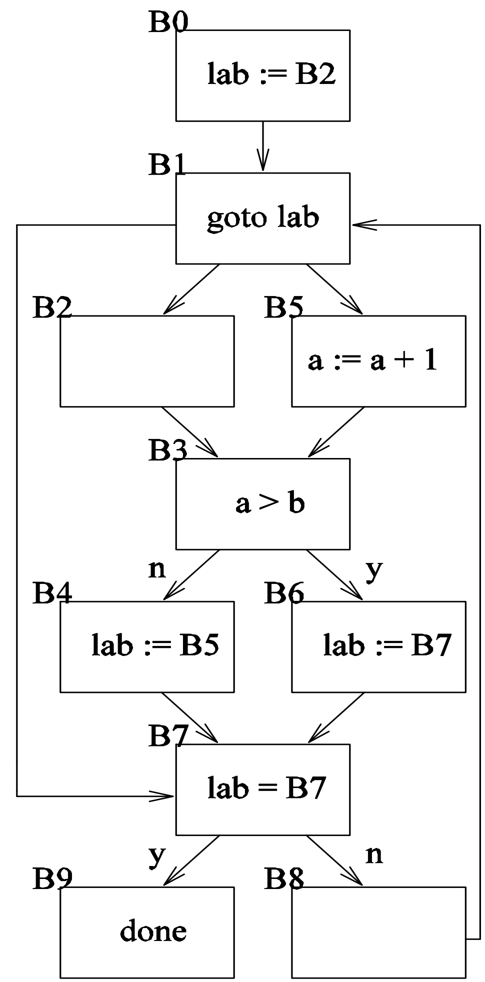
Client code

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

How Generators Work

```
generator(seg:Segment Int):Generator Int
== generate {
    a := lo seg;
    b := hi seg;
    while a <= b repeat {
        yield a; a := a + 1
    }
}

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



Example of Optimization (again)

From the domain Segment(E: OrderedAbelianMonoid)

```
generator(seg:Segment E):Generator E == generate {
    (a, b) := (low seg, hi seg);
    while a <= b repeat { yield a; a := a + 1 }
}
```

From the domain List(S: Set)

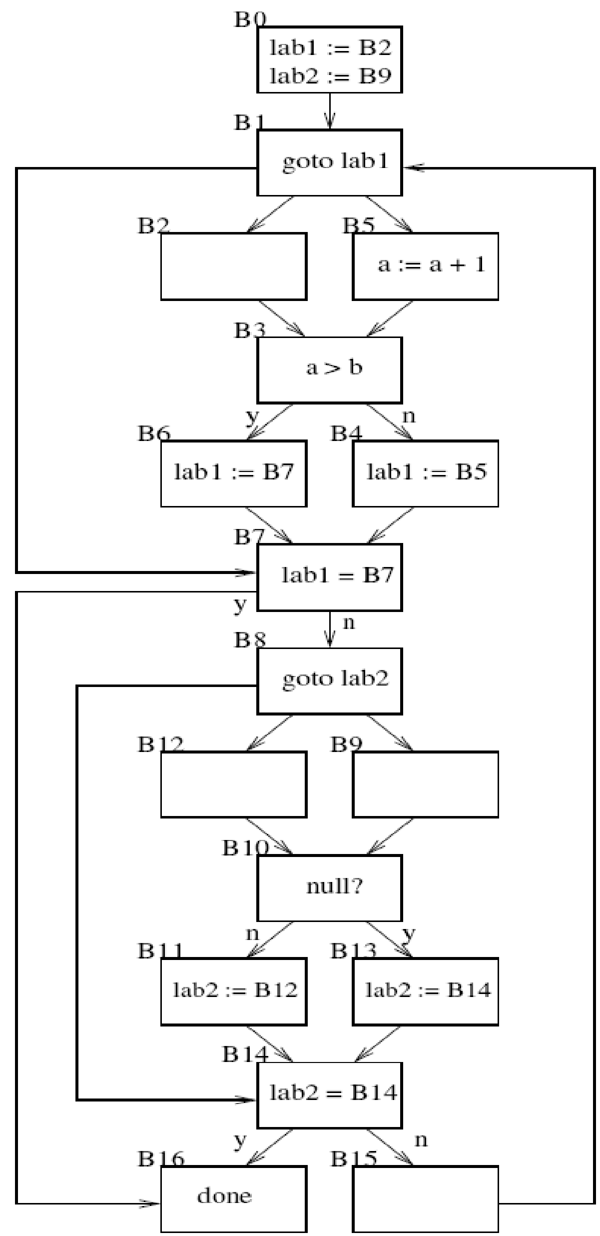
```
generator(l: List S): Generator S == generate {
    while not null? l repeat { yield first l; l := rest l }
}
```

Client code

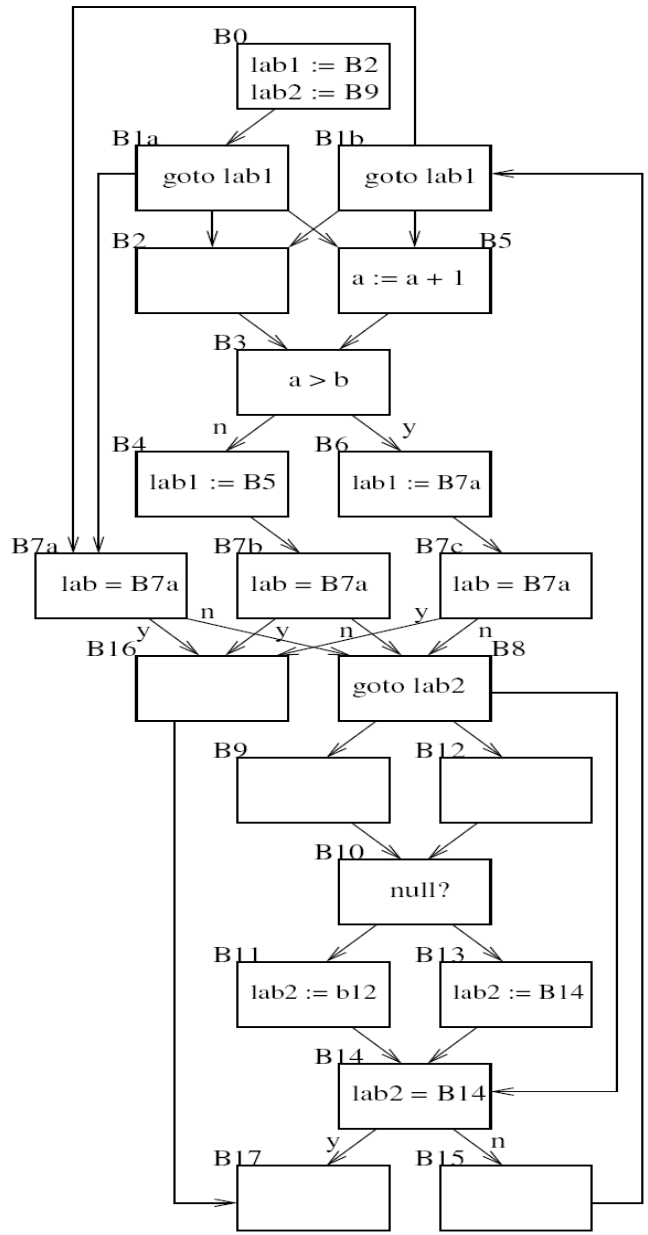
```
client() == {
    ar := array(...);  li := list(...);
    s := 0;   -- NOTE PARALLEL TRAVERSAL.
    for i in 1..#ar for e in l repeat { s := s + ar.i + e }
    stdout << s
}
```

Inlined

```
B0:  ar := array(...);
     l := list(...);
     segment := 1..#ar;
     lab1 := B2;
     l2 := l;
     lab2 := B9;
     s := 0;
     goto B1;
B1:  goto @lab1;
B2:  a := segment.lo;
     b := segment.hi;
     goto B3;
B3:  if a > b then goto B6; else goto B4;
B4:  lab1 := B5;
     val1 := a;
     goto B7;
B5:  a := a + 1
     goto B3;
B6:  lab1 := B7;
     goto B7;
B7:  if lab1 == B7 then goto B16; else goto E
B8:  i := val1;
     goto @lab2;
B9:  goto B10
B10: if null? l2 then goto B13; else goto B11
B11: lab2 := B12
     val2 := first l2;
     goto B14;
B12: l2 := rest l2
     goto B10
B13: lab2 := B14
     goto B14
B14: if lab2 == B14 then goto B16; else goto
B15: e := val2;
     s := s + ar.i + e
     goto B1;
B16: stdout << s
```



Clone Blocks for 1st Iterator



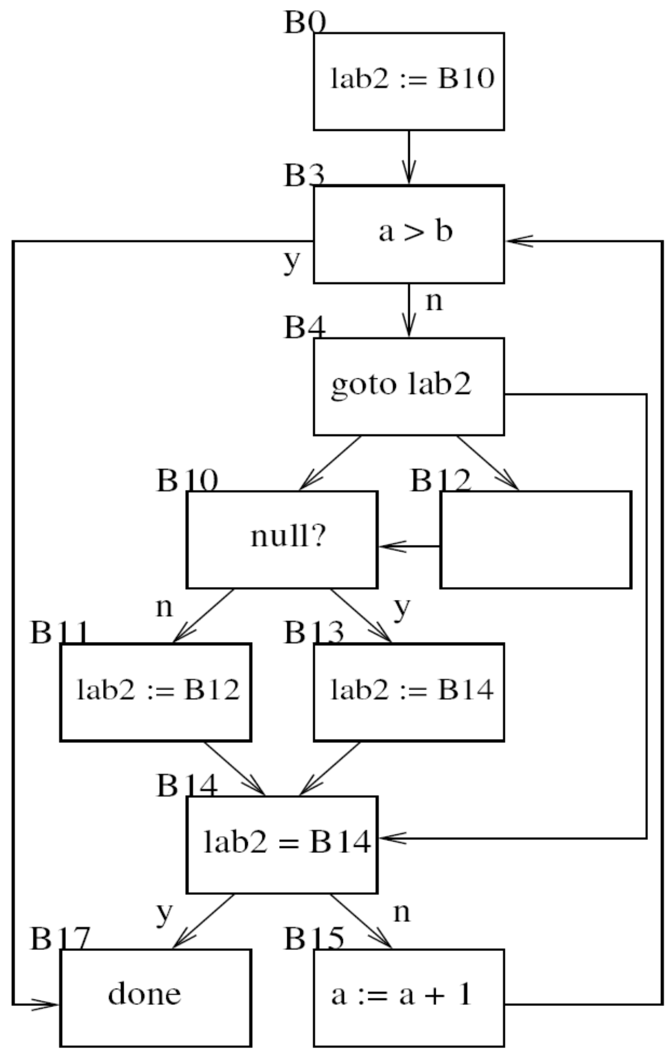
Dataflow

Block	Preds	Succs	Gen	Kill	In	Out
B0		B1a	1..	.11	...	1..
B1a	B0	B2 B5 B7a	...	...	1..	1..
B1b	B15	B2 B5 B7a	...	...	11.	11.
B2	B1a B1b	B3	...	...	11.	11.
B3	B2 B5	B6 B4	...	...	11.	11.
B4	B3	B7b	.1.	1.1	11.	.1.
B5	B1a B1b	B3	...	...	11.	11.
B6	B3	B7c	..1	11.	11.	..1
B7a	B1a B1b	B8 B16	...	...	11.	11.
B7b	B4	B8 B16	...	...	.1.	.1.
B7c	B6	B8 B16	...	...	..1	..1
B8	B7a B7b B7c	B9 B12 B14	...	..1	111	11.
B9	B8	B10	...	...	11.	11.
B10	B9 B12	B11 B13	...	...	11.	11.
B11	B10	B14	...	...	11.	11.
B12	B8	B10	...	...	11.	11.
B13	B10	B14	...	...	11.	11.
B14	B8 B11 B13	B17 B15	...	...	11.	11.
B15	B14	B1b	...	...	11.	11.
B16	B7a B7b B7c	B17	..1	11.	111	..1
B17	B16 B14		...	...	111	111

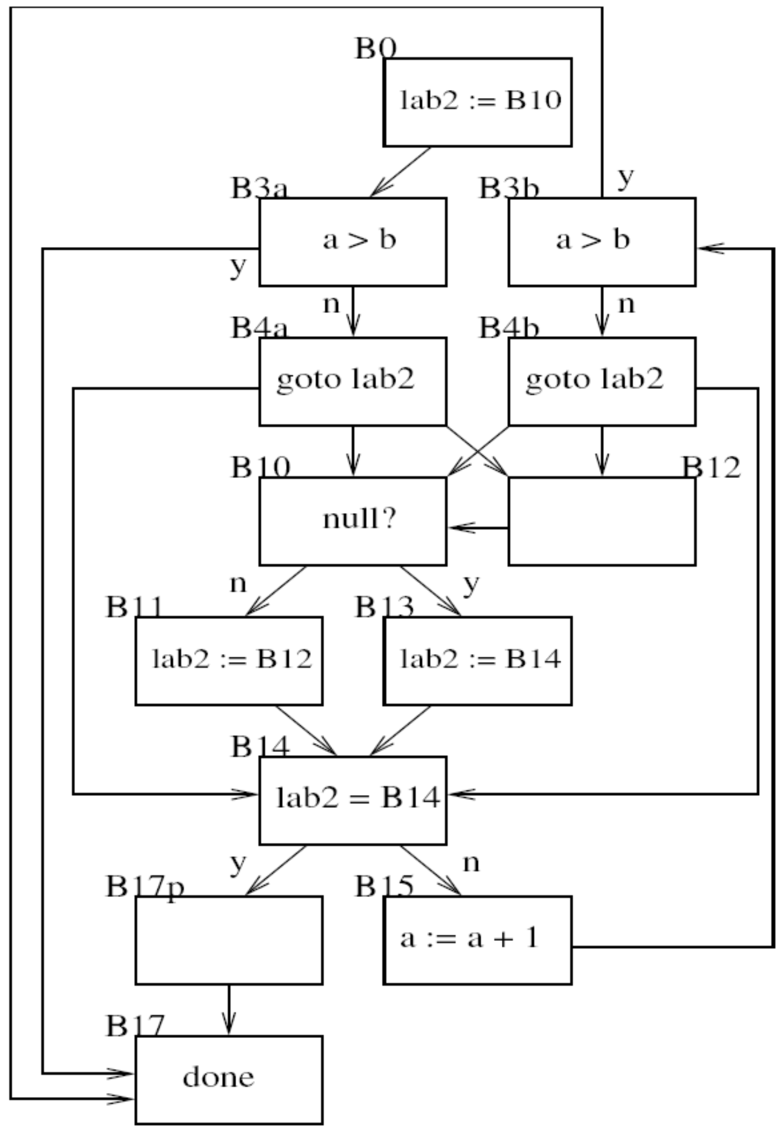
Block	Preds	Succs	Gen	Kill	In	Out
B0		B1a	1..	.11	...	1..
B1a	B0	B2	...	...	1..	1..
B1b	B15	B2 B5	...	...	.1.	.1.
B2	B1a B1b	B3	...	...	11.	11.
B3	B2 B5	B6 B4	...	...	11.	11.
B4	B3	B7b	.1.	1.1	11.	.1.
B5	B1b	B3	...	...	.1.	.1.
B6	B3	B7c	..1	11.	11.	..1
B7a	B1b	B8	...	...	.1.	.1.
B7b	B4	B8	...	...	.1.	.1.
B7c	B6	B16	...	...	..1	..1
B8	B7a B7b	B9 B12 B14	...	...	.1.	.1.
B9	B8	B10	...	...	.1.	.1.
B10	B9 B12	B11 B13	...	...	.1.	.1.
B11	B10	B14	...	...	.1.	.1.
B12	B8	B10	...	...	.1.	.1.
B13	B10	B14	...	...	.1.	.1.
B14	B8 B11 B13	B17 B15	...	...	.1.	.1.
B15	B14	B1b	...	...	.1.	.1.
B16	B7c	B17	...	...	..1	..1
B17	B16 B14		...	...	.11	.11

```
[lab1 == B2, lab1 == B5, lab1 == B7]
```

Resolution of 1st Iterator

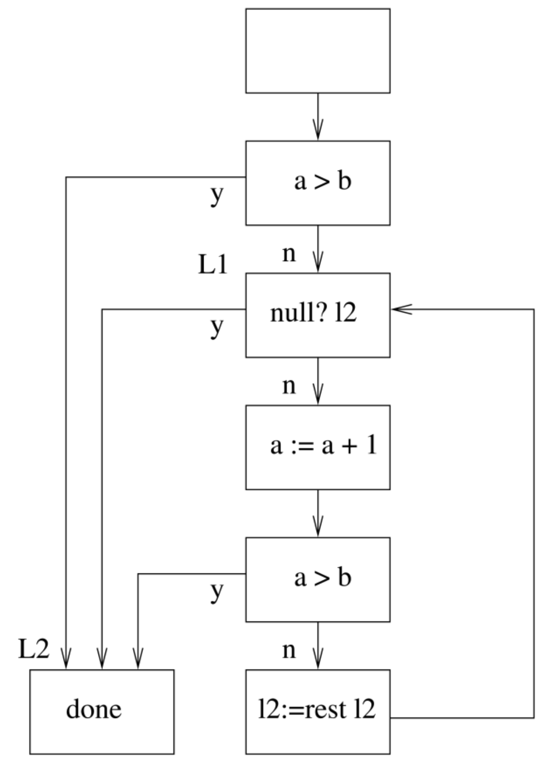


Clone Blocks for 2nd Iterator

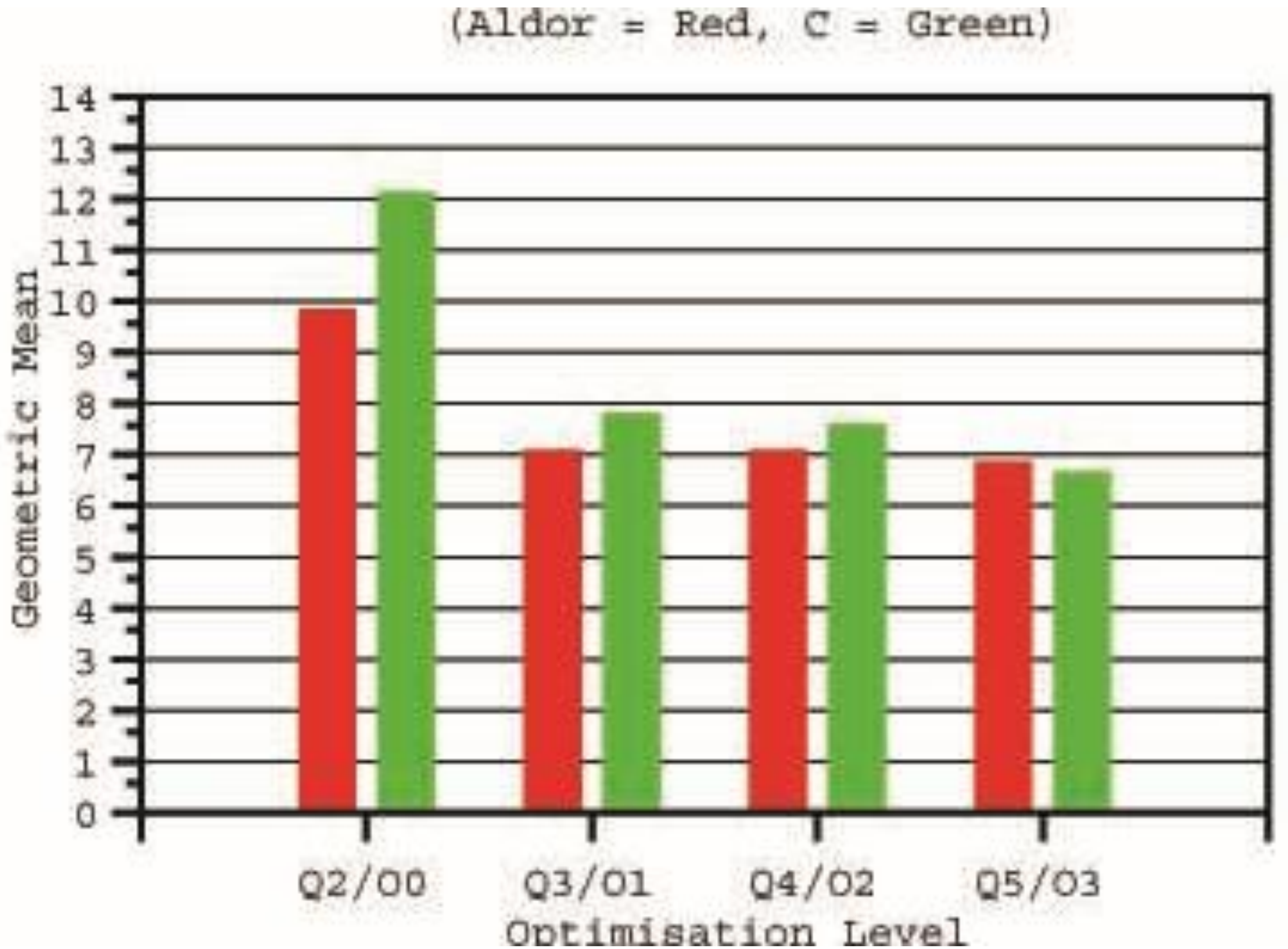


Resolution of 2nd Iterator

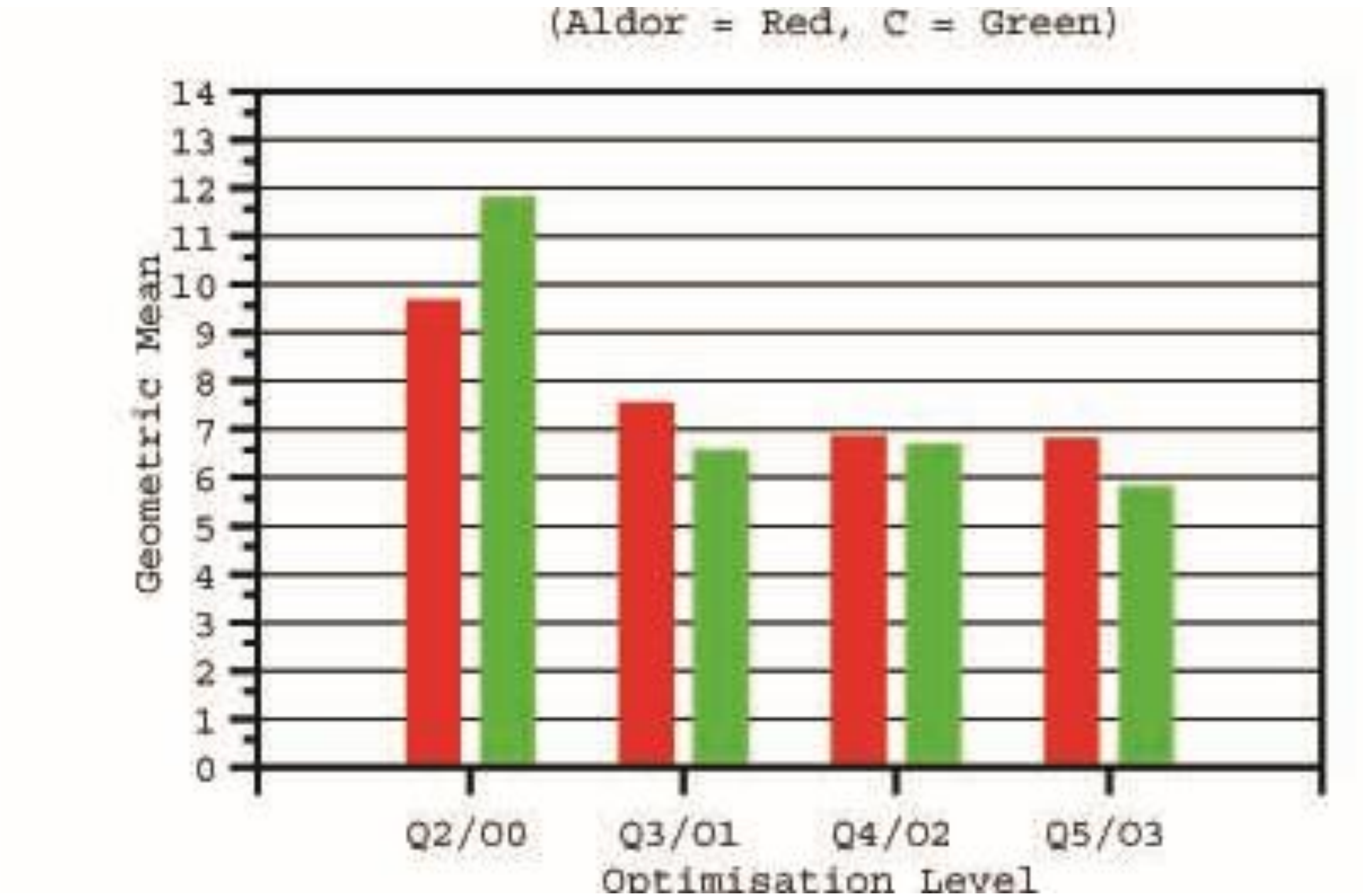
```
client() == {
      ar := array(...);
      l  := list(...);
      l2 := l;
      s  := 0;
      a  := 1;
      b  := #ar;
      if a > b then goto L2
L1:   if null? l2 then goto L2
      e := first l2;
      s := s + ar.a + e
      a := a + 1
      if a > b then goto L2
      l2 := rest l2
      goto L1
L2:   stdout << s
}
```



Aldor vs C (non-floating pt)



Aldor vs C (floating point)



Lessons Learned

- It is possible to be elegant, abstract and high-level without sacrificing significant efficiency.
- Well-known optimization techniques can be effectively adapted to the symbolic setting.
- Optimization of generated C code is not enough.

- Procedural integration, dataflow analysis, subexpression elimination and constant folding are the primary wins.
- Compile-time memory optimization, including data structure elimination, is important.
  - Removes boxing/unboxing, closure creation, dynamic allocation of local objects, *etc.* Can move hot fields into registers.

Conclusions

- Language design 20+ years old.
  - In the mean time, many of the ideas now mainstream.
  - Many still are not.
- Mathematics is a valuable canary in the coal mine of general purpose software.
  - The general world lags in recognizing needs.
- It has to be free.
  - Free^1 is the standard price.
  - Free^2 is required for engagement.

Prospectives

- New prospects for optimization:
  - Allowing opaque types to assert identities – use in optimization. Proof-carrying code should be within.
  - Rely more on JIT to optimize composed functors.
- Enhance semantics to support systematic parameterization.
  - Default views to limit exponential param explosion.
  - More operations on multi-sorted algebras.

Prospectives

- Mathematical Interface Definition Language
  - To make better sound use of external libraries.
  - E.g. BLAS, FLINT, etc with consistent semantics.
- More use of relevant modern standards.
  - MathML3, Unicode, HTTP, C1X, Modelica, …
- Support for collaboration.
  - Shared spaces, roll backs, etc.
- What kind of free?


2001
Muscadet Primeur
GRAND VIN
CHATEAU LATOUR
ALDOR
TAYLOR'S
1966
VINTAGE PORT
TAYLOR'S
1960
VINTAGE PORT
Penfolds
GRANDFATHER