Dependent Types and Categorical Programming

or What can we learn from Aldor?

Stephen M. Watt
University of Western Ontario

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

Computer Algebra’s Dirty Little Secret

Stephen M. Watt
University of Western Ontario

The Mathematics of Mathematical Handwriting Recognition

Stephen M. Watt
University of Western Ontario

TRICS Seminar, UWO CSD, 5 November 2008

15 Sept 2010, TRICS, U. Western Ontario, Canada
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:
  \[ A \rightarrow B \]

- Cartesian product types:
  \[ A \times B \]

- ... and combinations: \[ A1 \times A2 \times A3 \rightarrow R1 \times R2 \]
Properties

- Using these basic mathematical ideas allows one to reason about types.
  
  For example the projections on Cartesian products model record selection.

  first: \( A \times B \rightarrow A \)

  second: \( A \times B \rightarrow B \)
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.

\[
(n\colon\text{int}) \rightarrow \text{SquareMatrix}(n, \text{double})
\]

\[
(n\colon\text{int}) \times \text{SquareMatrix}(n, \text{double})
\]
Types and execution time values

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

- 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 A♯ (ISSAC 1994).
• Several new features back-ported to Scratchpad II/Axiom.

Numerical Algorithms Group
• Distributed by NAG with Axiom 2 1995-2000

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 1

#include "aldor"

double(n: Integer): Integer == n + n
#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
#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
}
\[
\begin{align*}
\text{[g: Generator S]} & \Rightarrow \{ \\
\quad \text{r} \leftarrow \text{empty}(); \text{for } s \text{ in } g \text{ repeat } \text{r} \leftarrow \text{cons}(s, \text{r}); \\
\quad \text{l} \leftarrow \text{empty}(); \text{for } s \text{ in } \text{r} \text{ repeat } \text{l} \leftarrow \text{cons}(s, \text{l}); \\
\quad \text{l}
\end{align*}
\]

\[
\begin{align*}
\text{generator(l: \%)} & \Rightarrow \text{Generator S} \Rightarrow \text{generate} \{ \\
\quad \text{while not empty? } \text{l} \text{ repeat } \{ \\
\quad\quad \text{yield first } \text{l}; \text{l} \leftarrow \text{rest } \text{l}
\quad \}
\}
\end{align*}
\]

\[
\begin{align*}
\text{(l1: \%)} & = (\text{l2: \%}) \Rightarrow \text{Boolean} \Rightarrow \{ \\
\quad \text{while not empty? } \text{l1} \text{ and not empty? } \text{l2} \text{ repeat } \{ \\
\quad\quad \text{if first } \text{l1} \leftarrow \text{first } \text{l2} \text{ then return false; } \\
\quad\quad \text{l1, l2} \leftarrow (\text{rest } \text{l1}, \text{rest } \text{l2})
\quad \}
\}
\end{align*}
\]

\[
\begin{align*}
\text{empty? } \text{l1} \text{ and empty? } \text{l2}
\}
\end{align*}
\]

\[
\begin{align*}
\text{...}
\}
\]
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:

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

\[
x, y \in \text{DoubleFloat} \subset \text{SG}
\]

\[
p, q \in \text{Permutation} \subset \text{SG}
\]

\[
x \ast y \checkmark
\]

\[
p \ast q \checkmark
\]

\[
p \ast y \checkmark \text{!!!}
\]
Why Two Levels?

- OO inheritance pb with multi-argument fns:

\[
\text{SG} == \ldots \{ "*": (\%, \%) \rightarrow \%; \} \\
\text{DoubleFloat: SG} \ldots \\
\text{Permutation: SG} \ldots \\
\]

\[
x, y \in \text{DoubleFloat} \in \text{SG} \\
p, q \in \text{Permutation} \in \text{SG} \\
\]

\[
x * y \; \checkmark \\
p * q \; \checkmark \\
p * y \; \times
\]
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.

\[ f: \text{Integer} \rightarrow \text{Integer} \]

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

• Recover OO through dependent products:

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

• With categories, guarantee required operations available:

\[ f(R: \text{Ring})(a: R, b: R): R = a*b + b*a \]
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 {
    ...
} == 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

  \[ \text{Polynomial}(x) \cdot \text{Matrix}(n) \cdot \text{Complex} \cdot R \approx \text{Complex} \cdot \text{Matrix}(n) \cdot \text{Polynomial}(x) \cdot R \]

- Slightly simpler example

  \[ \text{List} \cdot \text{Array} \cdot \text{String} \cdot R \approx \text{String} \cdot \text{Array} \cdot \text{List} \cdot R \]
Higher Order Operations

\[ \text{Ag} \Rightarrow (S: \text{BasicType}) \rightarrow \text{LinearAggregate} S; \]

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

\[ \text{al: Array List Integer} := \]
\[ \text{array(list(i+j-1 \text{ for } i \text{ in } 1..3) \text{ for } j \text{ in } 1..3}); \]

\[ \text{la: List Array Integer} := \]
\[ \text{swap(Array, List)(Integer)(al)}; \]
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)

```lisp
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)

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

*Client code*

```lisp
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 F

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

<table>
<thead>
<tr>
<th>Block</th>
<th>Preds</th>
<th>Succs</th>
<th>Gen</th>
<th>Kill</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td></td>
<td>B1a</td>
<td>1..11</td>
<td></td>
<td>1..1</td>
<td></td>
</tr>
<tr>
<td>B1a</td>
<td>B0</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1b</td>
<td>B2</td>
<td>B6</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>B1a B1b</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>B2 B5</td>
<td>B6 B4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>B3</td>
<td>B7b</td>
<td>.1.11</td>
<td>11.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>B1a B1b</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td>B3</td>
<td>B7c</td>
<td>.111.11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7a</td>
<td>B1a B1b</td>
<td>B8 B16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7b</td>
<td>B4</td>
<td>B8 B16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7c</td>
<td>B6</td>
<td>B8 B16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td>B7a B7b B7c B9 B12 B14</td>
<td>.1</td>
<td>111</td>
<td>11.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B9</td>
<td>B8</td>
<td>B10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B10</td>
<td>B9 B12</td>
<td>B11 B13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B11</td>
<td>B10</td>
<td>B14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B12</td>
<td>B8</td>
<td>B10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B13</td>
<td>B10</td>
<td>B14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B14</td>
<td>B8 B11 B13</td>
<td>B17 B15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B15</td>
<td>B14</td>
<td>B1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B16</td>
<td>B7a B7b B7c B17</td>
<td>.111</td>
<td>111</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B17</td>
<td>B16 B14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Block</th>
<th>Preds</th>
<th>Succs</th>
<th>Gen</th>
<th>Kill</th>
<th>In</th>
<th>Out</th>
</tr>
</thead>
<tbody>
<tr>
<td>B0</td>
<td></td>
<td>B1a</td>
<td>1..11</td>
<td></td>
<td>1..1</td>
<td></td>
</tr>
<tr>
<td>B1a</td>
<td>B0</td>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B1b</td>
<td>B15</td>
<td>B2 B5</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B2</td>
<td>B1a B1b</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td>B2 B5</td>
<td>B6 B4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B4</td>
<td>B3</td>
<td>B7b</td>
<td>.1.11</td>
<td>11.1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B5</td>
<td>B1b</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B6</td>
<td>B3</td>
<td>B7c</td>
<td>.111.11</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7a</td>
<td>B1b</td>
<td>B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7b</td>
<td>B4</td>
<td>B8</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B7c</td>
<td>B6</td>
<td>B16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B8</td>
<td>B7a B7b</td>
<td>B9 B12 B14</td>
<td>.1</td>
<td>111</td>
<td>11.1</td>
<td></td>
</tr>
<tr>
<td>B9</td>
<td>B8</td>
<td>B10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B10</td>
<td>B9 B12</td>
<td>B11 B13</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B11</td>
<td>B10</td>
<td>B14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B12</td>
<td>B8</td>
<td>B10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B13</td>
<td>B10</td>
<td>B14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B14</td>
<td>B8 B11 B13</td>
<td>B17 B15</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B15</td>
<td>B14</td>
<td>B1b</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B16</td>
<td>B7c</td>
<td>B17</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B17</td>
<td>B16 B14</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

[lab1 == B2, lab1 == B5, lab1 == B7]
Resolution of 1st Iterator
Clone Blocks for 2\textsuperscript{nd} Iterator
Resolution of 2\textsuperscript{nd} Iterator

client() == 
\[
\begin{align*}
\text{ar} & := \text{array}(\ldots); \\
\text{l} & := \text{list}(\ldots); \\
\text{l2} & := \text{l}; \\
\text{s} & := 0; \\
\text{a} & := 1; \\
\text{b} & := \#\text{ar};;
\end{align*}
\]
\text{if } \text{a} > \text{b} \text{ then goto } \text{L2}

\text{L1: if null? } \text{l2} \text{ then goto } \text{L2}
\text{\hspace{1em} e := first } \text{l2};
\text{\hspace{1em} s := s + ar.a + e}
\text{\hspace{1em} a := a + 1}
\text{\hspace{1em} if a > b then goto } \text{L2}
\text{\hspace{1em} l2 := rest l2}
\text{\hspace{1em} goto } \text{L1}

\text{L2: stdout } \text{<< } \text{s}
\text{}}
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?