The Role of Categorical Languages

Stephen M. Watt
Stephen.Watt@uwo.ca

Ontario Research Centre for Computer Algebra
University of Western Ontario

NSF CDI Workshop on Cyber-enabled Discovery and Innovation (CDI)
30 Oct 2007, Arlington VA
Why Math in Prog Language Research?

- Rich relationships among non-trivial concepts.
- Well-defined domain.
- Many programming language 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.
Computer Algebra vs Symbolic Computation

Computer algebra

- Arithmetic on defined algebraic structures.
- Polynomials, algebraic functions, linear algebra, quotients, ...
- May involve symbols, parameters, indeterminates.

Symbolic computation

- Computation with expression trees ("terms").
  - Symbols denoting operations ("+", "x", "sin")
  - Constants and variables (and what is the difference?)
- Transformation, simplification
- Expression equivalence
- What is defined?
Computer Algebra

- Solving polynomial systems
- Antidifferentiation (≠ integration)
- \( \mathbb{Q}[\alpha]/\langle \alpha^2 + 3\alpha + 7 \rangle : \frac{1}{5\alpha^2 + 2\alpha - 7} \rightarrow \alpha^2 + \frac{3940}{1309}\alpha + \frac{9160}{1309} \)
Symbolic Computation

\[ ax^2 + bx + c = \left(ax^2 + bx + \frac{b^2}{4a^2}\right) + \left(c - \frac{b^2}{4a^2}\right) \]

\[ = \left(x + \frac{b}{2a}\right)^2 + \left(\frac{\sqrt{4ac - b^2}}{2a}\right)^2 \]

\[ = \left(x - \frac{-b + \sqrt{4ac - b^2}}{2a}\right) \times \left(x - \frac{-b - \sqrt{4ac - b^2}}{2a}\right) \]
Models of Symbolic Computing?

- Computing with exact mathematical values.
- Computing with approximate mathematical values.
- Proving theorems.
- Computing with algebraic values.
- Term rewriting.
- Recording and retrieving mathematical facts.
- Manipulating notations.
A Problem in Computer Algebra Software

- Systems usually have several implementations of the same algorithm for different structures. E.g. Gaussian elimination over \( \mathbb{Q}, \mathbb{Z}/p\mathbb{Z}, \mathbb{Z}(x),... \)

- Sometimes in alternative views E.g. repeated squaring \((f^n)\) vs repeated doubling \((n \times p)\).

- 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
  - implement them efficiently, and
  - compose constructions flexibly.
Mathematical Interfaces

- Well-defined interfaces between different views of the same thing.
- Well-defined unified view of multiple things.
- This is where “categorical” programming languages come in.
Mathematical Interfaces

• Example: Modelling
  – Components of an automobile each have their own models.
  – 3D Geometry
  – Electrical properties
  – Mechanical properties (static)
  – Mechanical properties (dynamic)
  – Thermal properties

• Tier 1 manufacturer demands of Tier 2 suppliers

• System simulation
Mathematical Interfaces: Modelling

- Operating on system:
  - differentiation
  - simplification
  - exact solution of parts
  - automated approximation of regions
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 sold 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.
• Some user’s libraries 200-400 Kloc.
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.
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, compositibility, efficiency, interoperability
- Express the requirements and the rich relationships among inputs. Express guarantees on the results.
- Then have a language encouraging one to weaken the requirements and strengthen the guarantees.
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

  \[
  \begin{align*}
  \text{Semigroup: } & \text{ Category } \equiv \text{ with } \{ \ast : (\%, \%) \to \% \} \\
  \text{DoubleFloat: } & \text{ Join(Semigroup, ...) } \equiv \ldots \\
  \text{Permutation: } & \text{ Join(Semigroup, ...) } \equiv \ldots
  \end{align*}
  \]

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

  \[
  \begin{align*}
  x, y \in \text{DoubleFloat} & \subset \text{Semigroup} \\
  p, q \in \text{Permutation} & \subset \text{Semigroup}
  \end{align*}
  \]

  Liskov recognized this problem with binary operations already with CLU.

- With categories, the two levels allow \( x \ast y \) but prevent \( x \ast p \).

  \[
  \begin{align*}
  x, y \in \text{DoubleFloat} & \in \text{Semigroup} \\
  p, q \in \text{Permutation} & \in \text{Semigroup}
  \end{align*}
  \]

  Aldor
Dependent types

- 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.

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

- 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

```plaintext
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.
- Provide “aspect oriented” programming, or “separation of concerns”

```plaintext
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;
}
```

- Allow 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:

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

```plaintext
extend DirectProduct(n: Integer, S: Semigroup): Semigroup == ...
extend DirectProduct(n: Integer, S: Monoid): Monoid == ...
extend DirectProduct(n: Integer, S: Group): Group == ...
```

```plaintext
extend DirectProduct(n: Integer, S: Ring): Join(Ring, Module(S)) == ...
extend DirectProduct(n: Integer, S: Field): Join(Ring, VectorField(S)) == ...
```

```plaintext
extend DirectProduct(n: Integer, S: DifferentialRing): DifferentialRing == ...
extend DirectProduct(n: Integer, S: Ordered): Ordered == ...
```

- Normally these extensions would all be in separate files.
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
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>14</td>
<td>13</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aldor vs C (Part II)

Floating-point
(Aldor = Red, C = Green)

<table>
<thead>
<tr>
<th></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>Optimisation Level</td>
<td>Q2/00</td>
<td>Q3/01</td>
<td>Q4/02</td>
<td>Q5/03</td>
</tr>
</tbody>
</table>
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:

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

\[
\text{NNI} \implies \text{NonNegativeInteger};
\]

\[
\text{SUP} \implies \text{SparseUnivariatePolynomial};
\]

**LinearOrdinaryDifferentialOperator**

\[
\text{LinearOrdinaryDifferentialOperator}(\text{A: DifferentialRing, M: LeftModule(A) with differentiate: } \% \rightarrow \%)
\]

\[
\text{D: } \%;\]

\[
\text{apply: } (\%, M) \rightarrow M;\]

\[
\ldots\]

\[
\text{if A has Field then }\{\]

\[
\text{leftDivide: } (\%, \%) \rightarrow \text{Record(quotient: } \%, \text{ remainder: } \%);\]

\[
\text{rightDivide: } (\%, \%) \rightarrow \text{Record(quotient: } \%, \text{ remainder: } \%);\]

\[
\}\]
== 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.
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
Lessons Learned – Type Categories

Type categories widely useful:

• to describe algebraic interfaces
• to describe numerical interfaces
• to describe hardware properties
• to describe exceptional behaviour
• to describe families of models
• ...

Lessons Learned – Type Categories

Next generation:

- Should provide axioms and theorems *about* the code
- Should provide rich interface for multi-sorted algebras
  (avoid exponential parameter build up)
- Should provide an interface between symbolic computation and
  computer algebra.
- ...


Lessons Learned – Code Transformation

Optimization success:

• Procedural integration important.

• Data integration *more* important.

• Can provide efficient realization of very high-level code.
Lessons Learned – Code Transformation

Optimization challenges:

- Need optimization \textit{at} and \textit{across} levels of abstraction.

- Therefore need to be able to describe exactly \textit{what} the components are supposed to do.

- Use of theorems for high-level code transformation. (E.g. coset construction.)

- Data needs to be writable. Data data to be read-only.
What does “working symbolically” mean?

- Allow us to consider families of problems all at once.
- Should be able to adjust level of abstraction:
  - Polynomial of degree $d$.
  - Polynomial of degree 7 with rational function coefficients.
  - Polynomial of degree 7, with rational coefficients.
  - Set of values at particular points.
Bringing Approaches Together

• Well-specified interface, then...

• can *algebratize* symbolic computation
  (initial algebras, free algebras of various sorts, adjoint functors)

• can *symbolicize* algebraic computation
  (more varied structures)

• Amount to the same thing

• Need *algorithms* for these formal structures.
Example

\[ x^{2n} - y^{2m} = (x^n + y^m)(x^n - y^m) \]

\[ x^{n+3} - y^{2m} = (x^{n(n+3)/2} + y^m)(x^{n(n+3)/2} - y^m) \]

\[ 16^n - 81^m = (2^n - 3^m)(2^n + 3^m)(2^{2n} + 3^{2m}) \]

\[ x^{n^4-6n^3+11n^2-6(n+2m-3)-1000000m} \]

\[ = x^{-12m} \times (x^{p_1} + 10^m x^{p_2+2m} + 10^{2m} x^{4m}) \times (x^{p_2} + 10^m x^{2m}) \times (x^{p_1} - 10^m x^{p_2+2m} + 10^{2m} x^{4m}) \times (x^{p_2} - 10^m x^{2m}) \]

\[ p_1 = x^{1/3n^4-2n^3+11/3n^2-2n+6} \]

\[ p_2 = x^{1/6n^4-n^3+11/6n^2-n+3} \]
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.
- **Necessary** for connection between multiple paradigms.