An Introduction to Aldor and its Type System

Martin Dunstan

Numerical Algorithms Group (NAG Ltd)
and
Division of Computer Science
University of St. Andrews

Road Map

- What is Aldor?
- Introduction to syntax and basic concepts
  - constants, variables and literals
  - functions, iteration and generators
  - categories and domains
- Inheritance of Categories and Domains
- Resolving overloading and extensions
- Conclusions

What is Aldor?

- **Strongly-typed, imperative language with:**
  - **two-level object model with inheritance (*c.f.* Haskell)**
  - **overloading of symbol names**
  - **types and functions are (constant) values**
  - **generators, *post facto* extensions, literals *etc.***
- **Foreign language interface:**
  - **LISP (for AXIOM)**
  - **C, C++, Fortran 77**
- **Automatic garbage collection (mark and sweep)**
- **Compiles to FOAM, LISP or C**

Constants, Variables and Literals

```
+++ PI: a well-known constant
PI == 3.1415926535_897932385;

ThisYear == 1999;
   ++ The current year

+++ Who am I?
MyName == "Martin Dunstan";

-- Variables are treated likewise
fortyTwo := 42;
```

Simple Functions

```
fib == (n:Integer):Integer +-> {
   if (n < 2) then 1
   else fib(n - 1) + fib(n - 2);
}

-- More traditional definition
fib(n:Integer):Integer == {
   if (n < 2) then 1;
   else fib(n - 1) + fib(n - 2);
}

-- Bounded, polymorphic, curried function
Add(R:Ring)(a:R)(b:R):R == a + b;
```

Looping and Iteration

```
-- Simple conditional loop
while (i < 10) repeat doSomething(i);

-- Bounded iteration
for i in 1..10 repeat doSomething(i);

-- Unbounded iteration
for i in 1.. repeat doSomething(i);

-- Combined conditional and unbounded
for i in 1..
while (notDone) repeat
   notDone := testIt(i);
```

Generators (Coroutines)

```
primeStream == generate {
   for i in 1.. repeat
      if (prime? i) then yield i;
}

firstPrimes(n:Z):Generator(Z) == generate {
   for prime in primeStream
   for total in 1..n repeat
      yield prime;
}

-- Construct a list of the first ten primes
tenPrimes := [p for p in firstPrimes 10]
```

Categories

- **What are they?**
  - **similar to Haskell type classes**
  - **define the interface of domains**
  - **values of type `Category`**
- **Particular features**
  - **write categories *first*, then domains**
  - **often parameterised: `Aggregate(T:Type)`**
  - **conditional: `if (C has Order) then`**
  - **special symbol `%` for domain implemented**

A Simple Category

```
define Logic:Category == with {
   BasicType;

   ~  :  %      -> %; ++ Logical complement
   /\ :  (%, %)-> %; ++ Logical `meet', e.g. `and'
   \/ :  (%, %)-> %; ++ Logical `join', e.g. `or'
   xor:  (%, %)-> %; ++ `Exclusive or'

   default {
      (x:%) \/ (y:%):% == ~(~x /\ ~y);
      xor(x:%, y:%):%  == (x /\ ~y) \/ (~x /\ y);
   }
}
```

A More Complex Category

```
define LinAg(S:Type):Category == Aggregate S with {
   empty  :  ()                  -> %;
   bracket:  Generator S         -> %;
   bracket:  Tuple S             -> %;
   map    :  ((S,S)->S, %, %)    -> %;
   apply  :  (%, SingleInteger)-> S;

   default {
          empty():% == [];
          map(f:(S,S)->S, a:%, b:%):% ==
             [f(x,y) for x in a for y in b];
   }
}
```

Category Documentation

```
++ Description:
++    The category of associative rings, not
++    necessarily commutative, and not necessarily
++    with a 1.  This is a combination of an abelian
++    group and a semigroup, with multiplication
++    distributing over addition.
++ Axioms:
++    x*(y+z) = x*y + x*z
++    (x+y)*z = x*z + y*z
++ Conditional attributes:
++    spadnoZeroDivisors      ab = 0 => a=0 or b=0
Rng():Category == Join(AbelianGroup,SemiGroup);
```

Domains

- What are they?
  - collections of exported constants
  - abstract data types or packages
  - public view (interface) defined by category
- Particular features
  - often parameterised: `Zmod(n:Integer)`
  - category often written with domain
  - conditional definitions

A Simple Domain

```
define ZmodCat(n:Integer):Category == Ring with {
   if (prime? n) then Field;
}

Zmod(n:Integer):ZmodCat(n) == add {
   Rep == Integer;

   (a:%) + (b:%):% ==
      per mod((rep a) + (rep b), n);
   if (prime? n) then {
      inv(x:%):% == ...
   }
}
```

Complex Numbers

```
Complex(R:Field):Field with {
   *      :  (R, %)-> %;
   complex:  (R, R)-> %;
   real   :  %      -> R;
   imag   :  %      -> R;
} == add {
   Rep == Record(real:R, imag:R);
   import from Rep;

   complex(r:R, i:R):% == per [r, i];

   (r:R)*(c:%):% == complex(r*real c, r*imag c);
}  -- (Other exports omitted ...)
```

Post facto Extensions

- **What are they?**
  - **a way of changing the behaviour of existing domains without modifying existing code**
- **Benefits**
  - **prevent explosion of domains**
  - **incremental library development**
  - **allow for future development**
- ```
  extend Integer:DifferentialRing == add {
     differentiate(n:Integer):Integer == 0;
  }
  ```
  ```

Inheritance in Aldor

- **Categories**
  - **multiple inheritance**
  - **union of exports; disjunction of conditions**
  - `Join`$(C_1, C_2, \ldots, C_n) \equiv$ `with` $\{C_1; C_2; \ldots; C_n\}$
- **Domains**
  - **single inheritance**
  - **representations must be consistent**
  - `ParentDom add ChildDom`

Binding: Resolving Overloading[†]

- **Searching for export *op* in domain *dom*:**
  - **two passes, first pass ignoring all defaults**
  - **first look for *op* in *dom***
  - **try parent of *dom* (skipping defaults)**
  - **domain *dom* extended (skipping defaults)**
  - **try the defaults of *dom***
  - **try the defaults of parent**
  - **try the defaults of domain extended**

[†] See the notes at end of the talk.

Foreign Language Interface

```
import {
   getenv:  String -> String;
   puts:    String -> SingleInteger;
} from Foreign C;

export {
   fib: SingleInteger -> SingleInteger;
} to Foreign C;

import {
   senddbl: (DoubleFloat -> DoubleFloat) -> ();
} from Foreign Fortran;
```

Conclusions

- **Compiler**
  - generates efficient code (comparable to C)
  - FOAM provides platform independance
  - variety of platforms: UNIX, VMS, PC
- **Language**
  - ideal for computer algebra implementations
  - types and functions first class
  - categories permit bounded polymorphism
  - generators provide general iteration

- **Things not mentioned:**
  - **exceptions, case statements**
  - **default arguments for functions, records**
  - **conversion of literals at runtime**
  - **cross products, enumerations, records**
  - **coerce, retract, restrict, qualify**
  - **interactive use of compiler**

Some Comments

After answering the questions that followed my talk two points sprang to mind that I would like to note:

- **late binding would prevent the compiler performing important optimisations such as inlining so we don't do it. The compiler will bind references to identifiers as soon as possible. This can be delayed by using defaults in category definitions.**
- **the qualify "operator" (`export$Domain`) can be used to pin down the domain that an export comes from. You still have to work out which export is actually invoked if your domain is the result of extending or inheriting from other domains. If your categories often use defaults then life is slightly harder.**
- **resolving binding of exports is tricky to explain in a single slide so I'll explain it more fully here. To look for the export `op` in the domain `Dom` we apply the following algorithm twice: the first time defaults are ignored, the second time they aren't:**

  1. **see if `op` is in `Dom`**
  2. **see if `op` is in the parent of `Dom`**
  3. **see if `op` is in the domain that `Dom` extended**

  The important point is that this is recursive: to achieve step 2 we need to look in the parent of `Dom` (apply step 1), the parent of the parent of `Dom` (apply step 2) and so on before we can apply step 3. As soon as an export is found we stop.