4.1.3 Aldor

Aldor is a programming language originally intended to develop compiled libraries for computer algebra. The design of the language was influenced by several factors: It had to be *expressive* enough to capture naturally the high-level objects and relationships which arise in modern mathematics. An implementation had to be *efficient* enough for resource-intensive symbolic and numeric computing needs. Finally, the language had to be *modular* enough to allow large libraries of independently developed facilities to be used together in any combination.

The resulting formulation of the programming language has attempted to balance the mathematical desire for generality and uniformity, on one hand, with the practical requirements of demanding symbolic and numeric computation, on the other. This has required certain trade-offs. As an example, types and programs are first class values in *Aldor*. This means that they may be created and used dynamically, and provides a natural realization of mathematical sets and functions. The language does require, however, that certain information about the types and functions be known statically (before program execution) for efficient execution.

An optimizing compiler has been developed for *Aldor*. Programs are first compiled to a low-level intermediate language, this undergoes a number of optimizing transformations, and from this C, Lisp or native object code is generated. Interlanguage interfaces support natural linking with programs written in C, C++ and Fortran. The *Aldor* compiler was originally used to produce libraries for *Axiom* and its predecessor, while more recent emphasis has been on the production of stand-alone programs or modules for use in Maple.

The main reference for Aldor is [4].

Programming Language Characterization Aldor may be characterized as a strongly typed functional programming language with a higher order type system and strict evaluation. All values are treated uniformly and memory is managed automatically.

The type system has two levels: Each value belongs to some unique type, known as its *domain*, and the domains of expressions can be inferred statically. Each domain is itself a value belonging to the domain Type. Domains may additionally belong to some number of subtypes (of Type), known as *categories*. Categories can specify properties of domains such as which operations they export, and are used to specify interfaces and inheritance hierarchies.

The biggest difference between the two-level domain/category model and the single-level subclass/class model is that a domain *is an element of* a category, whereas a subclass *is a subset of* a class. This difference eliminates a number of problems in the definition of functions with multiple related arguments.

Ex post facto extension allows existing domains to belong to new categories. This supports a programming style which has recently come to be known as "aspect oriented programming."

Dependent products and mapping types are fully supported. Dependent products are tuples where the *type* of some component depends on the *value* of another, e.g. (n: Integer, m: IntegerMod(n)). Dependent mappings are functions where the type of the result depends on the value of an argument, e.g. mod: (Integer, n: Integer) -> IntegerMod(n).

Generic programming is achieved through explicit parametric polymorphism, using functions which take types as parameters and which operate on values of those types, e.g. f(R: Ring, a: R, b: R): R == a * b - b * a. Object oriented programming can be achieved naturally using dependent product values, e.g. (R: Ring, r: R).

Control flow is explicit and uses a fixed set of language-defined primitives, including the usual conditionals, loops, function call and return, as well as stackoriented exception handling (try/catch). Control abstraction is provided by suspendable/resumable generators which yield values over the course of a computation. Continuations are *not* supported in order to retain interoperability with languages such as C and Fortran.

In order that application-defined types have equal privilege and power as built-in types, language properties are defined independently of type. For example, overloading of names is not restricted to function valued quantities. Similarly, new types may support literals which appear in source programs. The language itself has very little in the way of predefined types, and in practice the basic types, e.g. Integer, DoubleFloat, Array(T) are all defined in standard libraries.

History The design of Aldor spanned a the period 1985-1995, and successive implementations, led by Stephen Watt. The original use of the language was as a compiled extension language for the Scratchpad II system [1]) at IBM Research. During its development, Aldor was known internally to IBM as A^{\sharp} ("A sharp") [3].

In 1990/91 IBM partnered with the Numerical Algorithms Group Ltd to release Scratchpad II, and did so under the under the trademark Axiom. The second release of Axiom included the A^{\sharp} compiler. The interim name "Axiom XL" (for Axiom extension language) was used for a short period by NAG before the legal trade name Aldor was established. From 1996 to 2000, NAG continued to extend Aldor, partially through the support the ESPRIT Frisco project, e.g. adding a Fortran foreign function interface.

The language has drawn ideas from a number of programming languages, including Ada, C++, CPL, Clu, Fortran, Lisp, ML, Pebble and Russel. It has borrowed heavily from an earlier language designed by Richard Jenks and Barry Trager [2] at IBM Research. The most notable differences between *Aldor* and this predecessor are that *Aldor* is more heavily functional and dependent types are completely integrated so all expressions have types expressible in the language. The language of [2] required programs to be very highly stylized, with type constructions occurring in particular specialized settings and at top-level only. The ideas *Aldor* has adopted from this language have been reworked to be strongly orthogonal.

In 2001 Aldor.org was formed to distribute *Aldor* more widely, and independently of *Axiom*.

Aldor

Examples

Function definition

```
miniSqrt(x: DoubleFloat): DoubleFloat == {
    r := x;
    r := (r*r + x)/(2.0*r);
    r
}
```

 This function computes a square root by six steps of Newton's method. The value returned by the function, is the value of the expression of its body.

Category definition

- A with expression forms a category.
- BasicType is a previously defined category which this extends.
- The comments beginning with ++ are retained as documentation.
- The symbols "%" are ultimately replaced by the domain which belongs to the category. E.g. importing Boolean: Logic will give the operations such as xor: (Boolean, Boolean) -> Boolean.
- The keyword define makes the value of Logic public information. This is necessary in order to know what operations are exported when Logic is used. Normally only the name and type are public information.

Category producing function

- This definition gives a function which computes a category.
- The Join primitive constructs a category which is a subtype of all its arguments.

Domain producing function

```
MiniList(S: BasicType): LinearAggregate(S) == add {
        Rep == Union(nil: Pointer, rec: Record(first: S, rest: %));
        import from Rep, SingleInteger;
        cons (s:S, 1:%):%
                                == per(union [s, 1]);
        first(1: %): S
                                == rep(1).rec.first;
        rest (1: %): %
                                == rep(1).rec.rest;
        empty (): %
                                == per(union nil);
        empty?(1: %):Boolean
                                == rep(1) case nil;
        sample: %
                                == empty();
        [t: Tuple S]: % == {
                1 := empty();
                for i in length t..1 by -1 repeat
                         l := cons(element(t, i), l);
                1
        }
        [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);
                1
        }
        apply(1: %, i: SingleInteger): S == {
                while not empty? 1 and i > 1 repeat
                         (l, i) := (rest l, i-1);
                empty? l or i ~= 1 => error "No such element";
                first 1
        }
        (out: TextWriter) << (1: %): TextWriter == {</pre>
                empty? 1 => out << "[]";</pre>
                out << "[" << first 1;</pre>
                for s in rest l repeat out << ", " << s;</pre>
                out << "]"
        }
}
```

- An add expression constructs a domain object. The domain defines a number of exports, including empty?, generator, << and [_].
- Within the domain the types Rep and % are distinct, begin the representation and abstract types for values in the domain. The operations per: Rep -> % and rep: %-> Rep allow values to be viewed in either way.
- The constructor [_] taking the generator as an argument first extracts the elements from the generator (for s in g repeat ...) secondly reverses the list to place them in the correct order.
- The expression apply(a,b) may be written simply as a b or a.b so defining apply allows the selection notations 1.3. (Juxtaposition and dot associate as (f (g h)) and (f.g).h.)

Aldor

Conditional category definition

```
define SimplifiedComplex(R: Ring): Category == Ring with {
    if R has Field then Field;
    if R has DenseStorageCategory then DenseStorageCategory;
    complex: (R, R) -> %;
    *: (R, %) -> %;
    real: % -> R;
    imag: % -> R;
    %i: %;
}
```

 The if expressions in the category expression assert additional categories under particular conditions.

Post facto extension of a domain

```
extend Symbol: Ordered == add {
    import from String;
    (u:%) < (v:%): Boolean == printName(u) < printName(v);
    (u:%) > (v:%): Boolean == printName(u) > printName(v);
    (u:%) <= (v:%): Boolean == printName(u) <= printName(v);
    (u:%) >= (v:%): Boolean == printName(u) >= printName(v);
}
import from List Symbol;
```

sort [+"a",+"x",+"i",+"o",+"m"];

Add ordering operations to an existing domain for symbols.

- The syntax **+**"xxx" forms symbols.

A higher order program

Ag ==> (S: BasicType) -> LinearAggregate S;
<pre>swap(X:Ag,Y:Ag)(S:BasicType)(x:X Y S):Y X S==[[s for s in y] for y in x];</pre>
al: Array List Integer := array(list(i+j-1 for i in 13) for j in 13); print << "This is an array of lists: " << al << newline;
<pre>la: List Array Integer := swap(Array,List)(Integer)(al); print << "This is a list of arrays: " << la << newline;</pre>

 The swap function takes two aggregate type constructors as arguments and produces a new function to swap aggregate data structure layers.

An algebraic domain constructor

```
PolynomialCategory(R: Ring, E: AbelianMonoid): Category == Ring with {
        *:
                        (R, %) -> %;
        *:
                        (%, R) -> %;
        degree:
                       % -> E;
                        (R, E) -> %;
        monomial:
                       % -> %;
        reductum:
        leadingCoefficient: % -> R;
        coerce:
                       R −> %;
}
Polynomial(R: EuclideanDomain, Expon: OrderedAbelianGroup):
    PolynomialCategory(R, Expon)
== add {
   Rep == List Term(R,Expon);
    -- Some details omitted ...
   0: % == per nil;
    degree(x: %): Expon == {
        empty? rep x => 0;
        first(rep x).expon
    }
    leadingCoefficient(x: %): R == {
        empty?(1 := rep x) => 0;
        first(l).coef
    }
    (xx: %) + (yy: %): % == {
        (x, y) := (rep xx, rep yy);
        not x => yy;
        not y => xx;
        (x0,y0):= (first x,first y);
        y0.expon > x0.expon =>
                    per cons(first y,rep(xx + per rest y));
        x0.expon > y0.expon =>
                    per cons(first x,rep(per rest x + yy));
        r: R:= x0.coef + y0.coef;
        r = 0 \Rightarrow per rest x + per rest y;
        per cons([x0.expon,r],rep(per rest x + per rest y));
   }
}
```

Stephen Watt (London Ontario)

Aldor

References

- R.D. Jenks, R.S. Sutor, and S.M. Watt. Scratchpad ii: An abstract datatype system for mathematical computation. In J.R. Rice, editor, *Mathematical Ascpects of Scientific Software*, volume 14 of *IMA Volumes in Mathematics and Its Applications*, Berlin-Heidelberg-New York, 1988. Springer-Verlag.
- R.D. Jenks and B.M. Trager. A language for computational algebra. In P. S. Wang, editor, 1981 ACM Symposium on Symbolic and Algebraic Computation, pages 6–13, New York, 1981. Academic Press.
- S.M. Watt, P.A. Broadbery, S.S. Dooley, P. Iglio, J.M. Steinbach, and R.S. Sutor. A first report on the A[#] compiler. In M. Giesbrecht, editor, ISSAC '94, International Symposium on Symbolic and Algebraic Computation, Oxford, United Kingdom, pages 25–31, New York, 1994. ACM.
- Stephen M. Watt, Peter A. Broadbery, Samuel S. Dooley, Pietro Iglio, Scott C. Morrison, Jonathan M. Steinbach, and Robert S. Sutor. AXIOM Library Compiler User Guide. NAG Ltd, Oxford, 1994.

This article was processed using the $\mathbb{A}T_FX$ macro package with LMAMULT style