# Domains and Subdomains in Scratchpad II

by
Stephen M. Watt

The implementation of the Scratchpad II computer algebra system is based upon a compiled, strongly typed language. This language provides packages of polymorphic functions and parameterized, abstract datatypes with operator overloading and multiple inheritance. To express the intricate interrelationships between the datatypes necessary for the description of mathematical objects, several techniques based on the notion of *category* have been used.

Categories have been used to enforce relationships between type parameters and to provide the mechanism for multiple inheritance. They also allow the language to be statically type checked and the generation of efficient code. This article describes the role of categories in Scratchpad II. (Scratchpad II uses the word "category" consistently with the computer science terminology of algebraic specification, which is somewhat inconsistent with the mathematical terminology of category theory.) Please note that this article refers to some language features that are not yet available in publicly distributed versions of the system. The compiler supporting these deatures is scheduled for release in early 1988.

## *Domains*

In an environment where each operation may have many meanings, it is often useful to give a "domain of computation" to specify the operations of interest. A *domain* is a Scratchpad II object which provides a set of operations and attributes. Domains are used to implement datatypes and packages.

*In Scratchpad II every object belongs to a unique domain.* For example, 2 belongs to the domain Integer. Domains are first class run-time values which belong to the domain Domain. Objects belonging to a domain may be manipulated by operations it exports. For example, two of the operations Integer provides are

```
"+": (Integer, Integer) -> Integer
"=": (Integer, Integer) -> Boolean
```

The operations exported by a domain need not combine only members of the domain itself. The signatures of the exported operations may involve any types whatsoever. For example, the domain RationalNumber exports the operations

```
"/": (Integer, Integer) -> RationalNumber
characteristic: () -> NonNegativeInteger
```

Note that RationalNumber does not appear in the signature for ***characteristic***. A domain which does not export any operations for creating or manipulating members is a ***package***.

Domains may be created by functions, providing parameterized types and packages of polymorphic functions. As an example of a parameterized type, when Integer is passed to the function Stack the domain which we denote Stack(Integer) is returned.

The need for polymorphic functions stems from the desire to implement a given algorithm only once, and to be able to use the program for any values for which it makes sense. For example, the Euclidean algorithm for computing greatest common divisors can be used for values belonging to any type which is a Euclidean domain. The following package takes a Euclidean domain as a type parameter and exports the operations ***gcd*** and ***lcm*** on that type.

```
GCDpackage(R: EuclideanDomain): with
        gcd: (R, R) -> R
        lcm: (R, R) -> R
    == add

        -- Euclidean algorithm
        gcd(x,y) ==
           while y ~= 0 repeat (x,y) := (y,x rem y)
           normalize x

        lcm(x, y) ==
           (x exquo gcd(x,y))::R * y
```

The exported operations can equally well be used for many types, the integers or polynomials over GaloisField(7) being two examples. Although the same ***gcd*** program is used in both cases, the operations it uses (***rem***, ***normalize***, etc.) come from the type parameter R.

## *Subdomains*

*An object may belong to any number of subdomains.* The number 2, for example, belongs to many subdomains of Integer, including PositiveInteger, EvenInteger and SmallInteger. The domain Integer belongs to many subdomains of Domain, in-

cluding Monoid, AbelianGroup, Ring and Algebra(Integer).

A ***subdomain*** consists of

- a domain
- a boolean function that characterizes which members of the domain belong to the subdomain
- additional operations defined on the subdomain.

A variable may be declared to belong to a domain or to a subdomain. When a variable is declared to belong to a subdomain, its domain is that which the subdomain restricts. That is, values lying in a subdomain also lie in the domain and may be used in all the appropriate domain operations.

The subdomain may provide operations which supplement or supersede those of the domain but which are restricted to values lying in the subdomain. This restriction is for two reasons. First, operations which are closed on the domain may not be closed on the subdomain. Second, if a value has been determined to lie in a subdomain or is the result of a closed subdomain operation, then the subdomain may provide a more efficient implementation than the domain operation.

Often the subdomain predicate can be determined at compile-time. In places where this cannot be done, a run-time check may be inserted when a value must belong to a subdomain. Certain subdomains which are known to the compiler can be more highly optimized than others.

## *Categories*

While polymorphic packages allow the implementation of algorithms in a general way, it is necessary to ensure that these algorithms may *only* be used in meaningful contexts. It would *not* be meaningful to try to use GCDpackage above with Stack(Integer) as the parameter. To restrict the use to cases where it makes sense, Scratchpad II has the notion of categories.

A ***category*** in Scratchpad II is a restriction on the class of all domains. It specifies what operations a domain must support and certain properties the operations must satisfy. A category may be created using a category constructor such as the one below.

```
OrderedSet(): Category == Set with
   -- operations
      "<": ($,$) -> Boolean
      max: ($,$) -> $
      min: ($,$) -> $
   -- attributes
      irreflexive "<"    -- not (x < x)
      transitive  "<"    -- x < y and y < z
                         --   implies  x < z
      total "<"          -- not(x < y) and not(y < x)
                         --  implies x=y
```

OrderedSet gives a category which extends the category Set by requiring three additional operations and three properties, or *attributes*.

A declaration is necessary in order for a domain to belong to a given category: having the necessary operations and attributes does not suffice. This is because the attributes are not intended to give a complete set of axioms, but merely to make explicit certain facts that may be queried later. It is usually the case that belonging to a category implies that a domain must satisfy conditions that are not mentioned as attributes. For example, in OrderedSet there is no attribute relating ***min*** and " < ", although such relations are implied.

A type parameter to a domain or package constructor is usually required to belong to an appropriate category. For example, in the previous section the parameter R to GCDpackage was declared to belong to the category EuclideanDomain.

## *Separation of Contract and Implementation*

The use of categories in restricting the type parameters to a domain or package constructor allows algorithms to be specified in very general contexts. For example, since all table types belong to a common category, algorithms can be written that do not need to know the actual implementation (for example, whether it is a hash table in memory or a file on disk). As an example of algebraic generality, consider the domain of linear ordinary differential operators, which is declared as follows

```
LinearOrdinaryDifferentialOperator(A,M): T == B where
   A:  DifferentialRing
   M:  Module(A) with deriv: $ -> $

   T == GeneralPolynomialWithoutCommutativity(A,
     NonNegativeInteger) with
       D:        () -> $
       ".":      ($, M) -> M
   ...
   ...
```

This domain defines a ring of differential operators which act on an A-module, where A is a differential ring. The type of the coefficients, A, is declared to belong to the category DifferentialRing and type of the operands, M, is declared to belong to the category Module(A) with a derivative operation. The constructed domain of operators is declared to belong to a category of general polynomials with coefficients A and two additional operations. The operation D creates a differential operator and "." provides the method of applying operators to elements of M.

### *Multiple Views and Multiple Inheritance*

It is often necessary to view a given domain as belonging to different categories at different times. Sometimes we want to think of the domain Integer as a belonging to Ring, sometimes as belonging to OrderedSet, and at other times as belonging to other categories. For a domain to have multiple views, it should be declared to belong to the Join of the appropriate categories. For example, the following keyed access file datatype may be viewed either as a table or as a file:

```
KeyedAccessFile(Entry: Set): T == B where
    FileRec  ==> Record(key: String, entry: Entry)
    ErrorMsg ==> String

    T == Join(FileCategory(LibraryName, FileRec),
          TableCategory(String, Entry, ErrorMsg)) _
        with
            pack: $ -> $

    B == add ...
```

An important use of categories is to supply default implementations of operations. So long as certain primitive operations are provided by a domain, others can be implemented categorically. For example, supplying only "<" and "=" allows definitions of ">", "<=" and ">=". Thus a domain may inherit operations from a category. The use of Join provides multiple inheritance.

### *Compile-Time Binding*

A domain object contains several vectors of function/environment pairs. When operation bindings are not known at compile-time (as for the operations exported by a type parameter), the operations are performed by calling the function in the appropriate slot of a vector. When operation bindings are known at compile time, a code fragment for the operation may be placed in line.

The scope rules in Scratchpad II and the operations applicable to domain values have been designed so that when a domain is known to belong to a particular category, it is also known exactly what operations it exports. From this, the precise location of each function is determined. Thus when a function is called using the general mechanism, a hard-coded offset is used.

Scratchpad II is implemented on top of LISP/VM. Since the exact number and type of arguments are known at compile time, much of the usual function can be omitted. This, in conjunction with compile-time knowledge of function offsets, makes function calling in Scratchpad II faster than that of the underlying Lisp system.

## Construction of Algebraic Error Control Codes (ECC) on the Elliptic Riemann Surface

by
Martin Hassner
William H. Burge
Stephen M. Watt

In this paper we make use of the power series facility of Scratchpad II to construct algebraic ECC on the elliptic Riemann surface of genus one. We present the construction of a specific example that highlights the method.

Linear algebraic ECC are formally defined as ideals in a function field. A message word which consists of $k$ symbols in a ground field is mapped by a matrix encoder over the ground field into a code word which consists of $n$ symbols, $n > k$. The encoder matrix is the null space of a check matrix that constrains the symbols of the code word to be the residues at $n$ distinct singular points of first order of a function whose poles are a fixed set of pointers that locate the symbols inside each code word. An additional constraint imposed on each code function is its divisibility by a fixed function or linear system of functions whose zero set is disjoint from the pole set. A code consists of all the $n$-vectors of residues associated with such a system of functions which form an ideal. Within this algebraic framework it is possible to classify and determine (lower) bounds on the performance, i.e, efficiency vs. correction power, of algebraic ECC.