The Scratchpad II Type System: Domains and Subdomains

Stephen M. Watt
Richard D. Jenks
Robert S. Sutor
Barry M. Trager

Mathematical Sciences Department
IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598 USA

Abstract

Scratchpad II is a language developed at Yorktown Heights for the implementation of a new computer algebra system. The need to model the intricate relationships among the datatypes representing mathematical objects has provided a number of challenges in the design of a type system for the programming language.

In languages in which a datatype constructor may take multiple parameters, ensuring compatibility between them is extremely important. Scratchpad II addresses this issue by basing its implementation of abstract datatypes on *categories*. Categories provide a convenient and useful method for specifying requirements on operations from datatypes. These requirements can be very complex when modelling mathematics.

We show how categories provide multiple inheritance and how inheritance of specification is separated from inheritance of implementation. We also present implications of the type system on compilation of efficient code and flexibility of a weakly typed interactive user interface.

Finally, the mechanisms of Scratchpad II are compared with those of traditional abstract datatype and object-oriented programming languages.

1. Introduction

Computer algebra provides tools for the manipulation of symbolic mathematical expressions. It is a fascinating field, rich in problems in the structuring of data, analysis of algorithms and constructive mathematics [&COMALG.] [&SYMSAC86.] [&FRENCH.] [&KNUTH2.]. This paper is not about computer algebra, however. It is about certain general programming language issues which become increasingly important when writing software for nontrivial mathematical applications.

Scratchpad II is a language developed for the implementation of a new computer algebra system. It has been undergoing an evolution since 1979 and is by now fairly mature [&MODLISP.] [&KEYS.] [&APLPAP.] [&SNOWBIRD.]. A modern dialect of the Scratchpad II has been used to implement approximately of 50,000 lines of code, including a library of facilities for the manipulation of symbolic mathematical expressions, a compiler, and an underlying library of data structures.

From an early stage, Scratchpad II was intended to be used both as a vehicle for the implementation of mathematical algorithms in their most general natural contexts and as a publication language. These goals are closely related — in both cases it is necessary to specify precisely the range of applicability.

An important criterion for the language from the point of view of this paper was that the library of code for symbolic mathematics was to be structured according to the established formalisms of modern abstract algebra [&BOURBAKI.].

A similarity between mathematicians and computer scientists is that both both participate in the activities of generalization and abstraction. A difference between them is that mathematicians have been doing this for several hundred years. It is not surprising that by now the mathematical community has produced thousands of inter-related abstractions.

Scratchpad II is certainly not specific to mathematical applications. Considerable effort has been applied to keep specifics of the mathematical applications confined to the library. However, as with any other language, the requirements of the initial application area have made us consider certain language design aspects more than others. In the design of Scratchpad II we have been forced to examine issues relating to relationships among datatypes.

To be able to give formal specifications of the scope of algorithms, a requirement was that the language have mechanisms to be able to express the rich set of relationships among the standard abstractions of modern mathematics. This paper describes the language mechanisms developed for this purpose.

In sections 2 and 3, we begin by presenting the foundations of the type system. It is based on the primitive notions of *domain*

In section 4, we go on to define the *category* abstraction in terms of domains and subdomains. In section 5, we define *type* as synonymous to *subdomain* and discuss how types are used in the language.

In section 6, we indicate how categories provide various linguistic facilities. We describe how we obtain parameterized abstract datatypes,

polymorphic packages, multiple views and multiple inheritance. We show how categories are used to make assertions about and relations between domain parameters. Finally, we show how packages fit naturally into the model.

In section 7, we present certain implementation aspects of the type system. In particular, we show how categories are used to specify compile-compile time bindings for overload resolution.

In section 8, we illustrate how a weakly typed interactive interface is used in conjunction with the strongly typed compiled language. A number of interesting issues arise here, largely because the number of types that need to be considered is not finite.

Finally, in section 9, we compare the type system of Scratchpad II with that of other abstract data type languages and with object oriented systems.

2. Domain

In an environment where an operation symbol such as + may have many different meanings, it is useful to specify a "domain of computation", or, simply, "domain", to restrict attention to specific meanings of interest. A *domain* is a Scratchpad II object providing a set of executable operations,[1] often called the operations *exported* by the domain.

In Scratchpad II every value belongs to a unique domain. For example, **2** belongs to the domain **Integer**. Domains are themselves first class run-time values which belong to the domain **Domain**.

The operations of a domain are syntactically described by a *signature* which pairs an operation name with source and target domains (see section 5 for a precise definition). For example, two of the operations **Integer** provides are

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

Values belonging to a domain are generally manipulated by operations it exports. The operations exported by a domain however may manipulate any values 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 exports *no* operation for creating or manipulating members is a *package*.

Creating Domains

---

[1] An *operation* consists of a name (e.g. + or **gcd**) and a description of a mapping from one set of values to another.

Domains are run-time values created by functions[2] called *domain constructors*.

Domain constructors are often parameterized. For example, function Stack is a domain constructor with one formal argument parameter; when **Integer** is passed to the function **Stack** a domain which we denote **Stack(Integer)** is returned.

The operations of a domain are implemented by functions defined by the domain constructor. These functions are called *polymorphic* since they are defined over a class of domains. For example, the operation

```
push: (S,$) -> $
```

is exported by domain **Stack(S)** for any domain **S**. This operation is implemented by a single function regardless of the value **S**.

Stack is an example of how a domain constructor is used to create a datatype. Domain constructors may also be used to implement a package of polymorphic functions. Here there is 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. The following package takes a Euclidean domain as a domain parameter and exports the operations ***gcd*** and ***lcm*** on that domain.

```
GCDpackage(R: EuclideanDomain): with
        gcd: (R, R) -> R
        lcm: (R, R) -> R
    == add
        gcd(x,y) ==  -- Euclidean algorithm
           while y ¬= 0 repeat (x,y):= (y,x rem y)
           normalize x
        lcm(x, y) ==
           u: Union(R, "failed") := y exquo gcd(x,y)
           x * u::R
```

The exported operations here can be used equally well for many domains, 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 domain parameter R.

The above definition illustrates Scratchpad II as an "abstract datatype language" for defining domains. Everything before the first "==" is the public part of the definition of the constructor **GCDpackage**. This part describes its parameters (R) and exported operations (**gcd** and **lcm**). Everything after the first "==" is private information as to how the operations are implemented.

---

[2] A *function* is a program implementing an operation.

3. Subdomains

A value may belong to any number of subdomains. The number **2**, for example, belongs to many subdomains of **Integer**, including **NonNegativeInteger**, **EvenInteger**, and **PrimeInteger**.

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.

For convenience, we will regard any domain to be a subdomain of itself.

Values 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 three 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. Third, the operation may be defined only on the subdomain.

Often the subdomain predicate can be determined at compile-time. In places where this cannot be done, run-time checks may be required when values must belong to a subdomain. Certain subdomains known to the compiler and the checks can be more highly optimized than others.

Creating Subdomains

The simplest way to construct a subdomain is to specify a base domain and a subdomain condition:

```
NonNegativeInteger() == Subdomain(n ↦ (n >= 0),  Integer)
```

The signature of the binary subdomain constructor above is given by

```
Subdomain(condition: D -> Boolean, D: Domain): Subdomain(D)
```

That is, given a domain **D** and a mapping from **D** into **Boolean**, it produces a subdomain of **D**.

If additional operations are desired they are provided in a body similar to that of a domain:

```
EvenInteger(): Subdomain(Integer) with
        half: $ -> PositiveInteger
    == Subdomain(n ↦(n rem 2 = 0), Integer) add
        half(n) == n quo 2
```

A subdomain may be created as a refinement of another subdomain. In this case, the new subdomain has the same base domain as the first and

the conjunction of the predicates is used. We use the same notation as for creating a subdomain from a domain.

```
PositiveInteger() == Subdomain(n ↦ (n>0), NonNegativeInteger)
```

Finally, it is possible to construct a subdomain by combining the conditions of several other subdomains. To do this one uses the **Join** constructor. One could define

```
PositiveEvenInteger() == Join(PositiveInteger, EvenInteger)
```

4. Categories

A *category* in Scratchpad II is a restriction on the class of all domains. We formally identify **Category == Subdomain(Domain)**.

The domain **Integer** belongs to many subdomains of **Domain**, including **Monoid**, **AbelianGroup**, **Ring** and **Algebra(Integer)**.

A category may specify what operations a domain must support, properties the operations must satisfy, or other criteria. Categories may be parameterized by domains or other values and may guarantee additional properties, based on the values of the parameters.

Categories existed as primitives in Scratchpad II from a very early stage [&SNOWBIRD.] [&ALIST.]. Their use was inspired by algebraic specification [&ADJ.] and previous experiments in computer algebra. [&ANDANTE.]

More recently, the language has been simplified by redefining categories in terms of `Subdomain(Domain)`.

Creating Categories

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  -->  x < z
      total "<"        -- not(x < y) & not(y < x) --> x=y
```

OrderedSet gives a category which extends the category **Set** by requiring three additional operations ("<", ***max***, and ***min***) and three properties, or *attributes*. The attributes are not intended to give a complete set of axioms, but merely to make explicit certain facts that may be queried.

A parameterized category may be given in a similar fashion. When identities must hold between operations, it is often possible to give default implementations for certain ones in terms of the others.

The following example shows a parameterized category with conditional operations and a default implementation.

```
FiniteSetCat(S: Set): Category == Set with
        null:       $ -> Boolean
        union:      ($,$) -> $
        intersect:  ($,$) -> $
        difference:($,$) -> $
        member:     (S,$) -> Boolean

        if S has Finite then
            Finite with
                all: () -> $
                complement: $ -> $
              default
                complement s == difference(all(), s)
```

5. Types

In Scratchpad II, *type* is synonymous with *subdomain*. The type of a value is not unique: *the type of a value is any subdomain of its domain*. The number 2, for example, simultaneously has type Integer, NonNegativeInteger, PositiveInteger, PositiveEvenInteger, among other subdomains of Integer. Likewise, the domain Integer simultaneously has type OrderedSet, EuclideanDomain, Ring, among other subdomains of Domain.

Types form *hierarchies* as a result of subdomain definitions (Figure 1). Once a value is known to have some type, it immediately has the type of all of its ancestors up the hierarchy. For example, once a value is known to have type PositiveInteger, it also has type NonNegativeInteger and Integer. Likewise, once a domain has type Ring, it also has type SimpleRing, Module(Integer), AbelianGroup, and others.

Although types are values in Scratchpad II, their most important use is in "declarations" and signatures. A declaration has the form "x : T" for variable x and type T. This means that variable x must be bound to a value of type T. Thus the declaration

```
n : PositiveInteger
```

is used to only allow positive integers to be assigned to the variable n.

A *mapping* is a primitive type which pairs a "source" type with a "target" type using an infix operator "->". For example, the declaration

```
f : NonNegativeInteger -> Integer
```

requires that the value of variable f be a function which maps values of type NonNegativeInteger to those of type Integer. Given the above declarations, the application f(n) is type-correct.

The package form **GCDpackage(RationalNumber())** is likewise type-correct. By definition,

```
RationalNumber(): Join(Field,Algebra(Integer)) with ..
```

and so, RationalNumber has type Field, Algebra(Integer), and all of their ancestors. One of these ancestors is EuclideanDomain as required by the package constructor **GCDpackage**.

A *signature* is similar to a declaration except that it pairs the name of the operation with a type, usually a mapping. For example, the above declaration for RationalNumber could be written

```
RationalNumber: () -> Join(Field,Algebra(Integer)) with ..
```

Signatures are used to describe the exported operations from domain and category constructors. Since types in signatures are not required to be mappings, domains and categories may export constants as well.

The notions of subdomain for three primitive types: mappings, records, and unions are all system defined. A mapping M is defined to be a subdomain of M′, for example, if the source of M is a subdomain of the source of M′ and the target of M′ is a subdomain of the target of M.

An operation in a given type will *subsume* an operation in an ancestor type with the same name if the type in the operation for the given type is a subdomain of the corresponding type in the ancestor's operation. For example, category SemiGroup and Group respectively export the exponentiation operation

```
"**": (%,NonNegativeInteger)
"**": (%,Integer)
```

Since, Group is a subdomain of SemiGroup, the second operation subsumes the first.

6. Linguistic Uses of Types

Relations among Parameters and Results

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.

Algorithms rarely require values to be from a particular domain. Even algorithms such as *gcd* which are originally intended to work over the integers actually work over a more general classes of values. Categories serve as classifications of domains. In Scratchpad II, a user may implement may then choose the most general class of domains for which the any algorithm so as to be applicable to the most general class of values where the algorithm makes sense.

Consider for example a user wishes to implement the algorithms **quickSort**, **heapSort**, and **shellSort** for sorting an array of integers. Once these algorithms have been written, the implementer will notice that these algorithms apply to a wider class of domains. From a system classification of data structures, this class is determined to be LinearAggregate with the *shallowMutability* attribute. The attribute assures that datatypes have operations which allow component values to be reset "in place".

These sorting functions are then embedded in a package so as to be parameterized by the domains over which they are defined.

```
SortPackage(S:OrderedSet,
      V:(FiniteLinearAggregate(S) with shallowlyMutable)):
        with
          quickSort: (S,V) -> V
          heapSort:  (S,V) -> V
          shellSort: (S,V) -> V
  == add ...
```

The algorithm may now be used not only for arrays but lists, doubly linked lists, strings, flexible arrays, and others, whose elements are integers, rational or floating point numbers, or, given that ordering is lexicographic, essentially any datatype. Moreover, the definition of the algorithms which follow the **add** are essentially unchanged from the original versions restricted to arrays of integers.

Multiple Views

It is often necessary to view a given domain as belonging to different categories at different times. Sometimes we want to think of a vector of integers as a ring, sometimes as an ordered set, sometimes as a module over the integers. In Scratchpad II, this is equivalent to regarding the type of a value from **Vector(Integer)** as **Ring**, **OrderedSet**, or **Module(Integer)**. Likewise, it is often useful to view values in subdomains in different ways, e.g. a prime integer as an odd integer, a positive integer, or simply an integer.

A domain has multiple views whenever it is 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 == C where
    FileRec  ==> Record(key: String, entry: Entry)
    ErrorMsg ==> String

    T ==> Join(FileCategory(LibraryName, FileRec),
               TableCategory(String, Entry, ErrorMsg))

    C ==> add ...
```

Inheritance

When two sets of operations are **Joined**, identical operations coalesce and are assumed to have the same meaning. This implies all implementations of an operation in a domain and its subdomains have identical semantics, i.e. return equivalent values and produce equivalent side-effects.

A type may always define an operation it exports. In particular, an operation exported by a type may always be reimplemented and exported by one of its subdomains. Usually this is to make it more efficient. While **isPrime?** is exported by **Integer**, it may be redefined in **PrimeInteger**, to simply return **true**. Thus primality testing would be immediate within any algorithm expecting values from List(Integer) but receiving values from PrimeInteger instead.

An implementation of an operation in a type serves as a *default implementation* for all subdomains of that type. Thus a type need not provide an implementation for an operation if an ancestor does. In this case, the type is said to *inherit* the implementation from the ancestor. The + operation in PositiveInteger, for example, is inherited from Integer.

The use of **Join** provides a mechanism for multiple inheritance. When a type such as **PositiveEvenInteger** has two or more ancestors, the implementation is currently chosen to be the first offered by the components of the Join taken in left-to-right order. A default + for **PositiveEvenInteger** would be the same in each ancestor so that the order of components in the Join is irrelevant.

Categories as types provide default implementations for domains. So long as certain basic operations are provided by a domain, default implementations of others can be implemented categorically. For example, supplying only "<" allows definitions of ">", "<=" and ">=". Thus a domain may inherit operations from a category.

Abstract Hierarchies and Implementation Hierarchies

Categories are used to describe the specification part of an abstract datatype and might be said to constitute the *abstract* component of the system. Since categories are subdomains, they form a hierarchy described by a directed acylic graph with root note Domain. Each node of the graph is a category representing a class of domains. Any domain which is a member of a category at a given node is also a member of every category which is an ancestor of that node.

A domain and all of its subdomains form an *implementation hierarchy* described by a similar graph. Each node of the graph is a subdomain representing a class of values. Any value which is a member of a subdomain at a given node is also a member of every ancestor. In each of the two kinds of hierarchies, an implementation missing at a given node are inherited from a "nearest" ancestor node which supplies one.

Another kind of implementation hierarchy is formed by *add* chains. A QuotientField domain, for example, is defined as an *extension* of a LocalAlgebra domain:

```
QuotientField(D: IntegralDomain):
  Join(Field,Algebra(D),...) with ...
     == LocalAlgebra(D,D,D) add ...
```

Implementations of operations in a domain serve as default implementations for any extension of that domain.

The general inheritance of operations in Scratchpad II may be described as follows. The search for a default operation for a type, a subdomain of some domain, say, D, is in the following order:

1. up the implementation hierarchy from the subdomain to D,
2. up the add-chain for D,
3. up the abstract hierarchy of categories (types) of D.

Semantics of Types

A declaration is necessary in order for a domain to belong to a type: having the necessary operations and attributes does not suffice. This statement is obvious for domains. A category is more than a macro, that is, a name which stands for the list of exported operations and attributes. Its name implicitly implies *all* of the mathematical properties associated with the class of domains it denotes. For example, ordered rings are defined by

```
OrderedRing(): Category == Join(OrderedAbelianGroup,Ring)
```

Knowing that a domain is both an ordered abelian group and ring does not make it an ordered ring.

Stated another way, the system will not 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 the category **OrderedSet** there is no attribute relating ***min*** and "<", although such a relation is implicit.

A domain object contains a number of vectors of function/environment-pointer pairs. When operation values 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. In cases where operation values can be fixed 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 known statically that it exports certain operations. Compile-time operator overload resolution is performed based on this knowledge.

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 [&LISPVM.] running on IBM VM/SP operating system. 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.

8. The System Interpreter

In addition to the compiler and the constructor library, the Scratchpad II system contains an experimental user interface with an interpreter for interactively creating objects and applying functions from domains and packages. Although the user cannot define new category and domain constructors in the interpreter, the rest of the Scratchpad II language is supported. It is a design goal to provide an easy to use interface in the spirit of those found in other computer algebra systems such as Maple [&MAPLE.] and Macsyma [&MACSYMA.]. The interpreter relaxes the strong typing requirement of the compiler. The combination of the sophisticated type system of Scratchpad II and the desire to have few (if any) user declarations and explicit function package calls requires the interpreter to have extensive facilities for type inference and automatic coercion.

The basic analysis of expressions in the interpreter proceeds via a bottom-up pre-order traversal of an attributed tree created from parsed user input with some top-down information propagation. The interpreter reduces each successive subtree to a value by attempting to select and apply a function identified by an entry in an *operation database* created from the constructors in the system library.

Each entry in the operation database contains a signature, a predicate, and an implementation location expressed in terms of pattern variables *1, *2, etc.. These variables may represent domains or objects of domains. For example, an entry for a "less than" function might look like

$$< : (*1,*1) \rightarrow *2 \text{ from } *1$$

if *1 has **OrderedSet** and *2 is **Boolean**

This states that there is a binary homogeneous function "<" on objects of a domain (*1) belonging to the category **OrderedSet** and that the function is implemented in that domain and returns a **Boolean** result (*2). This entry refers to a function declared in a category; other entries may refer to functions from particular domains or packages.

In the Scratchpad II interpreter, a coercion is a conversion that may be performed whenever it is needed to satisfy a type requirement on a function argument or an assignment to a declared variable. Ideally, all coercions would be 1 to 1 mappings and thus be completely reversible and information preserving. In practice, other transformations are included among the embedding coercions. For example, integers may be coerced

to be elements of finite fields, though it is not possible to uniquely retract the images. Since coercions are data driven, their occurrence is predictable from the contents of the constructor library. Thus a "+" operation will be chosen from **Integer** rather than **GaloisField(2)** for the expression 3 + 4. Some conversions that are not coercions are those that pass from exact to inexact forms; there is no coercion from **RationalNumber** to **Float** (though there is a ***convert*** function).

In order for a database entry to be considered applicable, the operation name and arity must be the same as those of the expression subtree. Coercions may be necessary for actual arguments to match the types or predicate specified in the operation entry. The process of coercing objects may recursively involve the operation selection procedure. Whenever a pattern variable occurs more than once in an entry, the interpreter must compute a minimal coerceable type from those types of the actual arguments corresponding to the variable. For example, the minimal coerceable type computed from **RationalNumber** and **Polynomial(Integer)** is **Polynomial(RationalNumber)**.

Another kind of type computation is shown by the following example. Given a 2 by 2 square matrix of integers, the following operation entry is found when the ***inverse*** function is invoked with the matrix as argument:

inverse: *1→ Union(*1,"failed") from *1
if *1 is SquareMatrix(*2,*3) and
*2 : PositiveInteger and *3 has **Field**

A simple substitution might yield *2 = 2, *3 = **Integer** and so *1 = **SquareMatrix(2, Integer)**. This is not quite right though, as *3 must belong to the category **Field** and **Integer** does not. The interpreter computes that **RationalNumber** is the "minimal" type belonging to the category **Field** to which all objects of **Integer** can be coerced. Thus the correct substitutions are *3 = **RationalNumber** and *1 = **SquareMatrix(2, RationalNumber)**. The original matrix will need to be coerced before this particular ***inverse*** can be applied.

If multiple database entries are found to be applicable, they are ranked according to the cost of coercing the actual function arguments to the required types. When a final selection is made, the function is gotten by instantiating the completed constructor form indicated in the implementation location of the database entry. Further details of the interpreter type inference and coercion facilities are given in [&SIGPLAN87.].

9. Comparison with Other Abstract Datatype Languages

The difficulty of understanding, changing, and maintaining a program appears to grow exponentially with its size. Abstract datatype languages provide for the decomposition of a large program into a number of small "modules" which can be independently defined and tested.

Examples of abstract datatype languages are CLU, [&CLU.] Ada,[3] [&ADA.] Modula-2, [&MODULA2.] Russell, [&RUSSELL.] and Poly [&POLY..] An abstract datatype language characteristically separates the definition of a datatype in two parts: (1) a *specification part* which describes the set of operations exported by the datatype, and (2) an *implementation part* which defines functions to implement the exported operations. Users of the datatype may refer only to the specification part. The implementation part is private to the implementer of the datatype and may be reimplemented and replaced without semantic consequence. The datatype is called *abstract* since implementation-specific information is hidden from user-view.

In Scratchpad II, modules are called *domain constructors*, are generally parameterized, and are used to create *domains*, which may be either datatypes or *packages* of functions.

Like Russell, Poly, and an experimental version of CLU, [&CLUIMP.] datatypes in Scratchpad II are run-time objects. The type system of Scratchpad II was designed to model *domain towers* of mathematics, e.g. polynomials of matrices of gaussian integers. Many algorithms in computer algebra are most naturally expressed in terms of domain towers where some component domains are expressed categorically, e.g. polynomials over a field.

In most abstract datatype languages, the specification part of a module definition lists the operations exported by a module. The specification part of a Scratchpad II module is a category. Unlike other languages, specifications also include attributes. Attributes are used to assert important properties such as mathematical axioms and theorems which may be conditionally present and which may be queried by programs.

Abstract datatype languages also vary according to the extent to which restrictions can be placed on type parameters. Ada allows modules to be created without specifying semantic requirements on the parameters. Clu provides a mechanism for requiring type-valued parameters for type-constructors to export a certain set of operations. Scratchpad II type parameters may be required to belong to a category and/or to export a certain set of operations and attributes.

Scratchpad II appears to be unique in allowing the exported operations depend on the parameters of that module. This feature, while a practical necessity for computer algebra, is also useful in data structure definitions. For example, Polynomial(R) exports ***gcd*** if R is a Field, Polynomial(R) has a commutative homogeneous multiplication if R does, and List(S) exports a " < " function if S is an OrderedSet.

Bibliography

[1] Abdali, S. K., Cherry, G. W. and Soiffer, N., "A Smalltalk System for Algebraic Manipulation," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, Volume 21, Number 11, New York: Association for Computing Machinery, November 1986, pp. 277-283.

[3] Ada is a trademark of the United States Department of Defense.

[2] Atkinson, R., Liskov, B. H. and Scheifler, Robert W., "Aspects of Implementing CLU," ??, 1978

[3] Bobrow, D.G., Kahn, K., Kiczales, G., Masinter, L., Stefik, M. and Zdybel, F., "CommonLoops: Merging Common Lisp and Object-oriented programming," Intelligent Systems Laboratory Series ISL-85-8 (Xerox Palo Alto Research Center: August 1985).

[4] Bobrow, D.G. and Stefik, M., "The Loops Manual," Technical Report KB-VLSI-81-13, Knowledge Systems Area, (Xerox Palo Alto Research Center: 1981).

[5] Boehm, Hans-J., and Demers, A., "Implementing Russel," Proceedings of the SIGPLAN '86 Symposium on Compiler Construction, ACM SIGPLAN Notices, Vol 21 No 7, July 1986.

[6] Boehm, Hans-J., Demers, A. and Donahue, J., "A Programmer's Introduction to Russell," Technical Report 85-16, Department of Computer Science, Rice University.

[7] Bourbaki, N., Algebre, Paris: Hermann, 1950.

[8] Buchberger, B., Collins, G. E. and Loos, R., editors, Computer Algebra: Symbolic and Algebraic Computation, Second Edition, New York: Springer-Verlag, 1982.

[9] Burge, W. H. and Watt, S. M., "Infinite Structures in Scratchpad II," Proceedings of the 1987 European Conference on Computer Algebra (EUROCAL 87), June 2-5, 1987, Leipzig German Democratic Republic, to appear (Springer Verlag). IBM Research Report RC 12794.

[10] Cardelli, L., "Amber," AT&T Bell Laboratories Technical Memorandum, 11271-840924-10TM, 1984.

[11] Char, B. W.,Geddes, K. O., Gonnet, G. H. and Watt, S. M., Maple User's Guide, Waterloo, Ontario: WATCOM Publications Limited, 1985.

[12] Char, B. W., editor, Proceedings of the 1986 ACM Conference on Symbolic and Algebraic Computation, July 21-23, 1986.

[13] Davenport, J. H., "A New Algebra System," unpublished, c. 1982.

[14] Davenport, J. H., Siret, Y. and Tournier, E., Calcul Formel, Masson, Paris, 1987.

[15] Foderaro, J., Newspeak, Ph.D. Thesis, University of California, Berkeley, 1983.

[16] Futatsugi, K., Goguen, J. A., Jouannaud, J.-P. and Meseguer, J., "Principles of OBJ2," Proceedings of the 12th ACM Symposium on Principles of Programming Languages, New York: Association for Computing Machinery, 1985.

[17] Goldberg, A. and Robson, D., Smalltalk-80: The Language and Its Implementation, Addison-Wesley, Reading, Massachusetts, 1983.

[18] Goguen, J.A., Thatcher, J.W. and Wagner, E., "An Initial Algebra Approach to the Specification, Correctness and Implementation of Abstract Data Types," Current Trends in Programming Methodology IV (1978), 80-149.

[19] Gries, D, and Prins, J., "A New Notion of Encapsulation," SIGPLAN 85 Symposium on Language Issues in Programming Environments, ACM SIGPLAN Notices, Vol 20 No 7, July 1985.

[20] Jenks, R. D., "MODLISP: An Introduction," Proceedings of EUROSAM 79 (Springer-Verlag Lecture Notes in Computer Science 72), pp. 466 - 480 (also available in revised form as: "MODLISP: A Preliminary Design," IBM Research Report RC 8073, January 18, 1980).

[21] Jenks, R. D., "A Primer: 11 Keys to New Scratchpad," Proceedings of EUROSAM '84, 1984 International Symposium on Symbolic and Algebraic Computation, Cambridge, England, July 1984

[22] Jenks, R. D., Sutor, R. S. and Watt, S. M., "Scratchpad II: An Abstract Datatype System for Mathematical Computation," IBM Research Report RC 12327 (Yorktown Heights, New York: November 17, 1986).

[23] Jenks, R. D. and Trager, B. M., "A Language for Computational Algebra," Proceedings of SYMSAC '81, 1981 Symposium on Symbolic and Algebraic Manipulation, Snowbird, Utah, August, 1981.

Also ACM SIGPLAN Notices, November 1981.

[24] Knuth, D. E., The Art of Computer Programming Volume 2, Second Edition (Chapter 4), Addison Wesley, 1981.

[25] Liskov, B., Atkinson, R., et al. CLU Reference Manual, New York: Springer-Verlag, 1981.

[26] LISP/VM User's Guide, First Edition, Houston: IBM Corporation, July, 1984.

[27] Matthews, D. C. J., "Introduction to Poly," University of Cambridge Computer Laboratory, Technical Report No. 29, 1982.

[28] Milner, R., "A proposal for standard ML," Proceedings of the 1984 ACM Symposium on Lisp and Functional Programming, New York: Association for Computing Machinery, 1984.

[29] Moon, D. A., "Object-Oriented Programming with Flavors," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, Volume 21, Number 11, New York: Association for Computing Machinery, November 1986, pp. 1-8.

[30] Schaffert, C., Cooper, T., et al. "An Introduction to Trellis/Owl," OOPSLA '86 Conference Proceedings, SIGPLAN Notices, Volume 21, Number 11, New York: Association for Computing Machinery, November 1986, pp. 9-16.

[31] Snyder, A., "Object-Oriented Programming for Common Lisp" Report ATC-85-1, Software Technology Laboratory, Hewlett-Packard Laboratories, Palo Alto, California, 1985

[32] Snyder, A., "Encapsulation and Inheritance in Object-Oriented Programming Languages" Proceedings of OOPSLA '86, pp. 38-45

[33] Snyder, A, Creech, M. and Kempf, J., "A Common Lisp Objects Implementation Kernel," Report STL-85-08, Software Technology Laboratory, Palo Alto, California: Hewlitt-Packard Laboratories, 1985.

[34] Soiffer, N., "A Perplexed User's Guide to ANDANTE," unpublished, 1981.

[35] Stefik, M. and Bobrow, D. G., "Object-Oriented Programming: Themes and Variations" The AI Magazine, June 1987, pp. 40-62

[36] Stroustrup, B., "What is "Object-Oriented Programming"?," Proceedings ECOOP, Paris, June, 1987

[37] Sutor, R. S. and Jenks, R. D., "The Type Inference and Coercion Facilities in the Scratchpad II Interpreter," Proceedings of the SIGPLAN '87 Symposium on Interpreters and Interpretive Techniques, SIGPLAN Notices 22, 7, pp. 56-63, New York: Association for Computing Machinery, July 1987. IBM Research Report RC 12595.

[38] Swinehart, D.C., Zellweger, P.T., Beach, R.J. and Hagmann, R.B., "A Structural View of the Cedar Programming Environment," ACM TOPLAS V? No. 4., New York: Association for Computing Machinery, October, 1986.

[39] United States Department of Defense, Ada Reference Manual, ANSI/MIS-STD 1815 (Jan.), United States Government Printing Office, 1983.

[40] VAX UNIX MACSYMA Reference Manual, Version 11, Symbolics, Inc., November, 1985.

[41] Wirth, N., "Programming in Modula-2", Third, Corrected Edition, Texts and Monographs in Computer Science, Edited by David Gries, Springer-Verlag, Berlin, Heidelberg, 1985

[42] Zippel, R., "Capsules," Proceedings of the SIGPLAN '83 Symposium on Programming Language Issues in Software Systems, ACM SIGPLAN Notices, Vol 18 No 6, June 1983.