RC 12327 (#55257) 11/17/86
Computer Science 23 pages

Scratchpad II: An Abstract Datatype System for Mathematical Computation

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

IBM Thomas J. Watson Research Center
Yorktown Heights, NY 10598

Abstract:
Scratchpad II is an abstract datatype language and system that is under development in the Computer Algebra Group, Mathematical Sciences Department, at the IBM Thomas J. Watson Research Center. Some features of *APL* that made computation particularly elegant have been borrowed. Many different kinds of computational objects and data structures are provided. Facilities for computation include symbolic integration, differentiation, factorization, solution of equations and linear algebra. Code economy and modularity is achieved by having polymorphic packages of functions that may create datatypes. The use of categories makes these facilities as general as possible.

1. Introduction

Scratchpad II is

- an interactive language and system for mathematical computation
- a strongly-typed programming language for the formal description of algorithms, and
- a sophisticated tool kit for building libraries of interrelated abstract datatypes.

As an interactive system, Scratchpad II is designed to be used both by a naive user as a sophisticated desk-calculator and by an expert to perform sophisticated mathematical computations. Scratchpad II has very general capabilities for integration, differentiation, and solution of equations. In addition, it has an interactive programming capability which allows users to easily create new facilities or access those resident in the Scratchpad II library.

Scratchpad II is also a general-purpose programming language with a compiler used to add facilities to the system or user's library. Library programs are read by the system compiler, converted into object code, then loaded and executed through use of the system interpreter. The programming language and interactive language are identical except that library programs must be strongly typed. The unique abstract datatype design of Scratchpad II is based on the notion of *categories* and allows polymorphic algorithms to be expressed in their most natural setting and independently of the choice of data representation.

The Scratchpad II library consists of a set of parameterized modules (abstract datatypes) which collectively serve as a tool kit to build new facilities. Among these modules are those which create computational "types" (such as integers, polynomials, matrices and partial fractions) or data structures (such as lists, sets, strings, symbol tables, and balanced binary trees). These modules can be used to dynamically "mix and match" types to create *any* computational domain of choice, e.g. matrices of matrices, or matrices of polynomials with matrix coefficients.

In contrast with Scratchpad II, other existing computer algebra systems, such as MACSYMA, MAPLE, REDUCE and SMP use but a few internal representations to represent computational objects. To handle complicated objects, some of these systems overload the data structure for a canonical form (such as rational functions) and use flags to govern which coefficient and/or exponent domain is to be used. As more and more overloading is done to a single internal representation, programs become increasingly error prone and unmanageable. The complexity of systems designed in this way tend to grow exponentially with the number of extensions. The design approach of Scratchpad II has considerable advantages relative to these other systems with respect to modularity, extensibility, generality, and maintainability.

This paper introduces the reader to the language and concepts of Scratchpad II in a "bottom-up" manner, illustrating some interesting and varied interactive computations. Sections 1-8 of the paper systematically introduce the reader to some of the more interesting types in the Scratchpad II world. Sections 9-11 highlight the facilities of the computer algebra library. Sections 12-15 then discuss the underlying high-level concepts of the language and system.

2. Some Comparisons with APL

An interactive session with Scratchpad II resembles that of one with *APL*. The Scratchpad II interpreter reads input expressions from the user, evaluates the expression, then display a result back to the user. Input and output lines are numbered and saved in a history file. System commands to perform utilities such as reading files, editing, etc. are preceded by ")". Everything after "--" is a comment.

The Scratchpad II language, however, is very different from *APL*: it uses a more standard character set, has operator precedence rules, and offers control structures for structured programming. Whereas a dominant theme of *APL* is arrays, that of Scratchpad II is types.

The following produces the same result as `(5**2)+4`.

```
5**2 + 4

   (1)  29
```

The previously computed expression is always available as the variable named %.

```
% + 1

   (2)  30
```

Unlike *APL*, large integer computations remain exact.

```
2**1000

   (3)
  107150860718626732094842504906000181056140481170553360744375038837035105
  112493612249319837881569585812759467291755314682518714528569231404359845
  775746985748039345677748242309854210746050623711418779541821530464749835
  819412673987675591655439460770629145711964776865421676604298316526243868
  37205668069376
```

Floating point numbers can be allowed to have many digits. Here is π to 200 places.

```
precision 200

   (4)  200


numeric %pi

   (5)
  3.141 59265 35897 93238 46264 33832 79502 88419 71693 99375 10582 09749
  44592 30781 64062 86208 99862 80348 25342 11706 79821 48086 51328 23066
  47093 84460 95505 82231 72535 94081 28481 11745 02841 02701 93852 11055
  59644 62294 89549 30382
```

Symbols may be referenced before they are given values. It is easy to substitute something for the symbol at a later time.

```
(x + 11/111)**5

         5    55   4    1210   3     13310   2      73205          161051
   (6)  x  + (---)x  + (-----)x  + (-------)x  + (---------)x + -----------
             111       12321       1367631       151807041     16850581551


eval(%, x, 10)

        1770223341829601
   (7)  ----------------
          16850581551
```

In contrast to *APL* where arrays are used to contain collections of values, most Scratchpad II users employ lists as the "standard" aggregate datatype and other aggregate types are available. Many of the operations on *APL* arrays are supported by Scratchpad II lists, which are described in section 5.

3. Numbers

Scratchpad II provides many different kinds of numbers. Where appropriate, these can be combined in the same computation because the system knows how to convert between them automatically.

Integers can be as large as desired with the only limitation being the total storage available. They remain exact, no matter how large they get. Rational numbers are quotients of integers. Cancellation between numerators and denominators will occur automatically.

```
11**13 * 13**11 * 17**7 - 19**5 * 23**3 * 29**2

   (1)  25387751112538918594640918059149578
```

```
1/2 + 1/6 + 1/24 + 1/720 + 1/5040
```

$$(2)\quad \frac{1789}{2520}$$

For approximations, floating point calculations can be performed with any desired number of digits. The function ***precision*** sets the number of digits to use.

```
precision 39
```

(3) 39

A smaller precision might have given the impression that the following expression evaluated to 12. (Ramanujan wondered if it was actually an integer.)

```
numeric %pi * sqrt 310. / _    -- continued on next line
   log((2+sqrt 2.) * (3+sqrt 5.) * (5+2*sqrt 10.+sqrt(61+20*sqrt 10.))/4)
```

(4) 12.00 00000 00000 00000 00000 04945 80712 26995

Gaussian integers are complex numbers where both the real and imaginary parts are integers.

```
(5 + %i)**3
```

(5) 110 + 74%i

Of course, not all complex numbers have integer real and imaginary parts. The following number has floating point components.

```
(2.001 - 0.001 * %i)**2
```

(6) 4.004 - 0.004 002%i

Sometimes the form of a number is as important as the type of number. Here are a few ways of looking at integers and rationals in different forms.

```
factor 643238070748569023720594412551704344145570763243
```

$$(7)\quad 11^{13}\ 13^{11}\ 17^{7}\ 19^{5}\ 23^{3}\ 29^{2}$$

```
continuedFraction(6543/210)
```

$$(8)\quad 31 + \cfrac{1}{6 + \cfrac{1}{2 + \cfrac{1}{1 + \cfrac{1}{3}}}}$$

```
partialFraction(1,factorial(10))
```

$$(9)\quad \frac{159}{2^8} - \frac{23}{3^4} - \frac{12}{5^2} + \frac{1}{7}$$

```
-- now we expand the numerators into p-adic sums of the primes in the denominators

padicFraction %
```

$$(10)\quad \frac{1}{2} + \frac{1}{2^4} + \frac{1}{5^2} + \frac{1}{2^6} + \frac{1}{2^7} + \frac{1}{2^8} + \frac{2}{3^2} - \frac{1}{3^3} - \frac{2}{3^4} - \frac{2}{5^5} - \frac{2}{5^2} + \frac{1}{7}$$

We can also view rational numbers as radix expansions using various bases. Repeating sequences of digits are indicated by a horizontal line.

```
decimal(1/352)

   (11)  0.0028409

base(4/7, 8)

   (12)  0.4
```

Rational numbers raised to fractional powers can easily be created and manipulated.

```
(5 + sqrt 63 + sqrt 847)**(1/3)

                  1/2     1/3
   (13)  (14*7    + 5)
```

Integers modulo a given integer may be conveniently created and used.

```
123 mod 11     -- create an integer mod 11

   (14)  2

% + 79         -- operations involving this value are now done mod 11

   (15)  4
```

The following asserts that a is a number satisfying the equation a**5+a**3+a**2+3 = 0.

```
a | a**5+a**3+a**2+3 = 0
```

Among other things, this relationship implies that any expression involving a will never have it appear raised to a power greater than 4. We will define b so that it satisfies an equation involving a.

```
b | b**4+a = 0

2/(b-1)                    -- compute 2 times the inverse of (b-1)

   (18)
      4    3    2          3    4    3    2          2
    -(a  - a  + 2a  - a + 1)b  + (a  - a  + 2a  - a + 1)b
   +
      4    3    2             4    3    2
     (a  - a  + 2a  - a + 1)b + a  - a  + 2a  - a + 1

2/%+1                      -- check result

   (19)  b
```

There are many other varieties of numbers available, including cardinal numbers, which need not be finite, and quaternions, which are non-commutative.

```
Aleph 1 + Aleph 0

   (20)  Aleph(1)

quatern(1,2,3,4)*quatern(5,6,7,8) - quatern(5,6,7,8)*quatern(1,2,3,4)

   (21)  - 8i + 16j - 8k
```

Abbreviation	*Full Name*
A	Any
B	Boolean
BF	BigFloat
COMBINAT	CombinatoricFunctions
E	Expression
G	Gaussian
GF	GaloisField
I	Integer
L	List
P	Polynomial
QUEUE	Queue
RF	RationalFunction
RN	RationalNumber
S	String
SM	SquareMatrix
STK	Stack
ST	Stream
SY	Symbol
TBL	Table
UPS	UnivariatePowerSeries

Figure 1. Some Scratchpad II Type Names and their Abbreviations

4. Types

Every Scratchpad II object has an associated datatype, which can be thought of as a property of the object. The datatype determines the operations that are applicable to the object and cleanly separates the various kinds of objects in the system. If the user has issued

```
)set message type on
```

or, at least, has not turned it off,[1] the datatype of an object is printed on a line following the object itself. For example, if you enter `3.14159`, the system will respond with a display similar to

```
   (1)  3.14159

Type: BF
```

In the Scratchpad II interpreter, BF is the abbreviation for BigFloat, which is the datatype of the number you entered. If you had not known anything about BF, issuing the command

```
)show BF
```

would have told you the unabbreviated name, the name of the file containing the Scratchpad II source code for BigFloat and the functions provided in the BigFloat *domain*.[2]

In the interpreter, each type has an abbreviation and it may be used almost anywhere the full name is used. Some of the abbreviations that are used in this paper are listed in Figure 1.

In the previous section, each of the numbers really had a type, even though we chose not to display it. Some were simple, like Integer and BigFloat, and some were parametrized, like Gaussian In-

[1] By default, it is on.

[2] You can think of a *domain* as a collection of objects with a set of functions defined on the objects, plus a set of *attributes* that assert facts about the objects or the functions. For example, the domain Integer provides the integers, the usual functions on integers, and attributes asserting that multiplication is commutative, 1 is a multiplicative identity element, etc..

teger and ContinuedFraction Integer. Some of the types were fairly complicated, like SimpleAlgebraicExtension(RationalNumber, UnivariatePoly(x,RN), a**5+a**3+a**2+3). At no point did we actually have to tell Scratchpad II the types of the objects we were manipulating. Although it is true that usually the Scratchpad II interpreter can determine a suitable type for an object without any type declarations whatsoever, you may sometimes want to supply additional information. You might provide this to help guide the interpreter to a particular type choice among several or to view an object in a particular way.

It is useful to know about types because:

1. Scratchpad II really does use datatypes and they are present no matter how simple a model of the interpreter is discussed.
2. Types are Scratchpad II objects in their own right and information is associated with them. A knowledge of types allows you to access and use this information.
3. The use of explicit coercions with types provide a powerful way to transform an expression, be it to simplify the expression, change the output form, or to apply a particular function.

When you enter an expression in the Scratchpad II interpreter, the type inference facility attempts to determine the datatypes of the objects in the expression and to find the functions you have used. The following dialog demonstrates the types assigned by the interpreter to some simple objects.

```
23                  -- this is Integer

   (1)  23

Type: I

3.45                -- this is BigFloat

   (2)  3.45

Type: BF

"this is a string"  -- this is String

   (3)  "this is a string"

Type: S

false               -- this is Boolean

   (4)  false

Type: B

x                   -- this is Symbol

   (5)  x

Type: SY
```

The above expressions are *atomic*: they involve no function calls. When functions are present, things can get a bit trickier. For example, consider 2 / 3. By the basic analysis above, the interpreter determines that 2 and 3 belong to Integer. There is no function "/" in Integer so the interpreter has to look elsewhere for an applicable function. Among the possibilities are a "/" in RationalNumber that takes two elements of Integer and returns an element of RationalNumber. Since this involves no work in converting the arguments to anything else, this function is called and the rational number 2/3 is returned. This all happens automatically and is relatively transparent to the user.[3]

Associated with each type is a representation, a specific form for storing objects of the type. This representation is private and cannot be determined without examining the program which implements the type.

[3] Some loading messages may appear from time to time as the system tries to coerce objects from one type to another or starts applying functions.

Some types, like Integer, are considered basic and have their representations provided internally by the system. Others, like RationalNumber, are built from other types (Record and Integer, here). Once a type is defined it may be used to represent other types. For example, QuotientField is represented by using Record and the type of the numerator and denominator. RationalFunction is represented by QuotientField Polynomial, along with the type of the coefficients of the polynomials. However, we re-emphasize that these details cannot be seen by users or other programs that manipulate values of these types.

Scratchpad II now provides over 160 different datatypes. Some of these clearly pertain to algebraic computational objects while others, like List and SymbolTable are data structures. Although Scratchpad II was originally designed as an abstract datatype language for computer algebra, no distinction is made to treat mathematical structures differently than data structures. In fact, data structures usually satisfy certain axioms and have mathematical properties of their own.

Scratchpad II is actually a general purpose language and environment: the new compiler for the language is being written in the language itself!

5. Lists

Lists are the simplest aggregate objects in Scratchpad II.

```
u := [1,4,3,5,3,6]
   (1)  [1,4,3,5,3,6]

rotate(u,2)
   (2)  [3,5,3,6,1,4]
```

Lists do not have to be homogeneous

```
u := [-43,"hi, there", 3.14]
   (3)  [- 43,"hi, there",3.14]
```

and they may be ragged.

```
v := [[1], [1,2,3], [1,2]]
   (4)  [[1],[1,2,3],[1,2]]
```

A monadic colon is used to append lists.

```
w := [:u, :[1..5],:u]   -- [1..5] is the list [1,2,3,4,5]
   (5)  [- 43,"hi, there",3.14,1,2,3,4,5,- 43,"hi, there",3.14]
```

Lists have origin 0. A "dot" is usually used to indicate indexing.

```
w.0
   (6)  - 43
```

Reduction over a list by a binary operator is supported.

```
*/[1..100]              -- this is 100 factorial
   (7)
933262154439441526816992388562667004907159682643816214685929638952175999
932299156089414639761565182862536979208272237582511852109168640000000000
00000000000000
```

A function may be applied to each element of a list by using "!".

```
oddp ! [1..5]          -- oddp returns true for an odd integer agrument

   (8) [true, false, true, false, true]

![1..5] + ![10..14]

   (9) [11, 13, 15, 17, 19]
```

A list may be viewed as a mapping which takes integers and returns the elements. The following list is then seen as the mapping

$$0 \rightarrow 1,\ 1 \rightarrow 1,\ 2 \rightarrow 2,\ ...,\ 7 \rightarrow 21.$$

```
u := [1,1..3,5,8,13,21]

   (10)  [1,1,2,3,5,8,13,21]
```

Juxtaposition with an intervening blank is equivalent to dyadic "." and means application. Parentheses are used for grouping. For lists, all three notations mean to apply the list as a mapping.

```
[u(0),u 1,u.2]

   (11)  [1,1,2]
```

A "!" can be used to apply *any* mapping to each element of a list.

```
u ! [0,1,3,5,7]

   (12)  [1,1,3,8,21]
```

Lists may created in many different ways. The following creates a list of the squares of the odd elements in *u*.

```
[n**2 for n in u | oddp n]

   (13)  [1,1,9,25,169,441]
```

A variety of very general iterator controls are available. Besides the "such that" form above, Scratchpad II also provides *while* and *until* forms. Iterations may also be nested or performed in parallel.

We now define a function ***fib*** to compute the Fibonacci numbers. The definition will be incrementally built from several separate pieces.

```
fib 0 == 1  -- the first initial value
fib 1 == 1  -- the second initial value
fib         -- looks at fib's value now as a mapping: 0 -> 1, 1 -> 1

   (16)  [1,1]
```

The general term will give a recursive definition for the remaining arguments of interest.

```
fib n==fib (n-1) + fib (n-2) when n > 1

fib         --look at its entire definition as a mapping

   (18)  [(n | 1 < n) -> fib(n - 1) + fib(n - 2),0 -> 1,1 -> 1]
```

The first term in the above mapping means if ***fib*** is given an argument `n` which is greater than 1, then `fib(n)` is computed using the recursive form. Now we will actually apply our function.

```
fib ! [0,1,3,5,7]  --apply fib to each integer in our list of values
   compiling fib as a recurrence relation

   (19)  [1,1,3,8,21]
```

Note that we were able to determine that a recurrence relation was involved and specially compile the function.

6. Infinite Objects

Scratchpad II provides several kinds of infinite objects. We have already seen the example of a repeated decimal expansion of a rational number above. Other examples of infinite objects are streams and power series.

Streams are generalizations of lists which allow an infinite number of elements. Operationally, streams are much like lists. You can extract elements from them, use "!", and iterate over them in same way you do with lists.

There is one main difference between a list and stream: whereas all elements of a list are computed immediately, those of a stream are generally only computed on demand. Initially a user-determined number of elements of a stream are automatically calculated. This number is controlled by a *)set* user command and is 10 by default. Except for these initial values, an element of a stream will not be calculated until you ask for it.

The expression `[n..]` denotes the (primitive) stream of successive integers beginning with `n`. To see the infinite sequence of Fibonacci numbers, we apply ***fib*** to each member of `[0..]`, the primitive stream of nonnegative integers.

```
fibs==fib![0..]
fibs              --by default, 10 values of a stream are computed

   (21)  [1,1,2,3,5,8,13,21,34,55,...]
```

Streams, like lists, are applicable as mappings and can be iterated over.

```
fibs ! [0,1,3,5,7]

   (22)  [1,1,3,8,21]

[n for n in fibs | oddp n]

   (23)  [1,1,3,5,13,21,55,89,233,377,...]

oddOnes s== [n for n in s | oddp n]  --define a function to do the filtering
oddFibs == oddOnes fibs              --define a new stream from the old

3*!oddFibs -1                        --produce [3*n-1 for n in oddFibs]

   (26)  [2,3,9,15,39,63,165,267,699,1131,...]

%![2*i for i in 1..]                 --can apply streams to streams

   (27)  [9,39,165,699,2961,12543,53133,225075,953433,4038807,...]
```

A power series can be obtained from a stream by coercing it to type UPS.

```
fibs::UPS(x,I)     --convert a stream to a power series

   (28)
                2     3     4     5      6      7      8      9      10
     1 + x + 2x  + 3x  + 5x  + 8x  + 13x  + 21x  + 34x  + 55x  + 89x
   +
         11
     O(x  )
```

Another way to generate this power series is as follows:

```
1/ps(1-x-x**2)

   (29)
                2     3     4     5     6     7     8     9     10
     1 + x + 2x  + 3x  + 5x  + 8x  + 13x  + 21x  + 34x  + 55x  + 89x
   +
        11
     0(x  )

sin %    --the composition of one power series with another

   (30)
           2    17  3     4    541  5    13  6    15331  7    3713  8
     x + 2x  + (--)x  + 4x  + (---)x  + (--)x  - (-----)x  - (----)x
               6              120       4        5040        180
   +
        22536359  9    3046931  10      11
     - (--------)x  - (-------)x   + 0(x  )
          362880         20160
```

Power series can have coefficients from any ring, e.g. rational functions, gaussians, even other power series. Assuming m denotes a 2 x 2 square matrix with values 1,1,1,0, the following illustrates a power series with matrix coefficients.

```
1/ps(1-m*x)

   (31)
     |1  0|   |1  1|     |2  1| 2   |3  2| 3   |5  3| 4   |8  5| 5
     |   | + |   |x + |   |x  + |   |x  + |   |x  + |   |x
     |0  1|   |1  0|     |1  1|     |2  1|     |3  2|     |5  3|
   +
     |13  8| 6   |21  13| 7   |34  21| 8   |55  34| 9   |89  55| 10      11
     |     |x  + |      |x  + |      |x  + |      |x  + |      |x   + 0(x  )
     |8   5|     |13  8 |     |21  13|     |34  21|     |55  34|

%::ST SM(2,I)    --obtain the coefficients of the power series as a stream

   (32)
     |1  0|  |1  1|  |2  1|  |3  2|  |5  3|  |8  5|  |13  8|  |21  13|
   [ |   |, |   |, |   |, |   |, |   |, |   |, |    |, |    |,
     |0  1|  |1  0|  |1  1|  |2  1|  |3  2|  |5  3|  |8   5|  |13  8 |

     |34  21|  |55  34|  |89  55|
     |      |, |      |, |      |, ...]
     |21  13|  |34  21|  |55  34|

trace!%   --obtain a Fibonacci sequence, but with different initial conditions

   (33)  [2,1,3,4,7,11,18,29,47,76,...]
```

7. Functions

Functions can be as important as the values on which they act. In Scratchpad II functions are treated as first class objects; function-valued variables can be used in any way that variables of other types may be used.

Functions may be defined at top level, as were the maps from the previous section, or they may be obtained from a library of compiled code, as are the operations provided by types.

The simplest thing that can be done with a function object is to apply it to arguments to obtain a value.

```
5 + 6

   (1)  11

Type: I
```

If there are several functions with the same name, the interpreter will choose one of them. An attempt is made to choose the function according to certain generality criteria.

When a particular function is wanted, the plus on GF(7) for example, it can be specified by a *package call* using "$".

```
5 +$GF(7) 6

   (2)  4

Type: GF 7
```

Probably the next simplest thing is to assign a function value to a variable.

```
plusMod7  := _+$GF(7);  plusMod7(5, 6)  -- assigning + from GF(7) to a variable

   (3)  4

Type: GF 7
```

To access the value of the function object for a top level map it must be declared first.

```
double: I -> I
double n == 2*n


f := double;  f 13

   (6)  26

Type: I
```

Functions can be accepted as parameters or returned as values. Here we have an example of a function as a parameter

```
apply: (I -> I, I) -> I  -- apply takes a function as 1st parameter
apply(f, n) == f n       -- and invokes it on the 2nd parameter


apply(double, 32)

   (9)  64

Type: RN
```

and as a return value

```
trig: I -> (BF -> BF)    -- trig returns a function as its value
trig n == if oddp n then sin$BF else cos$BF


t := trig 1;  t 0.1

   (12)  0.099 83341 66468 28152 30681 4198

Type: BF
```

Several operations are provided to construct new functions from old. The most common method of combining functions is to compose them.

"*" is used for functional composition.

```
quadruple := double * double;  quadruple 3

   (13)  12

Type: I
```

"**" is used to iterate composition.

```
octuple := double**3;  octuple 3

   (14)  24

Type: I
```

diag gives the diagonal of a function. That is, if g is `diag f` then `g(a)` is equal to `f(a,a)`.

```
square := diag _*$I;  square 3

   (15)  9

Type: I
```

twist transposes the arguments of a function. If g is defined as `twist f` then `g(a,b)` has the value `f(b,a)`.

```
power := _**$RN;
rewop := twist power;  rewop(3, 2)

   (17)  8

Type: RN
```

Functions of lower arity can be defined by restricting arguments to constant values. The operations *cur* and *cul* fix a constant argument on the right and on the left, respectively. For unary functions, *cu* is used.

```
square := cur(power, 2);  square 4    --  square(a) = power(a,2)

   (18)  16

Type: RN
```

It is also possible to increase the arity of a function by providing additional arguments. For example, *vur* makes a unary function trivially binary; the second argument is ignored.

```
binarySquare := vur(square);  binarySquare(1/2, 1/3)

          1
   (19)   -
          4

Type: RN
```

The primitive combinator for recursion is *recur*. If g is `recur(f)` then `g(n,x)` is given by `f(n,f(n-1,..f(1,x)..))`.

```
fTimes := recur _*$NNI;  factorial := cur(fTimes, 1::NNI); factorial 4

   (20)  24

Type: NNI
```

Functions can be members of aggregate data objects. Here we collect some in a list. The unary function `incfn.i` takes the `i`-th successor of its argument.

```
incfn := [(succ$SUCCPKG)**i for i in 0..5]; incfn.4 9

   (21)  13

Type: I
```

In practice, a function consists of two parts: a piece of program and an environment in which that program is executed. The display of function values appear as `theMap(s, n)`, where `s` is a hideous internal symbol by which the program part of the function is known, and `n` is a numeric code to succinctly distinguish the environmental part of the function.

```
recipMod5 := recip$GF(5)

   (22)  theMap(MGF;recip;$U;17,642)

Type: GF 5 -> Union(GF 5,failed)

plusMod5  := _+$GF(5)

   (23)  theMap(MGF;+;3$;12,642)

Type: (GF 5,GF 5) -> GF 5

plusMod7  := _+$GF(7)

   (24)  theMap(MGF;+;3$;12,997)

Type: (GF 7,GF 7) -> GF 7
```

Notice above that the program part of `plusMod5` is the same as for `plusMod7` but that the environment parts are different. In this case the environment contains, among other things, the value of the modulus. The environment parts of `recipMod5` and `plusMod5` are the same.

When a given function is restricted to a constant argument, the value of the constant becomes part of the environment. In particular when the argument is a mutable object, closing over it yields a function with an *own* variable. For example, define `shiftfib` as a unary function which modifies its argument.

```
FibVals := Record(a0: I, a1: I)

   (25)  Record(a0: I,a1: I)

Type: DOMAIN

shiftfib: FibVals -> I
shiftfib r == (t := r.a0; r.a0 := r.a1; r.a1 := r.a1 + t; t)
```

Now `fibs` will be a nullary function with state. Since the parameter `[0,1]` has not been assigned to a variable it is only accessible by `fibs`.

```
fibs := cu(shiftfib, [0,1]$FibVals)

   (29)  theMap(%G12274,721)

Type: () -> I

[fibs() for i in 0..30]

   (30)
   [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, 233, 377, 610, 987,
    1597, 2584, 4181, 6765, 10946, 17711, 28657, 46368, 75025, 121393,
    196418, 317811, 514229, 832040]

Type: L I
```

8. Other Data Structures

We have seen that lists and streams can be used to hold values in a particular order.

```
[1980..1987, 1982, 1986]

   (1)  [1980,1981,1982,1983,1984,1985,1986,1987,1982,1986]
```

Scratchpad II provides many other structures that may better suit your applications. We will point out a few of them here.

Finite sets are collections of objects that contain no duplicates.

```
{1980..1987, 1982, 1986}

   (2)  {1980,1981,1982,1983,1984,1985,1986,1987}
```

A stack is a data structure where the last value added to it becomes the first one to be removed.

```
s : STK I := stack()
   (3)  stack(Bottom)
for i in 1980..1987 repeat push(i,s)
s
   (5)  stack(1987,1986,1985,1984,1983,1982,1981,1980,Bottom)
```

The value farthest from the bottom is the last one added.

```
pop s
   (6)  1987
s
   (7)  stack(1986,1985,1984,1983,1982,1981,1980,Bottom)
```

A queue is similar except that it is "first in, first out".

```
q : QUEUE I := queue()
   (8)  queue(Entry,Exit)
for i in 1980..1987 repeat enqueue(i,q)
q
   (10)  queue(Entry,1987,1986,1985,1984,1983,1982,1981,1980,Exit)
dequeue q
   (11)  1980
q
   (12)  queue(Entry,1987,1986,1985,1984,1983,1982,1981,Exit)
```

Scratchpad II provides several different types of tables to hold collections of values that can be looked up by some index set. The function *keys* gives a list of valid selectors to use to retrieve table entries.

Values of type Table(Key,Entry) are kept in memory in the workspace. Here Key and Entry may be replaced by any type.

```
colors : TBL(I, S) := table()
   (13)  table()
colors.1981 := "blue"; colors.1982 := "red"; colors.1983 := "green";
colors
   (15)  table(1981= "blue",1982= "red",1983= "green")
colors.1982
   (16)  "red"
```

KeyedAccessFile gives tables that are stored as random access files on disk. AssociationList is used for tables that may also be viewed as lists and have additional functions for looking up entries.

Record types are used to create objects with named components. The components of a record may be any type and do not all have to be the same type. An example declaration of a record is

```
bd : Record(name : S, birthdayMonth : I)
```

Here bd has two components: a String which is accessed via name and an Integer which has selector birthdayMonth.

You must set the value of the entire record at once if it does not already have a value. At this point is therefore illegal to enter bd.name := "Dick" because the birthdayMonth component has no value. However, bd := ["Dick", 11] is a legal assignment because it gives values to all components of the record. Issuing bd.name := "Chard" would now be legal.

A declaration such as

```
x : Union(I, S, BF)
```

states that x will have values that can be integers, strings or big floats. If, for example, the union object is an integer, the object is said to belong to Integer *branch* of the union.[4] The case infix operator returns a Boolean and can be use to determine the branch in which an object lies. The following function will display a message stating in which branch of the union the object x, defined above, lies.

```
sayBranch x ==
  if x case Integer then output "Integer branch"
  else if x case String then output "String branch"
  else if x case BigFloat then output "BigFloat branch"
  else output "don't know"
```

Now if we assign x := 8 and then issue

```
sayBranch x

   (3)  "Integer branch"
```

9. Algebraic Facilities

Scratchpad II provides a richs set of facilities for doing symbolic mathematical calculations. This section gives examples of integration, differentiation, solution of equations, and eigenvectors.

Integration

```
integrate(x**5/(x**4+x**2+1)**2,x)
```

$$(7)\quad \frac{-x^2+1}{6x^4+6x^2+6}+\sum_{\alpha^2+\frac{2}{27}=0}\alpha\log\left((x^2+2)\alpha+\frac{x}{3}\right)$$

Differentiation

```
pderiv((x+1)*exp(log(x)/x+x**2/3)/(x-1),x)
```

$$(33)\quad \frac{((-3x^2+3)\log(x)+2x^5-2x^3-3x^2-3)\%e^{\frac{3\log(x)+x^3}{3x}}}{3x^4-6x^3+3x^2}$$

[4] Note that we are being a bit careless with the language here. Technically, the type of x is always Union(I, S, BF). If it belongs to the Integer branch, x may be coerced to an object of type Integer.

```
integrate(%,x)     --check result
                                3
                  3log(x) + x
                  ------------
         x + 1         3x
   (34)  (-----)%e
         x - 1
```

Solution of a Polynomial Equation

```
first solve(x**3+x+1=0,x)    --look at only the first of several solutions
                                          1 2                  1
                                          -                    -
              1  1             1  1       3           1  1     3
              -  -             -  -                   -  -
            1 2  2   9    1   2  2   1          1   2  2   1
   (15)  (- (-)3 31  - -)(((--)3 31  - -) )  + ((--)3 31  - -)
            2         2    18         2          18         2

%**3+%+1      --check result

   (16)  0
```

Complex Zeros

```
solve(x**7+2*x**5-x**4+x**3-2*x**2-1=0,x,1/10000) --eqn, variable, precision

                   1    28377      1    28377
   (10)  [- %i,%i,- - - (-----)%i,- - + (-----)%i,1]
                   2    32768      2    32768
```

Solution of Systems of Polynomial Equations

```
solve({x**2-x+2*w**2+2*y**2+2*z**2=0, 2*x*w+2*w*y+2*y*z-w=0,_
       2*x*y+w**2+2*w*z-y=0,x+2*w+2*y+2*z-1=0},_          -- set of equations
     {x,y,z,w}, _                                        -- set of variables
     1/1000)                                             -- precision

 (6)
     683        683                                 901    215      271    629
  [{x=----,y=0,z=----,w=0}, {x=1,y=0,z=0,u=0}, {x=----,y=----,z=- ----,w=----},
     2048       2048                                2048   2048     2048   2048

     1527     383    165    479       1157   525      383    305
   {x=----,y=- ----,z=----,w=----}, {x=----,y=----,z=- ----,w=----},
     2048     2048   2048   2048      2048   2048     2048   2048

     387    155    515    161
   {x=----,y=----,z=----,w=----}]
     2048   2048   2048   2048
```

Eigenvectors and Eigenvalues of a Matrix

```
eigenvectors [[x,2,1],[2,1,-2],[1,-2,x]]

   (4)
                            |1|
                            | |
   [[eigval= x + 1,eigvec= [|0|]],
                            | |
                            |1|

                                        |   - 1    |
                                        |          |
                                        |x - %A - 1|
                          2             |----------|]]]
    [algrel= (%A - 1)x - %A  + 9,algvec= [|    2     |
                                        |          |
                                        |    1     |
```

10. Coercion

Scratchpad II provides very sophisticated facilities for changing an object of one type into an object of another type. If such a transformation involves no loss of information (as in creating a rational number from an integer), this process is called *coercion.* If some information may be lost (as in changing a rational number to a fixed precision floating point number), the process is called *conversion.* For the user, the major difference between coercions and conversions is that former may be automatically performed by the Scratchpad II interpreter while the latter must be explicitly requested.

Since coercions happen automatically, the interpreter sometimes gives the impression of doing what you mean, rather, perhaps, than what you say. This can be quite helpful (would you really want to give all the detail involved in the above example?) but might not always give you what you expect. The following is a definition of a function that computes Legendre polynomials.

```
leg(0) == 1
leg(1) == x
leg(n) == ((2*n-1)*x*leg(n-1)-(n-1)*leg(n-2))/n when n in 2..

leg 6
   Compiling leg with signature I -> RF I
   Compiling leg as a recurrence relation

           6       4       2
        231x  - 315x  + 105x  - 5
   (4)  ------------------------
                    16

Type: RF I
```

The only problem with this function is that it does not actually produce polynomials! What was actually produced was a rational function, that is, a quotient of polynomials. The denominator is the constant polynomial 2. To see this result as a polynomial with rational number coefficients, just do a coercion.

```
% :: P RN

         231  6    315  4    105  2   5
   (5)  (---)x  - (---)x  + (---)x  - --
         16        16        16       16

Type: P RN
```

The double colon is the symbol for explicit coercion/conversion, where you are telling the interpreter, "I know what I want, so try to give me an object of this type."

The interpreter decided that we wanted a rational function because of the way we did the division in the third line of the definition:

```
leg(n) == ((2*n-1)*x*leg(n-1)-(n-1)*leg(n-2))/n when n in 2..
```

If this is changed to

```
leg(n) == (1/n)*((2*n-1)*x*leg(n-1)-(n-1)*leg(n-2)) when n in 2..
```

we will get a polynomial without having to do the coercion.

As this example illustrates, coercion may be used to change the way an object looks. In this sense, coercion corresponds to the algebraic manipulation of formulas that one does, say, to simplify an expression or change it into a form that is more meaningful.

To illustrate this, let's start with a 2 by 2 matrix of polynomials whose coefficients are complex numbers. In this form, it doesn't make much sense to ask for the "real" part of the object. We will transform the matrix until we get a representation with a real and imaginary part, each of which is a matrix with polynomial coefficients. In the following, the symbol `%i` is the complex square root

of 1. G is the abbreviation for Gaussian, a parameterized type used to create domains such as the complex numbers.

```
m : SM(2,P G I)

m := [[(j + %i)*x**k - (k + %i)*y**j for j in 1..2] for k in 1..2]

       |                                    2             |
       |(- 1 - %i)y + (1 + %i)x   (- 1 - %i)y  + (2 + %i)x |
  (2)  |                                                  |
       |                        2           2            2|
       |(- 2 - %i)y + (1 + %i)x   (- 2 - %i)y  + (2 + %i)x |

Type: SM(2,P G I)
```

The matrix entries can be transformed so that they each have real and imaginary parts.

```
m :: SM(2, G P I)

       |                                 2          2        |
       | - y + x + (- y + x)%i     - y  + 2x + (- y  + x)%i  |
  (3)  |                                                     |
       |        2          2         2    2       2    2    |
       |- 2y + x  + (- y + x )%i  - 2y  + 2x  + (- y  + x )%i|

Type: SM(2,G P I)
```

Now we push the matrix structure inside the real and imaginary parts.

```
g := % :: G SM(2, P I)

       |               2      |   |             2     |
       | - y + x    - y  + 2x |   |- y + x   - y  + x |
  (4)  |                      | + |                   |%i
       |        2      2    2|   |       2     2    2|
       |- 2y + x   - 2y  + 2x |   |- y + x   - y  + x |

Type: G SM(2,P I)
```

It is now clearer what is meant by the "real part" of the object.

```
real(g)

       |               2      |
       | - y + x    - y  + 2x |
  (5)  |                      |
       |        2      2    2|
       |- 2y + x   - 2y  + 2x |

Type: SM(2,P I)
```

In fact, this is what would have been returned if you just asked for `real(m)`. If we would rather see this last object as a polynomial with matrix coefficients, a simple coercion will do it.

```
% :: P SM(2,I)

  (6)
    |0  - 1| 2   |- 1  0|     |0  0| 2   |1  2|
    |      |y  + |      |y + |   |   |x  + |   |x
    |0  - 2|     |- 2  0|     |1  2|     |0  0|

Type: P SM(2,I)
```

11. Output

Besides to the character-oriented two-dimensional output you have already seen in this paper, Scratchpad II provides facilities for viewing output in FORTRAN format and in forms suitable for TeX and the IBM Script Formula Formatter. The following equation is displayed in the standard Scratchpad II output format.

```
R = (2*x**2+4)**4/(x**2-2)**5
```

$$(1)\ R = \frac{16x^8 + 128x^6 + 384x^4 + 512x^2 + 256}{x^{10} - 10x^8 + 40x^6 - 80x^4 + 80x^2 - 32}$$

The FORTRAN-style output of the equation is

```
      R=(16*x**8+128*x**6+384*x**4+512*x**2+256)/(x**10 -10*x**8+40*x**6 -80*
     *x**4+80*x**2 -32)
```

A form suitable for input to the TeX formula processor is

```
$$
{R={{{16 \  {x \sp 8}}+{128 \  {x \sp 6}}+{384 \  {x \sp 4}}+{512 \  {x
\sp 2}}+256} \over {{x \sp 10} -{10 \  {x \sp 8}}+{40 \  {x \sp 6}} -{
80 \  {x \sp 4}}+{80 \  {x \sp 2}} -32}}}
$$
```

This is for input to the Script Formula Formatter:

```
:df.
<R=<<<16 % <x sup 8>>+<128 % <x sup 6>>+<384 % <x sup 4>>+<512 % <x sup
2>>+256> over <<x sup 10> -<10 % <x sup 8>>+<40 % <x sup 6>> -<80 % <x
sup 4>>+<80 % <x sup 2>> -32>>>
:edf.
```

When formatted by Script, the equation appears as

$$R = \frac{16x^8 + 128x^6 + 384x^4 + 512x^2 + 256}{x^{10} - 10x^8 + 40x^6 - 80x^4 + 80x^2 - 32}$$

The integration with respect to x of the right hand side of the equation produces a object which is a rational function plus a sum over the roots of a polynomial. The output produced by Scratchpad II for the Script Formula Formatter is

```
:df.
<<<-<10 % <x sup 7>> -<12 % <x sup 5>> -<24 % <x sup 3>> -<80 % x>>
over <<x sup 8> -<8 % <x sup 6>>+<24 % <x sup 4>> -<32 % <x sup 2>>+16>>
+<sum from <<< alpha  sup 2> -<9 over 2>>=0> of < alpha  % <log left ( <
<<x %  alpha > -3>> right )>>>>
:edf.
```

The processed form is much easier to understand!

$$\frac{-10x^7 - 12x^5 - 24x^3 - 80x}{x^8 - 8x^6 + 24x^4 - 32x^2 + 16} + \sum_{\alpha^2 - \frac{9}{2} = 0} \alpha \log(x\alpha - 3)$$

12. Packages

In a large system there will be thousands of functions and there must be some way to organize them. One would be like to be able to group similar functions together and to be able to think in terms of useful collections of functions. In Scratchpad II, this is done with *packages*. For example, functions to compute permutations, combinations and partitions are be grouped together in a package providing simple combinatoric functions.

To see what functions are available in a package, the *show* system command is used.

```
)show CombinatoricFunctions

CombinatoricFunctions is a package constructor.

Abbreviation for CombinatoricFunctions is COMBINAT
Issue )edit ARITHMET SPAD to see source code for COMBINAT

---------------------- Operations --------------------------
binomial : (I,I) -> I                combination : (I,I) -> I
multinomial : (I,L I) -> I           partition : I -> I
permutation : (I,I) -> I             selection : (I,I) -> I
```

To group a collection of functions as a package, they must be compiled together in the body of a package constructor. A package constructor is a function which returns a Scratchpad II package object. This act of calling such a function is called *package instantiation.*

The package constructor for the CombinatoricFunctions is

```
CombinatoricFunctions(): T == B where
    T == with
        binomial:     (Integer,Integer) -> Integer
        multinomial: (Integer, List Integer) -> Integer
        permutation: (Integer,Integer) -> Integer
        combination: (Integer,Integer) -> Integer
        selection:    (Integer,Integer) -> Integer
        partition:    Integer -> Integer


    B == add
        ArithmeticFunctions()  -- import factorial from another package

        binomial(n,k) ==
            k < 0 or n < k => 0
            k = 0 or n = k => 1
            n quo 2 < k    => binomial(n,n-k)
            t := 1
            for i in 1..k repeat t := (t*(n-i+1)) quo i
            t
                ...
                ...
        -- p is not exported, it is local to this package.
        p(m: Integer, n: Integer): Integer ==
            m = 1 => 1
            m < n => p(m-1,n) + p(m,n-m)
            m = n => p(m-1,n) + 1
            p(n,n)

        partition n == p(n,n)
```

This example serves to illustrate several points. The first line is the definition of the function CombinatoricFunctions which has type T and body B, with T and B defined further on. The type information for a package consists mainly of a list of the functions it exports and their types. The body gives the definitions of the exported functions. Because local variables in the body of the package constructor are invisible from outside, it is possible to maintain information which is private to the package.

13. Domains

One very natural way to group functions is to place together the operations for combining values of a given type. In one sense, the collection of operations which may be performed on values of a given type define what the type is. If these functions are provided by a single package, then it is possible to hide the representation of the values belonging to the type by keeping it local to the package. In Scratchpad II, using packages to so encapsulate a new types is the basic method of data abstraction.

For convenience we usually distinguish between packages which implement types and those which do not. We call the former *domains* and usually use the term *package* only for those which do not implement types.

We illustrate Stack below as an example of a domain constructor. The use of "$" in the signatures of exported operations (e.g. *pop*) represents the type which the domain implements.

```
Stack(S: Set): T == B where
    T == Set with
        stack:  ()-> $
        empty?: $ -> Boolean
        depth:  $ -> Integer
        push:   (S, $) -> S
        pop:    $ -> S
        peek:   $ -> S
        peek:   ($, Integer) -> S


    B == add
        -- Rep is a record so that the empty stack is mutable.
        Rep := Record(head: String, body: List S)

        Ex  ==> Expression
        coerce(s): Ex ==
            args: List Ex := []
            for e in s.body repeat args := cons(e::Ex, args)
            args := nreverse cons("Bottom"::Expression, args)
            mkNary("stack"::Ex, args)
        stack() ==
            ["Stack", []]
        empty? s ==
            null s.body
        push(e, s) ==
            s.body := cons(e, s.body)
            e
        pop s ==
            empty? s => error "Stack over popped."
            e := first s.body; s.body := rest s.body
            e
        peek s ==
            empty? s => error "Can't peek empty stack."
            first s.body
        depth s == #s.body
        peek(s,i) ==
            n := # s.body
            i > n-1 or i < -n => error "Out of bounds peek."
            s.body.(i<0 => n+i; i)
```

The coercion to Expression is used to give the output form of values in the domain.

14. Polymorphism

Whereas the package constructor for CombinatoricFunctions is a nullary function, in practice most package constructors take arguments as does Stack. Since package constructors may have type valued arguments, the exported functions may be used to express polymorphic algorithms.

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 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
        gcd(x,y) ==                  -- Euclidean algorithm
           x:= unitNormal.x.coef
           y:= unitNormal.y.coef
           while y ~= 0 repeat
              (x,y):= (y,x rem y)
              y:= unitNormal.y.coef
           x
        lcm(x, y) ==
           u: Union(R, "failed") := y exquo gcd(x,y)
           x * u::R
```

The exported operations are said to be *polymorphic* because they can equally well be used for many types, the integers or polynomials over GF(7) being two examples. Although the same *gcd* program

is used in both cases, the operations it uses (*rem*, *unitNormal*, etc.) come from the type parameter R.

15. 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. In order 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 is 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 on. 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.

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 upon 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.

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 " < " allows definitions of " > ", " < = " and " > = ". Thus a domain may inherit operations from a category. The use of Join provides multiple inheritance.

Acknowledgments

The authors would like to thank Barry Trager, William Burge and Rüdiger Gebauer of the Computer Algebra Group at Yorktown Heights, and Greg Fee of the Symbolic Computation Group at the University of Waterloo for their suggestions and examples.

Bibliography

[1] 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 *SIGPLAN Notices*, New York: Association for Computing Machinery, November 1981, and *IBM Research Report* RC 8930 (Yorktown Heights, New York).

[2] 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

[3] Computer Algebra Group, *Basic Algebraic Facilities of the Scratchpad II Computer Algebra System,* Yorktown Heights, New York: IBM Corporation, March 1986.

[4] Computer Algebra Group, *An Overview of the Scratchpad II Language and System,* Yorktown Heights, New York: IBM Corporation, April 1986.

[5] Computer Algebra Group, *Scratchpad II Examples from INPUT Files,* Yorktown Heights, New York: IBM Corporation, August 1986.

[6] Sutor, R. S., ed. *The Scratchpad II Newsletter*, Vol. 1, No. 1, Yorktown Heights, New York: IBM Corporation, September 1, 1985.

[7] Sutor, R. S., ed. *The Scratchpad II Newsletter*, Vol. 1, No. 2, Yorktown Heights, New York: IBM Corporation, January 15, 1986.

[8] Sutor, R. S., ed. *The Scratchpad II Newsletter*, Vol. 1, No. 3, Yorktown Heights, New York: IBM Corporation, May 15, 1986.