We find the equations defining the intersection of the two loci. This correspond to the sum of the associated ideals.

```
id := ideal m  + ideal n

                2   1  2   1
   (5)  [x  - -,y  - -]
                2      2

   Type: DIDEAL(RN,DP(2,NNI),[x,y],DMP([x,y],RN))
```

We can check if the locus contains only a finite number of points, that is, if the ideal is zero-dimensional.

```
iszerodim id

   (6)  true

   Type: B

iszerodim(ideal m)

   (7)  false

   Type: B
```

We can find polynomial relations among the generators (*f* and *g* are the parametric equations of the knot).

```
f := x**2-1

          2
   (8)  x  - 1

   Type: DMP([x,y],RN)

g := x*(x**2-1)

          3
   (9)  x  - x

   Type: DMP([x,y],RN)

relationsIdeal [f,g]

                 2      3     2
   (10)  [- %D  + %C  + %C ]

   Type: L P RN
```

We can compute the primary decomposition of an ideal.

```
l: L DMP([x,y,z],RF I)

   Type: VOID

l:=[x**2+2*y**2,x*z**2-y*z,z**2-4]

          2     2   2         2
   (12)  [x  + 2y ,x z  - y z,z  - 4]

   Type: L DMP([x,y,z],RF I)

ld:=primaryDecomp ideal l

                1    2              1    2
   (13)  [[x + - y,y ,z + 2],[x - - y,y ,z - 2]]
                2                   2

   Type: L DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))
```

We can intersect back:

```
"intersect"/ld

                1       2  2
   (14)  [x - - y z,y ,z  - 4]
                4

   Type: DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))
```

We can compute the radical of every primary component. Their intersection is equal to the radical of the ideal of *l*.

```
rr:="intersect"/[radical ld.i for i in 0..1]

             2
   (15)  [x,y,z  - 4]

   Type: DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))

ss:=radical ideal l

             2
   (16)  [x,y,z  - 4]

   Type: DIDEAL(RF I,DP(3,NNI),[x,y,z],DMP([x,y,z],RF I))
```

References

[1] Gianni, P., Trager, B., and Zacharias, G., "Gröbner Bases and Primary Decomposition of Polynomial Ideals," to appear in *Journal of Symbolic Computation.*

Patrizia Gianni

Mappings as First Class Objects

Defining and Applying Mappings

Mappings 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.

Mappings may be defined interactively in the interpreter or they may be defined in 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
```

Manipulating Mapping Values

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

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

   (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
```

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

```
-- apply takes a function as 1st parameter
-- and invokes it on the 2nd parameter

apply: (I -> I, I) -> I

apply(f, n) == f n

apply(double, 32)

   (9)  64

Type: RN
```

and as a return value

```
-- trig returns a function as its value
trig: I -> (BF -> BF)

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
```

Mappings 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
```

Mappings 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
```

Mappings as Program-Environment Pairs

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

```
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.

Creating "Own" Variables

When a given mapping 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 program with an *own* variable. For example, define `shiftfib` as a unary mapping 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 program 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
```

Fixed Point Operations

A fixed point x of a map f is a point in the domain of f such that $x = f(x)$. Given a function which operates on a recursively defined data type, it is often possible to compute a useful fixed point. A powerful method for manipulating infinite structures is to compute the fixed point of structure transforming functions. As well as providing a functional mechanism for constructing self-referential structures, a combination of lazy evaluation and self-reference may be achieved.

Consider a recursively defined data type T and the class of functions mapping T → T. Certain functions in this class have trivial fixed points: the identity and constant valued functions. Some functions in the class may have no fixed point. (The fact that negation has no fixed point leads to the Russell paradox.) Other functions may have a fixed point which it is impossible to compute effectively.

If we restrict our attention to functions which do not perform operations on their argument but rather just include it in a new structure which is returned as the value then we may always compute a fixed point.

As an example, an infinite repeating list of values can be obtained as follows:

```
cons1234(1) == cons(1,cons(2,cons(3,cons(4,1))))

repeating1234 := fixedPoint cons1234
```

A fixed point finding operation is provided which operates on a stream transforming function and finds its fixed point, a stream.

```
a:=integers 1

   (2)  [1,2,3,4,5,6,7,8,9,10,...]
```

The function below prefixes a 1 to an integer stream.

```
f1(x: ST I): ST I == cons(1,x)

f1 a

   (4)  [1,1,2,3,4,5,6,7,8,9,10,...]
```

and the fixed point of *f1* is an infinite stream of 1's

```
b := fixedPoint f1

   (5)  [1]
```

Similarly

```
f2(x: ST I): ST I == append([1,2,3,4,5,6], x)

fixedPoint f2

   (8)  [1,2,3,4,5,6,1]
```

Here is another way to define the Fibonacci number stream. The plus operation takes two streams and adds them pair-wise.

```
f3(fib: ST I): ST I == cons(1,fib+cons(0,fib))

f3 b

   (10)  [1,1,2,2,2,2,2,2,2,2,...]

fixedPoint f3

   (11)  [1,1,2,3,5,8,13,21,34,55,...]
```

The stream of Catalan numbers:

```
f4(cat: ST I): ST I == cons(1,cat*cat)

fixedPoint f4

   (14)  [1,1,2,5,14,42,132,429,1430,4862,...]
```

The function *integ* integrates a stream viewed as the coefficients of a power series.

```
integ b

                1 1 1 1 1 1 1 1 1
   (15)  [1,-,-,-,-,-,-,-,-,--,...]
           2 3 4 5 6 7 8 9 10
```

Here we compute the fixed point of the function g that integrates a stream, and adds the constant term 1.

```
g(e: ST RN -> ST RN) == cons(1,integ e)

fixedPoint g

                1 1 1   1   1   1     1     1
   (18)  [1,1,-,-,--,---,---,----,-----,------,...]
           2 6 24 120 720 5040 40320 362880
```

It is also possible to find the fixed point of a function that transforms a pair of streams to a pair of streams.

```
k(tr: L ST I): L ST I == [cons(0,tr.1),1/(1-tr.0)]

k([cons(0,b),b])

   (20)  [[0,1],[1,1,2,4,8,16,32,64,128,256,...]]
```

The fixed point of k is two mutually recursive streams. Computing this provides another way to obtain the stream of Catalan numbers.

```
fixedPoint(k, 2)

   (21)
   [[0,1,1,2,5,14,42,132,429,1430,...],

    [1,1,2,5,14,42,132,429,1430,4862,...]]
```

Stephen M. Watt
William H. Burge

Work in Progress

This section describes some work on the Scratchpad II system that is being undertaken but is not yet complete.

Support for Data Structures in Scratchpad II

An effort to "categorize" the data structures available in Scratchpad II is now under way. Until recently, the algebraic facilities in Scratchpad II have made use of only a few data structures, in particular, records, lists and vectors. However, now that the new compiler is being written in Scratchpad II, the need for other traditional data structures has arisen. For example, various parts of the new compiler make use of stacks, sets, tables and doubly linked lists. New applications of Scratchpad II will make use of dictionaries, graphs, priority queues, etc.. Furthermore, as the system evolves, the need for specialized and highly efficient data structures will arise. One such example is Stream which provides a mechanism for lazy evaluation. To systematically describe the relationships between data structures and the operations available for them, we are building a category hierarchy for data structures.

The following partial definition for the categories Collection, IndexedAggregate and FiniteLinearAggregate should illustrate our point. Note that Type is the category in which all domains in the system belong.

```
Collection(S:Type): Category with
   -- number of items in the collection
   #: $ -> NonNegativeInteger

   -- top level copy of the collection
   copy: $ -> $

   map: ((S,S)->S,$) -> $

   if $ has shallowlyMutable then
     mapInPlace: ((S,S)->S,$) -> $
```

```
IndexedAggregate(Index:Type,S:Type): Category ==
  Collection S with
    elt: ($,Index) -> S

    if $ has shallowlyMutable then
      setelt: ($,Index,S) -> S
```

```
FiniteLinearAggregate(S:Type): Category ==
  IndexedAggregate(Integer,S) with
    concatenate: ($,$) -> $

    if S has OrderedSet then
      sort: $ -> $    -- sort using the "<" from S

      if $ has shallowlyMutable then
        sortInPlace: $ -> $
```

The category Collection describes homogeneous aggregates of objects with the operations *#*, ***copy***, ***map*** and the operation ***mapInPlace*** (if a collection has the attribute shallowlyMutable). A collection must have the attribute shallowlyMutable if any of its operations update (mutate) it. The prefix "shallowly" indicates that we are referring to the replacement of one component with another, rather than updating the component itself. For example, the ***mapInPlace*** operation updates a collection "in place" whereas the ***map*** operation creates a new structure. Notice that this implies that for shallowlyMutable collections, the ***map*** operation may be defined by applying ***mapInPlace*** to a copy of the collection. Indexed aggregates are collections which are indexed by objects of some type. A table of key/entry pairs is an example of an indexed aggregate. A table is a collection of entries indexed by keys. A finite linear aggregate is an indexed aggregate where the index is an Integer. For example, a string is a finite linear aggregate of characters.

This categorization of the data structures provides two main benefits. The first and primary benefit is the inheritance and sharing of code. For example, a generic sorting routine can be written for shallowlyMutable finite linear aggregates whose components belong to an ordered set (one could implement quick-sort in terms of the operations *#*, ***elt*** and ***setelt***). Secondly, hierarchies give a useful and systematic method for defining consistent and comprehensive sets of operations on types. They explain certain similarities and differences between types. For example, lists and strings are both finite linear aggregates and hence can be concatenated, but lists are also recursive aggregates.

There is an additional benefit from having a hierarchical organization for the data structures. We intend to provide as part of the user interface a way in which a user can explore category hierarchies. The