RC 12794 (#57573) 5/27/87
Mathematics 9 pages

INFINITE STRUCTURES IN SCRATCHPAD II

William H. Burge
Stephen M. Watt

IBM Thomas J. Watson Research Center
Box 218, Yorktown Heights, NY 10598 USA

Abstract

An *infinite structure* is a data structure which cannot be fully constructed in any fixed amount of space. Several varieties of infinite structures are currently supported in Scratchpad II: infinite sequences, radix expansions, power series and continued fractions. Two basic methods are employed to represent infinite structures: self referential data structures and lazy evaluation. These may be employed either separately or in conjunction.

This paper presents recently developed facilities in Scratchpad II for manipulating infinite structures. General techniques for manipulating infinite structures are covered, as well as the higher level manipulations on the various types of mathematical objects represented by infinite structures.

INFINITE STRUCTURES IN SCRATCHPAD II

William H. Burge
Stephen M. Watt

IBM Thomas J. Watson Research Center
Box 218, Yorktown Heights, NY 10598 USA

Abstract

An *infinite structure* is a data structure which cannot be fully constructed in any fixed amount of space. Several varieties of infinite structures are currently supported in Scratchpad II: infinite sequences, radix expansions, power series and continued fractions. Two basic methods are employed to represent infinite structures: self referential data structures and lazy evaluation. These may be employed either separately or in conjunction.

This paper presents recently developed facilities in Scratchpad II for manipulating infinite structures. General techniques for manipulating infinite structures are covered, as well as the higher level manipulations on the various types of mathematical objects represented by infinite structures.

1. Introduction

An *infinite structure* is a data structure which cannot be fully constructed in any bounded amount of space. Examples are a list of the digits of π or an explicit representation of n^3: $Z \rightarrow Z$ as a set of ordered pairs. On the other hand a *finite structure* is the opposite: it is a data structure which can be be fully constructed in finite storage.

It is common practice to use different data structures to represent the same value, according to the operations which are expected to be performed. For example, a collection of strings may be stored in any one of a linked list, a binary search tree or a hash table. Normally, these data structures will have differing storage requirements.

A value which has a representation as an infinite structure might also be represented by other infinite or finite structures. For example, the Fibonacci numbers can be given either as an infinite sequence of integers or as a finite recurrence with initial values. Likewise, $\sin(x)$ can be represented by infinite structures (a power series, a continued fraction, an infinite product) or by finite structures (a differential equation with initial value, an expression tree).

Scratchpad II has a number of facilities for creating and manipulating infinite structures. Two principal techniques are employed: (i) folding recursive data structures to be self-referential, and (ii) lazy evaluation.

This paper describes how these techniques are used to create and manipulate infinite structures in Scratchpad II.

We begin in section 2 by summarizing the general techniques for manipulating infinite structures. In section 3 the use of these techniques in creating *streams*, a low-level data type for representing infinite sequences is shown. In sections 4, 5 and 6 we give a number of examples of mathematical objects in terms of infinite structures. Section 4 discusses radix expansion of numbers. Section 5 describes the implementation of power series in terms of streams. Section 6 shows infinite continued fractions in normal form. Finally, in section 7, future developments are discussed.

2. Techniques for Infinite Structures

2.1. Self-Referential Data Structures

Structures for composite data are often defined recursively. Two common examples are linked lists and binary trees. The tail of a linked list has the same type as the whole list, and both branches of a binary tree have the same type as the whole tree.

Programs which traverse recursively defined data structures often have a logical structure which mirrors the data structure. Such programs have only a local view of the data. On recursive calls, they consider their argument as being the whole data structure and positional information relating to the original structure is lost.

If it is possible to assign to parts of the data structure (e.g. record field assignment), then a self-referring cycled object may be created. This can lead to bugs in programs which are written to have only a local view of the structure and which are expected to completely traverse a finite structure.

On the other hand, self-referential data structures are ideal for representing infinite structures which have some form of periodicity. Programs written to traverse a logically infinite structure can use an equivalent self-referential finite structure so long as they do not modify it as they proceed. Such programs are written to traverse only a finite portion of the logically infinite structure or are intentionally non-terminating.

As an example, consider power series represented as an infinite list of coefficients, with the i^{th} element being the coefficient to the term of degree i. Then the series for $1/(x+1)$ is a cycle of two elements: $[1, -1]$ and the series for a polynomial has a one element cycle of zeros at the end.

2.2. Lazy Evaluation

It is common practice to use procedural abstraction in programming languages to represent mappings finitely. This allows a possibly infinite set of ordered pairs of the mapping to be represented by an equivalent structure: the program. Then when points from the mapping are needed they are computed on demand by invoking the program on elements of the mapping's domain.

This is the fundamental idea of lazy evaluation: construct a value when it is to be used rather than when it is defined. For extended or aggregate data structures, construct the parts when they are required.

To construct an object as it is used, each unconstructed portion is represented as a program with some state information. The program part may be explicit (e.g. a pointer to a function) or implicit (a fixed interpreter) and the state information, if any, may be stored in global variables, in the data structure, or in a closed environment of the program part.

2.3. 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 $T \rightarrow T$. 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.

Let us 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 follows:

```
fixedPoint(f) ==
    arg := generateUnique()
    ret := f(arg)
    if arg = ret then
        -- f is the identity
        return arb. element from the domain of f
    else
        ret := subs(arg = ret in ret)
        return ret
```

Here `generateUnique` is a function which returns a unique system-wide value. Since f does not perform any operations on its argument, it is safe to pass it this generated unique value, which strictly speaking does not lie in its domain.

From the definition of `fixedPoint` above we see that, for functions in our restricted class, the set of fixed points will be one of

- a single constant (for functions which ignore their argument),
- the entire domain (for the identity function), or
- a single infinite structure

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

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

repeating1234 := fixedPoint cons1234
```

To combine lazy evaluation and self-reference, one builds a lazily evaluated object in which the state information for the unconstructed part contains references back to the whole object. It is convenient to use a fixed point calculation to build such objects. Examples of this are given in sections 3.4.

3. Streams

3.1. Language Support

Streams have been in Scratchpad II for some time. They have recently been reorganized into a domain[1] similar to lists, but with the difference that they might be infinite.

[1] Here the word "domain" is Scratchpad II terminology for "abstract data type".

The language provides special syntax for creating and iterating over lists and streams. This syntax is merely a convenience – it is ultimately translated into calls to operations on a list or stream type, as appropriate. Some of the stream functions are `cons`, `null`, `first`, `rest` `take`, `drop` and `elt` (similar to the list functions), together with functions for creating finite and infinite streams.

We begin this section on streams by illustrating with some examples of the language support.

```
a := [1..]

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

b := [i+1 for i in a]

   (2)  [2,3,4,5,6,7,8,9,10,11,...]
```

Select the 20th element (0-based indexing):

```
b.20

   (3)  22
```

At this point, the first 21 elements of `b` have been evaluated.

```
b

   (4)  [2,3,4,5,6,7,8,9,10,11,12,13,14,15,
          16,17,18,19,20,21,22,...]
```

The stream of odd integers can be created using a filter.

```
[i for i in a | oddp i]

   (5)  [1,3,5,7,9,11,13,15,17,19,...]
```

It is possible to combine multiple streams using parallel `for` iterators.

```
[[i,j] for i in a for j in b]

   (6)  [[1,2], [2,3], [3,4], [4,5], [5,6], [6,7],
          [7,8], [8,9], [9,10], [10,11], ...]
```

`append(x,y)` is the concatenation of streams `x` and `y`.

```
append([i for i in a while i<7],a)

   (7)  [1,2,3,4,5,6,1,2,3,4,5,6,7,8,9,10,11,12,13,
          14,15,16,17,18,19,20,21,...]
```

The sum of a finite stream of integers:

```
reduce(0, _+$1, take(a,10))

   (8)  55
```

A stream of partial sums:

```
scan(0, _+$1, a)

   (9)  [1,3,6,10,15,21,28,36,45,55,...]
```

3.2. *Lazy Evaluation in Streams*

A stream is represented by a list whose last element is a function that contains the wherewithal to create the rest of the list from that point, should it ever be required. The function takes no arguments and when invoked returns a stream of the appropriate type. The stream that it returns is used as the remainder of the original stream, and the value it returns becomes the first member of the stream. The function is paired with an environment which contains whatever state information is required to extend the stream.

Since functions that extract elements from a stream must test whether it is null before proceeding, the necessity of extending the stream can be determined by the function which tests for a null stream. It is the function `null` which does this. If `null` determines that more of the stream need be computed, then the particular stream-extending function is invoked. The stream is updated in place to obviate re-evaluation.

Scratchpad II is an abstract data type language so the representation of streams is invisible outside of the code which defines the data type. To allow programs to create infinite streams using lazy evaluation the stream data type exports the primitive operation `delay`.

`delay` takes a nullary stream-valued function and returns a stream of the same type. I.e.

$$delay\colon (() \rightarrow Stream\ T) \ \rightarrow\ Stream\ T$$

This, by itself, is not very exciting. All that has happened is that the stream valued function has been saved away somewhere, to be evaluated when elements are required from the stream. The power of `delay` comes from its use in the recursive definition of functions.

To illustrate, consider the definition of the `scan` function. This function takes as parameters, an initial value b, a binary function h, and a stream x. The value returned is the stream

$$[b, h\,(b, x_0), h(h(b, x_0), x_1),\ \ldots\ ]$$

`scan` may be defined in terms of `delay` as follows:

```
scan(b,h,x) ==
    null x => nil()
    delay
       c := h(frst x,b)
       cons(c,scan(c,h,rst x))
```

The last two lines form the body of a nullary function to which `delay` is applied.

`delay` also forms the basis of a style of programming for the creation of infinite streams. The style is to use intentionally recursive functions with no base case. For example, the following function returns the stream of Fibonnaci numbers when invoked on (1,1).

```
fib(a0, a1) ==
   cons(a0, delay fib(a1, a0 + a1))
```

3.3. Self-Referential Streams

A second way to create infinite streams is through the use of self-reference. The simplest way to do this is with the function `repeating`. This function takes a list of elements and produces a stream which repeats them indefinitely.

```
repeating [1, 2, 3]
```

(1) $[\overline{1,2,3}]$

Although this could be implemented using lazy evaluation, it is more efficient to represent this stream as a list with the third tail pointing back to the beginning.

Other self-referential streams can be created using fixed point operations. The simplest are again repeating streams. A subtler form of self-reference can be achieved by computing a fixed point in which the state information paired with the function contains a pointer back into the stream itself. Examples of both sorts are given in the following section.

3.4. Fixed Points of Stream Transforming Functions

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 `f` is an infinite stream of 1's

```
b := fixedPoint f
```

(5) $[\overline{1}]$

Similarly

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

fixedPoint f2
```

(8) $[1,\overline{2,3,4,5,6,1}]$

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) $[[0,\overline{1}],[1,1,2,4,8,16,32,64,128,256,\ldots]]$

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,...]]
```

4. Radix Expansions

In Scratchpad II it is possible to evaluate certain numeric types to decimal expansions or radix expansions in other bases. The simplest of these is the expansion of rational numbers. Here we give some examples.

First we define a couple of functions for coercion.

```
decimal r    == r::DecimalExpansion
radix(n, r) == r::RadixExpansion(n)
```

All rational values have repeating decimal expansions

```
decimal(22/7)

   (49)  3.1̅4̅2̅8̅5̅7̅
```

The arithmetic of decimal expansions is exact.

```
% + decimal(6/7)

   (50)  4
```

The periods can be short or long:

```
[decimal(1/i) for i in 350..353]

   (51)

   [0.002̅8̅5̅7̅1̅4̅, 0.0̅0̅2̅8̅4̅9̅, 0.00284̅0̅9̅,

    0.0̅0̅2̅8̅3̅2̅8̅6̅1̅1̅8̅9̅8̅0̅1̅6̅9̅9̅7̅1̅6̅7̅1̅3̅6̅8̅1̅0̅1̅9̅8̅3̅]

decimal(1/2049)

   (52)
   0.
     OVERBAR
        000488042947779404587603709126403123474865788189
          360663738408979990239141044411908247925817471 9
          375305026842362127867252318204001952171791117 6
          183504148365056124938994631527574426549536359 1
          996095656417764763299170326988775012201073694 4
          851146900927281600780868716447047340165934602 2
          449975597852611029770619814543679843826256710 5
          90531966813079551
```

Scratchpad II can do radix expansions in other bases.

```
[radix(i, 5/24) for i in 2..10]

   (53)

   [0.0011̅0̅:RADIX 2, 0.01̅2̅:RADIX 3, 0.031̅:RADIX 4,

    0.1̅0̅:RADIX 5, 0.113:RADIX 6, 0.1̅3̅:RADIX 7, 0.152̅:RADIX 8,

    0.17̅:RADIX 9, 0.2083̅:RADIX 10]
```

For bases greater than 10, the ragits (radix digits) are separated by blanks.

```
[radix(i, 5/24) for i in 11..15]

   (54)

   [0 . 2̅ ̅3̅:RADIX 11, 0 . 2 6:RADIX 12, 0 . 2̅ ̅9̅:RADIX 13,

    0 . 2 12 11 9̅ ̅4̅:RADIX 14, 0 . 3 1̅ ̅1̅3̅:RADIX 15]
```

These numbers are bona fide algebraic objects.

```
p := decimal(1/4)*x**2 + decimal(2/3)*x + decimal(4/9)

                2
   (55)  0.25x  + 0.6̅x + 0.4̅

q := pderiv(p, x)

   (56)  0.5x + 0.6̅

g := gcd(p, q)

   (57)  x + 1.3̅
```

The function `fracRagits` gives the stream of ragits in the fractional part of its argument.

```
rr: RADIX(8) := 3/49

   (58)  0.0̅3̅7̅2̅6̅1̅5̅

fracRagits rr

   (59)  [0,3̅,7̅,2̅,6̅,1̅,5̅,0̅]

%.30

   (60)  7
```

5. Power Series

5.1. Construction via Defining Relation

The functions in the Stream domain and stream packages are particularly suitable for the implementation of algorithms on power series. The domain PowerSeries is provided as a field, and the domain UnivariatePowerSeries and an elementary function package adds to it the functions exp, log, sin, cos, tan, the hypergeometric function, composition, lagrange inversion, reversion together with the solution of linear differential equations in power series.

A general method of producing programs which solve recursion or differential equations in power series by the method of undetermined coefficients has been developed in which the program can be written down almost immediately from the defining relation. In the method of undetermined coefficients a trial series together with an initial value or two is substituted into the recursion or differential equation, and then coefficients of equal powers are equated.

In these programs the trial series is made up of the initial values followed by the as yet unevaluated stream. The tail of the stream is then defined in terms of the whole stream and when elements are required the trial series becomes the resulting stream. The program, because it uses functions that operate on whole streams, rather than stream elements has the same structure as the defining relation.

For example e raised to the power series power $A(x)$, has defining relation

$$(e^{A(x)})' \equiv A'(x)e^{A(x)}$$

The corresponding program for generating the power series exp A, in Scratchpad II, where A is a power series is

```
exp A == integrate(1,pderiv A*exp A))
```

in which `integrate` and `deriv`, respectively, integrate and differentiate power series.

5.2. Examples

The command

```
)set streams calculate n
```

will cause the series to be displayed up to n^{th} order. If s is a variable assigned a series as its value, then one way to view it to higher order is to re-issue the ")set" command with a higher value of n and then re-display the value of s. The following declares `x` to be a UPS(x,RN), in other words a UnivariatePowerSeries with variable x and with rational number coefficients.

```
x := ps x
   (12)  x

exp x
   (13)
              1  2    1  3    1   4    1   5
    1 + x + (-)x  + (-)x  + (--)x  + (---)x
              2       6      24       120
   +
      1   6     1    7      1     8      1     9
    (---)x  + (----)x  + (-----)x  + (------)x
     720       5040       40320       362880
   +
        1      10       11
    (-------)x   + O(x   )
     3628800

cos x ** cos x
   (14)
            1  2     7  4     19  6    1597  8
    1 - (-)x  + (--)x  - (---)x  + (-----)x
            2       24       180       40320
   +
        373   10       11
    - (-----)x   + O(x   )
       32400

x/(exp x-1)
   (15)
          1      1  2      1   4      1    6
    1 - (-)x + (--)x  - (---)x  + (-----)x
          2      12       720      30240
   +
          1     8       1       10      11
    - (-------)x  + (--------)x   + O(x   )
       1209600       47900160
```

The hypergeometric function:

```
hyp(1/2,1,3/2,-x**2)
   (18)
           1  2    1  4    1  6    1  8    1   10
    1 - (-)x  + (-)x  - (-)x  + (-)x  - (--)x
           3       5       7       9      11
   +
       11
    O(x   )
```

Power series provide a method of solving differential equations when all else fails. The function `lde` solves the n^{th} order linear differential equation, its argument is a list of power series coefficients. The two solutions of

$$y'' + (\cos x)y' + (\sin x)y = 0$$

are

```
lde([sin x,cos x])
   (19)
   [
           1  2    1  4     31  6     379   8
     1 - (-)x  + (-)x  - (---)x  + (-----)x
           2       6      720       40320
    +
         1639   10       11
     - (------)x   + O(x   )
        907200
    ,
          1  3    1   5     59   7     31   9       11
     x - (-)x  + (--)x  - (----)x  + (----)x  + O(x   )
          3       10      2520       6480
   ]
```

Power series are also used as enumerating generating functions and the power series may be expanded from its generating function. For example the generating function for the Legendre polynomials is

$$\frac{1}{(1-2xt+t^2)^{1/2}}$$

With suitable declarations for `x` and `t`, it may be expanded directly as follows:

```
(1-2*x*t+t**2)**(-1/2)

   (24)
                          2        2
    1 + x*t + ((3/2)x  - 1/2)t
   +
             3            3
    ((5/2)x  - (3/2)x)t
   +
              4          2         4
    ((35/8)x  - (15/4)x  + 3/8)t
   +
              5          3            5
    ((63/8)x  - (35/4)x  + (15/8)x)t
   +
                6            4             2          6
    ((231/16)x  - (315/16)x  + (105/16)x  - 5/16)t
   +
                  7            5             3
        (429/16)x  - (693/16)x  + (315/16)x
      +
        - (35/16)x
     *
       7
      t
   +
       8
    O(t )
```

It is also possible to expand certain infinite products as power series. The function lambert will transform one series into another in which the coefficient A_n of x^n is the sum of the coefficients of the original a_i for all i that divide n, including 1 and n. In other words, if $f(x)$ is a power series, then $lambert(f(x))$ is the power series

$$f(x) + f(x^2) + f(x^3) + f(x^4) + \ldots$$

The series for the number of divisors of n is

```
lambert(x/(1-x))

   (20)
             2     3     4     5     6     7     8
      x + 2x  + 2x  + 3x  + 2x  + 4x  + 2x  + 4x
    +
        9      10      11
      3x  + 4x   + O(x  )
```

Using this function it is possible to expand certain infinite products as power series. For example the enumerating generating function for partitions is

$$\prod_{n=1}^{\infty} \frac{1}{(1-q^n)}$$

```
partitions := exp(lambert(log(1/(1-x))))

   (21)
                 2     3     4     5      6      7
      1 + x + 2x  + 3x  + 5x  + 7x  + 11x  + 15x
    +
         8      9      10      11
      22x  + 30x  + 42x   + O(x  )
```

Euler's theorem is then:

```
1/partitions
                  2    5    7      11
   (22)  1 - x - x  + x  + x  + O(x  )
```

The function `h`, defined below expands the infinite product

$$\prod_{n=1}^{\infty} g(x^n)$$

where g is a power series with constant term 1.

```
h g == exp lambert log g
```

The coefficient of $y^i z^n$ below is the number of partitions of n into i parts

```
h(1/(1-y*z))
   (7)
                     2        2     3    2         3
      1 + y*z + (y  + y)z  + (y  + y  + y)z
    +
        4    3     2        4
      (y  + y  + 2y  + y)z
    +
        5    4     3     2        5
      (y  + y  + 2y  + 2y  + y)z
    +
        6    5     4     3     2        6
      (y  + y  + 2y  + 3y  + 3y  + y)z
    +
        7    6     5     4     3     2        7
      (y  + y  + 2y  + 3y  + 4y  + 3y  + y)z
    +
        8    7     6     5     4     3     2        8
      (y  + y  + 2y  + 3y  + 5y  + 5y  + 4y  + y)z
    +
         9
      O(z  )
```

Jacobi's celebrated result:

```
h(1-x)**3
                        3     6      10      11
   (11)  1 - 3x + 5x  - 7x  + 9x   + O(x  )
```

The Hermite polynomials:

```
h(1/(1-a*f))*h(1/(1-f/a))

   (15)
             2              4    3    2
            a  + 1         a  + a  + a  + a + 1  2
      1 + (------)g + (--------------------)g
              a                   2
                                 a
    +
        6    5     4     3     2
       a  + a  + 2a  + 2a  + 2a  + a + 1  3
      (---------------------------------)g
                       3
                      a
    +
        8    7     6     5     4     3     2
       a  + a  + 3a  + 3a  + 4a  + 3a  + 3a  + a + 1  4
      (-----------------------------------------------)g
                             4
                            a
    +
         5
      O(g  )
```

The examples above illustrate the present capability of writing expressions that denote power series.

6. Continued Fractions

We use the following notations for continued fractions:

$$\mathop{\Phi}_{i=1}^{n} \frac{a_i}{b_i} = \cfrac{a_1}{b_1 + \cfrac{a_2}{b_2 + \cdots \cfrac{a_n}{b_n}}}$$

$$= \frac{a_1|}{|b_1} + \frac{a_2|}{|b_2} + \cdots + \frac{a_n|}{|b_n}$$

The notation $\mathop{\Phi}_{i=1}^{\infty} a_i/b_i$ may be used to represent the limit of an infinite sequence of convergents.

The function `continuedFraction` provides one method of forming a continued fraction. It takes as arguments the whole part, the partial numerators and the partial denominators.

The continued fraction $\mathop{\Phi}_{i=1}^{\infty} \frac{i}{i}$ which has the value $1/(e-1)$ is entered as

```
s := continuedFraction(0, [1..], [1..])

   (46)
        1 |       2 |       3 |       4 |       5 |       6 |
      ,---' +  ,---' +  ,---' +  ,---' +  ,---' +  ,---' + ...
      | 1      | 2      | 3      | 4      | 5      | 6
```

If all the numerators are one, then `reducedContinuedFraction` may be used. Euler discovered the relation $\frac{e-1}{e+1} = \mathop{\Phi}_{i=1}^{\infty} \frac{1}{4i-2}$

```
t := reducedContinuedFraction(0, [4*i-2 for i in 1..])

  (47)

    1 |     1 |     1  |     1  |     1  |     1  |
  ,---' + ,---' + ,----' + ,----' + ,----' + ,----' + ...
  | 2     | 6     | 10     | 14     | 18     | 22
```

Arithmetic on infinite continued fractions is supported. The results are given in reduced form. We illustrate by using the values $s = 1/(e-1)$ and $t = (e-1)/(e+1)$ to recover the expansion for e.

```
e := 1/(s*t) - 1

  (48)

        1 |     1 |     1 |     1 |     1 |     1 |
  2 + ,---' + ,---' + ,---' + ,---' + ,---' + ,---' + ...
      | 1     | 2     | 1     | 1     | 4     | 1
```

The following command evaluates the 15th convergent to a floating point number.

```
convergents(e).15::F

  (49)  2.71828182847
```

Many univariate power series may be transformed to continued fractions. Here we show the conversion of series to continued fractions in normal form, that is a continued fraction in which the partial denominators are one and the partial numerators after the first are monomials of degree 1. The quotient-difference algorithm takes the coefficients of the series and produces the partial numerators of the continued fraction.

The first example is the series for exp x:

```
expx := exp ps x

  (50)

           1  2    1  3    1   4     1   5      6
  1 + x + (-)x  + (-)x  + (--)x  + (---)x  + O(x )
           2       6       24       120
```

The domain for these continued fractions is abbreviated CFPS.

```
expx::CFPS(x,RN)

                           1
(51) ---------------------------------------------
                            1x
     1 - -----------------------------------------
                               1
                              (-)x
                               2
         1 + -------------------------------------
                                 1
                                (-)x
                                 6
             1 - ---------------------------------
                                   1
                                  (-)x
                                   6
                 1 + -----------------------------
                                     1
                                    (--)x
                                     10
                     1 - -------------------------
                                       1
                                      (--)x
                                       10
                         1 + ---------------------
                                         1
                                        (--)x
                                         14
                             1 - -----------------
                                           1
                                          (--)x
                                           14
                                 1 + -------------
                                             1
                                            (--)x
                                             18
                                     1 - ---------
                                         1 - etc
```

Another example is:

```
qq := q::QF UP(q, RN);

[qq**(i**2) for i in 0..]

             4  9  16  25  36  49  64  81
  (53) [1,q,q ,q ,q  ,q  ,q  ,q  ,q  ,q  ,...]

%::UPS(x,QF UP(q,RN))
  (28)
               4 2    9 3    16 4    25 5    36 6
    1 + q*x + q x  + q x  + q  x  + q  x  + q  x
  +
     49 7    64 8    81 9    100 10      11
    q  x  + q  x  + q  x  + q   x   + O(x  )

%::CFPS(x,QF UP(q,RN))

                           1
(55) ---------------------------------------------
                            q*x
     1 - -----------------------------------------
                             3
                           (q  - q)x
         1 - -------------------------------------
                                 5
                                q x
             1 - ---------------------------------
                                7    3
                              (q  - q )x
                 1 - -----------------------------
                                       9
                                      q x
                     1 - -------------------------
                                      11     5
                                    (q   - q )x
                         1 - ---------------------
                                           13
                                          q  x
                             1 - -----------------
                                          15     7
                                        (q   - q )x
                                 1 - -------------
                                             17
                                            q  x
                                     1 - ---------
                                         1 - etc
```

7. Concluding Remarks

We have viewed lazy evaluation and self reference of data as particular techniques for infinite structures and we have shown how these techniques are particularly powerful when used together.

Scratchpad II provides the basic requirements for manipulating infinite structures: the ability to include programs as parts of composite data objects and the ability to create and modify self-referential data objects. These basic facilities have been used to build a variety of abstract data types which provide logically infinite structures.

The additions so far have been to build a number of domains so that infinite sequences (streams), power series, decimal expansions and continued fractions may be treated as first class citizens.

It should be possible in the future to enter differential or recursion equations that define new power series in terms of existing ones as suggested in the example for exp in section 5.1.

Bibliography

1. H. Rutishauser [1954], *Der Quotienten-Differenzen-Algorithmus,* Z. Angew. Math. Physik 5 233-251.
2. H.B. Curry and R. Feys [1958], *Combinatory Logic* North Holland, Amsterdam.
3. P. Henrici [1977], *Applied and Computational Complex Analysis, Volume 2,* John Wiley & Sons.
4. R.D. Jenks and B.M. Trager [1981], *A Language for Computational Algebra,* Proc. 1981 ACM Symposium on Symbolic and Albebraic Computation.
5. H. Abelson and G. Sussman (with J. Sussman) [1985], *Structure and Interpretation of Computer Programs,* The MIT Press, Cambridge Mass.
6. R.D. Jenks, R.S. Sutor and S.M. Watt [1986], *Scratchpad II: An Abstract Datatype System for Mathematical Computation,* RC 12327, IBM Research.