492 (simultaneous equations) We are given string variable X whose n items are rational, and a string of n functions f_i , each of which takes n rational arguments and produces a rational result. Assign to X a value satisfying

$$\forall i \colon 0,..n\cdot X_i = f_i@X$$

or, spreading it out,

$$\begin{array}{lcl} X_0 & = & f_0 \;\; X_0\, X_1 \cdots X_{n-1} \\ X_1 & = & f_1 \;\; X_0\, X_1 \cdots X_{n-1} \\ & \vdots & \\ X_{n-1} & = & f_{n-1}\, X_0\, X_1 \cdots X_{n-1} \end{array}$$

In other words, find n simultaneous fixed-points. Assume that a repetition of assignments of the form $X_i{:=}\, f_i\; X_0\; X_1 \cdots X_{n-1}$ will result in an improving sequence of approximations until the value of X is "close enough", within some tolerance. Function evaluation is the time-consuming part of the computation, so as much as possible, function evaluations should be done concurrently.

After trying the question, scroll down to the solution.

§ Let X be a string-valued interaction variable. The usual solution is to create one process for each function, which performs its assignment repeatedly. Consider the single assignment

$X_0 := f_0\, X_0\, X_1 \cdots X_{n-1}$

notice that it may not even satisfy equation 0, since it may change the value of X_0 and hence $f_0\, X_0\, X_1 \cdots X_{n-1}$. Worse than that, it may cause some other equation which was satisfied to become unsatisfied. So we now have the difficult problem of distributed termination detection. And these assignments cannot be placed in parallel anyway because they each change a variable or string item that the other assignments use.

Here is a solution without those problems. Let *solve* be the problem of solving all the equations.

$solve \;=\; \forall i{:}\ 0,..n\cdot X'_i = f_i@X'$

Here and throughout this exercise, we use the equality comparator between rationals to mean "close enough", and we assume that the system of equations is "close enough" to being solved just when each equation is "close enough". Let Y be a string variable to remember the result of the function evaluations, and let j be a natural variable for indexing.

new Y: $n{*}rat\cdot$ **new** i: $nat\cdot$

Define *solverest* to solve the equations from j onwards.

$solverest \;=\; \forall i{:}\ j,..n\cdot X'_i = f_i@X'$

Now we refine *solve* as follows.

```
$solve \;\Leftarrow\; j{:=}\,0.\ \ solverest.\ \ j{:=}\,0.\ \ check$
$solverest \;\Leftarrow\;$ if $j{=}n$ then $ok$ else $Y_j{:=}f_j@X \,\|\, (j{:=}\,j{+}1.\ \ solverest)$ fi
$check \;\Leftarrow\;$ if $j{=}n$ then $ok$ else if $X_j = Y_j$ then $j{:=}\,j{+}1.\ \ check$ else $X{:=}\,Y.\ \ solve$ fi fi
```

We are evaluating the functions concurrently, as required. Then we check, one by one, if the equations are close enough. If we find one that isn't, we update all the X values, and restart.

If we had a concurrent **for**-loop, we could refine as follows (note the use of a bunch, rather than a string, for the **for**-parameter, to indicate concurrency):

```
$solve \;\Leftarrow\;$ for $j{:=}\,0,..n$ do $Y_j{:=}f_j@X$ od.  $j{:=}\,0.\ \ check$
```

Here is a more interesting solution, again using a parallel **for**.

```
$solve \;\Leftarrow\;$ for $i{:=}\,0,..n$ do new $y{:=}f_i@X.$  if $X_i = y$ then $ok$ else $X_i{:=}\,y.\ \ solve$ fi od
```

For each equation concurrently, evaluate the function, and see if that equation is satisfied. If it is, do nothing; if not, make one assignment, and then restart the entire program. NO GOOD: WE CAN'T PARTITION THE VARIABLES BECAUSE EACH PROCESS CALLS *solve* . PRESSING ONWARD ANYWAY For the function evaluation $f_i@X$, there is no need for the items of X to be read all at the same time. There is an explosion of calls to *solve* : the body of the concurrent **for** is n processes, and each process may call *solve* resulting in n^2 processes, each of which may call *solve* resulting in n^3 processes, and so on. Soon the number of processes will exceed the number of processors available to execute them. The implementation of our programming language requires a scheduler that keeps track of waiting processes, and assigns them to processors when they become available. The explosion can be eliminated as follows.

Specification S is called <u>idempotent</u> if $(S.S = S)$. In words, S is idempotent if executing it twice in sequence has the same effect as executing it once. For example, sorting is idempotent. For each idempotent process P , the scheduler maintains a binary variable w with the informal meaning " P is awaiting a processor for execution"; its initial value is $\bot$. When P is called, it may already be awaiting execution from a previous call; if so, there is no point in calling it again because it is idempotent; if not, it must be called, so the scheduler indicates that it is now awaiting execution. When P is called, the scheduler executes

```
    **if** $w$ **then** $ok$ **else** $w{:=}\top$.  place $P$ in the queue of waiting processes **fi**
```

When P reaches the front of the queue and a processor becomes available, the scheduler executes $w{:=}\bot$ to indicate that P is no longer waiting, and gives it to the free processor.

In this example, *solve* is idempotent, as is the body of the concurrent **for** , which we now name *process i* for reference.

```
    $solve \Leftarrow$ for $i{:=}\ 0,..n$ do $process\ i$ od
    $process\ i = $ new $y{:=} f_i@X$.  if $X_i = y$ then $ok$ else $X_i{:=}\ y$.  $solve$ fi
```

If there are n processors, and the scheduler always assigns *process i* to processor i , then there are never two copies of *process i* running at once. This implementation prevents an explosion of waiting processes, and is an efficient solution to the problem of solving simultaneous equations. THE SOLUTION WORKS IN PRACTICE EVEN THOUGH IT DOES NOT WORK IN THEORY.