Appendix 5 - Efficient Coding

This section gives some ideas about how to get the most speed and memory efficiency out of ReWrite code.

• Be careful about extra copies of structures

Because of the feature of unique references (a copy of a structure for each reference to it), it is easy to get ReWrite spending most of its time needlessly copying and destroying lists. It is possible to avoid this, possibly at the expense of somewhat more convoluted code (see the nprime example).

Another example is this (poorly written) function to insert an element into the nth position of a list:

```
insertnth[x,0,l] -> x,.l;
insertnth[x,n,l] -> car[l],.insert[x,n-1,cdr[l];
```

This will make a copy of the list `l` and destroy it n times - most inefficient.

A better version would be:

```
insertnth[x,0,l] -> x,.l;
insertnth[x,n,a,.rest] -> a,.insert[x,n-1,rest];
```

This is not much of a problem with atoms - they are still copied and destroyed, but this is very quick (see declaring types below).

A later version of ReWrite may spot some of the obvious places where this is unnecessary, and improve the code internally.

• How to avoid a 'stack full' error

In the example above,

```
insertnth[x,0,l] -> x,.l;
insertnth[x,n,a,.rest] -> a,.insert[x,n-1,rest];
```

each function call goes deeper onto the stack, since the outside function on the right hand side is `list` not `insertnth`, so tail recursion cannot be used. This means that `insertnth` would fill up the stack if it was used with a long list and large n. It is possible to rearrange this to make tail recursion possible.

```
insertnth[x,n,l] -> insertnthx[x,n,,l];

(* insertnthx has an extra parameter to store the list elements
```

```
that will appear to the left of x *)

insertnthx[x,0,l,rest] -> .l,x,.rest;
insertnthx[x,n,l,a,.rest] ->
          insertnthx[x,n-1,.l,a,rest];
```

This allows tail recursion to work, and very little stack space will be used. The basic technique here is to have an extra parameter to collect the 'bits done so far' then apply them all at once, rather than one at a time.

Unlike lisp (which is slow with the appending of lists), ReWrite appends and prepends to lists at the same speed, so both versions of `insertnth` are nearly equally efficient (for speed).

• Use conditions sparingly

If something can be done with either a condition of a match, the match is much clearer and faster. A problem with conditions is that all parameters used in them need to have copies made of them, because they may not damage the argument list until a particular rule has been chosen.

In the factorial example,

```
factorial[0] -> 1;
factorial[n] -> n * factorial[n-1];
```

（Time taken to find factorial[12] 10000 times on LC III: 162 ticks)

the 0 was included as part of the match. The following would be equivalent:

```
factorial[n]::n=0 -> 1;
factorial[n] -> n * factorial[n-1];
```

(Time taken to find factorial[12] 10000 times on LC III: 389 ticks)

except that ReWrite handles matching more efficiently than conditions, so the second version will be much slower.

• Declare types if known

ReWrite does not insist on types (unless they are being used as part of a match). However, type checking is very quick, and can save a lot of work as detailed below, so are worth using if a parameter is only supposed to have one particular type anyway.

The compiler can use type information to produce more efficient code in three ways:

• Without types, any copy, compare, destroy has to call a subroutine to do so on arbitrary types. With types, there is quite often a shortcut that can be used, and a small piece of code to do just the desired operation can be put inline.

• Some functions (noted with a * to the left in the Function List Appendix) can be optimised to be inline if the type is known.
For example:
`splice` produces inline code if it is know that it is working on a list.
`mult` can be greatly improved if both the arguments are known to be integers - there is a single machine code instruction to do this, and often the cell structure can be dispensed with completely, leading to very fast code. Also in this case, the result of `mult` is taken to be an integer.

• If all the functions in an expression are inline, as described in the last point, a whole expression can be optimised. This can be particularly useful for conditions on rules. Normally conditions are slow (see above), but if enough is known about the functions and types in a condition, they speed up dramatically.

For example:

```
factorial[0] -> 1;
factorial[n:int] -> n * factorial[n-1];
```

(Time taken to find factorial[12] 10000 times on LC III: 97 ticks)

and

```
factorial[n:int]::n=0 -> 1;
factorial[n:int] -> n * factorial[n-1];
```

(Time taken to find factorial[12] 10000 times on LC III: 107 ticks)

run at about the same speed (but the first is still clearer).

To summarise:

Disadvantages of typing parameters:
- Slightly more verbose code;
- Small overhead in type checking (typically about 2 machine code instructions).

Advantages of using types:
- Code can be clearer;
- The compiler can make some safe assumptions and produce much better code.
- Errors might be spotted earlier and more easily.

For example with:
`square[x:int] -> x*x;`
the type `:int` means that any error will be a mismatch in `square` rather than `mult`.

• Use the built in functions where possible

Because speed has been one of the main, objectives when code has been written, the internal functions are mostly designed to run quickly. For example, if you are using sets that contain non-negative integers, use the `oset***` functions.

One thing to note when accessing lists: the treetake function is significantly faster than the nth function for anything but very small values of n, so

`treetake[n,lis]` is much more efficient than `nth[n,lis]`.

This is because `treeslicesplice` has been direct coded, whereas `nth` is written in ReWrite.

• Use splice sparingly, but whenever useful

The `splice` operator is very efficient when it is used on the last pattern in a rule lhs or list, for example:

```
test[1,.r1,.r2] -> true;
```

will be very efficient.

The `splice` operator will be somewhat less efficient when used in other situations, as a match will have to be searched for, but usually still more efficient that any alternative coding that involves searching.

For example,

```
ismember[x,._,x,._] -> true;
ismember[x,y:lis] -> false;
```

while still requiring a search in the first rule, is still about 2-3 times as efficient as the following:

```
ismember[x,] -> false;
ismember[x,x,.rest] -> true;
ismember[x,y,.rest] -> ismember[x,rest];
```

So: use the `splice` operator whenever it is useful, but remember that it does have a time overhead. An algorithm that avoids searching altogether (if possible) will be faster (for example, pull a list to bits from the start rather than from the end).