Appendix 4 - Implementation

This section details some of the inner workings of ReWrite. Knowledge of this is not required to use ReWrite, but it gives some idea of why things are set up the way they are, and how ReWrite compares to other languages. It also discusses some aspects of efficiency (see next section).

Note: this section only applies to ReWrite 0.2.x - it may not be relied on for future versions because they may be different. Also, some of this has been somewhat simplified.

Storage

Everything in ReWrite is stored in cells. A cell has the following structure:

here type is

- 0 - data is a pointer to a list,
- 1 - data is a 32 bit integer,
- 2,3 are unused,
- 4 - data is a boolean,
- 5 - data is `null` (no data),
- 6 - data is a character,
- 7 - data is a token.

An integer is stored the following way:

list is stored in the following way:

Since the first and last elements of the list are both stored, concatenation of lists becomes easy, as the last element does not need to be searched for. For example, appending to a list (typically a slow operation in Lisp) is just as quick as prepend.

The cells are all stored in freelists, so a lot of memory management complications don't arise.

Another feature of ReWrite is that of unique references - all data structures are stored strictly as trees; i.e. more than one pointer to the same structure is not allowed. This has several major consequences:

- List surgery becomes completely safe, since no other structures will be interfered with. In ReWrite, all list operations are done surgically;

- The code can keep track of what values are in use, since if a particular function has no further need of a value, it can safely dispose of it. All functions in fact keep track of this, and any value is disposed of as soon as it is no longer required (see below). As a result, there is no need to perform any garbage collection.

- The disadvantage is that much more copying of structures needs to be done than would otherwise be necessary, and this can seriously impair performance. (Also see the section on coding efficiency).

An example of this last point is the following function that prepends the size of a list onto a list (for example for conversion to a Pascal string):

```
pstring[x:lis] -> length[x],.x;
```

The list `x` needs to be copied and passed to the `length` function, which counts the elements of `x` then destroys the list again. This turns what could otherwise be a quick function into something much less efficient.

Evaluation

There are several steps taken in function evaluation. Presented here is a simplified idea of what happens. In the interests of simplification, all of the optimisation that is done has been omitted. For example, in practice operations like arithmetic functions are all done inline wherever possible to save the overhead of list construction and a function call.

- The following is set up at the before a function is called:
  - all the arguments to the function are in a list to be passed to the function (which we will call `arglist`);
  - there is a list (which we will call `resultlist`) to which the result of the function call will be appended.

- When the function is called, there is a check for:
  - whether the free stack space is running low;
  - whether the user has told the program to abort.

If either of these conditions are met, then the program is aborted.

• The argument list is matched with the parameter list for the function (including any necessary condition checks). Although it is easiest when looking at how code works to imagine it done on a rule by rule basis, ReWrite may do checks that apply to several rules - it remembers what checks have succeeded and failed previously and never repeats them.

In the following example (which will be used throughout this section), the second and third rules have only the condition separating their left hand sides, so if the match for the second rule fails only on the condition, the match for the third rule will automatically succeed.

```
primex[lis,s,0,k:int] -> k-1;
primex[lis,s,n:int,k:int]::k>s*s -> primex[lis,s+1,n,k];
primex[lis,s,n:int,k:int] -> primex2[primetest[s,k,,.lis],s,n,k];
```

The pseudo-code for the match for this is something like (where `expr.n` refers to the `n` th element of the `expr` - this notation is not supported in ReWrite):

```
if arglist.1 exists then
  if arglist.2 exists then
    if arglist.3 exists is an integer then
      if arglist.4 exists is an integer then
        if arglist.3 = 0 then
          do right hand side of rule 1;
        else
          if arglist.4 > arglist.2 * arglist.2 then
            do right hand side of rule 2;
          else
            do right hand side of rule 3;
      else
        fail;
    else
      fail;
  else
    fail;
else
  fail;
```

The mechanism used for checking conditions is similar to that used for evaluating the right hand side with one exception - all arguments are copied, instead of just the duplicates (see below).

It is important to note that while doing the matching and condition checking, `arglist` is not tampered with at all. This is because if the match fails, `arglist` may be needed somewhere else - either as part of an error report, or there may be another version of the function later down the execution chain that can be tried.

• Once a successful match has been found (i.e. execution is now committed to a particular rule), any parts of arglist that will not be needed are destroyed. From the example above:

`primex[lis,s,0,k:int] -> k-1;`
`lis`,`s`,`0` and the list structure surrounding `arglist` are destroyed.

`primex[lis,s,n:int,k:int]::k>s*s -> primex[lis,s+1,n,k];`
Only the list structure surrounding `arglist` is destroyed.

`primex[lis,s,n:int,k:int] -> primex2[primetest[s,k,,.lis],s,n,k];`
Only the list structure surrounding `arglist` is destroyed.

- Execution of the right hand side is done in the following way:

  * If an argument is found, one of two things happen:
  - If this is the last use of the argument, it is appended onto the `resultlist`;
  - If this is not the last use, then a copy of the argument is appended onto `resultlist`.

  * If a function call is found, the following sequence is done:
  - `resultlist` is saved somewhere (on the machine stack);
  - `resultlist` is set to ;
  - the arguments of the function are executed in the normal way;
  - `resultlist` is put into `arglist`, and the old `resultlist` restored;
  - the function is called (or if it at the top level of the rhs, jumped to).

This method of evaluating the rhs has the effect that a function can return any number of results - they are all just appended to `resultlist`. Also note that tail recursion is supported (in fact required - an early version that had a bug and didn't do tail recursion properly required a 2 megabyte stack).

In the above example,

`primex[lis,s,0,k:int] -> k-1;`

- `resultlist` is saved;
- `resultlist` is set to ;
- `k` and `1` are appended to `resultlist`, `resultlist` is put into `arglist`, and the old `resultlist` restored;
- the function `minus` is jumped to with `arglist` = `k,1`, thus having used no space on the stack.

`primex[lis,s,n:int,k:int] -> primex2[primetest[s,k,,.lis],s,n,k];`

- `resultlist` is saved and set to ;
- `resultlist` is saved and set to ;
- a copy of `s` and a copy of `k` are appended to `resultlist`,
- is appended to arglist (actually another function call, but compressed for clarity);
- each of the elements of lis is appended to resultlist (technically another function call, but this is

optimised out);
`resultlist` is put into `arglist`, and the old `resultlist` restored;
- the function `primetest` is called;
- `s`, `n` and `k` are appended to resultlist;
`resultlist` is put into `arglist`, and the old `resultlist` restored;
- `primex2` is jumped to.

Compiling

The above describes the method used to evaluate rules. There are several steps required to turn it into machine code:

- Internal macro's (all those things beginning with AsmM in the rulelist) are used to expand something like the pseudo-code given above into a lump of intermediate code, which is sort of like a low level C or a high level assembly code program. This does not have any data structures, but is high enough level to be (mostly) machine independent. This intermediate code can be coded directly, but is unstable in its current version and is therefore undocumented. It may be documented in ReWrite 0.3.
  The intermediate code is designed to be easy to port, and all the low level routines are written directly in this. ReWrite itself contains no direct machine code (although there is some glue code to connect it to the Pascal interface);

- The intermediate code then goes through a register folder which assigns a register of the 680x0 to each of the variables in the code;

- The resulting code is then converted to instructions for the 680x0;

- The resulting code is saved to disk with the prefix l.name ;

- The last task is to link all the function calls up with the functions. This is done over the whole code when everything has been compiled.