RVM-FORTH, a Reversible Virtual Machine

Bill Stoddart
School of Computing and Mathematics
University of Teesside, North Yorkshire, U.K.

November 10, 2004

Abstract

RVM-FORTH is a version of Forth designed as a target execution platform for the compilation of a proposed experimental sequential programming language BB. Novel aspects include local variables with nested scopes, support for sets, and reversible computing mechanisms based on the primitive concepts of guard, choice and expression transformation. RVM-FORTH has support for locating and browsing operation definitions and for communicating with the the Unix shell. The ANS Forth draft Standard is available as on-line help. This paper describes the RVM-FORTH system in terms of a brief user guide.

key words: Forth, Local Variables, Sets, Reversibility, Expression Transformers

1 Introduction

Forth is a set of named operations for a stack based virtual machine. These operations provide the elements of an operating system, an interpreter and a compiler. Forth often runs "stand alone" without the need for an operating system. The version described here, however, runs under Unix. It consists of the following components:

- The RVM-FORTH nucleus. A basic virtual machine which provides primitive programming capabilities and is capable of extending itself by compiling new definitions. The nucleus is defined in a set of files which describe its operations in a mixture of Forth and structured assembler language.
- The Meta-Compiler. A program which reads the RVM-FORTH nucleus files and compiles them into a monolithic gnu assembler file.
- C functions for those parts of the nucleus written in C.
- A script to build the RVM-FORTH nucleus..
- The RVM-FORTH utilities layer. These are files containing Forth code. They are compiled by the nucleus when it commences execution.
- Example programs and test programs.

2 Installation

We describe the installation procedure in a way that gives an overview of the system.

RVM-FORTH comes as a zip file. The installation procedure assumes super-user privileges.[1]

Copy the distribution file (say `rvm.zip`) to `/usr/share`. Then `cd` to this directory and unzip with:

```
unzip rvm
```

This will create a directory rvm with a subdirectory of the same name. Now cd to this directory:

```
cd rvm/rvm
```

You are now in the "installation" directory of RVM. This contains the following sub-directories and programs:

- `nucleus/` containing the RVM-FORTH source code files that describe the RVM-FORTH nucleus. RVM-FORTH source files have the extension `.r`
- `metacompiler/` containing the source code and executable for the metacompiler (which is written in C) along with a make file.
- `sets/` contains the RVM-FORTH sets package (written in C) and a make file.
- `ccalls/` contains funtions to support i/o and signal handling (written in C) and a make file.
- `utils/` contaning the RVM-FORTH utilities layer files
- `examples/` containing the provided example programs
- `tests/` containing the provided test programs
- `mcomp` The script to build the RVM nucleus and copy the resulting executable file to `/usr/bin`.

Now, still from the installation directory `/usr/share/rvm/rvm` invoke the meta compilation script:

```
./mcomp
```

This completes the installation. The meta-compilation procedure produces the file `rvm_home.r` which contains the definition:

[1] A minor variation is needed to install RVM-FORTH in your home space without being a super user. Follow the same instructions but unzip the distribution file in your home space in a directory of your choice, with path x say. cd to x/rvm/rvm and edit the last command of the script mcomp to copy the executable to a directory of your choice instead of to /usr/bin. Then run the script ./mcomp

```
: RVM-HOME ( -- s ) C" /usr/share/rvm/rvm/" ;
```

i.e. an operation which returns a counted string giving the location of RVM's installation directory. RVM uses this to locate the Forth source programs which form part of the RVM system. NB: Users of the `csh` and `tcsh` shells will need to do a `rehash` before running the command.

3 Running RVM-FORTH

Now leave super-user mode and create and `cd` to the directory you will use for RVM-FORTH: say:

```
mkdir rvm; cd rvm
```

Now create a file my.r in this directory containing the following Forth definition:

```
: HW ." HELLO WORLD " ;
```

invoke Forth with:

```
RVM_FORTH HI L my.r
```

The rest of the line following `RVM_FORTH` is passed to the Forth interpreter. The command `HI` loads the utility layer. `L my.r` loads the file `my.r`.

Now we can try running the new program:

```
HW<enter>HELLO WORD ok
```

`L` interprets the following token (terminated by space) as a path relative to the current working directory. There is also a system load command `SL` which interprets the following token as a path relative the the RVM-FORTH installation directory. An example usage is:

```
SL examples/sendmory.r<enter>
T runs the puzzle: SEND+MORE=MONEY ok
```

4 Exploring the System

4.1 Locating and browsing source code definitions

We can locate the source code definition for any Forth operation. E.g assuming the file `sendmory.r` has been loaded as described above:

```
LOCATE T<enter>/usr/share/rvm/rvm/examples/sendmory.r  line 70 ok
```

And assuming nedit is installed we can view the definition of an operation. E.g. with `SEE T`. This runs nedit in read only mode on `sendmory.r`, opening the file at the definition of `T`, which we see has the following definition.

```
: T ( -- ) SOLUTIONS REPORT ;
```

`SOLUTIONS` finds the solutions to the problem and leaves them, in the form of a set, on the top of the stack. `REPORT` prints the solutions. We will return to this example application later.

In Forth it is quite acceptable to re-define a word. `ALL <name>` will list all the definitions of `<name>` in the current search path.The most recent definition is found and listed first.

```
ALL HI<enter>
utils/hi.r  line 90
nucleus/cold.r  line 52 ok
```

`SEEALL` reads the following token, searches for entries for that word, and will open an edit screen for each copy found. In response to `SEEALL HI` the editor opens the system files `nucleus/cold.r` and `utils/hi.r` at the following definitions.

```
: HI ( -- ) C" utils/hi.r"  SLOAD-FILE ;
: HI ;
```

The original definition of `HI` is a nucleus word which loads the utility layer definitions. In the utilities layer it is redefined to do nothing. Overall, `HI` ensures that the RVM-FORTH utilities layer is loaded.

The techniques described above for locating the source code for an operation from the operation name are not adequate for operations that use "vectored execution". For example to allow for redirection of keyboard input the ANS Forth word[2] `KEY`†is defined as:

```
: KEY ( -- c ) 'KEY @ EXECUTE ;
```

Here the execution token for the actual code to be executed is stored in the variable `'KEY`. To find the name of the actual operation we have to use `>NFA` to convert this token into a name field address. The actual operation name can be printed as follows:

```
'KEY @ >NFA COUNT TYPE<enter>IKEY ok
```

4.2 Wordlists

The Forth dictionary is organised as a number of wordlists. `.WORDLISTS` prints a list of all wordlists present in RVM-FORTH. `.ORDER`†tells us which of these will currently be searched (the first listed is the first searched).

```
.ORDER<enter> FORTH ROOT ok
```

The `COMPILER` wordlist holds words that are only used during compilation, and we can add it to the search order with `ALSO`†`COMPILER`. We can see exactly how this works be doing it in two steps:

[2]When an ANS Standard Forth word first appears in this text it is decorated with a dagger. The function of these words can be looked up in the on line help system which we will describe presently

```
ALSO .ORDER<enter> FORTH FORTH ROOT
COMPILER .ORDER<enter> COMPILER FORTH ROOT
```

The `COMPILER` wordlist includes program control structure words, such as `IF`†, `ELSE`†, and `THEN`†.

We can remove the last wordlist added to the search order with `PREVIOUS`†.

```
PREVIOUS .ORDER<enter> FORTH ROOT ok
```

`EVERY` allows us to locate words which are not in the current search path. For example with the search order of FORTH and ROOT, `EVERY IF` will generate the following form of output:

```
EVERY IF<enter>
COMPILER /usr/share/rvm/rvm/utils/syntax.r  line 81
COMPILER /usr/share/rvm/rvm/utils/control_structures.r  line 3 ok
```

Wordlsts can be created by the defining word `VOCABULARY`. To set the wordlist into which operations will be compiled we use `DEFINITIONS`†, which makes the first wordlist in the search order into the "compilation wordlist". The following classic example taken from an English/French Dictionary application. For brevity our dictionaries will contain the translation for just a single word!

```
VOCABULARY FRENCH   ALSO FRENCH DEFINITIONS
: AMI ." FRIEND " ;
( more entries can be added here )
PREVIOUS DEFINITIONS

VOCABULARY ENGLISH  ALSO ENGLISH DEFINITIONS
: FRIEND ." AMI " ;
( more entries can be added here )
PREVIOUS DEFINITIONS ALSO
```

We use our dictionary as follows:

```
FRENCH AMI<enter>FRIEND ok
ENGLISH FRIEND<enter>AMI ok
```

5 Compiler extensions: immediate words

The RVM-FORTH nucleus has a simple compiler which is able to define new operations by the sequential composition of existing operations. At this stage it lacks almost all the other features normally associated with a high level sequential programming language. It has no program control structures, no local variables... These capabilities are provided by operations defined in the utilities layer.

Program control structure words such as `IF ELSE THEN` compile branch instructions, record the locations of branch destinations and unresolved branch offsets, and resolve branch offsets. They are tagged with an `IMMEDIATE`†attribute that causes them to be executed during compilation rather than compiled as part of the current definition..

Using the exploration facilities described above, we found that `IF` has two definitions. The first defined performs the required compilation function:

```
: IF ( -- addr ) POSTPONE ZBRANCH MARK  ; IMMEDIATE
```

Here `POSTPONE`† `ZBRANCH`†compiles a virtual machine branch on zero instruction. When subsequently executed, this will remove the top stack item and if it is zero will branch over the IF clause in the control structure. However, when `IF` is executed, it does not know where this branch destination will be. `MARK` leaves the address of the branch instruction on the stack, so the branch destination can be amended by a following `ELSE` or (where there is no ELSE clause) `THEN`.

In the same file are corresponding definitions for `ELSE` and `THEN` which together implement the Forth IF ELSE THEN control structure. These definitions operate correctly and efficiently for valid source code, but if used incorrectly will not produce error reports. To impose syntax checks and error reporting they are redefined. The redifinition of `IF` is:

```
: IF ( -- addr ) POSTPONE IF -IF- ; IMMEDIATE
```

`POSTPONE IF` refers to the previous definition of `IF` and causes it to be treated like a non-immediate command, that is to be compiled as part of the current definition rather than executed immediately. `-IF-` leaves a syntax token on a separate syntax stack, and this is checked by a subsequent `ELSE` or `THEN`.

The corresponding re-definitions for `ELSE` and `THEN` are in the same file.

6 On line help

The final committee draft of the ANS Forth Standard is available as on line help. `ANS HELP` prints relevant help information. The `ANS` package assumes the Mozilla browser is available in the current unix search path. You can however use any browser which will navigate to a file path given on the command line, as Mozilla does with e.g.

```
mozilla /usr/share/rvm/rvm/dpans/welcome.htm
```

To select a different browser edit the definition of `MY-BROWSER`, which returns the browser name to the stack as a character array and count.

7 Invoking Unix commands

RVM-FORTH has a simple wordset for communicating with the Unix shell.

`SYS` is used in interpret mode, and passes the rest of the input line to the unix shell: e.g.

```
SYS pwd<enter> /home/fred/rvm ok
```

Words used for communicating with the shell within compiled definitions are defined in the `SYSMESS` vocabulary.

`SYSMESS-INIT` initialises the system message buffer.

`+MESS ( addr n -- )` adds the message consisting of n characters at addr to the system message buffer.

`TELL-UNIX` passes the text in the system message buffer to the Unix shell.

`ASK-UNIX ( -- addr n )` passes the text in the system message buffer to the shell and returns the shell's response as a string.

The following definitions illustrate the use of these words.

```
ALSO SYSMESS
: T1 ( --, tell unix shell to execute the pwd command )
  SYSMESS-INIT S" pwd" +MESS TELL-UNIX ;

: T2 ( -- addr n, ask the shell for the current directory )
  SYSMESS-INIT S" pwd" +MESS ASK-UNIX ;

: T3 ( p-- addr n, parse next token, terminated by a full stop,
and ask the unix shell for its response to this unix command )
 SYSMESS-INIT  [CHAR] . PARSE  +MESS  ASK-UNIX ;
PREVIOUS
```

You can try them out like this:

```
T1<enter> /home/fred/rvm ok
T2 TYPE<enter> /home/fred/rvm ok
T3 pwd. TYPE<enter> /home/fred/rvm ok
```

Note: we have made use of ANS Forth Standard words `S" [CHAR] PARSE` and `TYPE`. You can check the functionality of these words using the online help facility with:

```
ANS WORDS<enter> mozilla /usr/share/rvm/rvm/dpans/dpansf.htm&
request to launch browser issued ok
```

This points the browser at an ascii ordered list of links to Standard Forth word descriptions (known as "glossary entries").

In fact (exercise) you could now write your own Forth definition to do this using the `SYSMESS` commands.

Solution:

```
ALSO SYSMESS
: FW ( open browser at list of Forth words)
  SYSMESS-INIT  MY-BROWSER +MESS RVM-HOME +MESS
  S" dpans/welcome.htm" +MESS  TELL-UNIX ;
PREVIOUS
```

8 Defining words

The nucleus level of RVM-FORTH provides two "defining words". These are operations which can be used to add new words to the Forth dictionary. These are[3] $\boxed{:}$ and `CONSTANT`. An example constant definition is.

```
7 CONSTANT FRED
```

Using `HERE`†, which returns the next free location in Forth's data memory. we define in the utilities layer the defining word `CREATE`†

```
: CREATE HERE CONSTANT ;
```

`ALLOT`†, removes an integer from the stack and advances the memory allocation pointer that no. of memory units (bytes for the i386). `CELL`†is a constant giving the no. of memory units in a cell. With these we define the defining word `VARIABLE}`†

```
: VARIABLE CREATE CELL ALLOT ;
```

Suppose we want to define some words which return constant strings. We can define:

```
: MSG CREATE [CHAR] . WORD  ,CSTRING ;
```

`[CHAR]`† `.` compiles code that returns the character code for '.'.

`WORD`†takes a delimiting character from the stack (in this case a full stop) and scans the following text from the input stream until it finds this character (or end of line/file). It places the scanned text in a buffer, where the first byte of the buffer holds a count giving the string length, and subsequent bytes hold the characters (a form known as a "counted string"). It returns the address of the buffer.

`,CSTRING` takes a counted string address from the stack and compiles it into the first free locations in Forth's data memory.

We use `MSG` as follows:

```
MSG M1 HELLO WORLD .
```

And we can type the messages using:

```
M1 COUNT TYPE<enter> HELLO WORLD ok
```

Words defined with `CREATE` may be modified with `DOES>`†. When used within a compiled definition, `DOES>` exits from the definition after appending the action performed by the code which follows it to the most recently defined word. For

[3]We usually distinguish Forth words, Unix commands etc from surrounding text by presenting them in typewriter font. However, since Forth words can be any sequence of characters, including a single punctuation mark, we sometimes present them as $\boxed{.}$, $\boxed{,}$ etc.

example:

```
: +PRINT ." appending COUNT TYPE to most recent definition "
  DOES> COUNT TYPE ;
```

Let's define a word to be modified by `+PRINT` and modify its behaviour.

```
MSG M2 GOODBYE TO ALL THAT . +PRINT
```

```
M2<enter> GOODBYE TO ALL THAT ok
```

We generally use `CREATE` and `DOES>` within the same defining word. E.g.

```
: ANNOUNCE CREATE [CHAR] . WORD ,CSTRING DOES> COUNT TYPE ;
ANNOUNCE ANN1 HELLO WORLD .   ANNOUNCE ANN2 GOODBYE TO ALL THAT .

ANN1<enter> HELLO WORLD ok
```

When used in this way, code between `CREATE` and `DOES>` defines the compile time action for a class of words, and code following `DOES>` defines the run time action for the class. Communication between compile time and run time takes place via the address passed to the `DOES>` clause, which is the address where the compile time actions were performed.

The online help package (described earlier) uses `CREATE .. DOES>` along with other techniques we have discussed such as word lists and communication with the Unix Shell. To review this application enter `SEE ANS`.

9 Variables

32 bit global variables may be defined in Forth with either `VARIABLE`†or `VALUE`†. The difference between these forms is that a `VARIABLE` just pushes its address to the stack. Accessing or storing the value of the variable is then done by $\boxed{\texttt{@}}$†or $\boxed{\texttt{!}}$†. A `VALUE`[4] by constrast, is responsible for accessing its own data. It will either return its value to the stack or take on a new value from the stack. Which of these behaviours it exhibits is controlled by the presence or otherwise of the prefix[5] `to`.

```
100 VALUE X
```

```
X . 77 to X  X .<enter>100 77 ok
```

Arrays and array references are defined by `VALUE-ARRAY`. Arrays are indexed from 1. Examples:

[4]The definition of VALUE provides an interesting example of the use of CREATE .. DOES>.. VALUEs in RVM-FORTH are immediate words which, when invoked in compiled definitions, decide at compile time which action to perform.

[5]This prefix syntax is at variance with Forth's normal use of postfix. We use lower case $\boxed{\texttt{to}}$ to emphasise this. The ANS Standard uses $\boxed{\texttt{TO}}$.

```
2 VALUE-ARRAY T1   2 VALUE-ARRAY T2
10 to 1 of T1   20 to 2 of T1

size of T1 .  size of T2 .<enter> 2 2 ok
1 of T2 .  2 of T2 .<enter> 10 20 ok
```

The use of prefixes gives rise to an occasional need for brackets to control which object a prefix applies to. E.g. given:

```
VALUE I    10 VALUE-ARRAY TABLE
```

we might want to assign 100 to position `I` in `TABLE`. The following would not work: `100 to I of TABLE`. The prefix `to` would be consumed by `I`. Instead we must write: `100 to << I >> of TABLE`

Reference arrays may be declared with `VALUE-ARRAY^`. Example:

```
T2 VALUE-ARRAY^ T3

1 of T3 .<enter>10 ok
```

9.1 Local variables

Within a compiled definition, local variables may be declared within an argument list bracketed by (: ... :) as shown in the following example. We calculate the createst common divisor of two numbers using Euclid's algorithm, in which the smaller of the pair is subtracted from the larger to give a new pair. This process is repeated until the two numbers are equal

```
: GCD0 ( n1 n2 -- n3, pre n1>0 & n2>0, post n3 = gcd(n1,n2) )
  (: VALUE X  VALUE Y :)
    BEGIN
      X Y <>
    WHILE
      X Y >
      IF
        X Y - to X
      ELSE
        Y X - to Y
      THEN
    REPEAT
    X
  1LEAVE ;
```

Values X and Y are initialised from the stack, X taking the value of n1 and Y the value of n2. `1LEAVE` specifies that just one item (the current top of stack) will be returned. `0LEAVE 1LEAVE 2LEAVE` and `3LEAVE` are available to specify from 0 to 3 returned values. Only one of these may be used per operation, with the syntactic pattern:

```
: <name> ... (: ... :) ... nLEAVE ... ;
```

VALUEs may be declared after the argument list. In this case the first of these VALUEs should be declared at a point where one additional item has been pushed to the stack by code subsequent to the argument list. It obtains its initial value from that item. A second may be declared when another value has been pushed and so on.

Static scopes for locals may be opened and closed with (SCOPE and SCOPE). The following example illustrates these features:

```
1 VALUE X
: T  2  3
 (: VALUE X  VALUE Y :)
   4 VALUE Y
   (SCOPE 5 VALUE X   X .  Y . SCOPE)
    X . Y .
  0LEAVE
  X .  Y . ;

T<enter> 5 4 2 4 2 3 ok
```

The above descriptions have been given in a way that is independent of whether stack frame items are removed from the parameter stack or accessed by indexing into the stack. However, we cannot make this design decision completely transparent, as it will show up in the DEPTH†of the stack and in use of the stack print operation $\boxed{\texttt{.S}}^\dagger$.

Local array variables may be declared with VALUE-ARRAY or VALUE-ARRAY^ respectively. Both forms expect to be passed the address of an array. They implement call by value (where the whole array is copied into local variable space) and call by reference semantics respectively. Examples can be found in the file tests/argtests.r

10 Sets

A set containing the integers 1, 2 and 3 can be expressed as `1 3 ..` or alternatively `INT { 1 , 2 , 3 , }`. A set containing the strings "tom", "dick" and "harry" can be written as: `STRING { " tom" , " dick" , " harry" , }`

The open set bracket must be preceded by an expression which gives the type of the elements. We restrict ourselves to homogeneous sets (every element of a set must be of the same type). A set occupies one stack location (which is a reference into the heap). We have operations for set union \/, intersection /\ and subtraction \. For set subtraction and some other set operations we discuss below, the order of arguments is important. In all cases they should be provided in the same order as the mathematical infix form.

POW takes a set and returns the set of all its subsets. PROD takes two sets and returns their cross product. Two sets can be tested for equality by SET=. The value of a set can be printed with .SET. Examples:

```
1 4 .. VALUE X  2 5 .. VALUE Y
STRING { " aa" , " bb" , } VALUE Z
```

```
X Y /\ .SET   X Y \ .SET<enter> {2,3,4} {1} ok
Z POW .SET<enter> {{},{aa},{bb},{aa,bb}} ok
Z 1 2 .. PROD .SET<enter> {(aa,1),(aa,2),(bb,1),(bb,2)} ok
1 5 ..  X Y \/ SET= .<enter> -1 ok
```

As well as listing elements within set brackets, we can generate them there with a program. The implementation of $\boxed{\texttt{..}}$ provides a good example. (Use $\boxed{\texttt{EVERY ..}}$ to check whether the current dispensation has different versions for interpret and compile mode).

Order is not important in sets. The order in which elements are printed need not correspond to the order in which they are added to the set:

```
STRING { " tom" , " dick" , " harry" , } VALUE NAMES
```

```
NAMES .SET<enter> {dick,harry,tom} ok
```

We use sets of pairs to express relations e.g. between names and telephone numbers:

```
STRING INT PROD { " bill" 2673 |->$,I , " frank" 4012 |->$,I ,
" rob" 4012 |->$,I , } VALUE TEL
```

`|->$,I` takes a string and an integer from the stack and returns a string integer pair. A nicer notation[6] would be `|->` but unfortunately we were not able, without incurring a run time performance overhead, to do without some information about what kind of elements a pair is being formed from. We need 16 different pair constructors to handle 4 basic classes of element: integers, strings, pairs and sets. In naming these operations we use the characters I, $, P and S in a systematic way. e.g. `|->$,S` constructs a pair from a string and a set.

The type of the elements in the set is given by the preceding postfix expressions STRING INT PROD. Conceptually we can think of STRING as the set of all strings and INT as the set of all integers. Their product would then be the set of all possible string/integer pairs. We call this a "maximal set". Any postfix expression made up of INT STRING PROD and POW (and representing a single value) will represent a maximal set. Maximal sets are types in our system.

We can index through a set with `@ELEMENT`. The following operation prints the contents of `TEL`. In the definition we make use of `CARD`, which returns the no. of elements in a set, `FIRST` and `SECOND`, which return the first and second elements of a pair, and `.AZ` which prints an asciiz string (the form of string used in our sets package).

```
: .TEL ( -- ) CR  TEL CARD 0
  DO
    TEL I @ELEMENT   DUP FIRST .AZ SPACE  SECOND . CR
  LOOP ;
```

We can obtain the domain of a relation (all left hand elements of its pairs) with

[6]There are two mathematical notations for ordered pairs, (a, b) and $a \mapsto b$.

`DOM`. .We can obtain its range (right hand elements) with `RAN`

```
TEL DOM .SET  TEL RAN .SET<enter> {bill,frank,rob} {2673,4012} ok
```

Tests for membership, subset inclusion, and strict subset inclusion are provided by `IN`, `<:` and `<<:`. The following test is to reveal whether "steve" is in the telephone book:

```
" steve" TEL DOM IN .<enter> 0 ok
```

Each person listed in the telephone book has only one entry. We call such a relation a function, and we can look up a telephone number using function application.

```
TEL " rob" APPLY .<enter> 4012 ok
```

We can obtain the inverse of a relation with $\boxed{\sim}$. The relational image of a set is given by `IMAGE`. To enquire which people have the same number as frank we can use:

```
TEL ~ INT { TEL " frank" APPLY , } IMAGE .SET<enter>
{frank,rob} ok
```

Domain and range restriction and subtraction are provided by `<|`, `<<|`, `|>` and `|>>`. Domain restriction takes a set of type x and a relation from x to y. It yields that part of the relation whose domain coincides with the set. If we define:

```
STRING { " frank" , " rob" , } VALUE STUDENTS
```

We can obtain that part of the telephone book which records entries for students with:

```
STUDENTS TEL <| .SET<enter> {(frank,4012),(rob,4012)} ok
```

We can add a single new entry to `TEL`:

```
TEL " steve 4395" |->$,I  ADD-ELEMENT  to TEL
```

We can add a set of new entries. Assuming they are held in a set `NEW` of the same type as `TEL`:

```
TEL NEW \/ to TEL
```

To perform updates on `TEL` is slightly more complex, and it will be worth defining a new set operation for "function override". Its mathematical definition (written with our ascii set symbols) is:

```
S <+ U = (DOM(U) <<| S) \/ U
```

We can think of `S` as a set containing our database expressed as a mathematical relation. `U` is another relation of the same type containing updates. These may be either new entries or updates. In the context of our telephone directory example it works like this: `DOM(U)` is the set of names which occur in the updates. `DOM(u) <<| S` consists of the entries in `S` for those names which do not occur in the updates. i.e. the database with the entries that are going to be updated deleted. `(DOM(U) <<| S) \/ U` adds the updates.

Exercise. Code the Forth operation `<+`

Solution: `: <+ ( s u -- s<+u ) DUP DOM ROT <<| \/ ;`

Exercise: Assuming VALUEs `TEL` and `UPDATES` hold a telephone directory and its updates, what Forth phrase will assign the updated directory to `TEL`?

Solution: `TEL UPDATES <+ to TEL`

10.1 Set types

From the given types INT and STRING, which conceptually represent the set of all integers and the set of all strings.[7], We can constuct additional types using `PROD` and `POW`. The type of a set is the maximal (largest) set to which it belongs. For instance 3 belongs to the set {1,2,3}, but the largest set to which it belongs is INT and that is its type. The largest set to which any set of integers belongs is INT POW. The largest set to which a string integer pair belongs is STRING INT PROD. The largest set to which a set of string integer pairs belongs is STRING INT PROD POW. This is the type of $\boxed{\texttt{TEL}}$ in the above examples.

Given any set types x y we can construct further types x POW, x y PROD, x y PROD POW, x POW y POW PROD etc. Since there are an unlimited number of these and our set operations are type sensitive, we sometimes use *type expressions* in our stack effect descriptions. For these expressions we employ a postfix syntax using n and $ for the types INT and STRING, * for PROD and P for POW. Examples:

```
DOM   x.y.*.P -- x.P
IMAGE   x.y.*.P  x.P -- y.P
```

We use a full stop between elements of the same type expression. This is redundant, but helps readability. Sometimes, for the written description of a glossary entry, we would like to refer to the stack elements by name. For this we employ a notation of the form < *name* >:< *type* >. E.g.

```
DOM   r:x.y.*.P -- s:x.P
```
s is the domain of the relation r

11 Sequences

Sequences are functions whose domain consists of integers from 1 to n. They are described in the same way as sets except that instead of enclosing our descriptions within set brackets we use the sequence brackets $\boxed{\texttt{[}}$ and $\boxed{\texttt{]}}$.[8]

[7]It sounds from this description as if we can represent infinite sets, which is not the case. We need INT to be an infinite set for purposes of type theory, but we do not need this for our implementation.

[8]These names clash with the ANS operations to leave and enter compilation mode. Sequence brackets are defined in the SEQUENCE vocabulary. We provide $\boxed{\texttt{[[}}$ and $\boxed{\texttt{]]}}$, equivalent to $\boxed{\texttt{[}}^\dagger$and $\boxed{\texttt{]}}$,†to support compiler switching in this context.

```
CR STRING [ " tom" , " dick" , " harry" , ] DUP .SEQ .SET<enter>
[tom,dick,harry] {(1,tom),(2,dick),(3,harry)} ok
```

The type of a sequence with elements of type `x` is `n.x.*.P`, but we abbreviate this to `x.seq`. In terms of postfix type algebra, which borrows some notation from Forth, this abbreviation is defined as: $: \mathit{seq}\, n\, \mathit{swap} * P ;$.

The following sequence operations are provided:

`^  s1:x.seq s2:x.seq -- s3:x.seq, "concat", s3=s1^s2`
Concatenates two sequences. `s3` is formed by appending the elements of `s2` to `s1`.

`<-  s1:x e:x -- s2:x, "cat element", s2 = s1 ^ [e]`

`\|/  s1:x.seq n -- s2:x.seq, "remove"`
pre: `n <` card(s1)
`s2` is formed by removing the first n elements of `s1`

`/|\  s1:x.seq n -- s2:x.seq, "retain"`
pre: `n <` card(s1)
`s2` is the first n elements of s1

Exercise: Code the following.

`INSERT  s1:x.seq e:x n -- s2:x.seq`
`s2` is formed by inserting `e` at the nth position of `s1`, Note: we recommend using local variables. A definition using stack manipulations is possible (though not easy). However, it has the disadvantage that if the specifier decides on a different order of arguments, the code must be completely redesigned.

Solution:

```
: INSERT ( s1:x.seq e:x n -- s2:x.seq )
  1- (: VALUE S1  VALUE E  VALUE N-1 :)
    S1 N-1 /|\    E <-   S1 N-1 \|/   ^
  1LEAVE ;
```

12 Reversibility choice and potential values

The guard command `-->` removes a flag from the stack. If non-zero, execution continues ahead. Otherwise execution reverses.

During reverse execution, changes made to memory state by a previous forward execution may be reversed. For this to happen the code must be written using reversible operations where appropriate. Reversible versions of all primitives that change memory are provided. These have the normal names with an added underscore: $\boxed{\texttt{!\_}}$, $\boxed{\texttt{C!\_}}$, $\boxed{\texttt{+!\_}}$, `CMOVE_`, `CMOVE$>_`. Reversible versions of global and local variables can be declared with: `VALUE_`, `VALUE^_`, `VALUE-ARRAY_`, and `VALUE-ARRAY^_`.

Once invoked, reverse execution continues until a point at which a choice had been previously made is reached. If an untried alternative exists at that point, forward execution re-commences with a new choice. Otherwise, reverse execution continues. If reverse execution arrives back at the start of a command entered by the user, the prompt of "ko" is given rather than "ok" to indicate that the requested command was infeasible.

The construct: `<CHOICE S1 [] S2 ..[] Sn CHOICE>` provides choice between two or more alternative code sequences `S1`, `S2`... The alternatives are tried in order. Example:

```
: T1 <CHOICE 10 [] 5 [] 8 CHOICE> . FALSE --> ;
```

```
T1<enter>10 5 8 ko
```

`CHOICE` provides a choice from a set. Choices are always made in the same order.

```
1 3 .. VALUE S   : T1 S CHOICE . FALSE --> ;
```

```
T1<enter> 3 2 1 ko
```

`RANDOM-CHOICE` provides a random choice from a set.

```
: T2 S RANDOM-CHOICE . FALSE --> ;
```

```
T2<enter> 2 1 3 ko
T2<enter> 1 3 2 ko
```

Potential values are values a computation would yield were it to be carried out. We compute them by carrying out the computation, noting the result, then reversing so that no side effect of the computation remains. Let `S` be a code sequence which generates an integer stack value whilst leaving existing stack elements unchanged. `INT { <RUN S INT> }` is the set of possible values that can be left by S. There are equivalent forms where S generates an asciiz string, a pair or a set: `<RUN S STRING>`, `<RUN S PAIR>` and `<RUN S SET>`. They must all be used within set brackets preceded by a type expression unless `S` is deterministic (can only produce a single result). Potential values are related to the formal theory of "Expression Transformers".

Reverse computation of compiled code collects any garbage generated by forward computation. This removes a major disadvantage of the use of reference semantics in the sets package. Maximum garbage is collected when an application is organised as a potential value computation.

13 Example Application

We now return to the application mentioned briefly earlier, and listed below. It finds solutions to the pseudo-arithmetic problem SEND+MORE=MONEY. The code begins with global variable declarations for the set of digits, for each of the letters in the problem, and for the carry values for units, tens and hundreds.

`PUZZLE` finds a single solution to the puzzle, leaving the solution in the digit variables. It guesses values for each independent digit, beginning with the least significant. Each digit chosen is subtracted from `DIGIT` to ensure that each choice will be unique. Having guessed D and E, we calculate Y and C0 and apply a feasibility check: Y must be different from both D and E. If the check fails we reverse and try different values. We proceed in this way until all digit variables have been assigned using feasibility checks that ensure the assigned values are a solution to the puzzle.

`SOLUTIONS` returns the set of all solutions to the puzzle, each solution being a function from the names of the digits to their values. `.SOLN` takes such a solution from the stack and prints it. `REPORT` takes a set of solutions and prints them all.

Note that only `DIGIT` is declared as a reversible variable. The other variables are always assigned before use following any choice, so it serves no semantic purpose to restore their previous values during reverse execution.

```
0 9 .. VALUE_ DIGIT
0 VALUE S ( -- n )   0 VALUE E ( -- n )   0 VALUE N ( -- n )
0 VALUE D ( -- n )   0 VALUE M ( -- n )   0 VALUE O ( -- n )
0 VALUE R ( -- n )   0 VALUE Y ( -- n )
0 VALUE c0 ( -- n )  0 VALUE c1 ( -- n )  0 VALUE c2 ( -- n )

: PUZZLE ( --, find a solution to the puzzle )
  DIGIT CHOICE to D    DIGIT INT { D , } \ to DIGIT
  DIGIT CHOICE to E    D E + 10 /MOD   to c0   to Y
  Y E <>  Y D <>  AND  -->  DIGIT INT { E , Y , } \ to DIGIT
  DIGIT CHOICE to N   DIGIT INT { N , } \ to DIGIT
  DIGIT CHOICE to R
  N R + c0 + 10 MOD  E = -->
  N R + c0 + 10 / to c1
  DIGIT CHOICE to O
  E O +  c1 +  10 MOD  N =  -->
  E O +  c1 + 10 /  to c2
  DIGIT INT { R , } \ to DIGIT
  DIGIT CHOICE to S
  1 to M   S M + c2 + 10 MOD  O =  -->
  S M + c2 + 10 / M = --> ;

: SOLUTIONS ( -- $.n.*.P.P, returns the set of solns to the
sendmory puzzle, each solution is in the form of a function from
strings (" S", " E" etc) to values)
  STRING INT PROD POW {
    <RUN PUZZLE
      STRING INT PROD {
        " S" S |->$,I ,  " E" E |->$,I ,  " N" N |->$,I ,
        " D" D |->$,I ,  " M" M |->$,I ,  " O" O |->$,I ,
        " R" R |->$,I ,  " Y" Y |->$,I ,
      }
    SET>
  } ;
```

```
: .SOLN ( $.n.*.P -- ) (: VALUE SOLN :)
  SOLN " S" APPLY .  SOLN " E" APPLY .  SOLN " N" APPLY .
  SOLN " D" APPLY .  ." + "
  SOLN " M" APPLY .  SOLN " O" APPLY .  SOLN " R" APPLY .
  SOLN " E" APPLY .  ." = "
  SOLN " M" APPLY .  SOLN " O" APPLY .  SOLN " N" APPLY .
  SOLN " E" APPLY .  SOLN " Y" APPLY .  OLEAVE  ;

: REPORT ( -- ) (: VALUE SOLS :) CR
  BEGIN
    SOLS CARD
  WHILE
    SOLS  SOLS CHOICE   DUP .SOLN CR   SUBTRACT-ELEMENT   to SOLS
  REPEAT  OLEAVE ;

: T ( -- ) SOLUTIONS REPORT ;

CR .( T runs the puzzle: SEND + MORE = MONEY ) CR
```