Non-standard Features of kForth

Krishna Myneni and David P. Wallace
http://ccreweb.org

November 5, 2001

Abstract

We discuss two non-standard features of the kForth interpreter, a program based largely on ANS Forth. First we demonstrate how a limited form of data typing and type checking can catch a significant set of Forth programming errors, with almost no modification to standard Forth code. Second, we discuss the benefits and restrictions imposed by using a dynamically growable dictionary. The two new features of our Forth system require the addition of two new words: `A@` and `?ALLOT`.

1 Introduction

kForth [1] is a compact interpreter, largely based on ANS Forth [2]. However, several new features, which are not a part of the ANS standard, have been introduced in kForth. In this paper we discuss two of these non-standard features:

1. Data typing and type checking
2. Dynamic dictionary

A third non-standard feature, called *deferred execution*, will not be discussed here since it does not affect the structure of Forth code. We give the motivation for including these features in kForth, particularly as it relates to our original goal of writing an interpreter to be embedded into other applications. A parallel goal was to allow code written for kForth to be easily ported to other ANS Forth systems — kForth may be viewed as a subset of ANS Forth. The two new words, `A@` and `?ALLOT`, introduced in kForth for supporting the new features described in this paper, have simple ANS Forth compatible definitions. We describe how the two new features are implemented and illustrate their use with examples.

2 Brief History of kForth

kForth has its origin as an embedded interpreter for the application XYPLOT, a plotting and data manipulation utility [1]. One useful feature of XYPLOT is its expression evaluator, which parses simple algebraic expressions and applies the operations to an entire data set. For example, the expression `y*2` multiplies all of the *y* values of an *(x,y)* data set by two. In its early stages of development, the expression evaluator consisted of a parser which broke down the expression into a vector of "op-codes", and an execution loop which performed the sequence of operations. A data stack held the intermediate values of the calculation. Thus, the beginnings of a stack based interpreter was written and incorporated into XYPLOT. Subsequently the expression evaluator was developed into a full-featured interpreter that allowed the main application to be extended with modules written in Forth source code. Forth was chosen as the language for the interpreter rather than developing a custom application language for several reasons:

1. It is relatively easy to interpret stack based code.
2. One of the authors had several years of past experience with Forth programming.
3. Forth provides a wide range of functionality, from low level bitwise operations to high level floating point operations.
4. Forth provides a foundation for constructing an application specific language.
5. The use of an established language such as Forth reduces the need to write extensive documentation, which would be required for any custom language.

In addition to its use as an embedded interpreter, kForth also functions as a standalone Forth computing environment. At present, kForth features a vocabulary of over 240 words, with 116 core words, 26 core extension words, and more than 60 words from the option sets for ANS Forth.

3 Data Typing and Type Checking

The rationale for data typing in kForth is to provide error checking. Unlike the statically typed Forth system, strongForth [3], which defines a heirarchy of data types, the implementation of kForth is limited to just two data types: `ADDR` for addresses and `IVAL` for all other data. Words listed in Table 1 verify that the address operand on the data stack (or return stack) has the proper type, i.e. type `ADDR`, to avoid processor exceptions caused by invalid memory access. Although a *signal handler* might be used to catch such exceptions [4], if the interpreter is to be embedded into another application, this method would preclude the main application from having its own handler for such exceptions. It is important to note that type checking in kForth is performed at *run-time* by associating data types with elements on the stack. This form of type checking is known as *dynamic type checking* [5]. Compilers for traditional languages such as `C` perform type checking at *compile-time*, a method known as *static type checking* [5]. Although implementation of static type checking in Forth has been discussed previously [6], it requires augmentation of the language itself to provide a means of specifying argument types to a word — the strongForth language provides a clever way to do this using the common stack diagram comment. In kForth, however, our motivation is not to implement strict data typing, but to use data typing to catch typical run-time errors, with virtually no modifications to the Forth language.

C@	C!	W@	W!	@	!
A@	2@	2!	SF@	SF!	DF@
DF!	+!	FILL	ERASE	COUNT	TYPE
CMOVE	CMOVE>	LOOP	+LOOP	FIND	’
ACCEPT	OPEN	READ	WRITE	NUMBER?	SYSTEM
CHDIR					

Table 1: Words which perform address type checking in kForth

Two kinds of errors are likely to be caught by our limited method of type checking:

1. The number of parameters on the data stack or the return stack is incorrect and one of the parameters is an address.
2. Parameters on the data stack or the return stack are in the wrong order and one of the parameters is an address.

These are typical conditions created by problem code in Forth. Take the following simple example of code with incorrect ordering:

```
variable v
v 3 !
```

The following error message is displayed:

```
VM Error(1): Not data type ADDR
```

An operand of type `ADDR` was expected on top of the stack by `!` and not found. The kForth virtual machine (VM) `QUIT` execution and returned an error code (the error message is actually displayed by the outer interpreter). The state of the stack at the time of the error may be examined by `.S` to diagnose the problem, since the VM performs `QUIT` rather than `ABORT`.

Corruption of the *return stack* can also be detected by run-time type checking. For example, a common problem is to push an item onto the return stack and forget to pop the item before the word returns. An extreme example is:

```
: BAD 3 >r ;
```

Execution of `BAD` results in

```
VM Error(5): Return stack corrupt
```

The error is detected by checking the data type of the item on top of the return stack upon return from BAD. Since it does not have type ADDR, for a valid return address, the VM returns an error indicating corruption of the return stack. The VM executes ABORT when the return stack is found to be corrupt.

Now consider a more subtle coding error involving the return stack:

```
: bswap ( n1 n2 n3 n4 -- n2 n1 n3 n4 )
     2r> swap 2>r ;

1 2 3 4 bswap
```

Entering the above statements into three untyped Forth systems produced different results. One system displayed an error while the other two responded with ok. With the latter, examination of the stack showed that the arguments were unchanged. It should be noted that all of the systems we tried displayed an error if the word bswap was used inside of another word. A variant of the above example was also tested:

```
: test 10 0 do i . 2r> swap 2>r loop ;

1 2 3 4 test
```

It is interesting to observe the number of loop iterations executed by the various untyped Forth systems. At least one went into an infinite loop. Worse, the other systems executed this code without complaint, returning with ok — the only indication of a problem given by our diagnostic print of the loop index. kForth detects problems with both of these code examples using type checking of the return stack. Upon every iteration, the words LOOP and +LOOP test for the presence of a branch address on the return stack via type checking. The price of detecting these kinds of problems is the added overhead for maintaining type information and performing type checking. We measured the impact of type checking in kForth on execution efficiency and found that it caused about a 15% increase in the time to execute standard benchmarks [7].

By design, kForth implements data typing so that it is almost entirely transparent to the user or programmer. Every cell in the data stack has associated with it one of two types: IVAL or ADDR. Data types are stored in a separate *type stack*, shown in Figure 1, which is manipulated in parallel with the data stack. kForth does not provide words for direct manipulation of the type stack. Instead, intrinsic words which operate on the data stack perform corresponding operations on the type stack. Consider the behavior of ROT:

```
( n1 n2 n3 -- n2 n3 n1 )
```

If n1 is of type ADDR, and n2 and n3 are of type IVAL, as shown in Fig 1, ROT also rotates the type stack so that the top element has type ADDR after the operation. Other words which affect the data stack also manipulate the type stack in an analogous manner. The behavior of two particular words, with respect to data typing, should be noted: + and -. Consider the case where an offset must be added to an address using +:

`( a1 n -- a2 )` or `( n a1 -- a2 )`

We expect that the result of + should produce an address if *either* the first operand *or* the second operand is an address, and indeed this is the typing rule observed by + in kForth. The behavior of - is different:

`( a1 n -- a2 )` and `( n1 a -- n2 )`

We may subtract an offset from address a1 to obtain address a2; however, it is not sensible to expect that by subtracting address a1 from integer n1, we will obtain a valid memory address. Therefore the data type of the result depends on the ordering of the data types of the operands for -. But the programmer need not be aware of these typing rules — for these cases, sensible Forth code produces sensible data typing, enabling subsequent error checking. The typing rules for + and - are implemented in an efficient manner and require little computing overhead in the virtual machine.

The return stack has an associated type stack, called the *return type stack*, also shown in Figure 1. In transfers from the data stack to the return stack and vice-versa the data types are also transferred between the type stack and the return type stack. As with the type stack, direct manipulation of the return type stack is not permitted. Intrinsic words that modify the return stack also modify the return type stack. All of the words which make use of the return type stack in kForth are listed in Table 2. In addition to the VM itself, words which explicitly perform type checking using the return type stack are

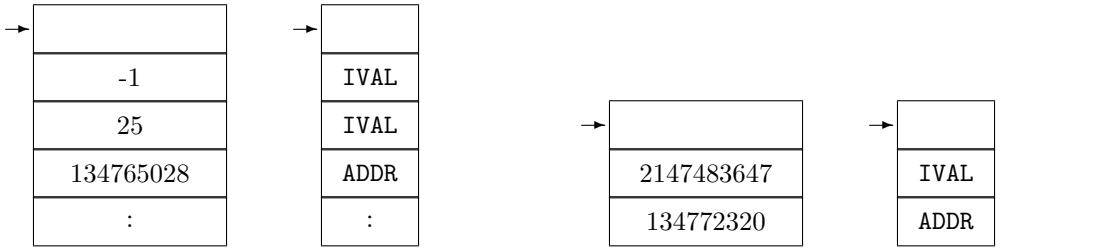

Figure 1: The kForth Stacks: the stack pointers are always aligned for each pair of stacks.

>R	R>	R@	2>R	2R>	2R@
DO	?DO	LOOP	+LOOP	UNLOOP	I
J	QUIT	EXECUTE			

Table 2: Words which use the return type stack

LOOP and +LOOP. The loop index words, I and J, place an item on the data stack with the same type as the starting loop index. It is therefore possible to loop over an address range and use an index word to place an item of type ADDR on the stack. The following example illustrates this point:

```
create tb1 20 allot

: byte_sum ( -- n | compute the sum of bytes in table tb1 )
     0 tb1 20 + tb1 do i c@ + loop ;
```

Use of C@ on the index value returned by I is valid since the starting index has type ADDR. The above code is no different from that used in an untyped system, once again demonstrating the transparency of data typing in kForth.

Next, we discuss the only known instance in which the programmer must be aware of data typing in kForth: fetching address values from memory onto the stack. An address is fetched from memory using A@ instead of @. The word A@ retrieves the same value as @, but it also sets the data type of the stack cell containing the value to type ADDR. In contrast, @ sets the data type to IVAL. The following example illustrates the use of A@:

```
variable current_table
create tb1 20 cells allot

tb1 current_table !

: @n ( n -- m | fetch the n^th element of the current table )
     cells current_table a@ + @ ;
```

The variable current_table holds the address of a table, set to tb1 in the example. The address of the table is fetched onto the stack by current_table a@ rather than by current_table @, as in an untyped Forth system. Notice that ! is used to store an address value to a memory location. Data has associated type information only while it resides on one of the two stacks (the data stack or the return stack). Type information is not retained for data stored at other memory locations. The need to provide a new word, A@, in the basic Forth dictionary may seem undesirable; however, it is a relatively small price to pay for the benefits of address type checking, which have been illustrated above. Use of A@ also makes clear to the reader of the code that an address is being fetched rather than an other kind of data value. This section concludes with the following point:

Porting kForth code to an untyped Forth system requires that A@ *be defined to be synonymous with* @.

4 Dynamic Dictionary

Traditional Forth implementations use a fixed size dictionary to hold word definitions and user created data such as small tables of numbers or counted strings [8]. The motivation to implement a dictionary which can grow as needed may be expressed in a simple code statement:

```
create array 1024 1024 * allot
```

We wish to `allot` 1 MB of space in the dictionary to hold an array of values. With a *static* dictionary, the above code is successful only if the dictionary happens to have been allocated with sufficient space. Otherwise, the Forth system may issue a dictionary overflow error or simply crash with a segmentation fault error. Some Forth systems allow the user to resize the dictionary from within the environment. A fixed size dictionary is not desirable for the use of kForth as an interpreter embedded into an application, since the useful dictionary size will depend on the application.

The ANS standard provides extension words for allocating dynamic memory, `ALLOCATE` and `FREE`, so one may argue that it is not necessary to be able to `allot` memory in the dictionary space for large blocks of user data. Also, having limited dictionary space to hold Forth code is usually not of practical concern. What are the benefits of a growable dictionary? A dynamically allocated dictionary provides the following conveniences to the programmer:

1. `ALLOT` may be used without size restriction. The programmer is not burdened by determining whether or not there is sufficient space to locate a block of data in the dictionary.
2. Addresses of data blocks made with the sequence, `CREATE` *name size* `ALLOT`, do not have to be managed by the programmer. With `ALLOCATE`, the returned address must be assigned to a constant or held in a variable until the memory block is freed.
3. Memory is freed automatically upon exit from the Forth system. In contrast, memory obtained using `ALLOCATE` should be released using `FREE` when it is no longer needed. Forgetting to do so reduces available system memory.

These features are of little value in programming embedded processors, which have stringent limits on available memory. However, for a desktop system using a modern operating system (e.g. Linux, Windows), programming may be made less cumbersome with these features. However, the benefits to the Forth programmer from using a dynamic dictionary come with some restrictions. The implications of using a dynamic dictionary are discussed below.

In kForth memory for both code and data is dynamically allocated as required. The dictionary itself is a vector of data structures, each containing the name of a word, the word's *precedence*, the *code field address* (CFA), and the *parameter field address* (PFA). In kForth, CFA is synonymous with *code pointer* or the ANS term, *execution token*. PFA is synonymous with the ANS term *data field*. During "compilation" of a word definition, a temporary code vector is built up. The size of this vector is unbounded. After a word definition has been compiled into code, which in kForth consists of pseudo op-codes, memory is dynamically allocated to hold the code sequence and then copied from the vector into the newly allocated block. The PFA of the new word is set to the address of the dynamically allocated block. The CFA is also set. If the word `RECURSE` was encountered during compilation, address placeholders inside the code are then replaced with the CFA. With this allocation scheme for the dictionary, the task of providing dictionary space and assigning addresses is passed on to the operating system (OS) rather than being handled by the Forth system, and growth of the dictionary is limited only by the OS.

Now we consider the behavior `ALLOT` may have in a system which implements a dynamic dictionary. First note that there is no `HERE` address in the dynamic system, since memory is not available until it is requested either through word definitions or by execution of `ALLOT`. In the traditional static dictionary Forth system, the code:

```
1024 allot
```

presumes that the programmer has access to the starting address of the memory region to be alloted, either because `CREATE` was invoked previously or because the starting address was obtained with `HERE`. Therefore `ALLOT` does not return an address, unlike its counterpart `ALLOCATE`.

In the dynamic dictionary system, kForth, the code `1024 allot`, must dynamically allocate 1024 bytes of memory, starting at some address which is determined by the OS. This address must somehow be made available to the programmer to allow use of the memory. We must change the behavior of `ALLOT`, but wish to do so in a way that use of `ALLOT` remains as consistent as possible with traditional Forth code. kForth imparts the following behavior to `ALLOT`: the requested memory is dynamically allocated and the starting address is assigned to the PFA of the last word defined in the dictionary. In kForth `ALLOT` must

be used only in a `CREATE` *name size* `ALLOT` sequence. The behavior of `CREATE` is also modified so that it sets the PFA to zero for the new dictionary entry. This allows `ALLOT` to verify that it is modifying the PFA of a word created with `CREATE`, instead of modifying a word that is already associated with data or code. Therefore, the statement:

```
1024 allot
```

by itself produces an error in kForth:

```
VM Error(9): Allot failed --- cannot reassign pfa
```

Two other core words are not provided in kForth owing to the lack of a `HERE` address. These are:

the comma operator (`,`)

`C,`

For creating initialized cell-size or byte-size tables, alternate, albeit somewhat less elegant, methods can be used in place of the above words. For instance, instead of the simple statement:

```
create tb1 100 , 200 , 300 , 400 ,
```

to make a 4 element table initialized to values, we could write

```
: t, ( n a1 -- a2 ) 2dup ! 1 cells - nip ;

create tb1 4 cells allot
100 200 300 400 tb1 3 cells + t, t, t, t, drop
```

Clearly the statement using the comma operator is superior, in terms of sheer simplicity, compared to the verbose method shown above. The problem with the above method is that an address for storing initial values into the table is not available until we use `ALLOT`, in conjunction with `CREATE`, to allocate the region for the table. Then the address must be manipulated to move the successive inital values from the stack into the table in the proper order.

The example given above begs the use of a *defining word* to mitigate the clumsy procedure for creating an initialized table in the absence of the comma operator. This points to the problem of how to program a defining word that requires access to a region that the word itself allocates at run-time, since `ALLOT` does not return an address. kForth provides the word `?ALLOT` to solve this problem. `?ALLOT` functions like `ALLOT` but also returns the starting address of the region on the stack. The compatible ANS Forth definition of `?ALLOT` is:

```
: ?allot here swap allot ;
```

Using `?ALLOT`, we may create a defining word for initialized tables:

```
: table ( ... n -- )
     create dup cells ?allot
     over 1- cells + swap
     0 ?do dup >r ! r> 1 cells - loop drop ;
```

Once the defining word `table` is included, it becomes trivial to create an initialized table:

```
100 200 300 400 4 table tb1
```

Note that this method works in ANS Forth as well, provided the compatible definition of `?ALLOT` is used. `?ALLOT` should not be equated with the ANS word `ALLOCATE` since, in kForth, `?ALLOT` must be used only with `CREATE` and it assigns the PFA of the created word.

Two simple examples of defining words having run time code further illustrate the use of `?ALLOT` in kForth:

```
: const ( n -- ) create 1 cells ?allot ! does> @ ;
: ptr   ( a -- ) create 1 cells ?allot ! does> a@ ;
```

The word `const` is equivalent to `CONSTANT` and `ptr` allows the creation of address constants for our typed Forth system. We close this section with the following point:

Porting kForth code to a static dictionary Forth system requires the compatible definition of `?ALLOT` *given above.*

5 Summary

We have discussed two features of kForth which depart from the current ANS standard. Data typing and type checking arise from a desire to supplement the Forth environment's error detection capability, particularly for its use as an embedded interpreter in other applications. We have demonstrated that our limited method of type checking can catch common Forth programming mistakes, particularly those associated with the return stack. Also, type checking in our implementation is largely transparent to the programmer and requires only one additional word, `A@`. The use of a dynamic dictionary offers convenience in making unrestricted use of the available system memory but at the cost of sacrificing the core words for compiling integer and byte constants into the dictionary: comma (`,`) and `C,`. Furthermore, `ALLOT` must be used only in conjunction with `CREATE`, and we must add the new word `?ALLOT` to allow the programming of defining words which require access to the alloted region. Whether or not the benefits of these features outweigh their costs can only be determined by real-world applications programming using kForth. It has been our experience, in using kForth both as an embedded interpreter and as a stand-alone computing environment, and for such diverse tasks as simulating microcontroller assembly code to demonstrating properties of hydrogen atom wave-functions, that there is an overall positive benefit from the new features we have discussed here.

References

[1] K. Myneni, kForth User's Guide, (`http://ccreweb.org`, 2001). kForth and XYPLOT are released under the GNU General Public License. Source code, executables, and documentation for both of these programs may be obtained from `http://ccreweb.org`. kForth runs on x86 compatible processors operating under Linux or the various Win32 systems (Windows 95/98/NT/2000).

[2] ANSI X3.215-1994, *American National Standards for Information Systems — Programming Languages — Forth*, (American National Standards Institute, New York, NY 1994).

[3] S. Becher, Introduction to strongForth, (`http://home.t-online.de/home/s.becher/forth/`, 2000).

[4] M. K. Johnson and E. W. Troan, *Linux Application Development* (Addison-Wesley, Reading, MA, 1998).

[5] A. V. Aho, R. Sethi, and J. D. Ullman, *Compilers: Principles, Techniques, and Tools* (Addison-Wesley, Reading, MA, 1988).

[6] B. Stoddart and P. J. Knaggs, Type Inference in Stack Based Languages, (Formal Aspects of Computing, 3: 1-000, 1992).

[7] A. Ertl, Performance of Forth Systems, (`http://www.complang.tuwien.ac.at/forth/performance.html`, 1996). Standard benchmark programs used are: `bubble-sort`, `matrix-mult`, `fib`, and `sieve`.

[8] L. Brodie, *Starting Forth* 2nd ed., (Prentice Hall, Englewood Cliffs, NJ, 1987).