Challenges for Forth

JAMES C. BRAKEFIELD
1214 Vista Del Rio, San Antonio, Texas 78216

ABSTRACT:
Three areas where Forth would seem to be lagging are presented along with suggestions for their resolution. They are: online help information, floating point, and parallelism

Forth seems to be stuck back in the eight and sixteen bit world of early microprocessors and low end minicomputers (remember them?). Microprocessors and memory chips are marching forward at what continues to be an exponential cost/performance rate. Megabyte and megaflop home computers are a reality. Some of this increasing cost/performance can be allocated to help the programmer.

ONLINE HELP INFORMATION

The computer I use at work has 24 megabytes of memory and the application software I use has some three hundred routines or operators. Finding the right operator and the information on how to use it involves searching a crib sheet and then looking up the routine in the manuals. There is online help but it is not very fast and does not have the right "hyper-text" organization. Part of the problem is that the CRT screen can not equal the resolution and page size of printed text. One would think that the speed of the computer could compensate for the lack of screen resolution.

I come home to a computer with only thee megabytes of memory and a Forth package with four thousand operators. Well, I have not yet done a crib sheet for it and the manual is half the size of the one at work. What can be done? What should be done? I suspect the answers are out there only waiting for universal acceptance so that they can become standards. One thing is for sure, I don't learn vocabularies very fast and I'm not getting any younger.

Solutions:

There are a number of psychological factors in designing a help facility. Usually the authors have a single particular help and debugging style in mind which may or may not fit that of the user. So there is the question of whether the user should match the style of the package or whether the package should try to match the style of the user. Adaptive software is still mostly a research topic.

I brought home the following gem from a User Interface Conference: "Designing the user interface is always 50% of the cost and effort in writing any software package with any set of tools." *i.e.*, No matter how you go about it, half your effort should be directed at ease of use. 50% is a larger than I had expected, but, I can find no contra-indication.

Well, from the indications given above, it may be evident that my style of online help is as follows:

If one knows the name of the routine, gradually expand the level of detail. *i.e.*, follow a path down an "information outline" starting with a

one-line description (name, stack diagram, half-line description). If one does not know the name, use one of several search techniques to find a name:

- If one knows part of the name use a wild-carded name search to obtain a list of possible names.
- If one knows a category use help on that category to obtain a list of possible names.
- If one knows a phrase use a KWIC search on the short descriptions to obtain a list of possible names.

Currently I use a paper based approach but see no reason other than screen resolution why it cannot be computerized. Looking at my approach psychologically, one can say that I use massive amounts of text while searching for a particular pattern. Once the right pattern is found, the level of detail is increased in small steps.

FLOATING POINT

Forth has been slow to bring floating point into its purview. In contrast Postscript uses floating point by default. Which has been more successful? I have asked the Adobe people if the floating point overhead was excessive and the answer was no. You look at the new microprocessors with on-chip pipelined floating point and you ask yourself is integer arithmetic just a bad habit?

At work I do image processing. I find floating point solves a lot of problems with scaling and calibration. If memory is so cheap and floating point so fast why use bytes and integers? The only real remaining reason for integer or byte is to save space, *i.e.*, a data compression issue and often that compression need not be exact (*e.g.*, data in memory as a byte or scaled 16-bit integer).

Thus I now question the value of integer being the default data type in any programming language or package.

Solutions:

It is relatively straightforward to make the data stack operate only in floating point: The various @ and ! words are rewritten to leave/take floating numbers on/from the data stack.

The issue of how to do address calculations then surfaces. There are three solutions that I can think of:

a) Use a set of address words which operate on the data stack.

b) Use a bit-flag as part of data to signify whether that data is floating or integer.

c) Do all of the address calculations on the return stack.

Option (a), of course, is the most pragmatic, being compatible with existing Forth.

Option (b), ideally requires hardware support and has occurred several times (Bendix G-20 of 1961, Burroughs 5500 of 1964).

Option (c) is the subject of papers (refs. 4 & 5) inspired by the 1991 SIGForth Conference and represents a considerable variation on Forth.

PARALLELISM

Parallel processing is winning. The practical reason is that it is the only way to make use of full memory bandwidth. When you add memory to your computer you are also adding to its potential bandwidth. It does not take many chips to get to say, 128 bits of memory data width. It would be nice if when you double the amount of memory you also double the processing rate.

Well, the theoretical framework is available (ref 1). The idea is that bit slice ALU's with local memory and the right ALU interconnections and pipelining can efficiently implement parallel versions of most algorithms. ALU's with thousands of bits are feasible.

Also chips designed for parallel processing are available (several of the DSP chips have dual

memory ports; members of the Inmos Transputer microprocessor series have one memory port and four communications ports; the Texas Instruments TMS320C40, a DSP chip, has two memory ports and six communications ports).

What all this leads up to is that a notation and memory data structure are needed which facilitates this type of processing. Again the package I use at work offers some suggestions. It uses a two level data structure. The first level is a descriptor which contains the element data type and precision, the number of dimensions and their extents. The second level is the data as a contiguous block. Such a feature can be added to Forth assuming some sort of memory management for the second level data (perhaps standard Forth should have built-in memory management routines?).

Solutions:

It turns out that a single data type suffices for parallel processing. It is the block of uniform data elements where the elements are: bits, characters, bytes, 16-bit or 32-bit integers, single and double precision floating point, or single and double precision complex. These data are always considered as a multi-dimensional object. A scalar is when the "span" or length of each dimension is one. Those dimensions with a span of one are always expanded via replication to match the span of the other operands in an expression.

Program code can also be represented as a multi-dimensional data set of tokens or addresses. A program could be a two dimensional set of tokens, the inner span being that of the longest expression and the outer span being that of the number of expressions.

This approach using blocks of data of uniform size and type represents a very hardware efficient organization of memory and has sufficient generality to encompass the programming scene (ref. 6).

REFERENCES

1) Guy E. Blelloch, *Vector Models for Data-Parallel Computing*, MIT Press 1990.

2) Adobe Systems Inc., *PostScript Language Reference Manual*, Addison-Wesley 1985.

3) *PV-WAVE Technical Reference Manual*, Precision Visuals, Inc. 1990.

4) James C. Brakefield, "Abstract FORTH", 1991, in preparation.

5) James C. Brakefield, "A Family of FORTH Architectures", 1991, in preparation.

6) James C. Brakefield, "Thoughts on Collection Oriented Programming", submitted to IEEE Software, 1991.