Transportable Forth and Cross Compilers

Rieks Joosten
Pijnenburg Software Development
P.O. Box 82
5270 AB St. Michielsgestel
The Netherlands

Abstract

This article describes the philosophy and experience with prototypes of a Transportable Forth Kernel (TFK). With TFK, the emphasis lies on transportability of the source code without sacrificing facilities such as flexibility and speed. To this end, some vital terminology is developed, nucleus words are stated, and future developments are discussed. TFK prototype systems prove to be not only decent "regular" Forth systems, but also a basis for writing transportable Forth. The latter requires some extra effort from the programmer.

1. Introduction

Forth has not been defined, it has grown. Efforts for standardization have not been very successful, resulting in a number of "standards". The fact that the language itself is extendable has played a significant role in this. One could say that contrary to classical languages as COBOL and FORTRAN, Forth is a living language. It is this property, that most other languages lack, that makes Forth comparable with a natural language.

In Holland it is not odd to find places that are hardly two miles apart, and yet have such different dialects that people cannot interact with each other. However, when they must work in another town, there is need for a generally accepted, yet workable language. In Holland this is found in what is called "high Dutch", and this is the language taught in schools (although dialects may be used in certain classes).

Comparing Forth with the above described situation induces the idea of creating a language-dictionary that contains the most often used Forth words and their definition. This can be done in two ways:
- descriptive (describe the words and their behavior from practical experience, as is done for some systems in the book "All About Forth", by Glen Haydon)
- prescriptive (define the words like a standard would)

Either method has (dis)advantages, but sitting back doing nothing is worse. Creating such a language-dictionary also will not inhibit the use of local dialects. However, it may enable the writing of transportable code, but that will take some learning for the people involved.

2. Another Forth System

The fact that a generally accepted, globally workable definition of Forth is lacking has led to the development of a new system that is called EVO-Forth. This system does not pretend to be generally accepted. It does claim to support transportable coding, (e.g. 32-bits systems), without any loss in efficiency and capabilities.

Reasons for designing yet another Forth system are many. First, there are the more philosophical reasons as described previously. Second, past experiences with operating systems, files/blocks, error handling, memory management, large machines as well as single board computers, etc. . ., have balanced ideas concerning a wide variety of hot Forth topics. Third, in order to get rid of outdated material, it is best to design and implement a system from scratch.

Such a new system may contain several modules, each for specific tasks such as floating point, error handling, memory management, filing, etc. Although they are projected to become parts of the system, they are not discussed here; the emphasis lies on the design and definitions with respect to the very basic system, the nucleus. To stress the nature of this nucleus and its importance, it has been given a separate name: TFK (Transportable Forth Kernel).

The starting points of the TFK design have been the following:
1. TFK has been defined in such a way that source code may be transportable, without restricting the possibilities of the language
2. TFK is to be maximally compatible with existing Forth systems
3. TFK has to run on very small as well as very big machines
4. A TFK has to be at least as efficient with memory and time as a comparable non-TFK Forth system.

The method by which TFK gains hardware independency (so that source code may be transportable) is primarily achieved by establishing appropriate terminology, and introducing models for memory, stacks, and data in a size and structure independent way.

The method by which TFK gains hardware independency (so that source code may be transportable) is primarily achieved by establishing appropriate terminology, and introducing models for memory, stacks, and data in a size and structure independent way.

In order to run TFK-based programs on very small as well as on very large machines requires a tradeoff to be made between facilities supplied by the system and machine size. This tradeoff is avoided by stating that an EVO-Forth development system includes facilities for meta-compilation with library and package support. This results in the possibility to generate stand-alone programs from which all superfluous code is stripped.

Such development environments are of such power, that it is straightforward to compile a TFK in such a way that its performance is equal or better than that of comparable Forth systems.

3. Towards a Nucleus Definition

In order to reach the goals that have been set, it is of vital importance that solid terminology is defined based on the use of Forth and of characteristics of portable programs. With this terminology, the difficulties in transporting Forth programs can be expressed, and subsequently be solved.

The compatibility of the TFK with other systems, especially with respect to addressing and memory store/fetch operators, is mainly due to the terminology that is used and the choice in names for associated operators. Making the program transportable requires the application to be rewritten in "TFK-style", which mainly consists of eliminating machine and implementation dependencies.

The complete list of terminology cannot be given in this article, due to limits on the length. One of the most important aspects of the terminology is to separate different data types and related operators to eliminate machine dependent code. For example, the data type "boolean" is defined as a separate data type, so the exact value of e.g. "true" can be implementation dependent. Also addressing types are strictly defined, with definitions for "cell" - addressing, "byte" - addressing and "extended" - addressing. A extended terminology list (as originally planned for this article) will be published, and can be obtained from Pijnenburg Software Development.

A number of constants and conversion routines supply the needed information to make code independent of a particular implementation. These include definitions as "BITS/CELL" etc. to retrieve the characteristics of the implementation in case e.g. precision of a compact storage of data is relevant. These can be used at compile-time for conditional compilation, thereby imposing no runtime penalty for using TFK.

4. Results

Prototype systems are built that run on the IBM pc/xt/at, as an extension of FYSForth. The most important modules thereof are:
- Transportable Forth Kernel
- Forth Compiler; text interpreter
- Extended Buffer Management
- File i/o
- Exception handling

Apart from that, there are some miscellaneous facilities such as a high level serial line interface, automatic document generation aids, i/o port words, vectors, execution chains, conditional execution, etc. They are specifically mentioned to acknowledge the dependency of the system thereon, albeit for a small part only.

Using these prototype systems, Evolution Assembler Environments (EAEs) have been created. These EAEs are machine-code programming environments that are all identical, except for the specifically target-dependent parts such as the mnemonics. These EAEs handle source files with speeds between the 300 and 500 kB/min, and yet have extensive logging facilities, macro support, forward referencing, etc.

It is found that writing the EAE-modules was uneasy in the beginning, and that it takes time to get used to the ideas behind TFK. Learning from sins against TFK is tolerated until a program is transported. It is felt that complying with TFK ideas is worth the effort of getting acquainted with it for a number of reasons.

First, it makes one more aware of what one is doing. Such results are important especially in modules that are difficult to oversee at first and afterwards turn out to be relatively simple. This has been the case with a very powerful forward-referencing module that is now part of the EAEs.

Second, it is a relief to know that the system is orthogonal in its wordset, so that it is unnecessary to think whether or not some routines are available. This saves considerable time looking whether or not such words exist, or having to recompile because words do not exist. Having some synonym words around is not a real disadvantage since this does not occupy dictionary space (headers are separated, and reside in symbol table buffers that are external to Forth).

Third, it is shown, by the speed of the assemblers, that the prototype systems are not slower than comparable Forth systems. As a matter of fact, the underlying FysForth would not be able to compile nearly as fast as the EAEs do. This speed is mainly due to the fact that headers and code are separated, allowing for other search-mechanisms to be utilized, but also to the fact that there are no limitations to the implementations, and that the words defined do not presume programmer knowledge of internal structures.

Apart from gaining experience with transportable Forth, the prototype system also allows for experimenting with other novelties for which a need was felt during the development of a complete high-performance yet robust Forth system. Naturally, the exception handling is amongst the new features, but also execution vectors (that allow initialization and finalization actions to be performed when switching vectors) and execution chains (that allow appending or insertion of routines to such chains; this enhances possibilities for initialization, finalization, resetting, etc., of modules in a system).

Each of these modules, as well as the more "classical" ones such as buffer management and symbol table handling, have been carefully designed with as major requirements that
- the syntax must be in accordance (and as orthogonal) as the TFK itself, and
- implementation details must be hidden from the user.

These two requirements allow for internal optimalization when necessary, without a priori sacrificing transportability of application programs that use these modules.

5. Conclusions

A prototype Transportable Forth Kernel has been developed for building professional Forth systems that support writing transportable Forth source code, without sacrificing flexibility or speed. The emphasis in the TFK design has been put on the definition of necessary terminology, and on maximum compatibility with existing Forth systems.

Forth cross compilers have been built for programming either Forth on a byte machine (6809-based), or a word machine (NOVIX NC4000-based). Running TFK-based test-applications on both machines confirms that the TFK is reasonably defined, and that there is no speed or memory penalty.

In order to explore the potentials and weaknesses of the prototype TFK, prototype Assembler Environments have been built on it. The performance (typically 400 kB/min on an IBM/AT) and general applicability thereof (identical environments for different cpu's, except for machine-code mnemonics), are direct consequences of the changing insights that are brought by the definition of the TFK.

Future work will be directed towards the extension of the transportability concepts that started with TFK, i.e. towards the definition of a high-performance, transportable, professional Forth system. Due to the promises embedded in the prototype assembler environments, an effort will be made to develop other programming environments based on this Forth system as well.