A Threaded Interpretive Language supporting Programming in the Large

April 8th, 1986

Mikael R.K. Patel
Department of Computer and Information Science
Linköping University
S-581 83 LINKÖPING
SWEDEN

INTRODUCTION

One crucial element that threaded programming languages such as FORTH [Brodie 81] lack is the ability to generate modules and support calls between modules. This is the main reasons why FORTH is so difficult to use in developing Large Software Systems. This paper discusses the definition and development of a FORTH inspired threaded language which offers the ability to generate sharable code and thus support Programming in the Large and idea of the Programmers Team [Brooks 80, Boehm 81, Ghezzi et al. 82].

Traditional Threaded Languages may be regarded to be closed systems, since the code generated maybe re-used only on source level. This leads to some problems when dealing with Very Large Software System. The use of vocabularies can solve some of the problems involved but not all. Many of FORTH system in use today are indirectly threaded and use absolute addresses in their threading scheme [Kogge 82]. This reduces the ability to share code among programmers without having to pay the price of recompiling the code for every usage.

In the present paper, a Threaded Interpretive Language supporting Module Management and dynamic linkage of modules is described. The first section of the paper deals with the definition of the goals for the language designed. In the second section some of the inspiration sources are examined and discussed, and in the third, some implementation details are reviewed.

GOALS

In order to support Module Management and programming of large software systems reusability of code should be on object code level instead of source level. Object code generated must thus be *relocatable* and *minimize the usage of memory* for the threading schema. A sub-goal of this project is the design of a threading approach that will achieve this. Minimization of the cost of threading should be taken into a count as the target system for the prototype implementation is to run on a 32-bit microprocessor. *Importing a function* from an other module present as low cost as possible, hopefully only as small as a threading operation.

Modules should be memory conservative and only be brought into memory when used. Traditional linking of modules must therefore be avoided as this leads to complete recompilation of all modules using a sub-module.

The execution kernel should be small and only give the basic functions. Higher level function (such as traditional block file management) should not be placed in the kernel. Further, the kernel should be *symmetric* and minimize the number of operations needed to manipulate different data objects on the stack. The goals for the design is summarized as following:

- Module code should be *relocatable*
- Implementation details (such as sub-definitions within a module) may be *hidden*
- The execution kernel should be *symmetric* and minimize the number of functions
- The execution time for calls between modules should be minimal
- The cost of threading and management of module should be small

INSPIRATION

In search for a method for module management, a survey of how it is solved in other programming languages gives some valuable insights into the problem domain and some of the common solutions. Languages such as MODULA-2 [Wirth 82, 86] support module management but not in an interactive manner. MODULE-2 is an interesting example as the language is small in comparisons to Ada [Pyle 81] but provides all the mechanisms for Programming in the Large. Other languages that give inspiration for how to deal with code modules are SIMULA [Dahl et al. 67] and SMALLTALK [Goldberg et al 83]. Both of these are object oriented and solves to the module problem by encapsulating function definitions in small units (objects), and reducing the visibility of the implementation of the object functions. Especially SMALLTALK is interesting in relation to FORTH since they both are interactive, and use interpretation and compilation modes.

CONTRIBUTIONS

A programming system prototype contributing to these goals has been implemented and is currently running on an Apple Macintosh. The current version, which is called MacTILE (the Macintosh Threaded Interpretive Language Environment) supports the usage of modules and fully relocatable module and kernel code. The kernel is minimized to a total of 220 functions and supports a number of data sizes (32-, 16-, 8- and bit operations) quick type conversion (such as number to text) and functions for symbolic manipulation such as association list used for implementing overloading [Pyle 81] and object oriented programming [Goldberg et al. 83, Dahl et al. 67]. The total size is about 4K bytes of machine code instructions and contains no threaded code. It is fully relocatable and may be ported to any computer system using the same microprocessor.

The syntax for a module definition has been adapted from MODULA-2.

```
MODULE TheModuleName

FROM ModuleToImportFrom
   IMPORT WordFromThisModule
   IMPORT AnOtherWord
   ( Internal definitions )
INITMODULE
   ( Initial code for the module )
ENDMODULE
```

When compiled, the module becomes a separate object code segment which is saved on file. The module may now be executed. The FROM defines a module from which definitions may be IMPORTed. The IMPORTed definitions are placed into the vocabulary of the current module.

The Module Compiler is also capable of detecting if a definition is IMPORTable or not. This is defined by the programmer by marking a definition as EXPORTable in the same fashion as IMMEDIATE. Using this mechanism definitions and sub-definitions may be hidden thus supporting one of the objectives of Programming in the Large. In addition to the attribute EXPORT four other modes are defined; NORMAL which is the normal attribute of a definition, EXECUTION being that the definition may only be used in execute (interpretation) mode only, COMPILATION for definitions that are only to be used when compiling and last the traditional IMMEDIATE mode.

An example, the Prime Benchmark using an external module for output management:

```
MODULE Prime

FROM InOut
  IMPORT WriteString
  IMPORT WriteCard
  IMPORT Writeln

8192 CONSTANT size
VARIABLE count
CREATE flags size 8/ ALLOT

: Primes ( --- )
  " 10 Iterations" WriteString WriteLn
  10 0 DO
    count OFF
    flags size 8/ true FILL
    size 0 DO
      I flags B@ ( a bit fetch operation )
      IF I 2* 3+ size OVER I+
         OVER OVER >
         IF DO
              false I flags B! ( a bit store operation )
              DUP
           +LOOP
         ELSE
           DROP DROP
         THEN
        DROP
         1 count +!
      THEN
    LOOP
LOOP
count @ 6 WriteCard "  Primes" WriteString WriteLn ;

INITMODULE
  Primes
ENDMODULE
```

The first reference to the external module `InOut` will force the module to be loaded into memory and initiated. The cost of executing a function in the external module is minimized, so that after the first call the cost of calling an imported function is an indirect jump—equivalent to a NEXT operation. The low cost is achieved by caching entry addresses at runtime. This strategy gives some problems when a module forced to be unloaded from memory, since cache addresses are no longer valid and must therefore be removed.The kernel support the loading and unloading of modules thus giving the programming full control of the utilization of memory. Words imported are placed in the vocabulary of the current module but may be renamed locally;

```
FROM InOut
  IMPORT WriteString RENAME Type ( FORTH Standard )
```

The renaming facility gives the ability to resolve local name conflicts and does not effect the vocabulary of the imported module. To reduce the size of the object code vocabularies are stored separately on disk and only used at compile-time. The module `InOut` may be viewed as having the following outline:

```
MODULE InOut
  ( Internal definitions in InOut )
: WriteString ( string -- ) ........... ; EXPORT
: WriteCard ( number\positions -- ) .... ; EXPORT
: WriteLn ( --- ) ......................; EXPORT
INITMODULE
  ( Initialization code for the module )
ENDMODULE
```

A side effect of relocatable and non-overlayed modules is that modules may be down-loaded into bare computer systems that use the same kernel and processor. The ability to compile and run code on a development system, to down-load sub-parts of the code into a target system and interactively and incrementally develop, and test the code on the target machine without target compilation has be an extension to the initial goals.

ACKNOWLEDGMENTS

I would like to thank my wife, Eva, for being so understanding and all personnel at the Department of Computer and Information Science at Linköping University for their help in this work. I would especially like to thank Ralph Rönnquist for his fertile suggestions and our fruitful discussions on the design and implementation of MacTILE.

REFERENCES

[Boehm 81] B. W. Boehm, *Software Engineering economics,* Prentice-Hall, Englewood Cliffs, N. J. , 1981.
[Brodie 81] L. Brodie, *Starting FORTH,* Prentice-Hall, Englewood Cliffs, N. J. 1981.
[Brooks 75] F. P. Brooks, Jr., *The Mythical Man-Month,* Addison-Wesley, Reading, MA, 1975.
[Dahl et al. 67] O. J. Dahl et al., *SIMULA 67 Common Base Language,* Norsk Regnesentral, Oslo, Norway.
[Kooge 82] P. M. Kooge, "An Architecture Trail to Threaded-Code Systems", *COMPUTER,* March 82, pp. 22-32.
[Ghezzi et al. 82] C. Ghezzi and M. Jazayeri, *Programming Language Concepts,* John Wiley & Sons, USA, 1982.
[Goldberg et al. 83] A. Goldberg and D. Robson, *Smalltalk-80: the language and its implementation,* Addison-Wesley, USA, 1983.
[Pyle 81] I. C. Pyle, *The Ada programming language,* Prentice-Hall, USA, 1981.
[Wirth 82] N. Wirth, *Programming in Modula-2,* Sperger-Verlag, N. Y., 1982.
[Wirth 86] N. Wirth, *Algorithms & Data Structures,* Prentice-Hall, London, 1986.