Technical Notes

Separate Headers

Siem Korteweg
J. v. Stolberglaan 16
3931 KA Woudenberg
The Netherlands

Hans Nieuwenhuijzen
Sterrewacht "Sonnenborgh"
Zonnenburg 2
3512 NL Utrecht
The Netherlands

Abstract

In classical FORTH, code is generated linearly in memory, and consists of appropriate, interwoven header and object code structures. To improve the memory usage characteristics of this classical dictionary structure, we have separated the headers from their associated object code to implement new header structures, supporting the removal of headers from memory when they are no longer required.

Objectives and Strategy

The object code of most applications will contain headers corresponding to:
— routines replacing parts of code by an identifier for extra clarity.
— parts of code frequently used in other routines.

All the headers may be available for debugging, but once a module has been tested, only the headers used for communication with other modules are needed. The other headers will not be used in other modules; i.e., they have a 'local' meaning. Although they have only a local meaning, they do have a global existence; i.e., they remain in memory until the code of the module is removed from memory.

An algorithm has been published, see [SCHL81], that supports the selective removal of headers. As it only temporarily adjusts the dictionary, we have tried to find a structural and natural mechanism to remove local headers from the object code when they are no longer functional. See [JOOS81] and [BART77] for two mechanisms to generate and handle header-less definitions (so called "orphans"). These orphans are used to save memory. We will generalize and clarify these mechanisms, and remove and clarify their extra administration.

When the FORTH system being used produces relocatable code, it is easy to construct an algorithm that performs the selective removal of local headers, although its execution may take quite some time. We will consider the general case in which the FORTH system produces non-relocatable code. In that case it is difficult to remove headers from compiled code, as header and executable code structures are interwoven.

Implementation

By separating headers from their associated code, we avoid the necessity of relocating object code when headers are removed from memory. In order to execute the object code of a routine whose header is found, we have to add a 'code pointer' to the headers. As all the headers can be placed adjacently in memory, we can remove the linkfield from the headers. All the present operations upon the dictionary can be adapted very easily.

We can implement vocabularies as a row of headers and a link to another vocabulary that is to be searched entirely when a header is not found in the first vocabulary. This implementation offers the fewest problems when vocabularies are to be removed in case one of the vocabularies overcrowds. It implies that the vocabularies are fully relocatable, although it is not equivalent to the usual vocabularies. This implies that we can implement a dictionary structure supporting headers separated from their object code without overhead in memory usage compared with usual dictionary structures.

In order to remove only the local headers from the dictionary, they should be recognizable from the global headers. This means that space has to be allocated in the headers to hold this property (one bit will suffice). Most headers of applications will be local, so we implement this property analogously to the precedence property. All headers are local, except when the routine GLOBAL is used analogously to the routine IMMEDIATE .

Most of the time we delete the local headers from the dictionary after an entire application is compiled. This implies a new routine DELETE.LOCALS to delete all the local headers from the entire dictionary. This routine can use a linear scan of the row of headers to erase the local headers by shifting the global headers.

As the headers and the object code are interwoven in usual dictionary structures, it is easy to make a decompiler as long as headers with their identifier strings are available in the code. The separation of headers and code makes it impossible to achieve an easy and direct correspondence between a part of object code and its associated header. We can use a binary search algorithm, using the historical order of the headers, but this will cause overhead. To reduce this overhead, one could use an array containing index pointers to the headers giving their historical order. However, the extra array would only be needed during decompilation, and decompilation is not a frequently used feature.

Highlevel Implementation

```
( bit configuration in a count byte )
( bit 7: parity, always up )
( bit 6: precedence )
( bit 5: local/global switch )
( bit 4: count )
( ..... )
( bit 0: count )

BASE HEX

0 VALUE BOH                                   ( begin of headers )
0 VALUE RUNNER                                ( traverses the header space )
0 VALUE HOLD                                  ( pointer for DELETE.LOCALS )
0 VALUE TEMP.END                              ( pointer for DELETE.LOCALS )

: ADVANCE 1+ 1 TRAVERSE 3 + ;                 ( 3: skip code pointer and last )
                                              ( character )

: ?DELETE C@ 20 AND ;
```

```
: DELETE.LOCALS
   BOH TO RUNNER                          ( HOLD = 0 => no headers in "buffer" )
   0 TO HOLD                              ( HOLD <> 0 => headers from HOLD to )
   BOH TO TEMP.END                                  (RUNNER remain in dict. )
   BEGIN RUNNER DP.H <>
   WHILE HOLD
         IF    RUNNER ?DELETE
               IF HOLD TEMP.END                              ( source.dest )
                  RUNNER HOLD -                                 ( #bytes )
                  DUP +TO TEMP.END
                  CMOVE                                  ( erase local headers )
                  0 TO HOLD                        ( no local headers in "buffer" )
               THEN
         ELSE RUNNER ?DELETE
               IF RUNNER TO HOLD THEN
         THEN
         RUNNER ADVANCE TO RUNNER
   REPEAT HOLD
   IF HOLD TEMP.END RUNNER HOLD -
      DUP +TO TEMP.END
      CMOVE
   THEN
   TEMP.END TO DP.H ;
```

Test Results of DELETE.LOCALS

Number of headers in the dictionary	Number of headers to be removed			
	25	50	75	100
100	1–1	1–2	2–2	2–2
200	1–2	2–4	3–5	4–7
300	1–3	3–6	4–8	6–11
400	2–4	3–8	5–12	6–15

Time in seconds

Table 1: The time needed to delete headers from the dictionary.

The recorded numbers are the times (in seconds) required to delete the denoted number of local headers. The first number corresponds to the deletion of headers taken from some source (resulting in a *non* uniform distribution of the identifiers, typical for most applications). The second number corresponds to the deletion of headers taken from the FYS FORTH kernel (resulting in a more uniform distribution of the identifiers).

Conclusions

The removal of local headers from compiled code is supported by the new structure, that is, it is possible to mark headers during compilation and to remove them from the dictionary afterwards. The implementation of this selective removal costs some extra code that should be considered, opposed to the possible savings. It offers the following advantages:

Saving of memory. When new code is generated all headers may be available for debugging. Once a module has been tested, only the headers for communication with other modules are needed. The other (local) headers can be deleted, thus allowing memory saving.

Long identifiers for clear documentation. It does not matter how long we make the identifiers of the local headers in the source code because they can be deleted from the object code (they do require extra mass-storage for the source code, but that is less important as the amount of available mass-storage is less critical than the amount of available central memory). So, we can choose long identifiers that are very clear to read and understand. This means that the programs can become clearer to read and easier to debug and maintain.

Meta- or cross-compilers. The generated object code does not contain headers, and the structure has built-in local headers. This makes it easy to write a meta- or cross-compiler using this structure.

Headers on mass-storage. It is possible to save all headers on mass-storage for possible later use.

Decompilation is not often used, but it could be implemented easily at the cost of some extra code and memory overhead, the latter only during decompilation.

Acknowledgements

We wish to thank prof. dr. J. v. Leeuwen of the department of Computer Science of the State University of Utrecht for giving one of us the opportunity to work a year at the Observatory for the final phase of his study. We also wish to thank Rieks Joosten and Frans Cornelis for the time they spent discussing and analyzing this concept and its use. We also thank Hans v. Koppen for the use of his room and his Apple II computer.

To obtain a copy of the FYS FORTH manual of FYS FORTH 0.3 please contact: Hans Nieuwenhuijzen, Sterrewacht "Sonneborgh", Zonneburg 2, 3512 NL Utrecht, the Netherlands.

References

[JOOS81] R. Joosten, *FYS FORTH 0.2/0.3 Preliminary User Manual*, State University Utrecht, Utrecht the Netherlands, 1981

[SCHL81] K. Schleisiek, Separate Heads, *FORTH Dimensions* vol. 2 no. 5, 1981, PO Box 1105, San Carlos, Calif. 94070, pp 147–151.

[BART77] P. Bartholdi, Pseudo-vectors, EFUG-notes, June 1977.

[BART79] P. Bartholdi, The TO solution, *FORTH Dimensions* vol. 1 no. 4, 1979, PO Box 1105, San Carlos, Calif. 94070, pp 38–41.

Appendix: Deviations from the '79-STANDARD

The TO-concept uses the routines TO, +TO and FROM to give a 'flag' certain values. Datastructures of type VALUE take appropriate actions by means of the routine EXECTO depending on the value of this flag. CF. also [JOOS81] and [BART79].

+TO

+TO –> <>

Sets the action of the following VALUE type to add the integer on top of the stack to the data contained in the VALUE.

1+

<INTEGER> 1+ –> <<INTEGER>+1>

Increments the number on top of the stack by 1.

CMOVE

<SOURCE> <DEST> <#BYTES> CMOVE -> <>

Moves <#BYTES> bytes of memory from address <SOURCE> through <SOURCE>+<#BYTES>-1 to <DEST> through <DEST>+<#BYTES>-1. The contents of these bytes are preserved, even when overlapping.

EXECTO

<+VALUE> <ADDR> EXECTO -(%val-flag = -1)-> <>
<ADDR> EXECTO -(%val-flag = 0)-> [<ADDR>]
<VALUE> <ADDR> EXECTO -(%val-flag = 1)-> <>

Acts on the address on top of the stack depending on the value of the %val-flag. The flag is reset to zero afterwards. The following cases are implemented:

-1: the <+VALUE> is added to the contents of <ADDR>.
0: the contents of the address replaces the address.
1: the <VALUE> is stored in <ADDR>.

TO

TO -> <>

Sets the action of the following VALUE type to store the number on top of the stack in it.

VALUE (IMMEDIATE)

<START.VALUE> VALUE #<VALUE.NAME># -> <>

Creates a value, initializing its contents to the number on top of the stack.