`State`-smartness — Why it is Evil and How to Exorcise it

M. Anton Ertl
Institut für Computersprachen
Technische Universität Wien
Argentinierstraße 8, A-1040 Wien
`anton@mips.complang.tuwien.ac.at`
`http://www.complang.tuwien.ac.at/anton/`
Tel.: (+43-1) 58801 4474
Fax.: (+43-1) 505 78 38

Abstract

`State`-smart words provide a number of unpleasant surprises to their users. They are applied in two contexts, and they fail in both: 1) for providing an arbitrary combination of interpretation and compilation semantics; 2) for optimizing with a special implementation of the (default) compilation semantics. This paper discusses these issues and shows programmers and system implementors how to avoid `state`-smart words. It also reports our experiences in converting the `state`-smart words in Gforth into a clean solution: little work and few problems.

1 Introduction

Global variables have a bad reputation — and they deserve it.[1] In Forth the nastiest global variables are those containing some system state: E.g., every Forth programmer can tell a horror story (or two) that involves `base`.

Among the global system variables `state` is the most insidious. Problems resulting from its use turn up long after their cause, when you least expect them. This apparently makes these problems so hard to grasp that a paper like this is necessary to discuss them.

`State` is mainly used in so-called `state`-smart (immediate) words. These words perform as expected as long as you interpret or compile them directly with the text interpreter; but when you tick or `postpone` them, the result is usually not what you want, but you normally don't notice this until much later.

[1]The reason given in *structured programming* and *software engineering* texts is that it enables interactions between every part of the code and every other. A more convincing (at least, to me) argument is that programmers usually want several instances of a global variable (at least if it is really used globally), depending on context; with a global variable they have to manage the context switch themselves, and make errors. Stacks and objects are clean alternatives to global variables.

The problems of `state`-smart words have been recognized a long time ago, resulting in their elimination in the Forth-83 standard. Shaw [Sha88] also discusses this topic.

A note on terminology: Unless otherwise noted, in this paper the verb *compile* means *append some semantics to the current definition* (or, in traditional implementation-oriented terms: *to store a CFA with* `,`).

Text interpreter is the ANS Forth term for the outer interpreter. The *compiler* is the text interpreter in compile state; the *interpreter* is the text interpreter in interpret state.

A request for interpretation (*RFI*) is a question to the ANS Forth committee about points in the standard that the questioner finds unclear. Sometimes RFI also means the answer to the question.

2 Example: Optimization

Consider the definition

```
: 2dup ( a b -- a b a b )
  over over ;
```

This definition works, and we want any "improvements" to have the same behaviour as this definition. Let us assume that this definition is too slow in your opinion. When `2dup` is compiled, you would prefer the definition

```
: 2dup ( a b -- a b a b )
  postpone over postpone over ; immediate
```

Unfortunately this definition does not work correctly when the interpreter processes it. Some people have tried to achieve the desired behaviour by making `2dup` a `state`-smart word:

```
: 2dup ( a b -- a b a b )
  state @ if
    postpone over postpone over
  else
```

```
    over over
  then ; immediate
```

This works in many cases, but in some cases it is incorrect; as Greg Bailey puts it [X3J96]:

- It compiles[2] correctly.
- It interprets correctly.
- (what it compiles[2]) executes correctly.
- Its tick, when `EXECUTE`d is correct if in interpret state at the time `EXECUTE` is invoked, but is incorrect if in compile state at the time.
- A definition into which its tick is `COMPILE,`d runs correctly if the definition runs in interpret state but fails if it is run in compile state.
- `[COMPILE]` does not work correctly with it.

To this, let me add:

- A definition that `postpone`s it executes correctly in compile state, but incorrectly in interpret state.

Here are some examples for the incorrect cases. They may look contrived because they are shortened to the essentials; keep in mind that in real applications there is lots of code between the parts, so the bug reveals itself quite far from its cause, the `state`-smart definition.

2.1 ' ... execute

```
: [execute] execute ; immediate
1 2 ' 2dup ] [execute] [
```

With the original `2dup` this results in having *1 2 1 2* on the stack. With the `state`-smart `2dup` this tries to compile `over over` (which is non-standard, because there is no current definition).

A typical real situation is having the execution token of `2dup` assigned to a `defer`ed word that is used in an immediate word, e.g.:

```
defer foo
' 2dup is foo
: [bar] ... foo ... ; immediate
: fnord ... [bar] ... ;
```

2.2 ' ... compile,

```
: [compile,] compile, ; immediate
: [2dup] [ ' 2dup ] [compile,] ; immediate
1 2 ] [2dup] [
```

The results are the same as above: With the original `2dup` this results in having *1 2 1 2* on the stack. With the `state`-smart `2dup` this tries to compile `over over`.

A typical real situation would be a macro (or run-time code generator), to which the execution token of `2dup` is passed as parameter, and that macro is used in another macro; e.g.:

```
: [foo] { xt -- }
  ... postpone do
    ... xt compile, ...
   postpone loop ... ; immediate
: [bar] ... [ ' 2dup ] [foo] ... ; immediate
: fnord ... [bar] ... ;
```

2.3 [compile]

```
: [2dup] [compile] 2dup ; immediate
1 2 ] [2dup] [
```

Again, the results are the same as above.

2.4 postpone

```
: compile-2dup postpone 2dup ;
: another-2dup [ compile-2dup ] ;
```

With the original `2dup` (and the `state`-dumb immediate `2dup`) the definition `another-2dup` does that same thing as `2dup`.[3] With the `state`-smart `2dup`, this tries to perform `over over` during the definition of `another-2dup` (which is non-standard, because the only thing on the stack at that time is a colon-sys, whose size is unknown).

A typical real situation would be a macro or run-time code generator:

```
: compile-dpower ( n -- )
  dup 1 ?do postpone 2dup loop
      1 ?do postpone d* loop ;
: foo ... [ 3 compile-dpower ] ... ;
```

2.5 Transformation using `immediate`

Here is one problem not mentioned in the list above: Forth programmers like to assume that

```
: foo ... [ bar ] ... ;
```

and

```
: [bar] bar ; immediate
: foo ... [bar] ... ;
```

[2]Here *compile* means: being processed by the compiler.

[3]This is how the standard currently defines it. The standard committee discusses outlawing (making ambiguous) compiling in interpret state in response to RFI 9; then this example would cause an ambiguous condition.

are equivalent. However, this is not guaranteed in the presence of `state`-smart words.

Some people have suggested avoiding the problem shown in Section 2.4 by making `compile-2dup` (and `compile-dpower`) immediate, and surround it's use with `]`...`[[`. They might also suggest avoiding the problem shown in Section 2.2 by surrounding `[2dup]` with `[`...`]`. However, as discussed above, these changes may have more effects than just working around the problem.

Moreover, consider the cases where both cases show up in the same word, e.g.:

```
: [foo] { xt -- }
  ... xt compile, ... ; immediate
: [bar]
  ... [ ' 2dup ] [foo] ...
  postpone 2dup ... ; immediate
: fnord1 ... [bar] ... ;
: fnord2 ... [ [bar] ] ... ;
```

In this case neither `fnord1` nor `fnord2` will behave correctly with the `state`-smart `2dup`. `Fnord1` suffers from the `' ... compile,` problem, `fnord2` suffers from the `postpone` problem. There are workarounds, but they are complex, and you have to notice the problem first; the first attempt at fixing one problem will probably directly lead to exposing the other problem.

3 Example: Combined Words

The example in this section involves not an optimization, but an arbitrary combination of interpretation and compilation semantics.

A word like (file wordset) `s"` is actually defined as the combination of two words: Its interpretation semantics is something like

```
: s"-int ( "ccc<">" -- c-addr u )
  [char] " parse copy-to-buffer ;
```

Its compilation semantics is something like

```
: s"-comp ( "ccc<">" -- )
          ( run-time: -- c-addr u )
  [char] " parse
  postpone sliteral ; immediate
```

If the interpreter processes `s"`, it should perform `s"-int`; if the compiler processes `s"`, it should perform `s"-comp`. I call these words *combined words*, because they combine the interpretation semantics of one word with the compilation semantics of a different word. Shaw calls such words state-unsmart [Sha88].

If `'` or `postpone` (`[compile]` etc.) encounter `s"`, the programmer usually either wants the semantics represented by `s"-int` or the semantics represented by `s"-comp`, not something else. The standard defines that `' s"` should give a result equivalent to `' s"-int`, and `postpone s"` should give a result equivalent to `postpone s"-comp`.

In traditional implementations every word has only one execution token and an immediate flag. When the interpreter processes a word, it `execute`s the execution token. When the compiler processes a word, it either `compile,`s (default compilation semantics) or `execute`s (immediate compilation semantics) the execution token, depending on the word's immediate flag.

It is impossible to implement a word like `s"` in such an implementation; as an approximation, these implementations implement `s"` with a `state`-smart word:

```
: s" ( state false: "ccc<">" -- c-addr u )
    ( state true: "ccc<">" -- )
    (            run-time: -- c-addr u )
  state @ if
    s"-comp
  else
    s"-int
  then ; immediate
```

This definition behaves correctly as long as it is only processed by the text interpreter, but it fails with `'`, `postpone` etc., as discussed in Section 2.[4]

4 Programs

This section discusses your options if you want to write a standard program.

Standard programs can only define words with default or immediate compilation semantics. As a consequence, it is impossible to implement words like `s"` in a standard program, and it is also impossible to implement optimizations such as those discussed in Section 2. So what is the best way to work around this restriction?

The general solution is to provide two words: one for the interpretation semantics and one for the compilation semantics. Examples: `'` and `[']`, `char` and `[char]`, and `s"-int` and `s"-comp`. This solution has the additional advantage of making it clearer for the user what it means when they `'` or `postpone` such a word. This solution was used in the Forth-83 standard.

4.1 Low-problem `state`-smart words

Mitch Bradley thinks that this solution is not user-friendly enough for novice users; he prefers the bugs induced by `state`-smart words to explaining

[4]To allow implementation of `s"` on traditional Forth implementations, the ANS Forth committee will probably outlaw `' s"`, and at least those uses of `postpone s"` that cause problems with these implementations.

to novice users, e.g., when to use ' and when to use [']. He also thinks that the problems with state-smart words are outweighed by the problems caused by mixing these two words up [Bra96].

If you agree with him, and want to write a combined word in a standard program, here's a compromise: Provide the combined word as state-smart word, but also provide the two constituents, e.g., s"-int and s"-comp. Advise the users to use the state-smart word only directly in the text interpreter. They should use only the constituents with ', postpone, etc.

4.2 Parsing words

This section takes a look at a common class of words:

The most frequent reason for wanting a combined word is *parsing words*: words that read the input stream. You typically want them to read the input stream when they are processed by the text interpreter. Later the read data is needed for some action. The difference between the interpretation and compilation semantics is that the compilation semantics needs to store the data between parsing time and the action, and it has to compile the action into the run-time of the current definition. This can be seen nicely by comparing the following definitions:

```
: .(  ( "ccc<">" -- )
  [char] " parse
  type ;
: ."  ( "ccc<">" -- ; run-time: -- )
  [char] " parse
  postpone sliteral
  postpone type ; immediate
```

In these definitions, [char] " parse is the parsing part, executed at parsing time; postpone sliteral takes care of storing, and type is the action.

In addition, there may be a conversion from the parsed string into some other format/type (e.g., into an execution token); this typically happens at parse time.

It is a good idea to factor these four components (parsing, conversion, storage, and action) into separate words. This allows using the functionality of the word in more situations: e.g., when the string is not in the input stream, or when the action has to be postponed by more than one level (as is done in run-time code generators).

Parsing words have another problem, apart from seducing people to write state-smart words: They take an argument from the input stream[5] and make it very hard or impossible to pass an arbitrary string as this argument. My advice is to write no parsing words at all; instead, write words that take string arguments (or suitably converted arguments), and use them in combination with words like s" that do only parsing (and storage); in this way you also avoid the temptation to write state-smart words.

[5]The input stream is another case of system state, with the additional handicap, that you have only limited influence on it.

5 Systems

This section discusses your options as systems implementor.

5.1 Combined Words

This section describes how you can implement words like s" without restricting the use of ', postpone etc.

The basic requirement is: the decision between interpretation semantics and compilation semantics must be made when the name is parsed and the execution token is looked up, not later at run-time.

A well-known implementation technique satisfying this requirement is the use of an interpretation and compilation wordlist, as in cmForth; the interpretation wordlist contains the interpretation semantics and is searched in interpret state; similaraly, the compilation wordlist contains the compilation semantics and is searched in compile state. One problem with this approach is that it is difficult to implement the search-order wordset on top of it.

A very clean technique has been proposed by Shaw [Sha88] and is used by DynOOF [Zsó96]: Every named word has two execution tokens, one for the interpretation semantics, the other for the compilation semantics. This is the approach I would use for a new Forth system, and it may also be a good solution for an existing system; however, we implemented solutions that had fewer repercussions for the header structure of our system, Gforth [Gfo].

First implementation

In addition to the immediate bit, Gforth also has a compile-only bit in the header. If the interpreter encounters a compile-only word, it reports an error (-14 throw).

The first implementation simply extended this mechanism: Instead of immediately reporting an error for a word with this bit set, the interpreter first looks in a table for the interpretation semantics of the word. If there is an entry, the interpreter performs this interpretation semantics; if there is no entry, the interpreter reports an error, as before.

The key for the table lookup is the name field address (NFA) of the word; thus we need only one table for all words, irrespective of wordlists.

In addition to the interpreter, `'` should be changed to give the execution token for the interpretation semantics of combined words.[6] Also, according to the draft for RFI 8, `find` should give the execution token for the interpretation semantics in interpret state, and the execution token for compilation semantics in compile state.[7]

Second implementation

After two months, Bernd Paysan replaced the first implementation with another one (for aesthetic reasons). In the following I will present the basic idea (the implementation in Gforth is slightly different); this presentation is based on a posting by Bernd Paysan [Pay98] and a followup by me:

```
: interpret/compile: ( xt-i xt-c "name" -- )
    Create immediate swap , ,
DOES> ( interpretation: ... -- ... )
      ( compilation: ... -- ... )
    state @ IF  cell+  THEN  @ execute ;
```

This is used like:

```
' s"-int ' s"-comp interpret/compile: s"
```

Until now, this gives us only a `state`-smart word; it will work correctly when processed by the text interpreter, but incorrectly with `'`, `postpone` etc. So we change these words to treat `interpret/compile:`-defined words specially.

In order to do this, we have to recognize these words. We do this by comparing the code fields. We cannot do this in a standard way, but the following works on traditional implementations:

```
' dup dup interpret/compile: i/c-prototype

: is-i/c? ( xt -- flag )
  [ 0 >body 1 chars / ] literal
  ['] i/c-prototype over compare 0= ;
```

Now we can define `'`, `postpone` etc.:

```
: i/c>int ( xt1 -- xt2 )
  >body @ ;
: i/c>comp ( xt1 -- xt2 )
  >body cell+ @ ;

: ' ( "name" -- xt )
  ' dup is-i/c? IF  i/c>int  THEN ;

: ['] ( compilation: "name" -- )
      ( run-time: -- xt )
  ' postpone literal ; immediate
```

`postpone` is a little harder, because we have to find out the immediacy; in the following I use `find` in a way that should work on traditional systems:

```
: comp' ( "name" -- w xt )
  bl word find dup 0= -13 and throw ( xt n)
  over is-i/c? if
    swap i/c>comp swap
  then
  0< if \ not immediate
    ['] COMPILE,
  else
    ['] EXECUTE
  then ;

: postpone, ( w xt -- )
   \ this can be optimized
   swap POSTPONE literal compile, ;

: postpone ( compilation: "name" -- )
           ( run-time: ... -- ... )
  comp' postpone, ; immediate
```

`[compile]` is easy now:

```
: [compile] ( compilation: "name" -- )
            ( run-time: ... -- ... )
  comp' drop compile, ; immediate
```

`find` should work correctly without changes (if you only use it for writing a text interpreter). As for `search-wordlist`, it is not clear what it is supposed to do.

Experiences

I coded the first implementation in an afternoon (except for the changes to `'`, because it was not clear to me at that time exactly what `'` should do); I changed the state-smart words into words using this mechanism in another afternoon. The changes affected less than 50 lines in the kernel and added less than 50 lines of code specifically for this feature. This implementation was used by the Gforth development team for about two months. As far as I remember, we encountered no problems.

The second implementation has been implemented in Gforth since July 1996 and was released to the general public[8] in December 1996.

The only problem we encountered and have heard about is this were the errors reported for ticking compile-only words. This feature is not directly related to the introduction of combined words, but

[6]However, the current draft for RFI 8 makes ticking such words ambiguous, so this is probably not a requirement.

[7]Yes, that makes `find` a `state`-smart word. However, apparently the only use that a standard program can make of `find` is writing a text interpreter, so this `state`-smartness is acceptable; also `find` is not immediate, so its `state`-smartness is not as insidious as that of immediate `state`-smart words.

[8]I estimate that Gforth has more than 1000 users, based on the number of bug reports, other email communication and downloads.

I implemented it when I rethought ticking in this context; an alternative behaviour would be to produce an execution token for the compilation semantics. Reporting an error is more cautious, but in hindsight the alternative would have been preferable (and I recommend it to implementors with a large base of legacy code), because the reported errors uncovered no real problem. Anyway, these error reports were easy to fix.

If you are still sceptical about changing from `state`-smart implementations of `s"` etc. to a parse-time state-checking implementation, in particular its impact on legacy code running on your system: In general, `state`-smartness proponents have reacted to my examples that show problems with state-smartness by telling me that I should not program like this (and that they hope that such programs are non-standard). In other words, their programs work the same whether the system uses `state`-smart words or combined words. So, their programs won't break when the system changes in this respect.

5.2 `[COMPILE]`

We could try to implement an optimizing `2dup`, as discussed in Section 2, using any of the techniques in Section 5.1; e.g., with `interpret/compile:`

```
:noname ( a b -- a b a b )
  over over ;
:noname ( run-time: a b -- a b a b )
  postpone over postpone over ;
interpret/compile: 2dup
```

However, with the implementation of `interpret/compile:` and `[compile]` shown above this would not be correct in the following case:

```
: my-2dup [compile] 2dup ;
1 2 2dup
```

This should put *1 2 1 2* on the stack, but it tries to compile `over over`.

The problem is that in the above definition `2dup` has default compilation semantics (i.e., to compile the interpretation semantics), but the system does not know that[9]. A problem with `[compile]` occurs in any implementation of combined words where the user specifies the compilation semantics without specifying whether this is the default compilation semantics or not. The following solutions are available:

- Add a flag to each combined word that indicates whether the compilation semantics are default. The user would have to supply the value for that flag, and `[compile]` would use this flag to decide what to do.
- Do not implement `[compile]` in the system; `[compile]` belongs to the *core ext* wordset, and words in this wordset are optional.
- Advise the users not to use combined words in this way (for optimization). Section 5.3 discusses an alternative mechanism.

[9]In general, the system cannot know by itself, whether the compilation semantics is equal to the default compilation semantics, because the problem is undecidable.

5.3 Optimizations

The example in Section 2 can be implemented with any technique for implementing combined words (provided you solve the `[compile]` problem discussed above), but there is one more option:

There is only one execution token for the word; it points to a record with two fields: One describes what to do when the execution token is `execute`d (in our example: `over over`); the other field describes, what to do when the execution token is `compile,`d (in our example: `postpone over postpone over`. `Execute` and `compile,` access their respective fields and just perform the action.

I call this technique *intelligent `compile,`* [EP96, Section 4.2]: in traditional systems `execute` does something different for every execution token encountered, whereas `compile,` stupidly always does the same: `,`.

The intelligent `compile,` has a slight advantage over combined words for optimization purposes: It generates better code in the `'` ... `compile,` case; it also avoids the `[compile]` problem.

On the other hand, it can only be used for optimization: For combined words two execution tokens are necessary, and the decision between them has to be made when they are looked up in the dictionary, not later at `execute` or `compile,` time.

The difference between a word with an optimizing compilation action and a general combined word is this: the general combined word (e.g., `s"`) has a non-default (and non-immediate) compilation semantics, whereas a word like `2dup` has default compilation semantics, and we just want to provide a special implementation of these semantics.

The intelligent `compile,` is used in bigForth and in the current RAFTS prototype [EP97].

5.4 `]]` ... `[[`

Several people have proposed the syntax

```
]] foo bar boing [[
```

as a more readable alternative to

```
postpone foo postpone bar postpone boing
```

One problem in implementing this syntax is how to deal with code that contains parsing words, like

```
]] ." hello, world" [[
```

Let us assume that we have extended `."` to be a combined word whose interpretation semantics are those of `.(`. In other words, `."` reads the string when it is processed by the text interpreter. Then, for consistent behaviour, `."` also should read the string as soon as it is parsed in `]]` ... `[[`. I.e., the code above should be equivalent to

```
[ s" hello, world" ]
sliteral postpone sliteral
postpone type
```

One approach would be to define all the parsing words as consisting of three parts, as discussed in Section 4.2: parse-time, storing, and run-time (action). The text interpreter knows about this, and executes, compiles, or postpones the parts depending on whether it is in interpret, compile, or postpone[10] state.

Given the execution tokens `parse`, `store` and `run`, the text interpreter would do the following in the three states:

interpret state

```
parse execute
run execute
```

compile state

```
parse execute
store execute
run compile,
```

postpone state

```
parse execute
store execute
store compile,
run postpone literal postpone compile,
```

I have not implemented this idea yet, and I am not sure if I will. One of the problems is how to integrate it with the existing `interpret/compile:` concept. Also, the benefits do not justify the complexity (at least if the programmers follow my advice and avoid parsing words).

[10]Postpone state is the state of the text interpreter between `]]` and `[[`.

6 Conclusion

`State`-smart words produce insidious bugs and should be avoided.

My advice to programmers is: Don't write parsing words; this will save you not only from the temptation to write `state`-smart words, but also from other problems. If you write parsing words, provide their factors. If you write `state`-smart words, provide their state-dumb constituents.

My advice to systems implementors is: Don't write state-smart words (except maybe `s"` and `to`, and then provide state-dumb constituents). If you want to provide convenient interpretation semantics for (normally) compile-only words, use one of the techniques for implementing combined words. We switched Gforth from state-smart words to combined words with little effort and few problems.

Acknowledgements

Jonah Thomas and the referees provided valuable comments on earlier versions of this paper. This paper has also benefited from email and Usenet discussions with Greg Bailey, Mitch Bradley, Loring Craymer, Bernd Paysan, Elizabeth Rather, Jonah Thomas and others.

References

[Bra96] Mitch Bradley. Re: Another solution for RFIs 8 and 9. Message 9609231729.AA06128@FirmWorks.COM to the mailing list ansforth@minerva.com, September 1996.

[EP96] M. Anton Ertl and Christian Pirker. RAFTS for basic blocks: A progress report on Forth native code compilation. In *EuroForth '96 Conference Proceedings*, St. Petersburg, Russia, 1996.

[EP97] M. Anton Ertl and Christian Pirker. The structure of a Forth native code compiler. In *EuroForth '97 Conference Proceedings*, pages 107–116, Oxford, 1997.

[Gfo] Gforth home page. http://www.complang.tuwien.ac.at/forth/gforth/.

[Pay98] Bernd Paysan. Re: State-smart etc was: Re: Facelifting my forth. Usenet newsgroup comp.lang.forth, message 351B70D4.F31@remove.muenchen.this.org.junk, March 1998.

[Sha88] George W. Shaw. Forth shifts gears. *Computer Language*, pages May: 67–75, June:61–65, 1988.

[X3J96] TC X3J14. Clarifying the distinction between "immediacy" and "special compilation semantics". RFI response X3J14/Q0007R, ANSI TC X3J14, 1996.

[Zsó96] András Zsóter. Does late binding have to be slow? *Forth Dimensions*, 18(1):31–35, 1996.