`State`-smartness: Applications, Pitfalls, Alternatives*

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 implementing words like `s"` that provide an arbitrary combination of interpretation and compilation semantics (combined words); 2) for optimizing using 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. 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.

`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.

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.

This paper explains the problems caused by `state`-smart words, and shows various alternatives to `state`-smart words that allow system and application programmers to avoid these problems. This version differs from the earlier version [Ert98] by an improved presentation and by taking two new developments into account: the answer to an RFI [X3J99], and the discovery of a way to implement combined words as ANS Forth programs (i.e., portably).

Section 2 and 4 present example applications of `state`-smart words and their pitfalls, before categorizing the applications of `state`-smart words (Section 5). Then we present alternatives to using `state`-smart words, both for ANS Forth programmers (Section 6), and for Forth system implementors (Section 7). Section 3 provides background information on semantics and combined words.

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.

An execution token (*XT*) is an abstract data type for a routine; the end users of XTs are `execute` amd `compile,`.

A *combined word* has an unusual combination of interpretation and compilation semantics, i.e., it is neither an immediate word nor a normal word (see Section 3 for an in-depth discussion).

2 Example: Optimization

Consider the definition

*This paper is an updated and significantly revised version of my EuroForth '98 paper; I submitted this draft version to JFAR in 2001, but it was not processed, so I am finally putting it online.

```
: 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[1] correctly.
- It interprets correctly.
- (what it compiles[1]) executes correctly.
- Its tick[2], when EXECUTEd 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.

In addition, the following problem may arise:

- A definition that postpones it executes correctly in compile state, but incorrectly in interpret state. While the problematic case is no longer allowed in a standard-conforming program [X3J99], it is still a good idea for a Forth system to support such programs.

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.

[1] Here *compile* means: being processed by the compiler.
[2] the XT produced by '.

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 code 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 defered 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 code 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 -- )
  >r
  ... postpone do
    ... r> 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 ] ;
```

This program is no longer standard-conforming, because it performs the compilation semantics of 2dup in interpret state [X3J99]. However, on most systems this program will perform as follows:

With the original 2dup (and the state-dumb immediate 2dup) the definition another-2dup does that same thing as 2dup. With the state-smart 2dup, this code tries to perform over over during

the definition of `another-2dup` (which will produce unpredictable results, because the only thing on the stack at that time is a colon-sys, whose size differs between systems).

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] ... ;
```

are equivalent. And indeed, this equivalence holds most of the time, except when dealing with `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:

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

```
: [foo] ( xt -- )
  >r ... r> 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 Semantics

Until now we have mostly avoided the concepts of semantics that were introduced in ANS Forth, because they are not reflected directly in most implementations, and people therefore find them confusing. This section explains them, and how `state`-smart words fit in.

The semantics of a word is its meaning. In ANS Forth the semantics of a Forth word is the action or behaviour that happens when the word is executed in a specific context.

Named Forth words have two semantics: *interpretation semantics* and *compilation semantics*. The standard also talks about execution, run-time and initiation semantics, but these are just used to define interpretation and/or compilation semantics.

3.1 Defining the semantics of words

Interpretation and compilation semantics are not necessarily connected, and the standard defines a few words of the standard by giving separate semantics in the glossary entries. However, in the typical case the standard defines just the execution semantics, and uses a default-mechanism to define interpretation and compilation semantics: the default interpretation semantics of a word are its execution semantics [ANS94, Section 3.4.3.2]; the default compilation semantics of a word are to compile the execution semantics.

For user-defined words the standard provides only two ways to define words:

Normal words have default interpretation and compilation semantics.

Immediate words have default interpretation semantics, and compilation semantics that are equal to the execution semantics.

Sometimes programmers want to create words with a non-default, non-immediate combination of interpretation and compilation semantics (*combined words*); or, for optimization, words with a non-default implementation of the default compilation semantics. `State`-smart words are a (not necessarily conscious) attempt to create combined words in one of the standard ways. But they are just immediate words with `state`-dependent interpretation and compilation semantics, and this difference from combined words results in behavioural differences in certain usage cases.

3.2 Using the semantics of words

The two semantics are used in different contexts:

text interpreter, interpret state: perform interpretation semantics

text interpreter, compile state: perform compilation semantics

`'`, `['`]: XT represents interpretation semantics[3]

`POSTPONE`: compile compilation semantics.

`[COMPILE]`: compile non-default or perform default compilation semantics.

`FIND`, `SEARCH-WORDLIST`: unclear, RFI 8 pending.

So, there is no relationship across all contexts between `state` and which semantics are used; when using just the text interpreter, there is a relationship. That is why `state`-smart words work correctly when processed by the text interpreter, and why they can fail in the other contexts. The other contexts occur relatively rarely, therefore it is easy to miss the problems of `state`-smart words during testing.

The decisive difference between the ANS Forth usage model and, say, an alternative model oriented towards `state`-smart implementations is this: The ANS Forth model decides which semantics to use (*binds*) when looking up the name, i.e., the programmer knows which semantics is used for a word from the static program text without requiring a lot of context; therefore this model is easier to work with (once the programmer understands it), and less error-prone. On the other hand, in a model catering to `state`-smart implementations (i.e., where semantics are directly associated with `state`), the binding would occur only at run-time, which is much harder to analyse and to work with, and more bug-prone.

The ANS Forth committee has relaxed (from a system view; *restricted* from a program view) its model lately to allow `state`-smart implementations of certain words [X3J99] (but not all [X3J96]), so programmers trying to comply with the standard lose some of the advantages of the original model. However, for system implementors it is a good idea to try to comply with the original model, to avoid bugs in their user's programs, and to hopefully establish more practice in the original model, such that it will become standard again some day.

4 Example: Combined Words

The example in this section involves not an optimization, but a combined word.

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 ;
```

If the interpreter processes `s"`, it should execute `s"-int`; if the compiler processes `s"`, it should execute `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`.

Many Forth implementations try to 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.

However, the ANS Forth committee apparently intends such an implementation to be legal, and will probably declare all the problematic uses nonstandard. They have already done so for the problematic uses of `postpone` and `[compile]` [X3J99], and will probably deal with `'`, `['`], `FIND`, and `SEARCH-WORDLIST` similarly, if asked.

So, a Forth implementor can probably use the `state`-smart definition of `s"` in a standard system. That does not make it a good idea, and we will look at alternatives in later sections.

5 Applications

Given the problems of `state`-smart words, why do programmers and system-implementors want to use them?

There seem to be two main uses:

Optimization (see Section 2) They want to implement the default compilation semantics of normal words in unusual ways.

Convenience They want to be able to use as much source code as possible both interactively and

[3] This follows from the last sentence of [ANS94, Section 6.1.0070].

in a colon definition, without requiring changes in the code (such as, changing uses of ' to [']) and vice versa).

In both cases state-smart words are used as an approximation to combined words; the effects on usage contexts other than text interpretation are usually not taken into consideration or are seen as undesirable, but unavoidable side effects (as we will see, they are avoidable).

The convenience issue leads to the following question: Most words can be just defined as normal words, and then be used without problems and without requiring changes between interactive use and use in a colon definition. Why does this not hold for all words? There are two classes of words, where it does not hold:

Parsing words These words read from the input stream[4] (e.g., s"). If a normal word reads from the input stream, the data has to follow right after the word during interactive use, but not when the word is used in a colon definition. Usually the input stream data such a word reads should follow directly after the word in both uses. To support this convenience, s" and to are defined as combined words in the standard. Other words have been defined in pairs (e.g., ', and ['']), making a change necessary when moving code between interactive use and a colon definition.

In parsing words we have to differentiate between parse-time actions (e.g., for ."), parsing) and run-time actions (e.g., for ."), printing).

Control structure words These words have only compilation semantics defined, because it is probably impossible to define the interpretation semantics in the desired way. Nevertheless, there are Forth systems that try to convenience their users by providing an interpretation semantics that approximates the desired effect.

6 Programs

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

6.1 Combined words

Although ANS Forth provides no direct way to implement words with an arbitrary combination of interpretation and compilation semantics, it is possible to get them to work on an ANS Forth system. The basic idea is to use a state-smart word for use in the text interpreter, and modify ', postpone etc. to use the correct part instead of the state-smart word.[5]

We use s" as example. The definitions of s", s"-int, and s"-comp are the same as in Section 4. We have to add definitions for ' and ['']:

```
: ' ( "name" -- xt )
  ' dup ['] s" = if
    drop ['] s"-int
  then ;
```

This just uses the old ', and replaces the XT of the state-smart word with the XT for the interpretation semantics.

Postpone is a bit more complex, because we cannot use ' to check if it is the word we are looking for, because there is no guarantee ' will be able to find the word at all (it might fail for words with undefined interpretation semantics). Find should be safer[6]:

```
: postpone ( compilation: "name" -- )
  >in @ bl word find ['] s" 1 d= if
    drop postpone s"-comp
  else
    >in ! postpone postpone
  then ; immediate
```

The definitions for [compile], find and search-wordlist should be similar (with the additional complication that it is not clear what find and search-wordlist should do).

All uses of ' etc. have to be redefined to use the new versions; among the standard words this is:

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

This approach can be generalized into dealing with several combined words by maintaining a table containing the xts for the state-smart words and their constituents; instead of comparing with ['] s", the generalized version performs a table lookup. This generalization has been implemented in a library that is available at http://www.complang.tuwien.ac.at/forth/combined.zip.

The drawbacks of this approach are: 1) It is relatively complex and requires more programming

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

[5] This technique was originally invented by Bernd Paysan, and refined and ANSified by me based on a suggestion by Jonah Thomas.

[6] Find is not defined completely in the standard, but at least for user-defined immediate words (such as our state-smart words) it should work and produce the same XT as produced by '.

than other approaches; however, the additional programming effort can be eliminated by using the library. 2) Uses of ' etc. in non-standard words are not superseded (unless you do it explicitly); so if you use one of these words on a supposedly-combined word, you run into the usual problems with `state`-smart words. This is only a problem if the program uses these non-standard words; a typical scenario where this problem arises is when the combined word is provided as part of a library written in ANS Forth, and then used on a particular system in combination with system-specific words.

6.2 Separating words by context

A simpler 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 reader what the programmer means when they ' or `postpone` such a word. This solution was used in the Forth-83 standard.

Even if you implement combined words (see Section 6.1), it is still a good idea to provide the parts of the combined word as separate words, such that users who favour clearness over convenience can use them.

6.3 Separating parsing words by factoring

For combined parsing words, the difference between the interpretation and the 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.

So, the two semantics have common factors, and 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 `postpone`d by more than one level (as is done in run-time code generators). Because of these advantages, most components of `.(/."` already are separate words in ANS Forth.

Parsing words have another problem, apart from seducing people to write `state`-smart words: By taking an argument from the input stream, they 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), i.e., words that would be factors of parsing words, 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.

7 Systems

This section discusses the options available to Forth system implementors. It partially also applies to writing programs that can make use of system internals.

Of course, a system implementor can use all the options available to ANS Forth programmers but the system implementor has additional options.

7.1 Combined Words

This section discusses a number of proposed or realized implementation schemes for combined words, but is not exhaustive.

Gforth's current implementation

Gforth's current implementation of combined words is mostly along the lines discussed in Section 6.1. We made use of our freedom as system implementors in the following ways:

We applied the change to ' etc. to the original definitions of the words, such that every word in the system using ' uses the version that produces the XT for the interpretation semantics.

Gforth does not use a separate table containing the various xts of combined words; instead, it defines combined words with a specific `does>`-based defining word, recognizes combined words by looking at the code field of the word, and accesses the interpretation and compilation xts in specific fields in the body of combined words.

The `does>`-based defining word does not produce a `state`-smart word, but a word that aborts when executed; consequently, this word cannot be used directly in the text interpreter, and therefore

the text interpreter has to access the interpretation and compilation semantics explicitly. This is slightly more complex and slower than the other way, but it has the following advantages: 1) It is easier to notice if the code field address of one of these words ever leaves the confinement of the few system-specific words that know how to deal with it. 2) It makes the implemented concepts clearer to readers of Gforth's code.

Dual-XT words

The most straightforward implementation of combined words is to have two xts per named word, one for interpretation semantics and one for compilation semantics.

A variation on this theme would be to have an XT for the interpretation semantics, and a Gforth-style compilation token representing the compilation semantics: For words with default compilation semantics the compilation token consists of the interpretation semantics XT, and the XT of `compile,`, so using the compilation token instead of a compilation XT would eliminate the need to define a compilation semantics definition for all words with default compilation semantics.

Such an approach would possibly be a good choice for a new implementation that can afford to waste a little memory in pursuit of elegance and simplicity. DynOOF [Zsó96] uses a dual-XT approach, but it is not clear if it implements combined words.

Compilation Wordlist

cmForth has a separate compilation wordlist containing the compilation semantics of some words, mostly for optimization purposes (but it also makes compile-only words really compile-only). The compiler searches this wordlist first, the interpreter does not search it at all.

This mechanism can be used to implement combined words, but it has a few pitfalls for ANS Forth implementors:

- ANS Forth requires that user-defined words with the same name as standard words shadow the standard word completely. This can be solved by having a separate wordlist for user-defined words that is always searched first.
- If the system implements the search order wordset, transparently dealing with these internal wordlists requires some additional magic.

Overall, the compilation wordlist approach appears relatively unattractive for implementing in an ANS Forth system.

Partial shadowing

Mark W. Humphries proposed an implementation of combined words by setting a compile-only flag of a word, such that a search for the compilation semantics would find the word, but a search for interpretation semantics would not find it, but possibly an earlier-defined word with the same name and without the flag. This would allow to define combined words like this:

```
: s" s"-int ; ( interpret-only )
: s" s"-comp ; immediate compile-only
```

Gforth's 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.

After two months, Bernd Paysan replaced the first implementation with the current one (mainly for aesthetic reasons).

`[COMPILE]`

`[Compile]` poses a special problem: `[compile] 2dup` should be equivalent to `2dup` (because `2dup` has default compilation semantics), whereas `[compile] s"` should be equivalent to `postpone s"` (because `s"` has non-default compilation semantics). If both `2dup` and `s"` are implemented as combined words, how should `[compile]` know that. 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.
- Let `[compile]` assume non-default compilation semantics, don't use combined words for optimization, and advise the users not to use

combined words for optimization. Section 7.2 discusses an alternative mechanism. Gforth takes this approach.

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 combined words in another afternoon. The changes affected less than 50 lines in the kernel and added less than 50 lines of code specifically for supporting combined words. This implementation was used by the Gforth development team for about two months. As far as I remember, we encountered no problems.

The current implementation has been implemented in Gforth since July 1996 and was released to the general public[7] 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 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.

7.2 Optimizations

The example in Section 2 can be implemented with any technique for implementing combined words, but there is an alternative: What we actually want to do is to provide an unusual implementation for the default compilation semantics. The natural place for performing such optimizations is `compile,`, because that is the latest time for doing this optimization, it avoids the problem with `[compile,]` and it offers slightly more optimization opportunities: `['] 2dup compile,` will not be optimized if the optimization works through a combined-word `2dup`, but it will be optimized if `compile,` performs the optimization.

A simple variant of this *intelligent* `compile,` just special-cases the xts we want to optimize:

```
: compile, ( xt -- )
  dup ['] 2dup = if
    drop postpone over postpone over
  else
    compile, \ the dumb compile,
  then ;
```

We use this kind of special-casing in the `compile,` of the current Gforth development version to compile non-primitives into using primitives (e.g., variables are compiled as literals), as preparation for combining a sequence of primitives into a super-primitive. BigForth also uses this technique.

If there are many different optimization or code generation cases (e.g., in a native-code compiler), special-casing in this way is impractical and unmodular. Then it is better to keep the primitive-specific optimization code near the other code for that primitive, and use a data structure to arrange a data-driven `compile,`.

E.g., RAFTS [EP96, Section 4.2] uses the following scheme: The XT is implemented as a pointer to a record with two fields: the address of the `execute` routine for the word; and the address of the `compile,` routine for the word. The generic `execute` and `compile,` just jump through these fields.

This scheme is essentially just an extension of the traditional code field scheme: in traditional systems `execute` does something different for every execution token encountered by dispatching through the code field, whereas `compile,` stupidly always does the same thing: `,`; the new scheme adds a `compile,` code field to the (execute) code field.

7.3]] ... [[

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" [[
```

If you want to provide the convenience of using code including parsing words in the interpreter, in

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

colon definitions, and within `]] ... [[`, combined words are not sufficient, because they do not work as desired within `]] ... [[`; `."` 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 6.3: 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[8] 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,
```

8 Conclusion

Combined words can be used for moving source code with parsing or control flow words between interactive use and colon definitions, and for implementing optimizations.

`State`-smart words are often used as approximation of combined words, but there are differences in behaviour when used with `'`, `[compile]`, `postpone`, etc., and these differences result in insidious bugs. The reason for these differences is that combined words bind the action to be performed at parse time, whereas `state`-smart words decide based on the run-time `state`.

There are a number of alternatives available to programmers and system implementors, starting with various ways to implement combined words, through avoiding parsing words (highly recommended), and splitting parsing words into two variants (like `'` and `['] `) or into factors. System implementors also have the additional option of using a different mechaninsm than combined words to achieve their goals, in particular implementing optimizations through `compile,`, and implementing parsing words by using a more complex text interpreter and special parsing-word definitions.

We switched Gforth from state-smart words to combined words with little effort and no significant problems.

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

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, Charles Esson, Mark Humphries, Bernd Paysan, Philip Preston, Elizabeth Rather, Jonah Thomas, and others.

References

[ANS94] American National Standards Institute. *American National Standard for Information Systems: Programming Languages: Forth*, 1994. Document X3.215-1994.

[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.

[Ert98] M. Anton Ertl. `State`-smartness — why it is evil and how to exorcise it. In *EuroForth'98 Conference Proceedings*, Schloß Dagstuhl, 1998.

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

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

[X3J99] TC X3J14. Regarding compilation while in interpretation state. RFI response Q99-027, ANSI TC X3J14, 1999.

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