The new Gforth Header

Bernd Paysan
net2o

M. Anton Ertl*
TU Wien

Abstract

The new Gforth header is designed to directly implement the requirements of Forth-94 and Forth-2012. Every header is an object with a fixed set of fields (code, parameter, count, name, link) and methods (`execute`, `compile,`, `(to)`, `defer@`, `does`, `name>interpret`, `name>compile`, `name>string`, `name>link`). The implementation of each method can be changed per-word (prototype-based object-oriented programming). We demonstrate how to use these features to implement optimization of constants, `fvalue`, `defer`, `immediate`, `to` and other dual-semantics words, and `synonym`.

1 Introduction

Forth started out with a word header (Fig. 1) that satisfied several requirements. As additional requirements arose, the header was adapted, resulting in the fig-Forth header and eventually the old Gforth header. A number of requirements were tacked onto the existing header design, resulting in a maze of `if`s when text-interpreting a word.

Yet more requirements came along that necessitated extending the header again. At that point Bernd Paysan decided to perform a major redesign of the header, following [Bro84, Tip 8.13]: "Use decision tables". The present paper explains this new design.

We start out by discussing header-related requirements in Standard Forth and in Gforth (Section 2); in particular, we discuss the desire to use the execution token as name token (Section 3). In Section 4 we describe the fields and methods of the new Gforth header, and related implementation issues. Section 5 shows examples of using the header features to implement words that the old header does not support well, such as `to` or `synonym`. Section 6 compares the performance of the old and new header. Finally, we look at related work in Section 7.

In this paper we use "Gforth 0.7" to represent old Gforth and Gforth 1.0 to represent new Gforth. Most of the statements about Gforth 0.7 are also true for several older versions; Gforth 1.0 has not been released yet, but all the code presented here works with Snapshot `gforth-0.7.9_20190829`.

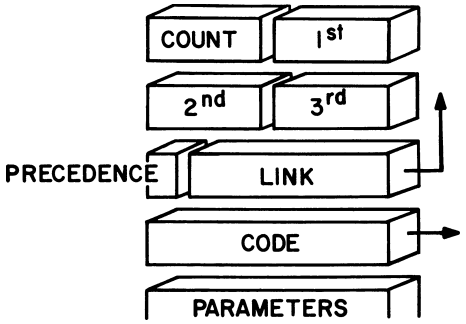

Figure 1: The original Forth word header [Moo74]

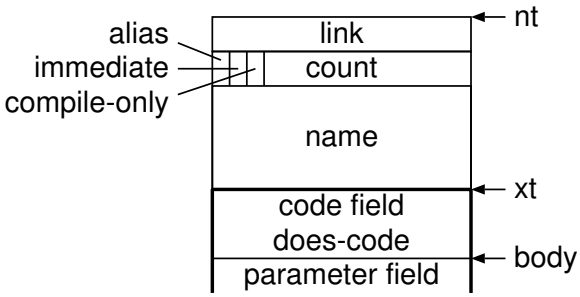

Figure 2: The Gforth 0.7 header

We use the notation `]] a b c [[` instead of `postpone a postpone b postpone c` for compactness and readability. We also use `find-name ( c-addr u -- nt )`. We embed nameless colon definitions inside colon defintions with the syntax `[: ... ;]`; this pushes the xt of the nameless colon definition. The Forth-2012 locals syntax `{: ...-- ... :}` defines the names before `--` as locals in stack effect order and treats the rest as comment.

2 Requirements

The old Gforth header (Fig. 2) is not that far from the original Forth header; it has more flags, a varible-length name field, a two-cell code field, and the link field has moved, but it mostly still has the same fields. It satisfies the following requirements:

*anton@mips.complang.tuwien.ac.at

Execution semantics The code address in the first cell of the *code field* determines which kind (colon definition, constant, etc.) the word is; for children of `create...does>` the *does-code* part points to the code after the `does>`. The body in the *parameter field* contains data (e.g., the value of a constant) or the threaded code of a colon definition.

These fields are all that is necessary for `execute` and `compile,`, so words defined with `:noname` or `noname` have only these fields (indicated by the thick line in Fig. 2).

`immediate` The *immediate bit* makes it possible to differentiate between words with default compilation semantics and immediate words (where the compilation semantics is the execution semantics).

`compile-only` Setting the *compile-only*[1] *bit* removes the interpretation semantics of a word; trying to interpret it produces an error on Gforth 0.7.[2]

`alias` An alias is a word with a different name (or in a different wordlist), but the same xt as another word. They are marked by setting the *alias bit*, and instead of a code field they contain the xt of the word they alias to.

Traverse wordlist e.g., for `words` or for building the hash table; the *link*, *count*, and *name field* are used for that. Words are usually searched through the hash table (a separate data structure), but locals are searched directly by traversing the wordlist.

Colon definition visibility The name of a colon definitions becomes visible only at the end. Gforth inserts (`reveal`s) the new header into the current word list at that time, and therefore does not need a `smudge` bit. We kept this approach in the new header, and therefore hardly mention it in the rest of this paper.

A number of additional requirements were addressed without redesigning the header:

`Compile,` In primitive-centric threaded code [Ert02] each word is `compile,`d to a primitive, usually with an inline argument, e.g., `call` *body* for a colon definition. Different word types have to be compiled to different code. In Gforth 0.7 `compile,` contains a big `case` control structure for this that looks at the contents of the code field.

Dual-semantics words (aka NDCS words [Pel17]) The compilation semantics of words like `s"` and `to` are neither default nor immediate. So these words have an interpretation semantics represented by one xt and a compilation semantics represented by a separate xt. In Gforth 0.7 such words have a special code field, and the two xts are stored in the parameter field.

Gforth uses `name>interpret` and `name>compile`[3] to access interpretation and compilation semantics. In Gforth 0.7 they abstract the following complexity: These words check for this special code field and produce the appropriate xt or ct (compilation token) [Ert98]. `Name>interpret` also looks at the compile-only bit. `Name>compile` also looks at the immediate bit. Both words also look at the alias bit.

`To` Consider `to` *name*: The action performed at run-time depends on how *name* was defined (e.g., with `fvalue` or as a (cell-sized) local). Again, there is a big `case` that looks at the code field of *name* to decide what to do.

As a consequence, in Gforth 0.7 text interpretation performs quite a bit of conditional control flow for every interpreted word. The new header eliminates most of this conditional control flow.

The following requirement is satisfied in the new header, but not reliably in the old header:

`>name` Reliably get from "the" xt of a word to its nt; this is useful in, e.g., a decompiler. Gforth 0.7 implements `>name` with a heuristic that usually works, but can also produce wrong results.

3 XT and NT

Originally, every word had one name and one semantics/action (the execution semantics) which also served as interpretation semantics, and from which either default or immediate compilation semantics were derived. Representing such words through a single address or token is a good idea.

Later, Forth acquired features that turn the relation between "named words" and "semantics" from 1:1 to m:n. Therefore, these separate concepts need separate tokens: execution token (xt) for semantics/actions; and name token (nt) for named words. Yet there is a strong desire among Forthers for a unified token, leading to repeated discussions about the necessity of nts.

[1] aka `restrict`

[2] In Gforth 1.0 all words have interpretation semantics.

[3] Called `name>int` and `name>comp` in Gforth 0.7.

3.1 New Gforth approach

This desire was also strong in the design of the new Gforth header, leading to the following design goals:

- For many words, the nt and the xt representing the interpretation semantics are the same address.
- You can pass an xt to a word that expects an nt, and you will get a plausible result.
- You can pass an nt to a word that expects an xt, and you will get a plausible result.

Most defining words produce words where the interpretation xt=nt. Exceptions are `interpret/compile:` (which defines a dual-semantics word), `synonym` and `alias`. There are other xts associated with a word, but they are usually thought of as being "the xt" of a separate word. E.g., `name>compile` returns "the xt" of `execute` or `compile,` on the top-of-stack (and another xt beneath it).

Words defined with `:noname` and `noname` can also use the xt as nt. `name>string` produces an empty string, and `name>compile` produces default compilation semantics.

Note that, for some words, `name>interpret` is not a noop, and `>name` does not get you back to the nt where you came from. This demonstrates that there is a conceptual difference between nt and xt. This is documented in the stack effect of words:

```
name>interpret ( nt -- xt|0 )
name>compile   ( nt -- w xt )
immediate?     ( nt -- flag )
name>string    ( nt -- c-addr u )
execute        ( ... xt -- ... )
compile,       ( xt -- )
>body          ( xt -- a_addr )
>name          ( xt -- nt|0 )
xt>name        ( xt -- nt )
```

Note that `xt>name` is a no-op; it documents that there is an xt on the stack before, and an nt afterwards.

In the following we present examples that demonstrate the difference. First we define some words:

```
: b ." b" ;
: c ." c" ;
:noname ." d" ; alias d
' b alias e immediate
synonym f b
' b ' c interpret/compile: g
create t
' t alias u
synonym v t
' t ' c interpret/compile: w
```

In the examples, the input is shown in **bold**, while the output is shown in blue.

Here, xt=nt.

```
s" b" find-name ' b = . -1  ok
s" b" find-name execute b ok
' b name>string type b ok
```

But in the following examples, xt≠nt:

```
s" d" find-name ' d = . 0  ok
s" f" find-name ' f = . 0  ok
s" g" find-name ' g = . 0  ok
```

Xts are usable as nts, but the nt is of a different word:

```
' d name>string type  ok
' e name>string type b ok
' f name>string type b ok
' g name>string type b ok
```

`Execute` and `compile,` exhibit the same behaviour for nts as for their interpretation xts:

```
s" d" find-name execute d ok
s" f" find-name execute b ok
s" g" find-name execute b ok
: d1 [ s" d" find-name compile, ] ; see d1
: d1
  ; ok
: f1 [ s" f" find-name compile, ] ; see f1
: f1
  b ; ok
: g1 [ s" g" find-name compile, ] ; see g1
: g1
  b ; ok
```

But `>body` does not maintain this illusion; we decided to make it a simple field access word rather than a method; if you want to get from the nt to the body of the interpretation xt, use `name>interpret >body`:

```
s" u" find-name >body u = . 0  ok
s" v" find-name >body v = . 0  ok
s" w" find-name >body w = . 0  ok
```

There are other xts associated with some words:

```
s" g" find-name name>compile  ok 2[4]
name>string type execute ok 1
name>string type c ok
```

In Gforth, immediacy is a property of a named word, not of the xt:

[4] Gforth~1.0 shows the number of items on the stack after `ok` unless the stack is empty.

```
s" e" find-name immediate? . -1  ok
s" b" find-name immediate? . 0  ok
: h1 e ; b ok
h1  ok
: h2 b ;  ok
h2 b ok
```

For some words, there is a difference between compilation semantics and `compile,`ing the interpretation semantics:

```
e \ interpretation semantics b ok
: j1 e ; \ compilation semantics b ok
j1  ok
: j2 [ ' e compile, ] ;  ok
j2 b ok
g b ok
: i1 g ; c ok
i1  ok
: i2 [ ' g compile, ] ;  ok
i2 b ok
```

3.2 Alternatives

The new Gforth approach is not for every Forth system, but every system has to deal with tokens for named words and tokens for semantics/actions. The difference between these concepts is most pronounced for separated-header systems where the name field and link field of a word are separated in memory from the code field and parameter field (e.g., to make it easy to produce headerless code). Here we explore the options for implementing nt and xt (for interpretation semantics) in this setting; we do not discuss implementing, e.g., dual-semantics words.

XT≠NT

The most straightforward approach is to let nt point to the name field or link field, and let the xt point to the code field or parameter field. You don't get a unified token, and you can only use xts for words that take xts, and only use nts for words that take nts. Getting from the xt to the nt is either slow[5] or requires adding a back pointer to every code field.

NFA as XT=NT

This approach works, but has several disadvantages:

- The name field part is needed whenever an xt is used (e.g., for `execute`), and therefore leaving the name headers away is either much harder or can be used only for applications that don't use xts (e.g., no `execute`, `is`, or `>body`).
- Words that consume an xt, in particular `execute` and `>body`, perform an extra indirection.

Other properties: A synonym has a different xt than its original; the code field can be stored with the name and link fields and separate from the parameter field.

CFA as XT=NT

This does not work with the most obvious way to implement this separated-header concept: If a synonym is implemented as a name that has the same code field as the original, both words have the same nt=xt, and `name>string` will produce the same name. Implementing `words` with `traverse-wordlist` and `name>string` will not show the name of the synonym, but show the name of the original for every synonym. E.g., with the definitions above we would get

```
s" f" find-name name>string type b  ok
```

One can complexify the implementation to avoid these problems, but the result is probably less attractive than the alternatives above.

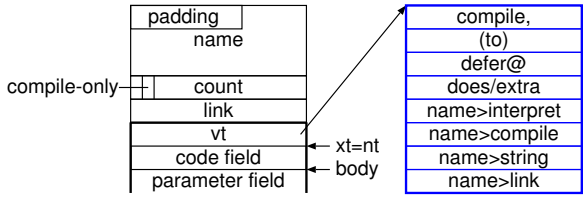

Figure 3: The new Gforth header. The thick lines surround fields present in nameless words.

4 The New Header

4.1 Organization and Fields

Figure 3 shows the new header. The xt/nt points to the code field, and the header structure grows in both directions from there, with variable-length fields (name field and parameter field) at both ends. The fields are:

code field This field contains the *code address* of native code that implements the execution sementics of the word (see also the `execute` method in Section 4.2).[6] The xt directly points to this field.

[5] If the separated headers were stored in an array and if we had the same order in memory for the separated headers and for the separated bodies (and code fields), we could use binary search for the body address in the headers. However, a straightforward implementation of `synonym` means that a header may point to an out-of-order body, so binary search cannot be used.

[6] Note that while NEXT and the threaded code in a colon definition use direct-threaded dispatch, `execute` (and other

vt This *virtual table* field points to a table of implementations for the various methods (see Section 4.2). All method implementations are represented by xts. You can get to this field with `>namevt ( xt -- addr )`.

parameter field This field serves the same purpose as it has since the dawn of Forth: It contains additional data or code, depending on the word type, e.g., the value of a constant, or the threaded code of a colon definition. You can get to this field with `>body ( xt -- addr )`.

Unnamed words have only the fields above (indicated by the thick line in Fig. 3). Named words have the following fields in addition:

count field This (almost) cell-sized field contains the length of the name in bytes. You can get to it with `>f+c ( nt -- addr )`, but it is better to get the count with `name>string ( nt -- c-addr u )` which also works on nameless words.

name field This field contains the name of the word. The arrangement is unusual in having the name before the count field; this is due to the variable-length nature of the name field, and because we already have the variable-length parameter-field at the end of the word header. The actual name starts *count* bytes *before* the count field; before the actual name, there is padding that ensures that the parameter field is maximally aligned (and, as a consequence, the other fields are cell-aligned). You can access the name with `name>string`.

compile-only flag While most flags have been eliminated by using methods, we kept the compile-only (aka `restrict`) flag with a much-reduced role: Gforth now has interpretation semantics for all words, but warns in some cases when the interpretation semantics of compile-only words are used. You can use `compile-only? ( nt -- flag )` to check whether a word is compile-only.

link field This field is just like the link field since the dawn of Forth: it contains the nt of the previous word in the same wordlist, or 0 if there is none. You can get to this field with `>link ( nt -- addr )`, but the method `name>link ( nt1 -- nt2 )` works even on nameless words.

Variations

A possible variation of this organization is to move the code field into the virtual table. This would mean that a word header would be one cell smaller, and, for a fixed virtual table (as for closures [EP18]) would be cheaper to create. The disadvantages would be: Each `execute` and other dispatch through the code field would incur an additional indirection. And we would need an additional virtual table for every primitive (each primitive has a different code address); with 413 primitives in Gforth, this would mean 3717 cells for virtual tables, i.e., 3717 words to break even; Gforth on AMD64 has 3922 named words, and a number of unnamed ones, so, before loading user code, the benefit of this size optimization is small.

To avoid the additional indirection, instead of storing the code address in the virtual table, a copy of the code itself could be stored there. Another way to look at this would be that the virtual table would be prepended to the code; the code would need to be duplicated for different virtual tables. With this variation, the vt field would again become a code field.

4.2 Methods

There are various words that take an nt or xt; the behaviour of many of these words depends on the nt/xt. In the old header conditional/`case` code decided what to do by looking at header flags or at the contents of the code field.

The new header uses an object-oriented design[7]: These words are method selectors, and the method implementations are determined when the word is defined. This section describes the methods; defining them is covered in Section 4.5. Apart from the `execute` method which is implemented in native code and represented by a native-code address, all methods are implemented as Forth code represented by an xt.

`Execute ( ... xt -- ... )`

`Execute` takes an xt and performs the semantics represented by that xt. This has been treated as a method since the dawn of Forth: Different word types have different (native) code addresses in their code field;[8] for some word types (colon definitions, `does>`-defined words), this (native) code then calls threaded code that allows varying the behaviour of the word by writing Forth code.

words going through the code field) use indirect-threaded code dispatch [Ert02].

[7] The design is for a fixed set of methods, not for general object-oriented programming. Mini-OOF is a general object-oriented Forth extension with similar features.

[8] In indirect-threaded code; direct-threaded and native code systems have some differences in details, but they also execute some native code when `execute`ing an xt.

applicable to	generated code	`set-optimizer` by	executed Aarch64 instructions
all words	lit <xt> execute	set-execute	14
`does>` words	does-xt <xt>	set-does>	11
constants	lit <value>	constant	2

Figure 4: Different correct implementations of `compile,` for a word defined with `constant` as shown in Section 5.1

`does ( ... body-addr -- ... )`

For a word defined with `set-does>` or `does>`, the native code pushes the body of the word *body-addr* on the stack, and then executes the xt passed to `set-does>` (or the xt representing the code following the `does>`. This xt is the `does` method of the word.

The `does` method is only used by words defined with `set-does>`/`does>`. Its slot can be used for other purposes by other word types; it is then called `extra`.

`Compile, ( xt -- )`

A classic indirect-threaded Forth system uses `,` to `compile,` an xt. But Gforth (since version 0.6) uses primitive-centric threaded code [Ert02], where different words have to be compiled differently ("intelligent `compile,`"); e.g., a colon definition is compiled to the primitive `call` followed by the body address of the called word.

The significance of `compile,` is that 98.8% of the dynamically executed primitives[9] are in threaded code produced by `compile,`, while only 1.2% of the primitives[10] are invoked through `execute` or a deferred word, which use the `execute` method above. So improving the code generated with `compile,` for a word has a much greater effect than improving its `execute`d code.

In Gforth 0.7, `compile,` is implemented as a `case` control structure that looks at the code address in the code field. In Gforth 1.0, `compile,` is a method, so new word types and their code generators can be added without changing existing code.

Note that `compile,` has to be equivalent to

```
: compile, ( xt -- )
  ]] literal execute [[ ;
```

Many uses of `compile,` rely on this relation between `execute` and `compile,`. Every implementation of the `compile,` method has to satisfy this equivalence, and therefore words like `set-does>` that change what `execute` does for a word also change what `compile,` does for a word. To get an optimizing `compile,` implementation, you have to call `set-optimizer` afterwards.

[9] in https://www.complang.tuwien.ac.at/forth/peep/sorted

[10] 1% calls, 0.2% other primitives

There have been attempts to define the `compile,` for some words in a way that does not satisfy this equivalence, as a shortcut to implementing dual-semantics words, but this does not work correctly in all cases. To implement dual-semantics words, set `name>compile` and/or `name>interpret` (see Section 5.5 and 5.6).

Figure 4 shows the effects that different `compile,` implementations have for compiling a constant as defined in Section 5.1, corresponding to different levels of specialization, resulting in fewer executed instructions for the more specialized code.

`Name>interpret ( nt -- xt )`

The Forth-2012 word `name>interpret` has been in Gforth since 1996 under the name `name>int`, but it used a number of `if`s that looked at various flags and fields to produce the correct xt. In Gforth 1.0, it is a method. For normal words (where nt=xt), it just performs a no-op.

`Name>compile ( nt -- xt1 xt2)`

The Forth-2012 word `name>compile` has been in Gforth since 1996 under the name `name>comp`, and it also used a number of `if`s to do its work. In Gforth 1.0 it is a method; for normal words, it just pushes the xt of `compile,`; for ordinary immediate words, it pushes the xt of `execute`. Examples of other implementations will be shown below.

`(to) ( val xt -- )`

Words defined with, e.g., `value` or `fvalue` can be used with `to`. What to do then depends on how the word was defined.

In Gforth 1.0, `to` and `is` are synonyms, so `to` can also be used for words defined with `defer`.

In Gforth 0.7 `to` *`name`* looks at the code field of *name*, and has a `case` structure for all the types of words known to it. Adding a new type requires changing the code of `to`.

In Gforth 1.0, we have a method `(to)` that performs the actual storing of the value *val* (of any type) into the word specified by xt. E.g., for a word defined with `fvalue` the implementation of `(to)` is (simplified):

```
: fvalue-to ( r xt -- )
  >body f! ;
```

One can see this method as the implementation equivalent to the approach used to specify `to` in the Forth-2012 document[11]: There, `to` first performs the generic part of `to`, and then the word-type-specific `to` *name* semantics; and these semantics are specified at each defining word.

`Defer@ ( xt1 -- xt2 )`

Gforth 1.0 also has several `defer`-like defining words, e.g., the standard `defer` and the per-task `udefer`. Words defined with these words all behave like words defined with `defer`: Running such a word executes the xt stored in it; that xt can be read with `defer@` or `action-of`, and it can be written with `defer!` or `is`.

Gforth 1.0 defines `defer@` as a method, which allows different ways of accessing the xt for the different word types. `Action-of` uses `defer@`. `Is` is a synonym for `to`, which makes `Defer!` a synonym of `(to)` (see above). Therefore we do not need a separate `defer!` method.

`Name>string ( nt -- c-addr u )`

In Gforth words defined with `:noname` and `noname` have no name field. Therefore `name>string` is a method:

- For named words, it just returns the address of the name field and the number of characters.
- For unnamed words, it returns an empty string.

`Name>link ( nt1 -- nt2|0 )`

`Name>link` is a method that provides the functionality of a link field even for (unnamed) words which have no link field.

Grouping

These methods can be divided into the following groups[12]:

- `Name>string` and `name>link` each have only two implementations: One for named words and one for noname words. They are otherwise independent of the other methods. Instead of implementing them as methods, one could also use a *noname* bit in the header, and implement these words with `if`.
- `Name>interpret` and `name>compile` define the semantics of named words. They make sense only for named words.
- `Execute`, `does`, `compile,`, `defer@` and `(to)` (in its role of `defer!`) define the behaviour of an xt, and are relevant for unnamed as well as named words.

These differences are also visible in the stack effect: The first two groups take nts, the last group an xt (or body).

Factoring

For the most part, these words were not introduced for this header design, but already existed earlier, and we just turned them into methods:

word	year	origin
`execute`	1970s	early Forth [Moo74]
`does`	1970s	early Forth
`compile,`	1994	Forth-94
`name>interpret`	1996	Gforth 0.2
`name>compile`	1996	Gforth 0.2
`defer@`	2005	Forth 200x
`name>string`	1996	Gforth 0.2

`(To)` corresponds to the "`to` *name* run-time" semantics factoring in Forth 200x introduced in 2009[13].

Only `name>link` was introduced with this header design.

So most of these words have proven their worth as factors of other words for a long time, but are they also good interfaces for method implementations? In our experience they are, and you can judge for yourself by reading Section 5.

One word that has been questioned is `name>compile`, because it represents compilation semantics with two xts instead of just one. The benefit is that `name>compile` is cheaper to implement for normal words and immediate words (i.e., the vast majority of words). Dual-semantics words require an extra layer (see Section 5.5), but that can be done once for all such words (Section 5.6).

4.3 `Find`

While the new header design does not change our implementation of `find` (we already implemented `find` based on `name>interpret` and `name>compile` in the old header), this is a related topic, and may be of interest to the reader. It also is relevant by showing why we do not want to exploit all the flexibility that the two xts returned by `name>compile` offer.

On success, `find` returns an xt and either 1 (immediate[14]) or −1 (otherwise). `Find`-based

[11] http://forth-standard.org/standard/core/TO

[12] These groups could be considered as *interfaces* or *traits* in object-oriented programming languages.

[13] http://www.forth200x.org/documents/forth09-3.pdf

[14] Actually, assuming that `find` should be usable for user-defined text interpreters, we need a different notion of *immediate* than "compilation semantics = execution semantics": If `find` returns an xt in compile state that the user-defined

user-defined text interpreters either `execute` or `compile,` the xt, depending on `state` and the number returned by `find`.

If `name>compile ( name -- xt1 xt2 )` returns either `execute` or `compile,` as xt2, `find` can be implemented in a way that allows such text interpreters:

```
: find {: c-addr -- c-addr 0 | xt 1/-1 :}
  c-addr count find-name dup if
    dup name>compile >r swap name>interpret
    state @ if drop else nip then
    r> ['] execute = if 1 else -1 then
  else
    drop c-addr 0
  then ;
```

The `state`-dependent part caters for dual-semantics words where the xt returned by `name>interpret` is different from the xt1 returned by `name>compile`. The part afterwards extracts the $1/-1$ from the xt2 returned by `name>compile`.

Our implementations of `name>compile` all heed the restriction mentioned above, and that is probably a good idea for all systems that implement `find`.

4.4 `To` optimization and locals

A simple way to compile `to` *name* is to compile a literal for the xt of *name*, followed by `(to)`. But what we actually do is to resolve the method dispatch already at compile time, and compile the implementation of `(to)` for that xt.

But optimization does not stop there. E.g., in the `fvalue-to` case above, we want to resolve the `>body` during compilation, resulting in compiling the body (instead of the xt) as literal, and compiling `f!`, eliminating the colon definition overhead and the `>body` at run-time. We first define the `compile,` implementation for `fvalue-to` as follows (simplified):

```
: fvalue-to-compile, ( xt -- )
  drop ]] >body f! [[ ;
```

We can use this word with `set-optimizer` (see Section 4.5):

```
: fvalue-to ( r xt -- )
  >body f! ;
' fvalue-to-compile, set-optimizer
```

Another syntax that does not require a name for the `compile,` implementation is to use `opt:`:

```
: fvalue-to ( r xt -- )
  >body f! ;
opt: drop ]] >body f! [[ ;
```

text interpreter should `execute`, `find` should also return 1.

We have also implemented Mecrisp's constant folding mechanism [Koc15] in Gforth [Pay19][15]: Compiling a literal pushes it on a compile-time literal stack; `compile,` implementations can access the literal stack to perform operations on the constants at compile time. In the present case, compiling `>body` takes *name*'s xt from the literal stack and pushes *name*'s body. Before generating actual code, the remaining contents of the literal stack are compiled as literals. In the present case, the body is compiled as literal before compiling `f!`. The end result is that `to` *name* (where *name* is an fvalue) is compiled to the same code as

```
[ ' name >body ] literal f!
```

In most cases the optimization is just nice to have. But there is one case where optimizing `to` *name* is essential: locals. Gforth does not keep the headers of locals around until run-time, so using the xt of a local at run-time would not work. The `(to)` implementation of a (cell-sized value-flavoured) local is called `to-w:`. The `compile,` implementation of `to-w:` is (simplified):

```
: to-w:-opt ( xt-to-w: -- )
  ?fold-to >body @ lp-offset ( offset )
  ]] laddr# [[ , ]] ! [[ ;
```

`Compile,` passes the xt of `to-w:` to `to-w:-opt`. `?fold-to ( xt-to -- xt-name )` drops this xt and moves *xt-name* from the literal stack to the data stack.[16] The rest of `to-w:-opt` computes the *offset* of the local from the locals-stack pointer (first line), and then (second line) compiles code for generating the run-time address of the local (`]] laddr# [[ ,`), and for storing the value there (`]] ! [[`).

Similar optimizations are also used for the `defer@` method (and, based on that, `action-of`).

Alternatives to the scheme above are:

- Keep the headers of locals around for the whole session. Then you can just use an unoptimized or minimally optimized implementation of `to` *name*.
- Have additional methods equivalent to `]] literal (to) [[` and `]] literal defer@ [[`. Then you do not need constant folding to avoid having to deal with the xts of locals at run-time. We used this approach in Gforth before we implemented constant folding.

[15] For a description in English, see news:<2019Aug5.121829@mips.complang.tuwien.ac.at>

[16] The other case (no literal) does not happen when the source code is `to` *name*, but is also handled correctly: it just compiles `to-w:` without optimization and exits `to-w:-opt` without performing the words after `?fold-to`.

4.5 Setting method implementations

By setting the method implementations appropriately, we can define words with capabilities that are not properly supported by the old header format (see Section 5), but how do we set them?

The basic approach is that of an object-oriented system based on prototypes [Bor86] rather than classes. A new word is created by building a new header, copying the methods of an existing one, and then changing individual method implementations.

In practice, the usual approach is to call an existing defining word (which at bottom level works by copying and then changing the behaviour of a pre-existing word, which in turn was created by the cross-compiler), and then modifying the behaviour of the resulting word.

You can change individual method implementations of the most recently defined word with the following words:

setter	stack effect	sets
`set-execute`	`( addr -- )`	code field `compile,`
`set-does>`	`( xt -- )`	code field `does compile,`
`set-optimizer`	`( xt -- )`	`compile,`
`set->int`	`( xt -- )`	`name>interpret`
`set->comp`	`( xt -- )`	`name>compile`
`set-to`	`( xt -- )`	`(to)/defer!`
`set-defer@`	`( xt -- )`	`defer@`
`set->string`	`( xt -- )`	`name>string`
`set->link`	`( xt -- )`	`name>link`

In order to preserve the relation between `execute` and `compile,`, every word that changes what `execute` does also has to change what `compile,` does. `Set-execute` changes the `compile,` implementation to the default code generator for all words. `Set-does>` changes the `compile,` implementation to the default code generator for `create...does>` words. Afterwards, you can change the `compile,` implementation with `set-optimizer` to one that generates faster code. Using `set-optimizer` before `set-execute` or `set-does>` (or `does>`) will not have an effect that survives the `set-execute`/`set-does>`.

If you want to change a method of an older word, you can make the older word take the place of the most recently defined word with

```
make-latest ( nt -- )
```

To avoid confusing mixups of behaviours, the behaviour of a word should not be changed after it has been used; the implementations of the behaviour can still be changed (e.g., for optimization).

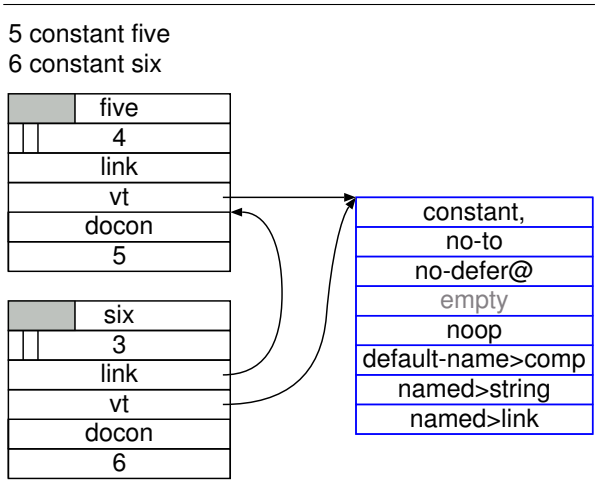

Figure 5: Two words with a shared (deduplicated) virtual table

4.6 Deduplication

Implementing this prototype-based approach is easy if each word has its own vt (virtual table): just copy the vt of the original, and then change it as you please. If you go that way, you can put the virtual table directly into the header instead of in a separate structure.

However, we want to avoid the memory costs of this approach.[17] Therefore we deduplicate the virtual tables: Just before the definition of the next word starts, Gforth checks whether the current word's virtual table is equal to one of the virtual tables that existed before (for words defined earlier). If so, the address of the earlier copy of the virtual table is stored in the current word's vt field, and the current virtual table's memory is reclaimed (see Fig. 5). As a result, we currently only have 117 vts in the Gforth image (for ≈ 4000 words).

You can keep virtual tables in the main dictionary with this scheme (we have done so for a while), but it causes complications in some places. An approach that avoids these complications is to keep the virtual tables in a separate section [Ert16].

Gforth performs deduplication and its reversal (duplication) automatically: If you change a method of a word with a deduplicated vt, Gforth duplicates the vt first. When switching to a new word by defining a new word or with `make-latest`, Gforth deduplicates the vt.

4.7 `Create-from`

Creating a word by starting out with some word, modifying it, and finally deduplicating the vt is

[17] ≈28 000 cells for the Gforth image alone. The memory cost can become a problem even on large machines for code that uses the dictionary for lookup tables, or otherwise creates many words at run-time).

somewhat expensive. This is especially relevant when defining a huge number of words for using a wordlist as a lookup table for data.

A part of that expense could be reduced by using a hash table for deduplication rather than the linear search we use now, but we did not pursue this direction for now. Instead, we introduced

```
create-from ( nt "name" -- )
```

This creates a hidden[18] word with an empty body that has the same method implementations as *nt*. Implementationwise, this means that we just copy the vt address instead of duplicating and later deduplicating the vt.

Gforth uses `create-from` to implement all the common definition words (such as `constant`). But if a word is modified after that (with `immediate`, `does>` or one of the `set-`... words), it incurs the cost of duplication and, later, deduplication.

In combination with `noname`[19], our current implementation still duplicates the vt, changes the `name>string` and `name>link` implementation and eventually deduplicates the vt; we plan to optimize this case in the future.

4.8 Out-of-band data

Gforth keeps some header-related data in other places than the header.

Hash table

Gforth uses a hash table to speed up dictionary searches. This hash table is in `allocate`d memory; it is built on system startup by inserting all the words from the linked-list representation of the wordlists, and is rebuilt in the same way when necessary (e.g., to increase the number of buckets).

The hash table is also needed with the new header design (linear search in a linked list does not become faster with the new header), and our implementation is actually hardly affected by the header change.

Location information

Gforth 1.0 keeps a lot of source code location information around, but it is all out-of-band. In particular, there is an array that contains a location for every dictionary cell. If a dictionary cell is the nt of a word, the corresponding location indicates where the word is defined. If a dictionary cell is some threaded-code cell, the corresponding location indicates the source code for which this threaded-code cell was generated.

[18] You have to `reveal` the word to make it visible.

[19] The next defining word after `noname` produces a nameless word.

4.9 +TO, ADDR

Many Forth systems, including Gforth, also support `+to` *name* (for incrementing *name*) and, in some cases, `addr` *name* (for taking the address of *name*[20]). We will only look at `+to` in the following, but the issue is similar for `addr`.

The most straightforward way to implement `+to` is to have another method `(+to)` and implement `+to` similar to `to`. This leads to similar code at the `+to` definition, but also the various `(+to)` implementations would look very similar to their respective `(to)` implementations; e.g., for `fvalue` they would look as follows:

```
: fvalue-to ( r xt-fvalue -- )
    >body f! ;
opt: drop ]] >body f! [[ ;

: fvalue-+to ( r xt-fvalue -- )
    >body f+! ;
opt: drop ]] >body f+! [[ ;
```

To avoid this code duplication, Gforth employs the following approach: At the start of `TO`/`+TO` text interpretation, the variable `to-style#` is set to indicate which of the two is currently being text-interpreted, then the same code is executed for both words, and in the end, this variable is used to pick the right xt (`f!` or `f+!` in this case) from a table, and then `execute` or `compile,` it. The corresponding code for `fvalue` looks as follows:

```
Create f!-table ' f! , ' f+! ,

: fvalue-to ( r xt-fvalue -- ) \ gforth
    >body f!-table to-!exec ;
opt: drop ]] >body [[ f!-table to-!, ;
```

While using a global variable makes the code smell, variants of this approach have been used in Forth systems for decades, without known problems; in contrast to the `state`-smartness problem [Ert98], the consumer of the value in `to-style#` is not separated from the producer if the `to`-like words parse (as they do in Gforth).[21]

For simplicity of exposition, we ignore this particular twist in the rest of this paper.

[20] `Addr` destroys one of the advantages of value-flavoured words: That they have no aliases and can therefore be allocated to registers, or their accesses reordered wrt memory accesses and accesses to other value-flavoured words. `Addr` exerts this destructive effect already when it *can* be applied to a word, even when it is not actually applied, because there is no guarantee that it will not be applied later. One way to deal with this would be to make it explicit at word definition whether `addr` can be applied to this word, and report an error if `addr` is applied to a word to which it cannot not be applied.

[21] Some other systems have non-parsing variable-setting `to` implementations, and one can produce funny effects with them <news:2017Jan7.150224@mips.complang.tuwien.ac.at>.

5 Examples

This section shows examples of using the `set-...` words to perform things that were much less elegant with the old header. The shown code is usually simplified: It does not show some of the complications for extra features of Gforth that are not the focus of the present work, e.g., `+to`.

5.1 Constant

```
: constant ( x "name" -- )
  create ,
  [’] @ set-does>
  [: >body @ ]] literal [[ ;] set-optimizer
;
```

This first defines `constant` as a `create...does>` word with the `does>` action `@`. The programmer is not allowed to change constants, so our `compile,` implementation (the quotation before `set-optimizer`) compiles the constant's value x as a literal, which has the same effect, but is cheaper than the code that would be produced without the `set-optimizer` part (2 Aarch64 instructions instead of 11).

The actual Gforth implementation of `constant` uses a native-code `docon`. This `docon` has been there since Gforth's inception, and was important for performance before we switched to primitive-centric threaded code (it takes 5 Aarch64 instructions, compared to 10 for the `does>`-based definition above).

5.2 Fvalue

```
: fvalue-to ( r xt-fvalue -- )
    >body f! ;
opt: drop ]] >body f! [[ ;

: fvalue ( r "name" -- ) \ float-ext
  fconstant
  [: >body ]] Literal f@ [[ ;] set-optimizer
  [’] fvalue-to set-to ;
```

`Fconstant` is defined analogously to `constant`; `Fvalue` reuses the `execute/does` part of `fconstant`, but defines a different optimizer that accesses the body at run-time in order to treat changing values correctly; the resulting code performs 3 Aarch64 instructions.

The `to` implementation is set to `fvalue-to`. Its definition and its optimizer are quite straightforward. One non-obvious thing that happens when compiling `to` *name* is that *name*'s xt is compiled as a literal, followed by `>body f!`; constant folding performs the `>body` at compile-time rather than at run-time, so the final code generated when compiling `to` *name* is a literal followed by `f!` (4 Aarch64 instructions, compared to 20 without this optimizer).

5.3 defer

```
: value-to ( x xt -- )
  >body ! ;
opt: ( xt -- ) \ run-time: ( x -- )
  drop ]] >body ! [[ ;

: defer-defer@ ( xt1 -- xt2 )
  >body @ ;
opt: ( xt -- )
  drop ]] >body @ [[ ;

: perform @ execute ;

: defer ( "name" -- )
  create [’] abort ,
  [’] perform set-does>
  [: >body ]] lit-perform [[ , ;]
                          set-optimizer
  [’] value-to set-to
  [’] defer-defer@ set-defer@ ;
```

This example shows setting the `execute/does>`, `compile,`, `to` and `defer@` methods, with `to` and `defer@` having optimizers.

5.4 Default and immediate compilation semantics

```
: default-name>comp ( nt -- xt1 xt2 )
    name>int [’] compile, ;

: imm>comp ( nt -- xt1 xt2 )
    name>int [’] execute ;

: immediate ( -- ) \ core
    [’] imm>comp set->comp ;

: immediate? ( nt -- flag )
    name>compile nip [’] execute = ;
```

The default compilation semantics are to compile the execution semantics, i.e. (in Gforth 1.0) the interpretation semantics. `Default-name>comp` implements the default compilation semantics. Most defining words (e.g., `create` and `:`) use `default-name>comp` as implementation of the `name>compile` method.

`Immediate` changes the compilation semantics to be the same as the execution/interpretation semantics. So this implementation changes the `name>compile` implementation to produce the xt of `execute` (instead of `compile,`) as xt2; so when you `execute` the result, it eventually executes the xt of the word.

`Immediate?` shows how one can determine the immediacy of a word based on `name>compile`. But there is actually rarely a need to check for immediacy. Instead, you can use `name>compile` directly. E.g., a text interpreter in compile state can just perform `name>compile execute` to perform the compilation semantics of a word, without worrying about immediacy. `Postpone` can also be implemented without worrying about immediacy:

```
: postpone ( "name" -- )
  parse-name find-name dup 0= -13 and throw
  name>compile swap ]] literal [[ compile,
 ; immediate
```

5.5 To

`To` is an example of a word that has neither default compilation semantics nor immediate compilation semantics.[22] We first show how to implement `to` directly:

```
: to-int ( v "name" -- )
  parse-name find-name dup 0= -13 and throw
  (to) ;

: to-comp ( compilation: "name" -- )
          ( run-time: v -- )
  parse-name find-name dup 0= -13 and throw
  ]] literal (to) [[ ; immediate

: to-name>comp ( nt -- xt1 xt2 )
  drop ['] to-comp ['] execute ;

synonym to to-int
' to-name>comp set->comp
```

`To-int` has the same interpretation semantics as `to`, and `to-comp` (explained below) the same compilation semantics. We then define `to-name>comp`, which later serves as the `name>compile` implementation for `to`. Finally, `to` is defined as synonym for `to-int`, copying (among others) the interpretation semantics of `to-int`. The compilation semantics is then overwritten with `set->comp`.

Note that `to-comp` resolves the `(to)` method at compile time through the optimizer of `(to)` (not shown here).

5.6 Interpret/compile:

Directly defining a word like `to` is cumbersome, so Gforth has two more convenient ways to define them. First, there is `interpret/compile:`, which has been in Gforth since 1996. You can use it to define `to` as follows:

```
' to-int ' to-comp interpret/compile: to
```

The interface `interpret/compile:` has been designed for the previous header implementation, but can be implemented with the new header implementation relatively straightforwardly:

```
: i/c>comp ( nt -- xt1 xt2 )
    >body cell+ @ ['] execute ;

: interpret/compile: ( i-xt c-xt "name" --)
    defer , lastxt defer!
    ['] defer-defer@ set->int
    ['] i/c>comp set->comp
    ['] no-to set-to
    ['] no-defer@ set-defer@ ;
```

The interpretive behaviour of *name* is like for a `defer`red word, so we implement `interpret/compile:` as inheriting from `defer`. It stores *i-xt* to the first cell of the body with `lastxt defer!`. The `name>interpret` implementation fetches *i-xt* by reusing `defer-defer@`; note that unlike most words, for *name* xt≠nt, so `name>interpret` is not a noop here.

In addition, it stores the *c-xt* in the second cell of the body (with `,`). The compilation semantics is to `execute` *c-xt*, and the `name>compile` implementation `i/c>comp` implements this behaviour.

We inherit from `defer`, so we inherit the `(to)` and `defer@` implementations for `defer` (i.e., `is`, `defer!`, `action-of` and `defer@` would work). We do not want that, so we override these methods with `no-to` and `no-defer@`, which report an error if these words are used on *name*. Note that the interpretation semantics explicitly uses `defer-defer@` rather than the generic `defer@`, because the latter no longer works after the `set-defer@`.

`Interpret/compile:` has been criticized on aesthetic grounds, so Gforth also has code for `compsem:`, which is used like `opt:`. The implementation of `compsem:` is short[23], but would require explaining Gforth features beyond the scope of this paper, so we skip it here.

[22] Standard programs must not tick or `postpone` `to`, making an immediate `state`-smart implementation possible. Other standard words (e.g., `s"`) do not have this restriction. We demonstrate a `to` that works even without this restriction, as an example for this whole class of words, because it also shows the usage of `(to)`.

[23] http://git.savannah.gnu.org/cgit/gforth.git/tree/set-compsem.fs

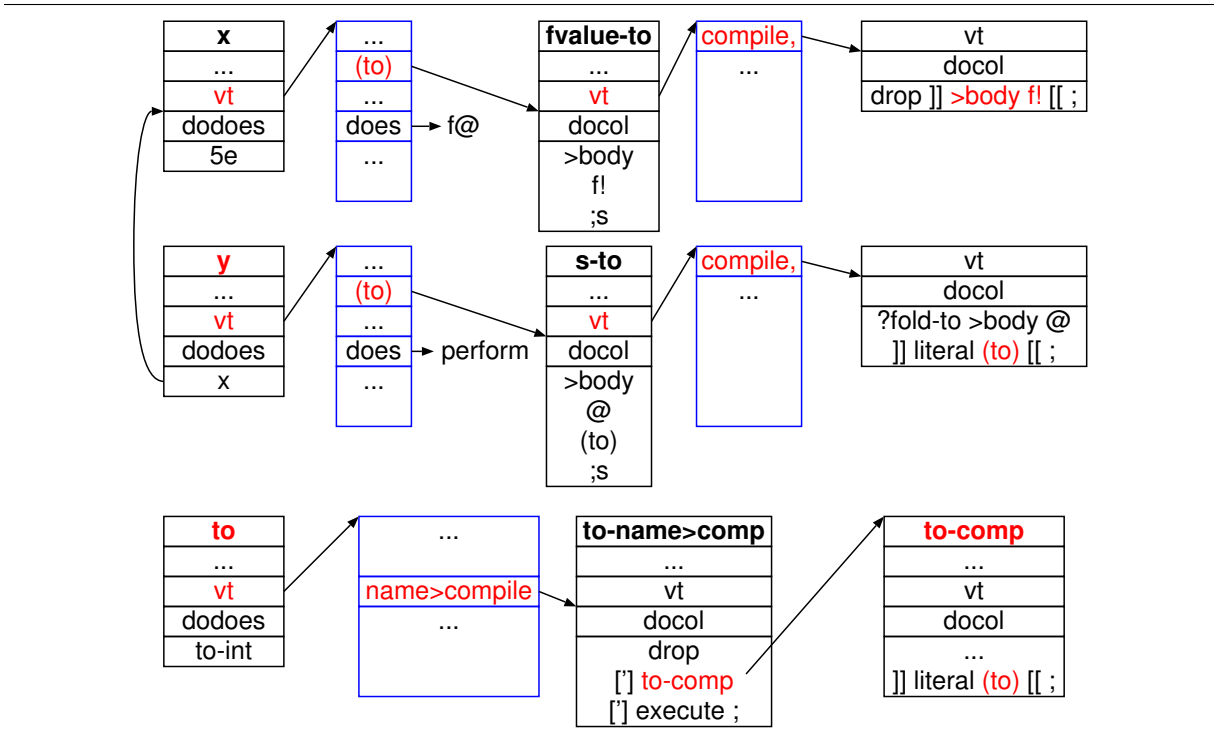

Figure 6: Data structures for the example

5.7 Synonym

```
: s-to ( val nt -- )
  >body @ (to) ;
opt: ( xt -- )
  ?fold-to >body @ ]] literal (to) [[ ;

: s-defer@ ( xt1 -- xt2 )
  >body @ defer@ ;
opt: ( xt -- )
   ?fold-to >body @ ]] literal defer@ [[ ;

: synonym ( "name" "oldname" -- )
  defer
  parse-name find-name dup 0= -13 and throw
  dup lastxt defer!
  compile-only? if compile-only then
  [: >body @ compile,     ;] set-optimizer
  [: >body @ name>interpret ;] set->int
  [: >body @ name>compile   ;] set->comp
  ['] s-to set-to
  ['] s-defer@ set-defer@ ;
```

`Synonym` stores the nt of *oldname* in the body of *name*. The normal way to deal with *name* is through words that use the nt. `name>interpret` gets the nt of *oldname* and gets its xt; likewise for `name>compile`. Text interpretation, ticking and `postpone` work through these words.

The implementation of `(to)` also gets the nt of *oldname* and performs its `(to)` implementation; if *oldname* has `to` ***oldname*** semantics, they will be performed; if not, *oldname*'s `(to)` implementation reports an error. The `(to)` implementation also has an optimizer that computes *oldname*'s nt and compiles its `(to)` implementation (which in turn triggers *oldname*'s `(to)` optimizer). The implementation of `defer@` is analogous.

Note that these things work for arbitrarily deep chains of synonyms, always delegating to implementation of the same method at the next level.

In order to make *name*'s nt also work as xt, `synonym` inherits from `defer`. So, if you `execute` the nt of *name*, this will `execute` the nt of *oldname*; eventually a word with nt=xt is performed. Synonym also has an optimizer for `compile,`ing the nt, which optimizes away the indirection.

5.8 Example

This example presents these words in action. Figure 6 shows the data structures involved in the example.

```
5e fvalue x
synonym y x
: foo to y ;
```

In this code, `to y` performs the compilation semantics of `to` by performing the `name>compile` method and its implementation `to-name>comp`. As a result, the text interpreter `executes to-comp`, which first parses `y` and produces its nt, and then compiles the nt as a literal followed by `(to)`. `(to)`

has an optimizer that takes the preceding literal (`y`'s nt) into account and then `compile,`s `s-to`. This calls the optimizer of `s-to`, which in this case consumes the preceding literal (still `y`'s nt), then fetches the body of `y` (giving `x`'s nt), and compiles that as literal, followed by `(to)`. Again, `(to)`'s optimizer resolves this, this time to `compile,`ing `fvalue-to`. This calls the optimizer of `fvalue-to`, which compiles `>body f!`. `>Body`'s optimizer consumes the preceding literal (`x`'s nt), and compiles `x`'s body as literal. The following `f!` is compiled as-is. As a result, the code for `foo` is the same as produced by directly writing

```
: foo [ ' x >body ] literal f! ;
```

6 Empirical Results

The goal of this work has been design cleanliness and flexibility, not performance. But here we demonstrate that performance does not suffer (and actually benefits a little).

Unfortunately, since the work on the new headers began in 2012, a lot of other changes have been made in Gforth (> 3000 commits), and the header-related changes cannot be isolated with reasonable effort.

Nevertheless, we take various measures to isolate the performance effects: We compiled the last old-header Gforth[24] and a recent new-header Gforth[25] with gcc-4.7.2, a compiler version where both the old and the new version use all the performance features of Gforth. We use `--ss-number=14` to ensure that both versions use almost the same static superinstructions. And finally, instead of measuring a task such as compilation of a complete file, where other features (in particular, IDE features like `locate` and `where`) impact the performance, we look at individual operations.

In the following results, we do not use the simplified implementations shown above, but the more complex implementations in Gforth.

All results are in cycles of user time on a Core i7-4790K (Haswell).

6.1 `Find-name`

Gforth uses an out-of-band hash table that does not use the link field of the header during `find-name`, so the new header structure should have little effect on `find-name` performance. Nevertheless, the text interpreter calls `find-name` once for every word it interprets or compiles, so we present performance numbers here. The following table shows the average run-time of one invocation of `find-name` on one word out of a set of 1602 words (all present in the search order):

cycles	old	new
`find-name`	3184	3855

Apparently the other changes between these version have slowed down `find-name` a little.

6.2 `Name>interpret` and `name>compile`

When text interpreting, the nt produced by `find-name` is then usually processed either by `name>interpret` (when interpreting) or by `name>compile` (when compiling). For the old header, normal words, aliases[26], and `interpret/compile:` words are treated in different paths of a cascade of `if`s spread across a number of words. For the new header, different method implementations are called. So for different word types, the performance may be different; we therefore measure the performance of `name>interpret` and `name>compile` for different word types:

cycles	old	new
`name>interpret` normal	123.8	15.0
`name>interpret` alias	123.3	24.6
`name>interpret` i/c	124.2	24.7
`name>compile` normal	136.4	32.1
`name>compile` alias	138.8	47.4
`name>compile` i/c	143.2	30.1

This part of text interpretation is quite a bit faster with the new header thanks to not having to perform a lot of tests, but in the overall scheme of things, this speedup vanishes in the noise.

6.3 `Compile,`

When compiling, Gforth eventually `compile,`s some xt (e.g., because `name>compile` produced the xt of `compile,` as xt2, and the text interpreter `execute`s that), and that `compile,` eventually compiles a primitive. The path from entering `compile,` to compiling a primitive has changed a lot: With the old header, `compile,` uses an `if`-cascade to decide the word type and what code to generate; the new header uses method dispatch instead. Depending on where in the `if`-cascade a word type is, the performance can vary; in particular, `does>`-defined and `value`-defined words are early in the `if`-cascade; by contrast, the old `compile,` recognizes primitives by excluding all other word types, so the old `compile,` performs the the longest chain of `if`s for them.

In the following, we present the run-time of the whole `compile,`, as well as a number *without code generation*. The latter number does not include the actual code generation of the primitive, nor (in the

[24] commit 617d4a8deccf5f4eefeb236f972171d6f65bb685
[25] commit 045ff553a7c6304015be66533ead115c93866882
[26] Aliases are similar to synonyms; synonyms are not supported in the old header.

new system only) recording which threaded-code location corresponds to which source code location; it does include the time spent on laying down the inline parameters of the primitives that execute the does and value words.

cycles	old	new
`compile,` does	1074	1156
without code generation	153	51
`compile,` value	963	1103
without code generation	151	49
`compile,` primitive	1013	985
without code generation	235	12

In the *without code generation* lines, we can see that the overhead of selecting what to compile has become much smaller. However, that overhead is just a small part of the costs of `compile,`, and the additional work of recording the source/threaded-code correspondence has more than made up for these savings.

6.4 `>Name`

Sometimes it is useful to get back from that xt of a word to the nt; e.g., the decompiler does it in order to print the name of a word. Gforth has a word `>name ( xt -- nt|0 )` that does this or returns 0 if its argument is not an xt. With the new header, the conversion from xt to nt is a noop (but if you want to make it explicit, you can use `xt>name`). But `>name` also does the checking, and Gforth 1.0 uses relatively reliable, but costly heuristics to do that. Gforth 0.7 uses heuristics for both conversion and checking.

cycles	old	new
`' create >name`	1281	2280
`' noop >name`	51971	2314

For the old scheme, `>name` is particularly expensive for primitives like `noop`, while for normal words like `create`, it is not so extreme. For the new headers, the heuristics for non-primitives are more expensive than for the old headers. But these costs have not been a performance bottleneck yet, and we can think of some ways to improve these costs if they ever prove to be a problem.

6.5 Header creation

As mentioned in Section 4.7, creating words is more expensive in Gforth 1.0 if the overhead of duplication and deduplication is incurred, but is fast if that is avoided with `create-from`. We measured this with a benchmark that defines 1,000,000 constants in a wordlist, using different implementations of `constant`. The numbers reported are cycles per created word; for the *new* results, we use snapshot `gforth-0.7.9_20190829` compiled with `gcc-4.9.2`.

		new	
cycles	old	deduplicated	`create-from`
`constant`	2168	9477	1742

This performance difference is not just relevant for applications that use wordlists as a data structure. Loading programs is also affected: We saw a speedup by 30% when loading the OpenGL and Xlib libraries.

7 Related work

In the beginning, Forth stored the length and 3 letters of the name [Moo74]. Fig-Forth supported full-length names, optionally shortened to the length stored in `width` [Tin81]. As a result, it was expensive to get from the name field to the link field (as required repeatedly during name search) and other fields. Smith proposed [Smi80] to reduce this cost by moving the name characters before the count byte, with the link field still pointing to the count byte; In Smith's scheme, the name characters are stored in reverse order, because thanks to `width` they system may store only a part of the name, and that part should be the start of the name. Our new header is similar in putting the variable-length name string before the rest of the header, at a negative offset from the nt address that we use for header accesses; however, we store the name characters in conventional order, because we can use the count to get at the start of the name.

Shaw [Sha88] puts multiple code fields in headers. In addition to the ordinary code field, a word can have a `to` code field, which corresponds to our `(to)` method, but uses a different execution mechanism. Shaw also uses the multiple code field mechanism to get rid of `state`-smart words, by having an optional code field for the compilation semantics (the ordinary code field implements interpretation semantics); it uses the same code field for that as for `to`, using flags to decide if that code field is used for compilation semantics. While there are significant differences from the new Gforth header, Shaw's work is conceptually still closer than all others.

CmForth[27] by Charles Moore and Pygmy[28] by Frank Sergeant implement dual-semantics words by putting a word for the interpretation semantics in the `FORTH` vocabulary and a word for the compilation semantics in the `COMPILER` vocabulary, with the text interpreter searching the appropriate vocabulary for the current use. Mark Humphries also implements dual-semantics words with multiple headers, but he puts them in the same wordlist, with flags that indicate whether the word should be found when looking for a certain semantics.

[27] https://raw.githubusercontent.com/ForthHub/cmFORTH/combined/cmforth.fth

[28] http://pygmy.utoh.org/pygmyforthmanual.html#h1id47

MPE's VFX Forth has `set-compiler` that works like Gforth's `set-optimizer`: It changes what `compile,` does for the preceding word [MPE16, Chapter 19.7.3]. This suggests that VFX implements the intelligent `compile,` in a similar way as the new Gforth header, but to the best of our knowledge this has not been published.

In 2004, Ertl sketched[29] a header structure with an additional field for the intelligent `compile,`, and, for named words, an xt2 field for implementing compilation semantics: for normal words, the xt2 field would contain the xt of `compile,`; for immediate words, the xt of `execute`; and for other words, it would contain something else. However, it requires some additional complexity to implement `find` in a way that supports user-defined text interpreters.

The new Gforth header differs from the 2004 header ideas in that it implements compilation semantics by defining what `name>compile` does for a word. This means that unlike in the 2004 ideas, the xt1 of a compilation token can be different from the xt representing the interpretation semantics, thus making it simpler to implement `find`. In addition, the new Gforth header allows changing what several other words do for the present header, which supports defining `value`-like words, `synonym`s, etc. And it stores the xts for all these method implementations in a separate structure (vt) that is deduplicated.

8 Conclusion

Gforth's old header (based on the original Forth header) leads to complex and inflexible implementations of words like `compile,`, `name>interpret`, `name>compile,`, and `to`; it supports dual-semantics words through an ugly hack; we did not even implement `synonym`, because that would have required adding more special cases.

By contrast, the new Gforth header has a prototype-based object-oriented design that allows extending the behaviour of words like `compile,`, `name>interpret`, `name>compile`, and `to` for individual words. This flexibility makes it relatively easy and compact to implement, e.g., `synonym` such that the created synonyms also work with `to` if the original worked with `to`.

Acknowledgments

We thank the reviewers for their comments, which helped to improve the paper.

[29] https://www.complang.tuwien.ac.at/forth/header-ideas.html

References

[Bor86] Alan Borning. Classes versus prototypes in object-oriented languages. In *ACM/IEEE Fall Joint Computer Conference*, pages 36–40, 1986. 4.5

[Bro84] Leo Brodie. *Thinking Forth*. Fig Leaf Press (Forth Interest Group), 100 Dolores St, Suite 183, Carmel, CA 93923, USA, 1984. 1

[EP18] M. Anton Ertl and Bernd Paysan. Closures — the Forth way. In *34th EuroForth Conference*, pages 17–30, 2018. 4.1

[Ert98] M. Anton Ertl. `State`-smartness — why it is evil and how to exorcise it. In *EuroForth'98 Conference Proceedings*, Schloß Dagstuhl, 1998. 2, 4.9

[Ert02] M. Anton Ertl. Threaded code variations and optimizations (extended version). In *Forth-Tagung 2002*, Garmisch-Partenkirchen, 2002. 2, 6, 4.2

[Ert16] M. Anton Ertl. Sections. In *32nd EuroForth Conference*, pages 55–57, 2016. 4.6

[Koc15] Matthias Koch. Flags, Konstantenfaltung und Optimierungen. *Vierte Dimension*, 31(arm):16–18, 2015. 4.4

[Moo74] Charles H. Moore. Forth: A new way to program a mini-computer. *Astron. Astrophys. Suppl.*, 15:497–511, 1974. 1, 4.2, 7

[MPE16] Microprocessor Engineering. *VFX Forth for x86/x86 64 Linux*, 4.72 edition, 2016. 7

[Pay19] Bernd Paysan. Constant Folding für Gforth. *Vierte Dimension*, 35(2):17, 2019. 4.4

[Pel17] Stephen Pelc. Special words in Forth. In *33rd EuroForth Conference*, pages 37–45, 2017. 2

[Sha88] George W. Shaw. Forth shifts gears. *Computer Language*, pages 67–75 (May), 61–65 (June), 1988. 7

[Smi80] Robert L. Smith. A modest proposal for dictionary headers. *Forth Dimensions*, I(5):49, 1980. 7

[Tin81] C. H. Ting. *Systems Guide to fig-Forth*. Offete Enterprises, Inc., San Mateo, CA 94402, 1981. 7