Memory Safety Without Tagging nor Static Type Checking

M. Anton Ertl*
TU Wien

Abstract

A significant proportion of vulnerabilities are due to memory accesses (typically in C code) that memory-safe languages like Java prevent. This paper discusses a new approach to modifying Forth for memory-safety: Eliminate addresses from the data stack; instead, put object references on a separate object stack and use `value`-flavoured words. This approach avoids the complexity of static type checking (used in, e.g., Java and Factor), and also avoids the performance overhead of dynamic type checking for non-memory operations. This paper discusses the consequences of this approach on the language, and on performance.

1 Introduction

Accessing arrays with an index outside the array bounds is usually a bug. Other memory access errors include reading an uninitialized location, accessing a field of the wrong structure, and accessing memory that has been freed. Memory access errors are common as sources of program crashes, but they are also a common source of vulnerabilities: Gaynor reports [Gay20] that at least 65% of vulnerabilities in various software environments are due to memory safety issues.

From pretty early on there were programming languages where such bugs could not happen or would be caught and reported, e.g., Lisp and Algol 60. Such programming languages are called *memory-safe languages*.

Forth, on the other hand, allows the programmers to shoot themselves in the foot, and relies on careful programmers to avoid such bugs; while this has some advantages, there are occasions when some programmers prefer memory safety. The Forth world has some answers in this area, e.g., Factor and Oforth, but they rely on static type checking and/or tagging all data for dynamic type checking.

In the present work, we introduce the approach of avoiding some of that type checking by eliminating addresses from the data stack, and instead keeping object[1] references on an object stack. The benefits of this approach are that it avoids both the complexity of static type checking as well as the slowdown of tagging and tag-checking.

In Section 2 we look at the problem and how others have solved it. Section 3 describes our own approach; in particular, the main contribution of this work is to keep object references separate from other data through separate stacks (Section 3.1) and value-flavoured words (Section 3.6). The rest of Section 3 describes various other changes necessary for memory safety, but most of that is relatively straightforward. Section 4 gives an idea about the difference between Forth and Safe Forth by dividing the Core wordset of standard Forth into words that are unchanged in Safe Forth, and words that need various changes. Section 5 discusses topics beyond 1:1-correspondence between Forth and Safe Forth. We present implementation approaches for some common operations in Safe Forth and their theoretical effect on performance in Section 6. Safe Forth is currently a paper design (Section 7). Finally, Section 8 discusses some related work.

1.1 Safe Forth or a memory-safe Forth?

The intention of this paper is to explore the idea of memory-safety without tagging and without static type checking in general. One can build a number of different languages on that foundation, which makes the exposition challenging: Describing only one of these languages loses generality, but makes the description easier to understand. Therefore we take this approach, and we call that language *Safe Forth*, but we occasionally describe alternative variants.

Safe Forth is object-oriented; this plays a minor role in much of this paper, but occasionally shines through. A more detailed description of the object model is given in Section 3.8, where the topic is first discussed in more depth.

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

[1] Here *object* refers to a piece of memory, similar to a contiguous region in standard Forth, or an object in the C standard. The division of memory into pieces instead of providing a flat address space is universal in memory-safe languages. However, elsewhere in this paper *object* typically is used in the object-oriented sense.

2 Existing approaches

Most languages in wide use today are memory-safe, with the exceptions being C, C++, assembly language and Forth; languages like Rust and Ada have unsafe escape hatches, and many others have interfaces to C that can also serve as escape hatches (but into C, not an unsafe dialect of the base language).

As an example, Java has primitive types (such as `int` or `double`) and reference types (objects including arrays). Java has static type rules and the Java compiler applies them and knows which type an expression has. On the level of objects, the type checker often does not know the exact class/type of an object, just that it is a subclass of a certain class; the exact class information is present at run-time in a field at the start of each object and is used for method dispatch and for subtype checking.

This scheme is roughly followed by all object-oriented languages, but in many language implementations the primitive types and object references are not distinguished by static type checking, but by tagging: A bit or a few bits of a machine word (a cell in Forth terminology) are reserved for indicating the type of the machine word: each primitive type represented as a machine word gets a different tag, and object references also get their tag (usually one tag for all object references, with more information available at the start of the object).[2] There are also cases like the Ocaml interpreter that uses tagging in a statically type-checked language to simplify the garbage collector.

In the world of Forth-like languages, Factor uses static type checking, while Oforth uses tagging.

These implementations ensure memory-safety as follows:

Out-of-bounds access: Array accesses are bounds-checked. Sophisticated compilers can eliminate a significant proportion of these bounds checks [BGS00].

Uninitialized location: All locations are initialized. Java and Oforth zero all locations: this initializes integers to 0, FP variables to 0.0, and reference types to `null`.

Access to the wrong field: Java only allows accesses to fields that belong to the (statically known) class of the object. Java allows casting to a class C in order to establish that static knowledge, but then Java tests during the cast (at run-time) whether the object is an instance of C. Likewise, other languages either have to establish static knowledge or check on every field access that the object has that field. Like for bounds checks, there are ways to reduce the number of necessary checks by increasing compiler sophistication.

Use after free: The common approach to avoiding use-after-free bugs is to let the language perform automatic storage reclamation, either through garbage collection or through reference counting (Python). Rust employs a sophisticated static type system that ensures that no reference to an object remains when the object is freed.

[2] There is also a technique called NaN-boxing which for the purposes of this paper is a variant of tagging and is not discussed further here.

3 Safe Forth

This section discusses how Safe Forth differs from Forth, and how its approach achieves memory-safety.

3.1 Basic approach

The basic idea in the present approach is to separate object references from other data types by putting object references on a separate objects stack, and no addresses on the data stack. Words that work with object references take them from the objects stack, and words that work with non-reference data take them from the data and FP stacks.

This makes it unnecessary to use tags or sophisticated type checkers, and yet (with the right setup) it is impossible to perform address arithmetic and similar things that are incompatible with memory safety.

In particular, consider the twenty dynamically most frequently executed primitives in the statistics at http://www.complang.tuwien.ac.at/forth/peep/sorted:

1	13.5%	`;s`
2	13.2%	`col:`
3	**9.0%**	`@`
4	*5.1%*	*`?branch`*
5	4.6%	`lit`
6	3.4%	`var:`
7	3.4%	`dup`
8	3.2%	`user:`
9	3.0%	`swap`
10	*2.8%*	*`+`*
11	2.5%	`con:`
12	2.0%	`>r`
13	1.9%	`r>`
14	*1.8%*	*`0=`*
15	*1.3%*	*`and`*
16	**1.3%**	`c@`
17	**1.2%**	`!`
18	1.2%	`over`
19	*1.1%*	*`cells`*
20	1.1%	`rot`

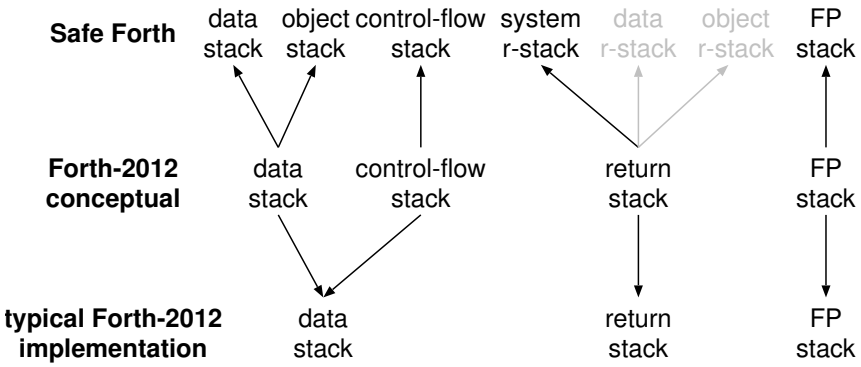

Figure 1: Stacks in Forth and Safe Forth

The black numbers are for words that do not need to handle tags in a tagged implementation, either because they don't handle data, they handle arbitrary data, or they push pre-tagged data (e.g., `lit`). The *slanted blue* lines are for words that would need to deal with tags in a tagged system, but do not need that with our approach (or with static type checking). The **bold red** lines are for words that deal with addresses; in a tagged system like Oforth they are replaced with words that have to check both the tags and the class descriptors of the object references, while with our approach (and with static type checking) you avoid the tag checking (but still have to check the class descriptor).

For the four problems mentioned above our approach is:

Out-of-bounds access: Array accesses are bounds-checked (and first the object is checked to be an array of the appropriate type).

Uninitialized location Values, arrays, and structures are zeroed on creation.

Access to the wrong field A field access checks whether the object actually has that field. It is possible to optimize this check away in some cases with relatively little complexity; e.g., in a method of class C we know that `this` is an instance of C.

Use after free Use garbage collection. Other approaches may be possible, but are unlikely to be simpler.

What this approach does not give us is type checking of the data on the data stack. If you want to add 1 to the letter `A`, you still can[3]. You also don't have to pay the compiler complexity or runtime cost of such type checking.

[3]While Safe Forth is probably not something Chuck Moore is interested in, at least in this respect it follows his preferences [RCM96]

3.2 Stack underflows and overflows

Stack overflows and underflows undermine memory safety if they are not caught. Fortunately they can be caught at no run-time cost on systems with a paged MMU, by putting unaccessible guard pages around each stack. This approach has been used by Gforth since almost its inception, and is a good approach for Safe Forth.

Gforth-fast keeps stack items in registers; to avoid spurious underflow exceptions from traffic between these stack items and the stack in memory, several memory slots below the stack bottom are left accessible. In a Safe Forth using this technique, the registers and the extra memory slots of the object stack have to be initialized to safe values (e.g., `null`); then, while accessing stack items below the bottom is possible, this cannot subvert memory safety.

If MMU-based stack bounds checking is unavailable, the stack pointer can be bounds-checked (more expensively) with code every time it is changed.

3.3 Return stack

In addition to keeping object references on a separate stack, we also have to ensure that we don't open a memory-safety hole through the return stack. Forth systems typically keep return addresses on the return stack, as well as counted-loop parameters and cells moved by the program from the data stack with words like `>r`. In ordinary Forth, a program can execute arbitrary code with `>r exit`, which is not compatible with memory safety.

Our solution to this problem is to just exclude the return stack words like `>r` from Safe Forth.

Another solution is to split the return stack into three stacks: A system return stack for system-execution types (return addresses and loop control paramaters), a data return stack for stashing away data stack items, and an object return stack for stashing away object stack items. However, if you

implement locals, the additional benefit of the latter two stacks does not seem to be worth the cost.

In either case, `exit` inside a counted loop poses a problem (unless we split return addresses and counted loop parameters into using different stacks). Standard Forth has `unloop` to remove counted-loop parameters, but memory-safety can be violated in typical implementations by using `unloop` in a non-standard way. However, because the stack for system-execution types now only contains these types, it is relatively straightforward to implement `exit` without needing `unloop`: Just count the number of counted loop nests, and let `exit` compile code for dropping the corresponding number of loop parameters before compiling the return.

Figure 1 shows the relation between the stacks in Forth and Safe Forth. Optional stacks are shown in light gray.

3.4 Control-flow stack

The control-flow stack contains information about incomplete control structures during compilation. On most systems it is implemented on the data stack (although the standard does not guarantee that).

We considered putting the control-flow stack items on the object stack in Safe Forth. However, in Safe Forth we have to ensure that `do`-like words are matched with `loop`-like words not only during compilation, but also at run-time (to avoid memory corruption by mixups of loop-control parameters and return addresses).

Eventually the simplest way to achieve this is to have a separate control-flow stack.

3.5 `Null`

`Null` is a value on the object stack without object; trying to access through a `null` reference throws an error. `Null` is implemented as address 0 (to make type-ignorant initialization possible). Programs can test whether a reference is `null`.

3.6 No `variables`

`Variables` push an address, so Safe Forth eliminates them. They are replaced with `values`. In addition to the classical values that communicate with the data stack, there are `ovalues` for holding object references; an ovalue `ov` pushes its content on the object stack, and `to ov` consumes an element from the object stack.

In all other areas variable-flavoured words for storing data are eliminated and replaced with value-flavoured words, with both data-stack and object-stack variants. In particular, field-access words are value-flavoured (Forth-2012 only provides variable-flavoured field words).

Locals are value-flavoured in Forth-2012, which is kept in Safe Forth. The object-stack variant is defined by using `o:` in front of the local. E.g.

```
{: a  o: b  c :}
```

defines two data-stack locals `a` and `c` and an object-stack local `b`.

All these value-flavoured words for storing data initialize the data to `0`, `0e` or `null` if there is no initialization value given (e.g., for locals after `|`).

3.7 Fields in structures

This section discusses the access of fields in structures in a non-object-oriented memory-safe Forth. We use the Forth-2012 structure syntax in this section, with modifications.

A (value-flavoured) field is defined with one of the defining words `ovalue: value: cvalue: fvalue:` etc.[4] Fields are defined in the context of a structure.

For a field `f`, performing `f` takes a reference from the object stack and checks if the reference points to a structure/object that has a field `f`. If so, it pushes the content of that field in the structure on the appropriate stack (data, object, or FP stack). If not, it throws an exception.

Performing `to f`[5] is very similar, except that it takes a value from the appropriate stack and stores it in the field.

Forth's field-defining words keep the offset on the data stack during the definition, but that would mean that the program could change offsets and thus undermine memory-safety, so in Safe Forth the offset is kept in a hidden variable.[6] This means that we have to use `begin-structure ... end-structure` rather than the alternative `0 ... constant`.

The structure name pushes an object representing the structure type on the object stack, and `new` allocates an object of that type and initializes it with the type, and all fields zeroed.

Figure 2 shows an example of a structure. We discuss the representation of the type in Section 6.

3.8 Objects and fields

Safe Forth uses an object model with single inheritance of fields and method implementations, where every method selector can be used with any class (duck typing). When invoking a method, the top of

[4] Many of these defining words are also defined for Forth in Gforth's `struct-val.fs`.

[5] The alternative syntax `->f` is already implemented in Gforth.

[6] Alternative: an opaque object referenced through the object stack.

```
begin-structure intlist
  ovalue: intlist-next
  value:  intlist-val
end-structure

intlist new ovalue x
5 x to intlist-val
```

type	intlist
intlist-next	null
intlist-val	5

Figure 2: Source code for creating a structure and the resulting memory for the structure

the object stack O is removed, put into `this`, and used for method dispatch (calling the right method implementation for the combination of the class of O and the method selector).

Fields in objects are similar to fields in structures, with one difference: they refer to the object in `this` rather than the object stack, and do not consume that object. Because `this` is set on method entry, we know that the class D of the object is a subclass of the class C for which the method implementation was defined; if `f` is defined in a superclass B of C, no checking is necessary, and the check can be eliminated when compiling `f` inside that method.

Note that, e.g., `postpone f` can put `f` in a context where the class of `this` is not guaranteed to be a subclass of B, so it's better to make this optimization dependent on the compilation context. The alternative is to ensure through language restrictions that `f` can only be used in the right context, but one might overlook some corner case, and the restrictions may be too limiting.

3.9 Arrays

Arrays are accessed through a reference on the object stack. They are either object arrays that contain only object references and `null`, or they are data arrays that contain only data, no object references. Safe Forth has typed data arrays, i.e., they contain only cells, only characters, or only floats.

Safe Forth uses polymorphic access words:

```
[]    ( u array -- v )
->[]  ( v u array -- )
```

The type of `v` depends on the type of the array, e.g., for an FP array `v` is a FP value on the FP stack, for an object array `v` is an object reference on the object stack.[7] If the index `u` is outside the array bounds, an exception is thrown.

[7] A disadvantage of this approach is that `[]` and `->[]` have a stack effect that depends on the passed array, which makes the code harder to analyse. Alternatively, we could have stack-effect-specific array access words such as `o[] ->o[] [] ->[] f[] ->f[]`.

```
50 farray oconstant a
3e 1 a ->[]
a 1 3 slice oconstant b
5e 1 b ->[]
```

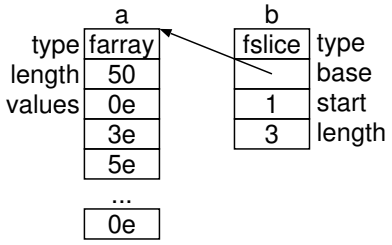

Figure 3: Source code for creating an array and an array slice, and the resulting memory

An alternative is to have object arrays and, for non-objects, untyped arrays of bytes with typed access words that use byte offsets rather than element indices. This approach would be closer to Forth and still memory-safe, but users of an object-oriented memory-safe Forth probably prefer the approach outlined for Safe Forth.

Figure 3 shows an example of an array. We discuss the representation of the type in Section 6.

The word `farray` creates a new FP array with the length (number of elements) given on the data stack; the elements are initialized to 0e.

3.10 Array slices

Forth supports representing parts of arrays by giving the start address and number of elements (or number of address units), the same representation as the full array. In Forth address arithmetic is used for that, but we cannot use that in Safe Forth, so we provide array slices instead. In the simplest case an array slice starts at some element of the array it is based on (with index 0 in the array slice), and has a length at most as long as the remaining elements of the base array.

Figure 3 shows an example of an array slice. This assumes (object-oriented) dynamic dispatch for `->[]` to work. Once we have polymorphism for array access words, we can have more fancyful kinds of array slices beyond what Forth can represent with address and length, e.g., with strided access.

3.11 Strings and String Buffers

Standard Forth uses *c-addr u* to represent both strings (e.g., in `type`) and string buffers (e.g., in `read-file`). In some cases (e.g., `move`), string buffers are represented by the address alone, and the word cannot prevent buffer overflows.

```
16 stringbuf oconstant s
s" abc" s move
s type
```

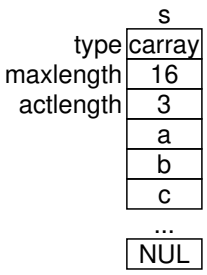

Figure 4: Source code for creating a string buffer, and for storing a literal string there, and the resulting memory

The use of two cells for this purpose leads to deep stacks and related complications, and so a significant number of programmers advocates counted strings or other alternative single-cell representations. So in Safe Forth we use a single reference on the object stack for representing a string or a string buffer.

A string buffer has a maximum length (used for words that write to the buffer, such as `read-file`), and an actual length (used for words that read from the string, such as `type`); see Fig. 4. Trying to store a too-long string into a string buffer results in an exception.

For read-only strings the maximum length is unnecessary. Such a specialized representation can easily be implemented in an object-oriented Safe Forth in addition to string buffers. Trying to write to a read-only string results in an exception.

3.12 Execution tokens

In Forth, `execute`, `compile,` and `defer!` take an xt from the data stack. In Safe Forth, we need to put execution tokens on the object stack; however, the execution tokens in Safe Forth will likely have a different representation than those in Forth, because it needs a type field like other objects referenced from the object header. One way to achieve this on top of an unchanged Forth system is to create objects that contain a type field and the Forth-level execution token. In Gforth a more efficient approach (one memory access less) would be to add a type field to the word header [PE19], but that requires deeper changes.

3.13 `does>` and `>body`

`Create` is eliminated. Instead, `does>` is combined with `oconstant`[8] and the code behind `does>` starts with the value of the oconstant pushed onto the object stack. Correspondingly, `>body` produces the value of the oconstant from its xt. E.g., the following is a possible implementation of `constant` in a non-object-oriented memory-safe Forth (Section 3.7):

```
begin-structure const-struct
  value: val
end-structure

: constant ( n "name" -- )
  const-struct new odup to val oconstant
does> ( -- n )
  ( const-struct ) val ;
```

3.14 Garbage collection

There are conservative garbage collectors that do not move the data around, and do not need to know if a cell contains a memory reference or some other data. The existing Forth garbage collector[9] is conservative.

Alternatively, precise garbage collection needs to know which cells contain memory references and which don't; the benefit is that it can move the data around, which eliminates fragmentation and makes allocation faster.

In the absence of tagging, it is difficult to keep track of all memory references in all situations: It's relatively straightforward to know it for object fields: objects are headed with a class address, and the class can contain the necessary information. But for locals, things are more difficult; either we have separate locals stacks for data and for object references, or we need some way to know which cells on the locals stack or return stack are references and which are data. These ways either require some runtime overhead or significant compiler complications.

To avoid such complications, we stick to conservative garbage collection.

4 Words

In order to see how similar and how different Safe Forth is from Forth, in this section we look at the Forth-2012 core words, and determine which are unchanged, which are changed, and how substantial the needed changes are.

[8] `Oconstant` is the object-stack equivalent of `constant`. An alternative view is that it is like `ovalue` except that you cannot use `to` on its children. Note that you can put mutable objects into oconstants

[9] http://www.complang.tuwien.ac.at/forth/garbage-collection.zip

In some cases one might choose the break in compatibility to perform other changes as well, e.g., to change the input stream handling, but in this paper we do not go into that and only work out changes that are related to the requirements of memory-safety.

4.1 Unchanged

These words typically work just on the data stack, but in some cases some stack items are taken off or pushed on the object stack, FP stack, or system-return-stack as discussed above: execution tokens are on the object stack, system compilation types (e.g., control-flow stack items like orig) are on the control-flow stack, system execution types (e.g., nest-sys, i.e., return addresses) are on the return stack, and floating-point values are on the FP stack.

```
 # #S ’ ( * */ */MOD + +LOOP - .  ." /
/MOD 0< 0= 1+ 1- 2* 2/ 2DROP 2DUP 2OVER
2SWAP : ; < <# = > >IN ?DUP ABORT ABORT"
ABS AND BEGIN BL CHAR CELL+ CELLS CHAR+
CHARS CONSTANT CR DECIMAL DEPTH DO DROP
DUP ELSE EMIT EXECUTE FM/MOD HOLD I IF
IMMEDIATE INVERT J KEY LEAVE LITERAL
LOOP LSHIFT M* MAX MIN MOD NEGATE OR OVER
POSTPONE QUIT RECURSE REPEAT ROT RSHIFT
S>D SIGN SM/REM SPACE SPACES SWAP THEN U.
U< UM* UM/MOD UNTIL WHILE XOR [ [’] [CHAR]
]
```

`CELL+ CELLS CHAR+ CHARS` are listed above, because they can be used to compute sizes or offsets, but they are not very useful in most Safe Forth variants (except those that use sizes and offsets for arrays instead of numbers of elements or indexes).

4.2 Adapted stack effects

```
#> ( xd -- string )
>BODY ( xt -- object )
>NUMBER ( ud1 string -- ud2 stringslice )
ACCEPT ( stringbuf -- )
COUNT ( string -- c stringslice )
ENVIRONMENT? ( string -- false | ... true )
EVALUATE ( ... string -- ... )
FILL ( stringbuf c -- )
FIND ( string -- xt +-1 | string 0 )
MOVE ( string stringbuf -- )
S" ( run-time: -- string )
SOURCE ( -- string )
TYPE ( string -- )
WORD ( c -- string )
```

Among these, `count` is useful only in its meaning as alias for `c@+`.

4.3 Slight changes

`DOES>` now works on oconstants.

`EXIT` now works in counted loops without preceding `unloop`.

4.4 Substantial changes

`@` is deleted. Values and value-flavoured words push their value, and `[]` loads a value from an array.

`!` is deleted. `to` is used for storing into values and fields, `is` for storing into deferred words, and `->[]` for storing into arrays.

`+!` is deleted. The system may have `+to`.

`Allot` is deleted. Words like `new` or `array` are used instead.

`State` is replaced with the value `state@`.

`Base` is replaced with the value `base@`.

4.5 Deleted words

```
 !  +!  , 2!  2@ >R @ ALIGN ALIGNED ALLOT
BASE C! C, C@ CREATE HERE R> R@ STATE
UNLOOP VARIABLE
```

4.6 New words

```
base@ ( -- u )
null= ( o -- )
o= ( o1 i2 -- )
odup? ( o -- null | o o )
oconstant ( o "name" -- )
odrop ( o -- )
odup ( o -- o o )
oliteral ( o -- ) ( run-time: -- o )
oover ( o1 o2 -- o1 o2 o1 )
orot ( o1 o2 o3 -- o2 o3 o1 )
state@ ( -- f )
oswap ( o1 o2 -- o2 o1 )
[] ( u array -- v )
->[] ( v u array -- )
```

This set of words only allows individual values and array data. You typically also want to add some words for defining classes, fields and methods (not listed here).

4.7 Statistics

Relative to a base of 133 Core words in Forth-2012, 96 (72%) are unchanged, 14 (11%) have adapted stack effects and 2 (2%) have other small changes. 21 (16%) are deleted, and 14 (11%) are new.

5 Advanced topics

5.1 Looping over arrays

Arrays are often accessed in loops, e.g., iterating from the first to the last element. The bounds

checks (and type checks) for these accesses can often be optimized away, and many memory-safe languages employ compiler optimization based on some form of data-flow analysis [BGS00] to achieve that.

Of course we want to use a simpler way in Forth. One way is to have array-walking words that walk the whole array (or array slice) and don't need to perform bounds checking. This could look as follows:

```
: type ( string -- )
  arraydo emit arrayloop ;
```

`Arraydo` would take an array and push one element of the array (from first to last) in every iteration; the loop would be closed with `arrayloop`.

In addition one might want variants that write an element in every iteration, or that push an element from one array and store a result into another array, etc. You may also want to work with a strided array slice, possibly with negative stride (for walking the array backwards). One problem with flexible `arraydo`s that can take arrays of various types, and array slices with all kinds of strides is that the way the array is handled is only known at run-time, so one typically has to call a run-time-dispatched handler. One could avoid that by having a variant of `arraydo` for every kind of array (slice) it can be applied to, but that leads to an explosion of words and would result in less flexible words that use these words.

An alternative to a `do`...`loop`-like control structure is to define a word like Postscript's `forall` or Oforth's `apply` that takes an array and an xt, and for each element of the array, pushes the element and calls the xt. One disadvantage of this approach is that either the code in the xt cannot access locals of the definition containing the `forall`, it requires explicit passing of the locals [EP18], or a complex implementation; by contrast, `arraydo`...`arrayloop` supports access to locals in the loop body.

5.2 Escape hatch

Memory-safe languages often have an escape hatch into code that is not guaranteed to be safe: E.g., Modula-2 has the module `SYSTEM` that allows access to low-level facilities, Java has the Java native interface (JNI) that allows calling C functions, and Rust has unsafe blocks.

The purpose of such an escape hatch is twofold:

- It allows doing things that are impossible in the memory-safe language, such as low-level access to I/O devices.
- It allows doing things more efficiently that are inefficient in the memory-safe language. E.g., `arraydo`...`arrayloop` could be implemented in Safe Forth by accessing each element individually and incurring a bounds check for each element, but it is more efficient to write these words in Forth without bounds checks (all accesses are within the array).

The typical approach is not to use the escape hatch at each place in application code that needs to deal with, e.g., I/O devices, but to define a memory-safe interface to these I/O devices, and use the escape hatch to implement this interface. This keeps the unsafe code to a minumum.

Two people reading a draft version of this paper suggested to include a way to specify absolute addresses for memory-mapped I/O in Safe Forth. We know of no way to achieve this without opening a hole that may subvert the memory-safety guarantees of Safe Forth. But using the escape hatch to achieve this is appropriate and seems simple enough: E.g., one could define words for the individual registers of the I/O device, or alternatively define the I/O device as a slice in an array of bytes (specifying the address through the escape hatch), and use offsets in that array to access various registers. In either case, the programmer of code beyond the escape hatch would guarantee the memory-safety of the words added to Safe Forth, or of the data structures modified through the escape hatch.

An advantage of having an escape hatch is that Safe Forth does not need to include words for every contingency; instead, you can define them as needed.

For Safe Forth implemented on top of Forth, the obvious escape hatch language is Forth, and the escape hatch is relatively simple to implement: Safe Forth provides its words in a wordlist different from the wordlists providing unsafe words, and in Safe Forth only this wordlist and wordlists defined in memory-safe code are on the wordlist. The escape-hatch word `forth` makes the Forth wordlist available again.

There may also be cases where you want to weld the escape hatch shut, in particular when you want to process untrusted code (e.g., coming from the Internet). It is straightforward to have a mechanism for disabling the escape hatch.

5.3 Multi-threading

Some of the challenges of multi-threaded multi-tasking are: Single-threaded stop-the-world garbage collection is relatively simple and is implemented in an existing garbage collector for Forth; extending it for a single-threaded cooperative multi-tasker is relatively easy, but has the unpleasant effect of stopping all tasks while it collects. Things become really complex when

multi-threading is involved, because other threads may be in states that are not safe for garbage collection. There are solutions to these problems, but they are quite complex.

An alternative is to set the system up such that only per-thread/task garbage collection is necessary. One way to do that is that every thread/task has its own memory and collects its own memory. The tasks communicate through channels, mailboxes, or the like, but cannot pass object references across these communication channels. Instead, if you want to pass a data structure to another task, it has to be marshalled (serialized) on the sender and unmarshalled on the receiver.

The disadvantage of this approach is that the marshalling and unmarshalling constitutes an overhead; the advantage is that programs organized in this way can put the tasks into different processes, and actually even different computers.

Alternative approaches are to avoid garbage collection, e.g., with a reference-counting scheme, or to implement full-blown multi-threaded garbage collection.

6 Implementation efficiency

This section does not describe the implementation in detail, but addresses performance concerns you may have. The implementation approaches suggested in this section are designed to go with an object-oriented system where every selector can be used with any class (duck typing), with single inheritance (especially of fields), where all methods of a class are defined before the first object of the class is created.

In this section, we describe the implementation of some Safe Forth features as Forth code, but it could just as well be some other lower-level language, e.g., assembly language.

The performance claims are not supported empirically in this paper, possibly in a future paper.

6.1 Type equality

Field accesses to structures that do not support extensions only have to check the structure type for equality. So an access to field `intlist-val` in Fig. 2 could be implemented in Forth as follows:

```
: intlist-val ( o -- n )
  o> dup @ intlist = if
    2 cells + @ exit then
  type-error throw ;
```

On processors with out-of-order cores (such as the performance cores on desktop, server, and current smartphone CPUs) the `if` will usually be predicted correctly, which means that the code needed to compute the condition (in **bold red**) is not on the critical path; it costs resources (of which most such CPUs have plenty), but not latency. Only the *slanted blue* part is on the critical path for computing the result. So, on this class of CPUs this checked version will be about as fast as the unchecked version, at least in latency-limited code (the usual case).

```
selector sum
selector foreach

object class
  ovalue: next
  m: foreach ... ;m
end-class list

list class
  value: val
  m: sum ... ;m
end-class intlist
```

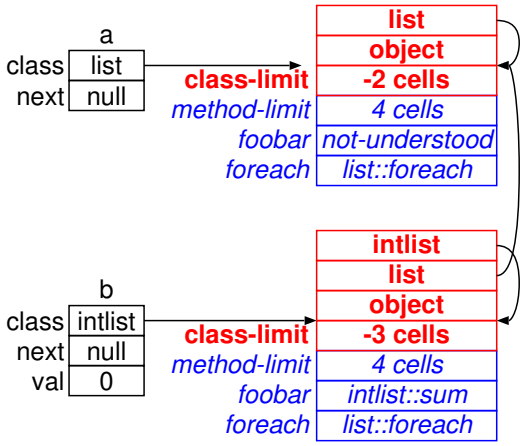

Figure 5: Simple example of some classes and objects. The **bold red** parts are used for subtype testing, the *slanted blue* parts for method dispatch.

6.2 Subtype test

If the Safe Forth supports structure extension, or equivalently inheritance of fields in classes, we need to check if the type of the structure/object is a subtype of the type the field was defined for.

We use Cohen's approach to subtype testing, [Coh91], see Fig. 5: Each class has a subtype table (shown in **bold red**) containing its own address, and the addresses of all its ancestor classes. The primal ancestor (`object`) has offset −1 cells, the next generation (`list` in the example) has offset −2 cells, etc. For each class the offset at which it is to be found is known. E.g., `list` and all of its descendents will find the address of `list` at offset −2 cells in their subtype table; nondescendents will either have a higher (closer-to-zero) `class-limit`, or will have a different class at offset −2 cells. So the code for an access to field `next` with a subtype

check looks as follows:

```
: next ( o1 -- o2 )
  o> dup @ dup @ -2 cells <= if
    -2 cells + @ list = if
      cell+ @ >o exit then then
  type-error throw ;
```

Concerning performance, the latency in the usual correctly-predicted case is again the same as for the unchecked case, but the check needs more resources than a type equality check.

Note that when accessing a field of `this` (the object used for method dispatch), an ancestor class of `this` is known, and if the field belongs to an ancestor of that class (the usual case), no check at run-time is needed.

6.3 Method dispatch

We use a per-class method table with bounds checking (a bounds-checked version of unhashed general selectors [Ert12, Section 3.1]). The table is shown in the blue part of Fig. 5: `method-limit` contains the offset from the start of the class descriptor (where the class addresses point to) to just beyond the last method implemented for the class. The code for a method dispatch looks as follows:

```
: sum ( o -- n )
  o> dup @ dup cell+ @ 3 cells u>= if
    2 cells + @ execute exit then
  not-understood ;
```

Again, the bounds check (in red) does not cost latency on an out-of-order CPU, and the whole dispatch is usually as fast as the unchecked version.

6.4 Array access

An array access as a colon definition has to check the type, and the bounds. If the array access is implemented as a method, the method dispatch has to be performed, but there is no need to check the type. The bounds check has to be implemented in any case. Here's how `f[]` implemented as method for `farray` (Fig. 3) might look:

```
m: f[] ( n farray -- r )
  ( farray is in THIS )
  dup length u< if
    floats values + f@ exit then
  bound-error throw ;m
```

The bounds check is in red, and if the `if` is correctly predicted (the usual case), it does not contribute to the latency. A correctly predicted method dispatch also does not contribute to the latency.

7 Status and further work

For now Safe Forth is a paper design. Implementing it and evaluating this implementation is on my agenda, but currently not at the top, and it depends on the interest of the community if it ever reaches the top.

One could implement it as a layer on top of Forth relatively easily, but that would mean that the object-stack access is slow. For decent performance, the object stack needs to be implemented with its stack pointer in a register and maybe an OTOS (top-of-object-stack) register, and a `this` register, which requires changing an existing system at a pretty basic level and is somewhat more involved.

8 Related work

For the main contribution of the present work, the use of a separate stack and value-flavoured words to reduce the need for static or dynamic type checking, there is surprisingly little related work. A number of Forth systems have a separate FP stack (standardized in Forth-2012), but the reason for that is not type-safety, and the separation ends as soon as the stack contents are stored to memory.

Similarly, a string stack has often been proposed for dealing with strings [MM81], but again, type safety has not been a primary goal. A vector stack has been used for dealing with vectors [Ert17]; there the main point is to provide vectors as an opaque data type. Still, the object stack can be seen as a generalization of the string and vector stacks.

There are several memory-safe languages based on Forth, e.g., Oforth[10] and Factor [PEG10]. However, Oforth uses type tagging and Factor uses static type checking in combination with run-time type checking, while Safe Forth uses the division between object references and integer/FP data to get rid of a large part of the type checking.

Apart from that, many design decisions in Oforth[11] are similar to those in Safe Forth: Oforth initializes all locations to zero. You can only read from and write to fields (with `@field !field`), not take their address; field accesses are allowed only to fields of `self`'s class, and no class check is necessary. Oforth keeps only system-defined stuff on the return stack, and it uses a separated control-flow stack. Oforth has no variables, only (task-local) values. `Execute` is a method selector for objects. `Does>` is eliminated, because it can be replaced with object-oriented dispatch. Unlike Safe Forth, Oforth implements control flow similar to Factor or Postscript: by passing closures (quotations with lexical scoping)

[10] http://www.oforth.com/

[11] Personal communication with Franck Bensusan

to control-flow words that `execute` the control-flow word multiple times).

There is a large body of work on subtype testing and method dispatch; Ducournau [Duc11] gives an excellent, although somewhat abstract overview. We choose Cohen's subtype testing approach [Coh91] because we comply with its single-inheritance limitation and because it is simple to implement incrementally. For method dispatch in objects2 [Ert12] we proposed using a number of these techniques (to allow different space/time tradeoffs), here we select one of those and enhance it with bounds checking similar to that used in Oforth [Ben18]; however, Oforth has a table of classes per selector, while the unhashed general selectors in objects2 and in the present work have a table of selectors per class.

9 Conclusion

Memory-safe languages eliminate a significant class of bugs and vulnerabilities. Forth can be turned into a memory-safe language by eliminating all operations involving addresses, e.g., `@` and `!`. In order to be still useful as a general-purpose language, we replace addresses with object references, but they have to be distinguished from other data. Other languages and virtual machines use tagging and/or static type checking to distinguish them; we instead put object references on an object stack, and have `ovalue` and other defining words that keep object references separated from other data.

Starting from this premise, and otherwise keeping relatively close to standard Forth results in surprisingly few changes to the core vocabulary.

Acknowledgments

Franck Bensusan and the reviewers provided helpful comments that helped improve the paper.

References

[Ben18] M. Franck Bensusan. Method dispatch in Oforth. In *34th EuroForth Conference*, pages 31–36, 2018. 8

[BGS00] Rastislav Bodik, Rajiv Gupta, and Vivek Sarkar. ABCD: eliminating array bounds checks on demand. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, pages 321–333, 2000. 2, 5.1

[Coh91] Norman H. Cohen. Type-extension type tests can be performed in constant time. *ACM Transactions on Programming Languages and Systems*, 13(4):626–629, October 1991. Technical Correspondence. 6.2, 8

[Duc11] Roland Ducournau. Implementing statically typed object-oriented programming languages. *ACM Computing Surveys*, 43(3):Article 18, April 2011. 8

[EP18] M. Anton Ertl and Bernd Paysan. Closures — the Forth way. In *34th EuroForth Conference*, pages 17–30, 2018. 5.1

[Ert12] M. Anton Ertl. Methods in objects2: Duck typing and performance. In *28th EuroForth Conference*, pages 96–103, 2012. 6.3, 8

[Ert17] M. Anton Ertl. SIMD and vectors. In *33rd EuroForth Conference*, pages 25–36, 2017. 8

[Gay20] Alex Gaynor. What science can tell us about C and C++'s security. Blog posting, https://alexgaynor.net/2020/may/27/science-on-memory-unsafety-and-security/, 2020. 1

[MM81] Michael McCourt and Richard A. Marisa. The string stack. *Forth Dimensions*, III(4):121–124, 1981. 8

[PE19] Bernd Paysan and M. Anton Ertl. The new Gforth header. In *35th EuroForth Conference*, pages 5–20, 2019. 3.12

[PEG10] Sviatoslav Pestov, Daniel Ehrenberg, and Joe Groff. Factor: a dynamic stack-based programming language. In William D. Clinger, editor, *Proceedings of the 6th Symposium on Dynamic Languages, DLS 2010, October 18, 2010, Reno, Nevada, USA*, pages 43–58. ACM, 2010. 8

[RCM96] Elizabeth D. Rather, Donald R. Colburn, and Charles H. Moore. The evolution of Forth. In *History of Programming Languages*, pages 625–658. ACM Press/Addison-Wesley, 1996. 3