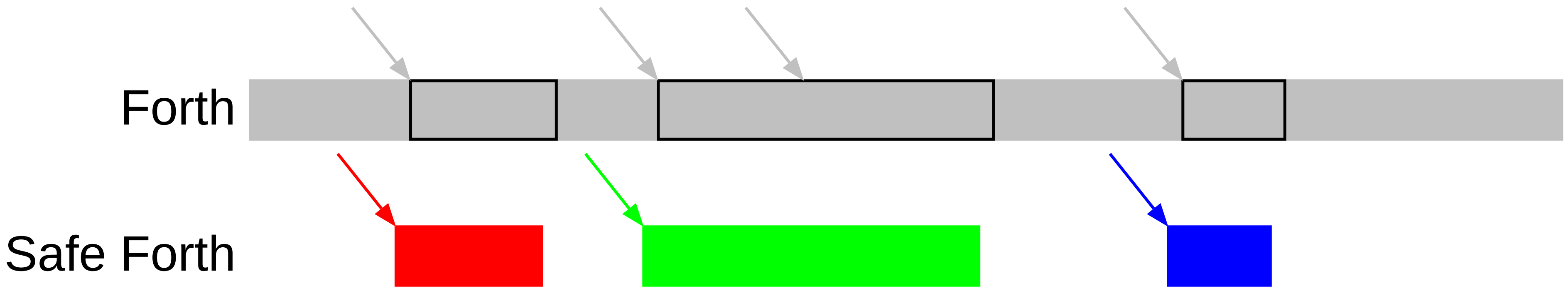
Memory Safety
Without Tagging nor Static Type Checking

M. Anton Ertl, TU Wien

Memory Safety



- Out-of-bounds memory accesses
- Accesses to the wrong structure
- Uninitialized memory
- Use after `free`

Memory Safety in Programming Languages

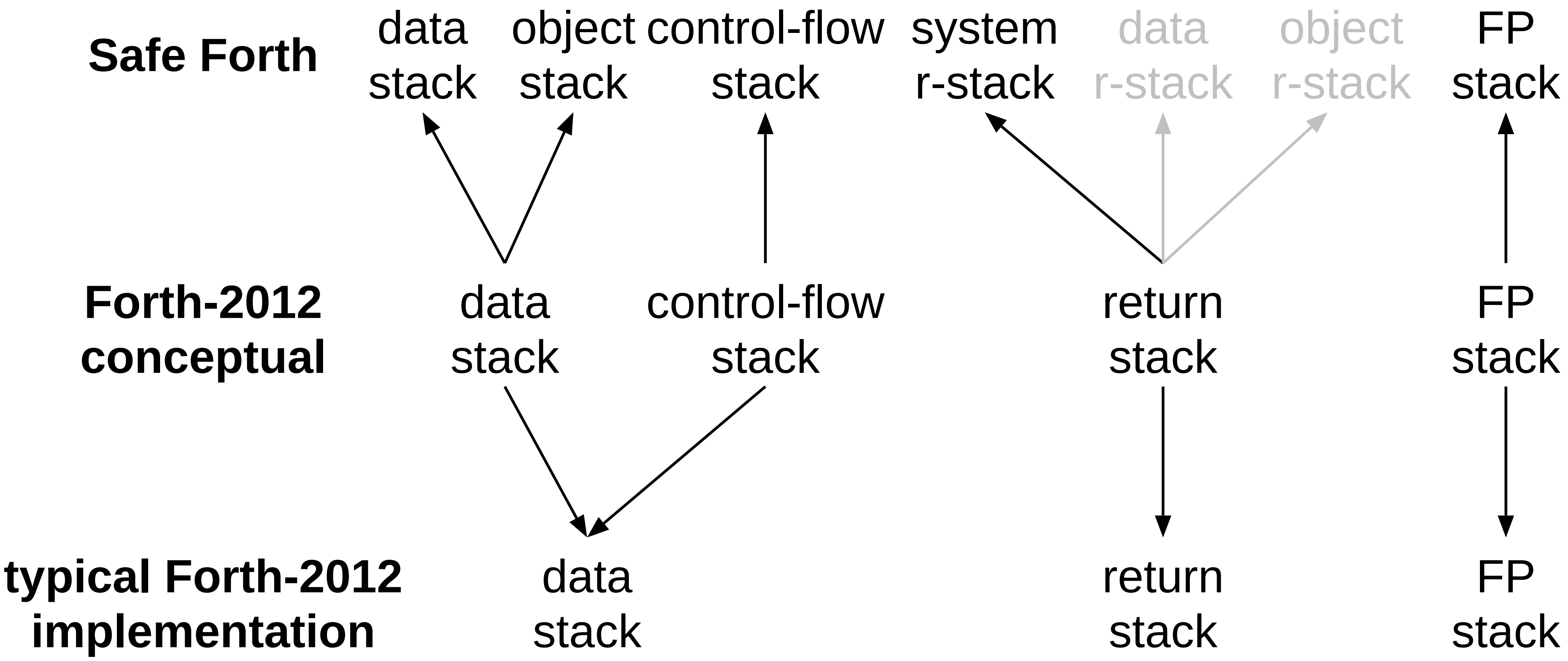
- Not memory-safe: Forth, Assembler, C, C++
- Memory-safe: most languages (e.g., Factor, Oforth, Java)
- Distinguish between references and data
  static type checking (Factor, Java)
  tagging (Oforth, Lisp)
- Out-of-bounds memory accesses: bounds checking
- Accesses to the wrong structure: (dynamic) type checking
- Uninitialized memory: zero everything
- Use after `free`: garbage collection etc.

Safe Forth

- A memory-safe Forth-family language
- no static type checking (unlike Factor)
- no tagging (unlike Oforth)
- no addresses on data stack
  no `@ !` etc.
  no address arithmetic
- object references on object stack
- values, value-flavoured fields
- array accesses with

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

Safe Forth: Stacks



Safe Forth: data stack, object stack, control-flow stack, system r-stack, data r-stack, object r-stack, FP stack

Forth-2012 conceptual: data stack, control-flow stack, return stack, FP stack

typical Forth-2012 implementation: data stack, return stack, FP stack

Catch stack underflows and overflows

Example Program

```
begin-structure intlist
    field: next
    field: val
end-structure

: insert {: n listp -- :}
    intlist allocate throw
    listp @ over next !
    n over val !
    listp ! ;

variable mylist 0 mylist !
1 mylist insert
2 mylist insert
```

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

: insert {: n o: list1 -- list2 :}
    intlist new
    list1 oover to next
    n odup to val ;

null
1 insert
2 insert
ovalue mylist
```

Example Program (cont.)

```
: .list ( list -- )
    begin ( list1 )
        dup while
            dup val @ .
            next @ repeat
    drop ;

mylist @ .list \ prints 2 1
```

```
: .list ( list -- )
    begin ( list1 )
        odup null<> while
            odup val .
            next repeat
    odrop ;

mylist .list \ prints 2 1
```

Statistics

- Out of 133 core words in Forth-2012
- 96 (72%) unchanged
- 14 (11%) adapted stack effects (e.g., `#> ( xd - string )`)
- 2 (2%) other small changes
- 21 (16%) deleted (e.g., `! >r`)
- 14 (11%) new (e.g., `null= oconstant`)
- Some non-core words required (e.g. `value to`) plus object-stack equivalents (e.g., `ovalue`)

Escape Hatch

- Sometimes we want to do things beyond Safe Forth (e.g., hardware I/O)
- Sometimes we want to eliminate the Safe Forth overhead/opportunity cost
- escape to Forth
  programmer responsible for memory-safety
  requirements beyond Forth memory-safety
- Weld escape hatch shut for processing untrusted code

Multi-threading

- Multi-threading and garbage collection: complex
  especially with decent performance
- Alternative:
  per-thread garbage collector
  no passing of object references between threads
  marshal and unmarshal objects for inter-task communication
  can also be used between computers

Implementation efficiency

- No implementation yet
- Direct overhead may be less than many expect
  Missed opportunities may be a bigger problem

Conclusion

- Memory Safety: references limited to within objects
- Safe Forth
  no addresses
  separate data and object stack
  separate data and object values, value-flavoured fields, etc.

Status

- Paper design
- May become reality if there is enough interest