func.new

Ben L. Titzer, Oct 2025 CG Meeting

2025-10-29

Proposal in a nutshell

[intro]

Proposal in a nutshell

[verse]
In the old days
We could just flip a bit
T.L.B. shootdown (OS kernel)
And new instructions were ready
All the baddies came (cash money)
And took over
And that was bad (privilege escalation)

Proposal in a nutshell

[chorus]
func.new (ooh-ooh)
is all we need
func.new (ooh-ooh)
is all we need
new code
with just the caps we want (no escalation)

Proposal in a nutshell

[verse]
So hey we did Wah-zum
Now there's no R...C...Es
Code and data izza sep-ah-rut
But sometimes
Interpreters go slow (oh oh oh)
A whole new module is a hassle

Proposal in a nutshell

[chorus]
func.new (ooh-ooh)
is all we need
func.new (ooh-ooh)
is all we need
new code
with just the caps we want (no escalation)

Proposal in a nutshell

[verse]
Inside a module, it's all our show
But the tools rearrange us
So better let them know
What the new code uses

Proposal in a nutshell

[chorus]
func.new (ooh-ooh)
is all we need
func.new (ooh-ooh)
is all we need
new code
with just the caps we want (no escalation)

Proposal in a nutshell

[interlude]

Proposal in a nutshell

[chorus]
func.new (ooh-ooh)
is all we need
func.new (ooh-ooh)
is all we need
new code
with just the caps we want (no escalation)

Proposal in a nutshell

[verse]
So we propose
a new mechanism
for new code
but no funny business

Background

- Wasm has separate code and data, aka Harvard architecture
  - Cannot change code at runtime (or even read it)
  - Important security properties:
    - Control-flow integrity (partly due to virtualized execution stack)
    - No remote code execution (RCE)
  - Important analysis properties:
    - Closed-world assumption: can analyze all of a module’s access to internals
    - wasm-opt: remove dead code and data from a module

Background

- Wasm has separate code and data, aka Harvard architecture
  - Cannot change code at runtime (or even read it)
  - Important security properties:
    - Control-flow integrity (partly due to virtualized execution stack)
    - No remote code execution (RCE)
  - Important analysis properties:
    - Closed-world assumption: can analyze all of a module's access to internals
    - wasm-opt: remove dead code and data from a module
- New code? That's a host capability!
  - Wasm MVP launched with JavaScript APIs to make new modules
  - `new WebAssembly.Module(bytes)`
  - `WebAssembly.compile(bytes)`
  - `WebAssembly.compileStreaming(bytes)`

Motivation for the proposal

- Guest virtual machines: a VM in a VM
  - Python, JS on Wasm (yes that’s a thing!), Lua, C#, Java, CPU emulators
  - Interpreter performance is bad: **10-100X** slower than JITing
    - Interpreters on Wasm also typically **2-4X** slower than on native
  - Is partial evaluation of an interpreter (Futamura) a solution?

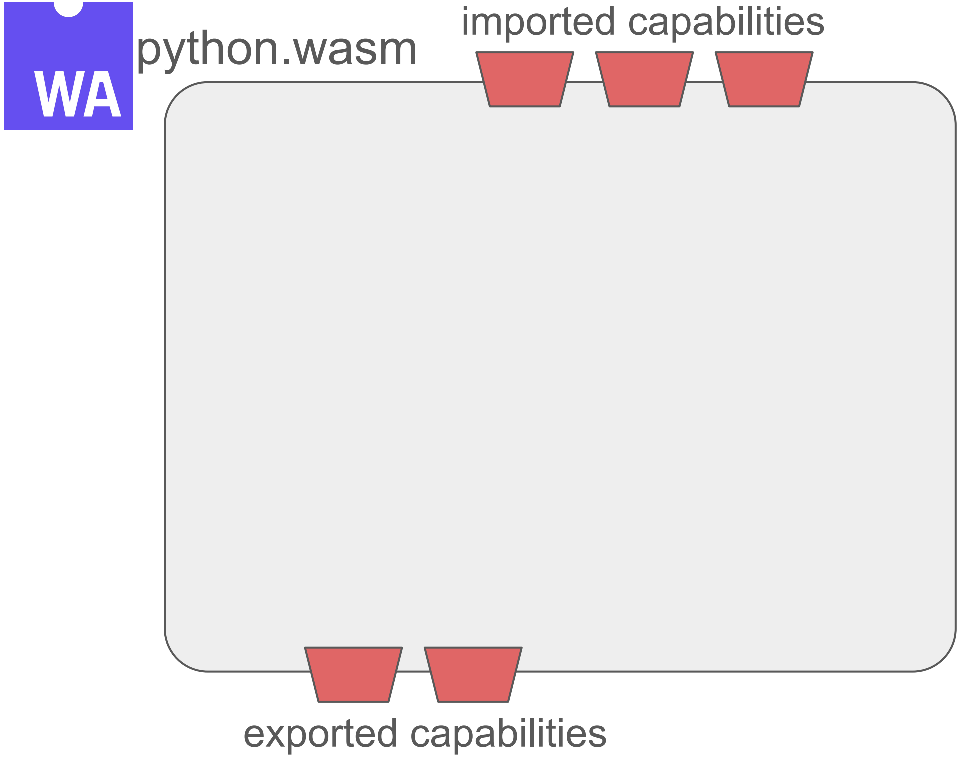
Motivation for the proposal

- Guest virtual machines: a VM in a VM
  - Python, JS on Wasm (yes that’s a thing!), Lua, C#, Java, CPU emulators
  - Interpreter performance is bad: **10-100X** slower than JITing
    - Interpreters on Wasm also typically **2-4X** slower than on native
  - Is partial evaluation of an interpreter (Futamura) a solution?
- Existing solutions on the Web
  - JS APIs require entire module at a time
  - Issues with synchronous compile limits

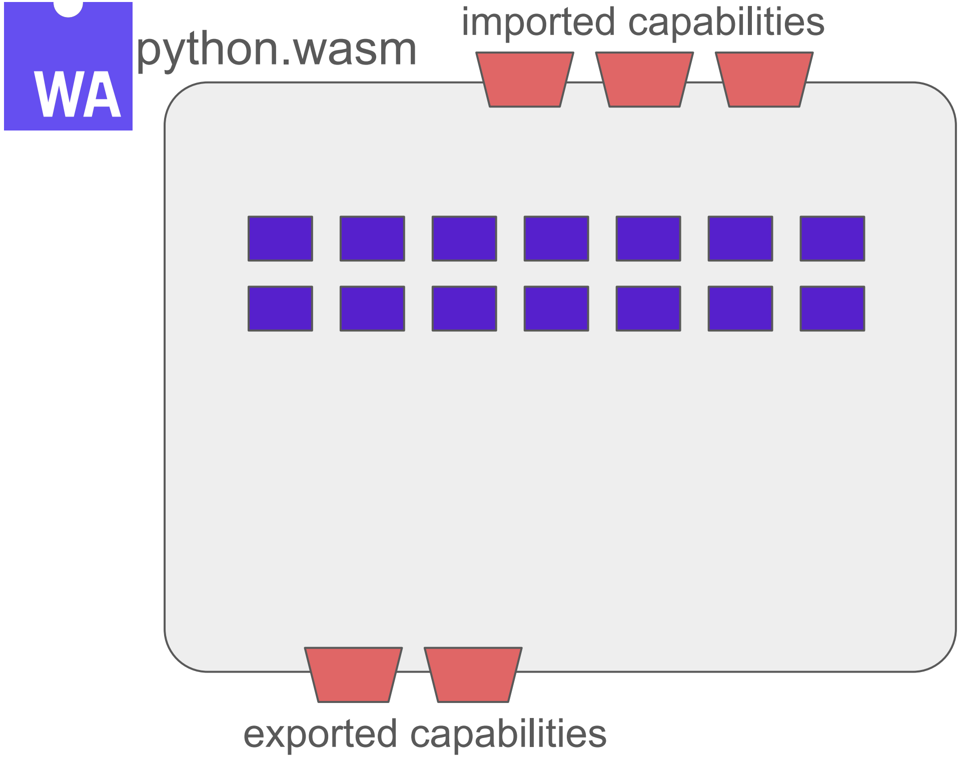
Motivation for the proposal

- Guest virtual machines: a VM in a VM
  - Python, JS on Wasm (yes that's a thing!), Lua, C#, Java, CPU emulators
  - Interpreter performance is bad: **10-100X** slower than JITing
    - Interpreters on Wasm also typically **2-4X** slower than on native
  - Is partial evaluation of an interpreter (Futamura) a solution?
- Existing solutions on the Web
  - JS APIs require entire module at a time
  - Issues with synchronous compile limits
- Existing solutions elsewhere
  - Wasm engine embedding API (`wasm-c-api`)
  - Engine-specific embedding APIs, e.g. WAMR, wasm3, wasmtime
  - Super-powered host function: `wizeng.new_funcref(bytes)`
  - => No portable solution yet

Motivating Example - Guest language runtime

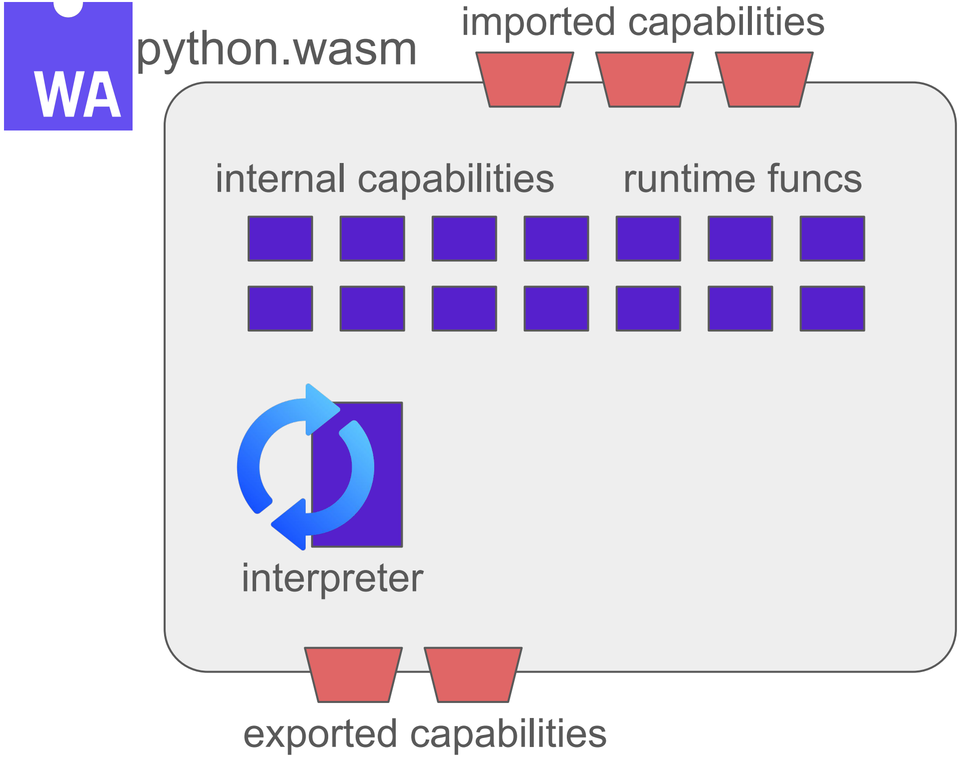


Motivating Example - Guest language runtime



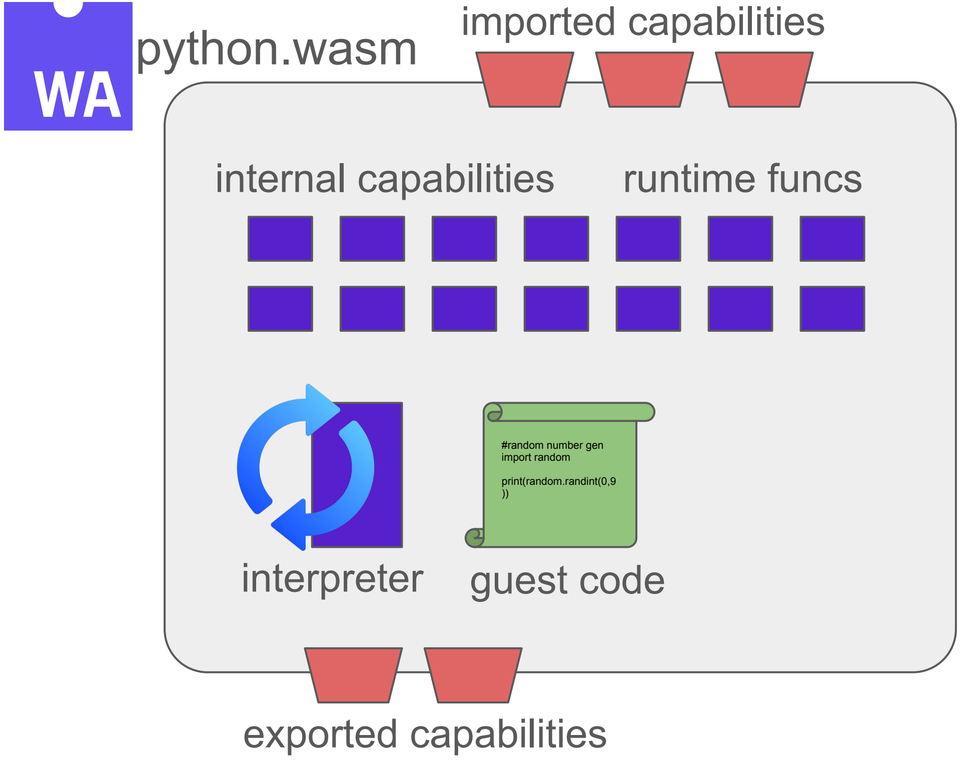
- What’s inside?
  - Just Wasm code!

Motivating Example - Guest language runtime



- What's inside?
  - Just Wasm code!
  - A division between internal runtime and an interpreter

Motivating Example - Guest language runtime

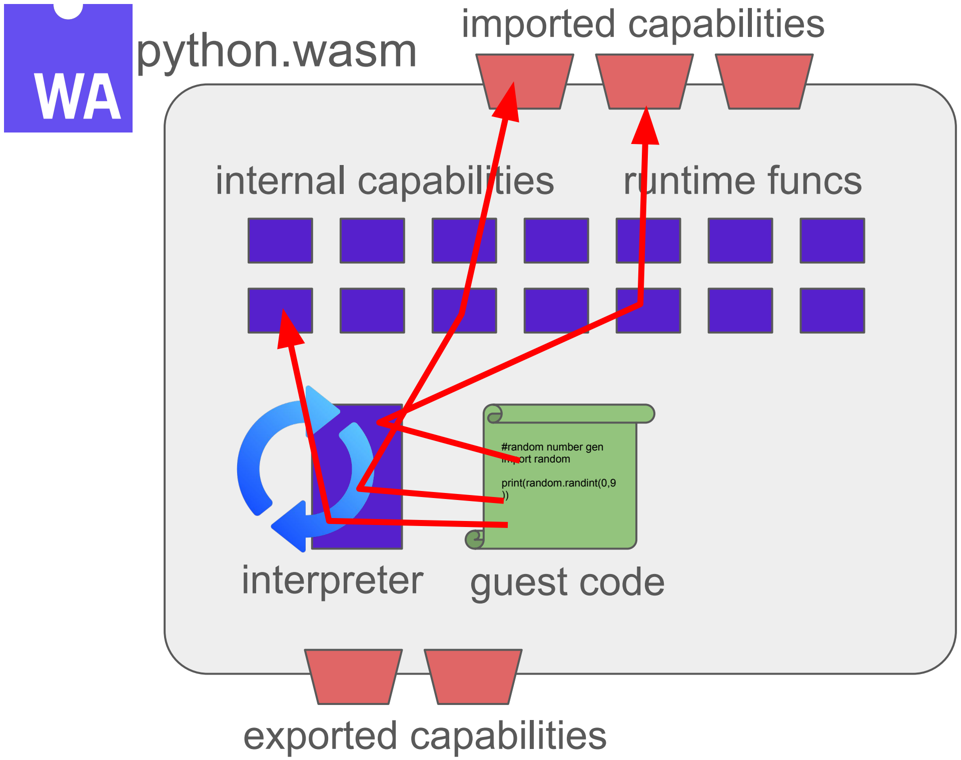


```
#random number gen
import random

print(random.randint(0,9
))
```

- What’s inside?
  - Just Wasm code!
  - A division between internal runtime and an interpreter
  - That happens to be Turing-complete

Motivating Example - Guest language runtime

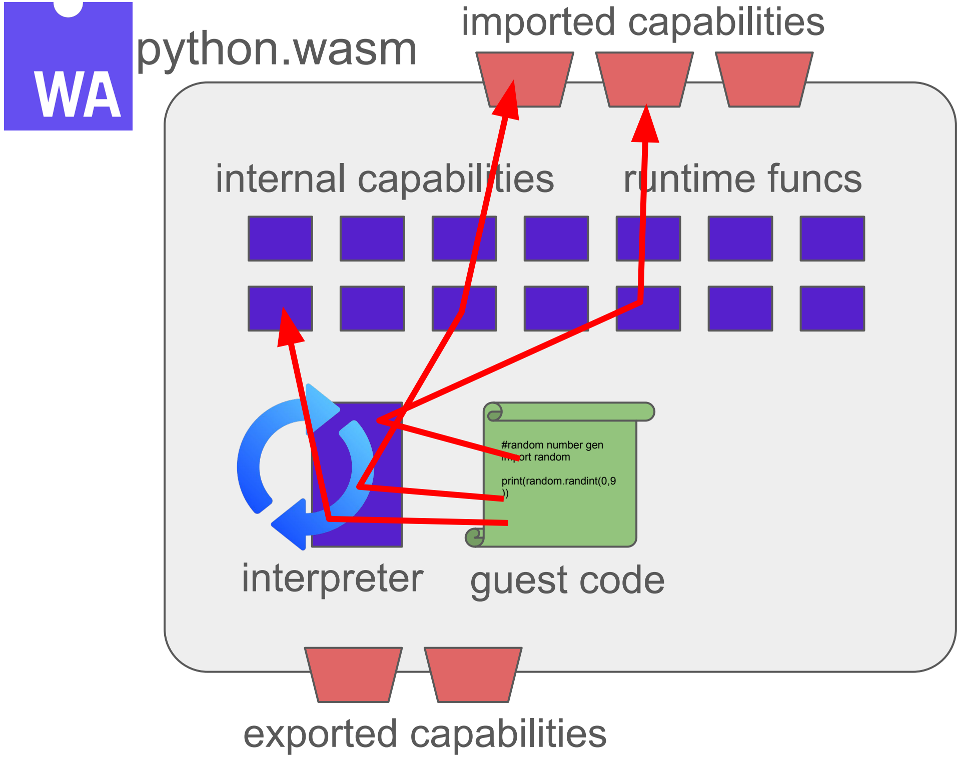


```
#random number gen
import random

print(random.randint(0,9
))
```

- What’s inside?
  - Just Wasm code!
  - A division between internal runtime and an interpreter
  - That happens to be Turing-complete
  - Through the interpreter loop, guest can “drive” the runtime’s internal and imported capabilities.

Motivating Example - Guest language runtime

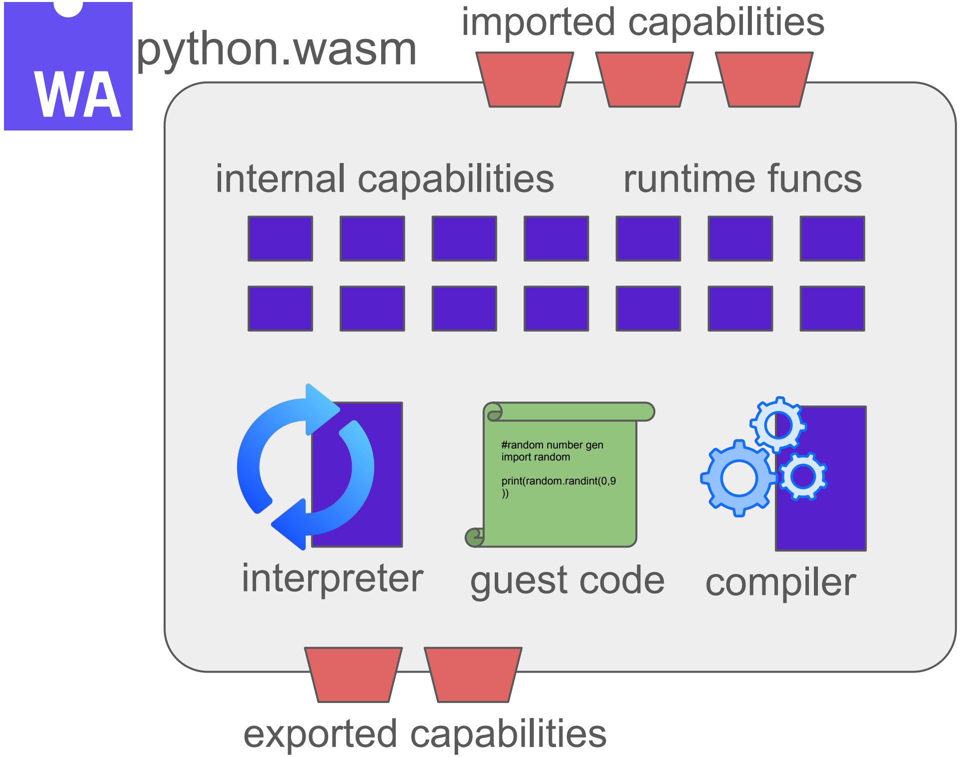


```
#random number gen
import random

print(random.randint(0,9
))
```

- What’s inside?
  - Just Wasm code!
  - A division between internal runtime and an interpreter
  - That happens to be Turing-complete
  - Through the interpreter loop, guest can “drive” the runtime’s internal and imported capabilities.
  - But interpreted code cannot break the encapsulation of its module.

Motivating Example - Guest language runtime

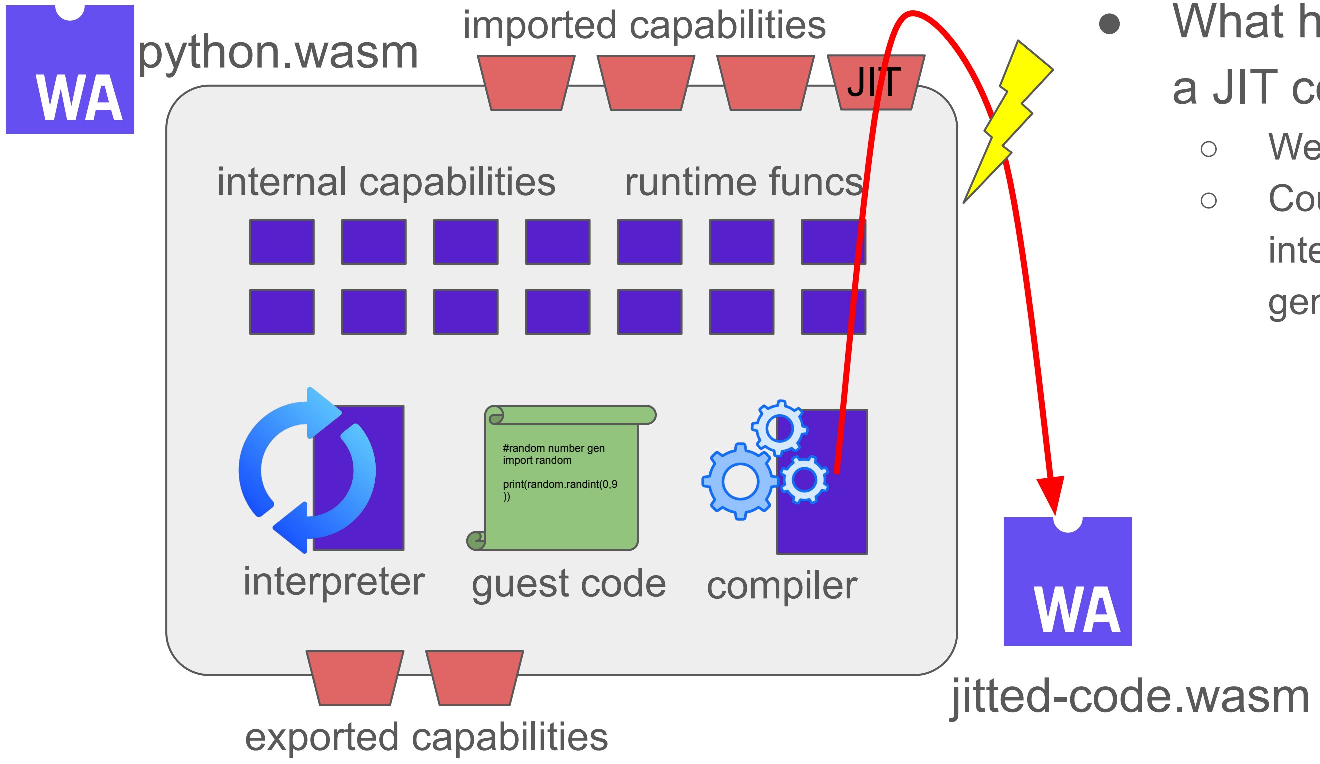


```
#random number gen
import random

print(random.randint(0,9
))
```

- What happens when we add a JIT compiler?

Motivating Example - Guest language runtime

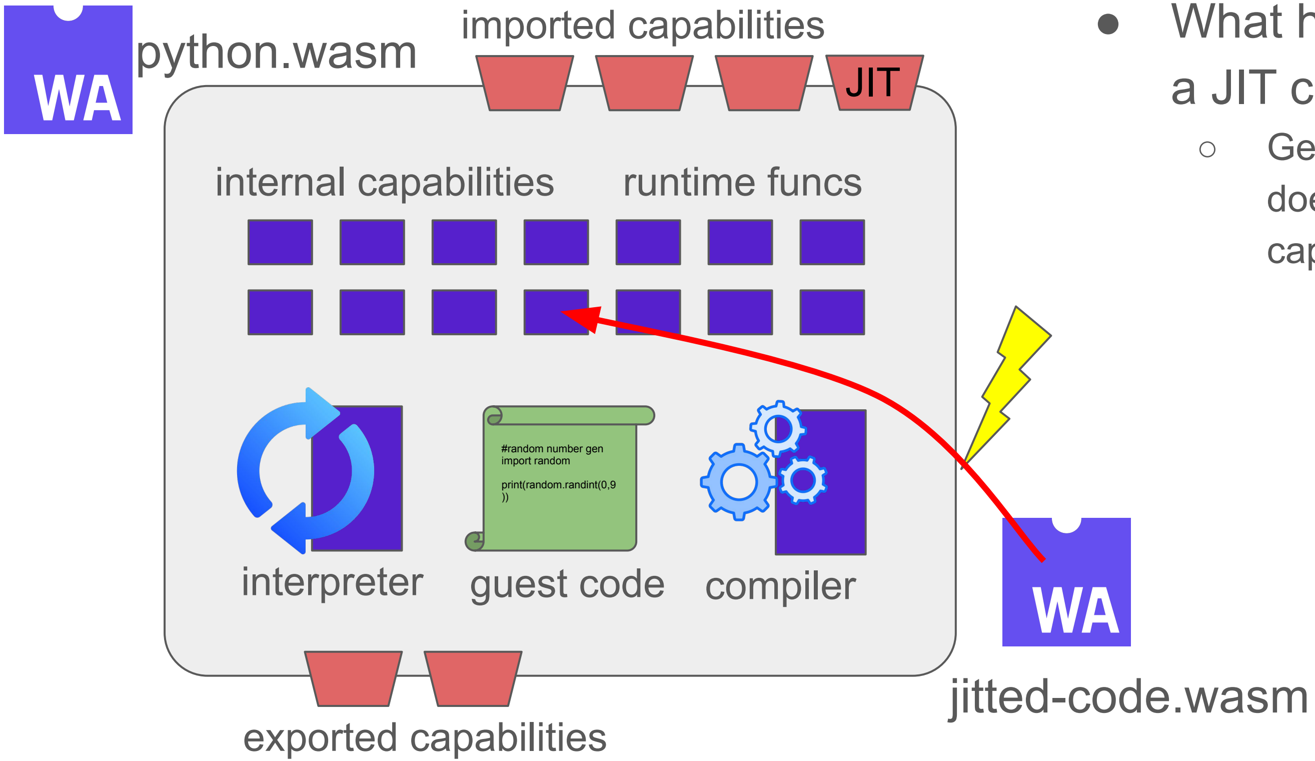


```
#random number gen
import random

print(random.randint(0,9
))
```

- What happens when we add a JIT compiler?
  - We need a new host capability
  - Could have accidental interposition or corruption on generated code

Motivating Example - Guest language runtime

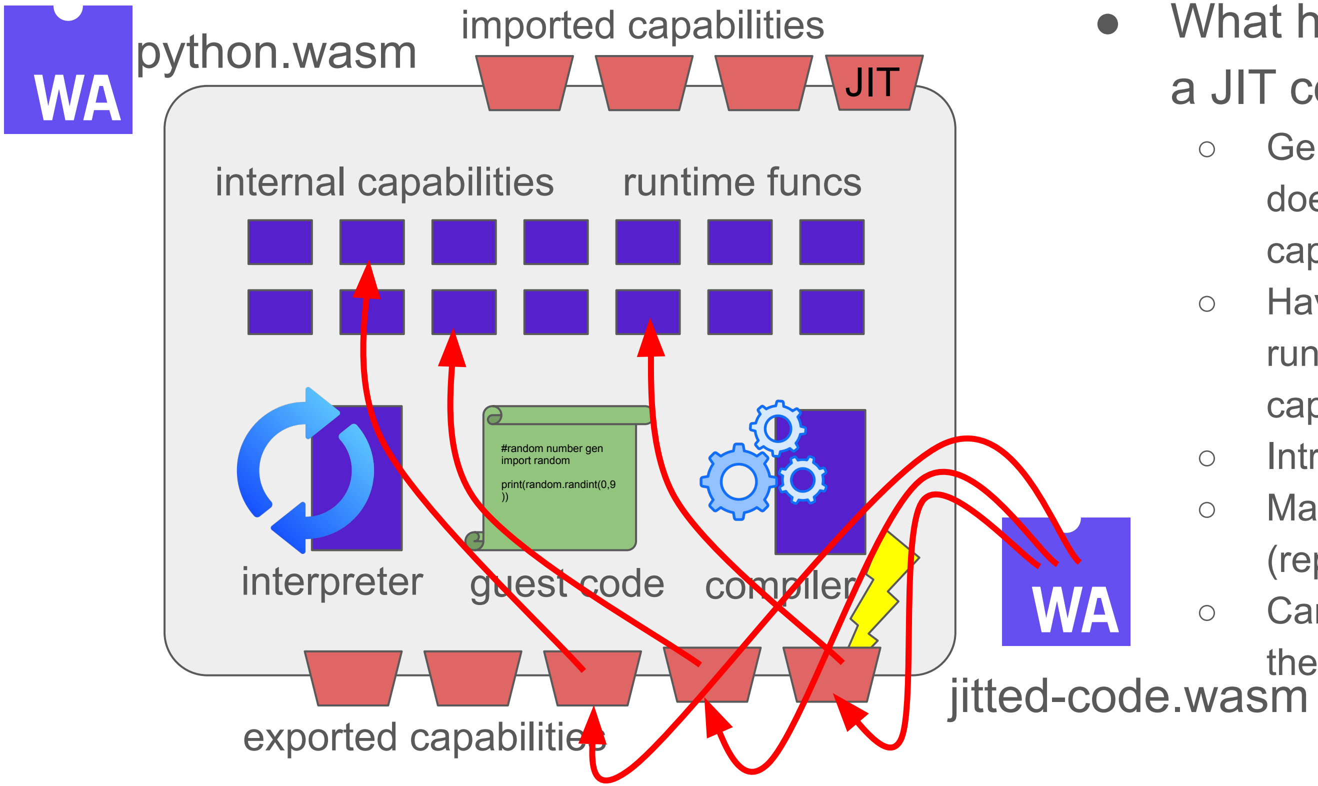


```
#random number gen
import random

print(random.randint(0,9
))
```

- What happens when we add a JIT compiler?
  - Generated module (rightly) doesn’t have access to internal capabilities of the runtime

Motivating Example - Guest language runtime

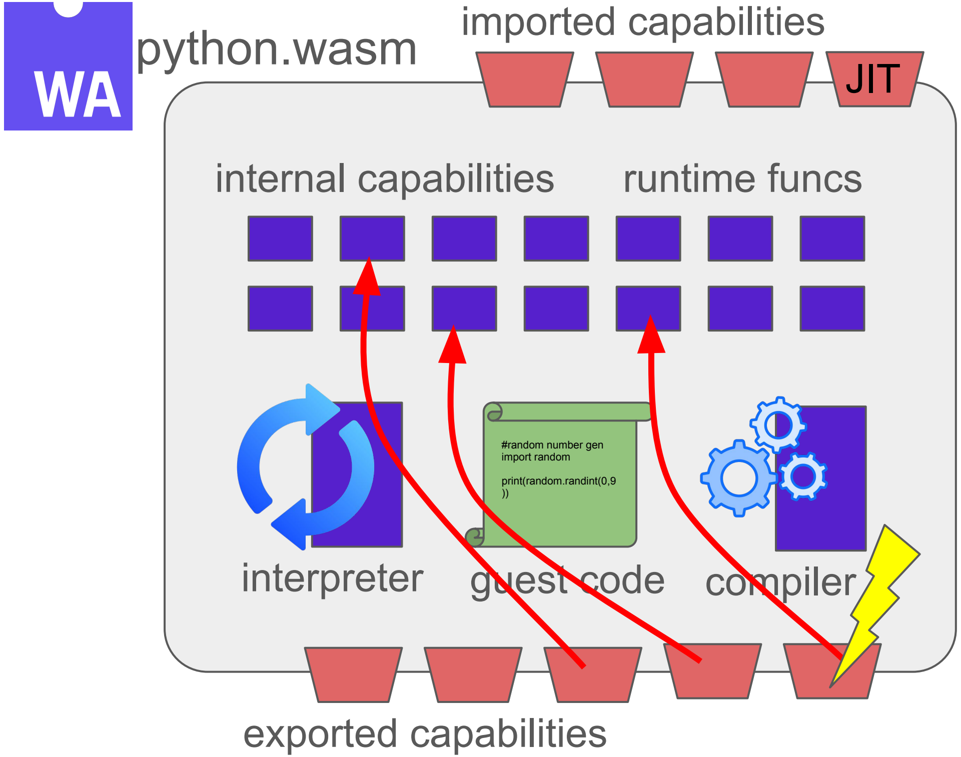


```
#random number gen
import random

print(random.randint(0,9
))
```

- What happens when we add a JIT compiler?
  - Generated module (rightly) doesn’t have access to internal capabilities of the runtime
  - Have to export internals of runtime as additional capabilities
  - Introduces indirections
  - Makes larger JITed artifacts (repeated imports)
  - Can’t export types, must repeat them

Motivating Example - Guest language runtime



```
#random number gen
import random

print(random.randint(0,9
))
```

- What happens when we add a JIT compiler?
  - Generated module (rightly) doesn’t have access to internal capabilities of the runtime
  - Have to export internals of runtime as additional capabilities
  - Breaks modularity of the guest language runtime

Proposal: fine-grained JIT interface

- A new bytecode: `func.new $mt $ft $env (start, end)`
  - `$mt`: memory index
  - `$ft`: function type index
  - `$env`: environment index
  - `(start, end)`: indexes into memory where bytecode is stored
- An engine executes `func.new` by:
  - Copying bytes from `(start, end)` in the memory `$mt`
  - Validating the code as if the body of a function with signature `$ft`
  - Returns a new funcref of type `(ref null $ft)` upon success

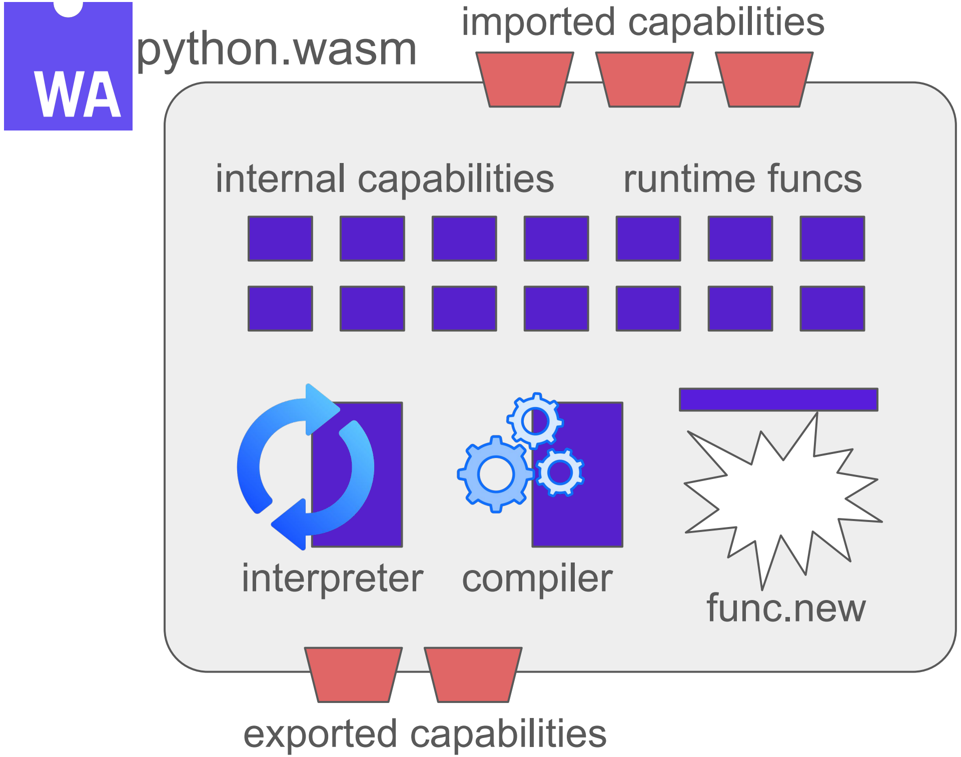
Proposal: fine-grained JIT interface

- A new bytecode: `func.new $mt $ft $env (start, end)`
  - `$mt`: memory index
  - `$ft`: function type index
  - `$env`: environment index
  - `(start, end)`: indexes into memory where bytecode is stored
- A new section: *environments* (scopes)
  - Declares a subset of the encapsulating module to be accessible to new code
  - Renumbers declarations starting from 0

Proposal: fine-grained JIT interface

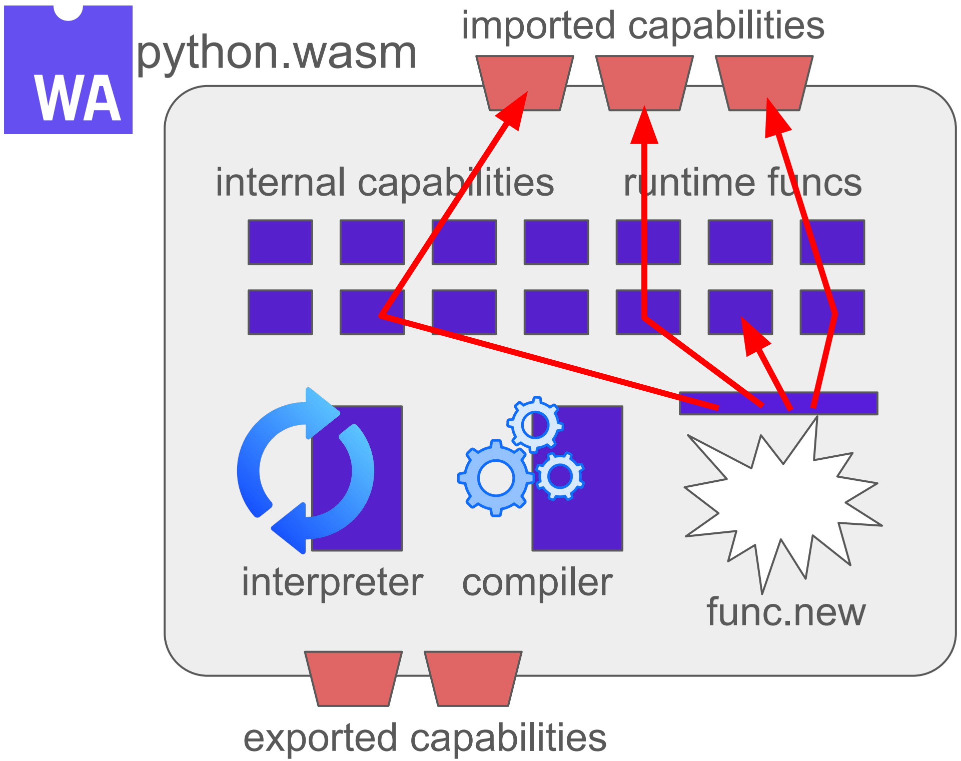
- A new bytecode: `func.new $mt $ft $env (start, end)`
  - `$mt`: memory index
  - `$ft`: function type index
  - `$env`: environment index
  - `(start, end)`: indexes into memory where bytecode is stored
- A new section: *environments* (scopes)
  - Declares a subset of the encapsulating module to be accessible to new code
  - Renumbers declarations starting from 0
- A flag for memories: `code`
  - Similar to `shared` flag, allows a memory to be used for `func.new`
  - Prevents accidental use of a memory to make code

Motivating Example - Guest language runtime



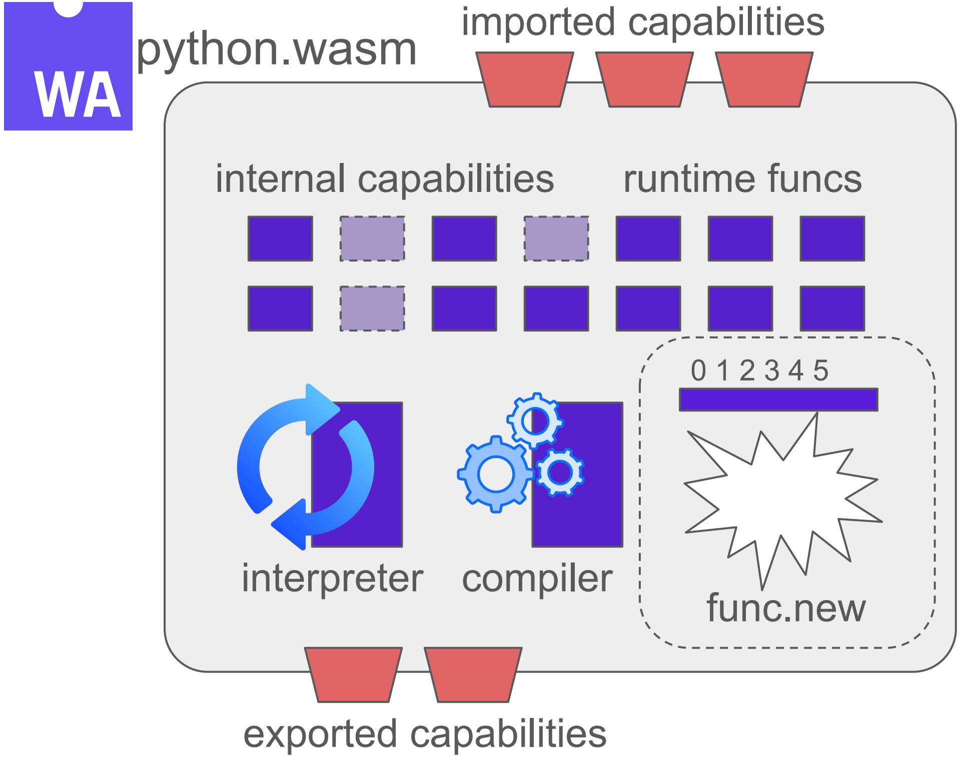
- func.new plus its environment is a controlled “hole” in the module for dynamically creating new functions

Motivating Example - Guest language runtime



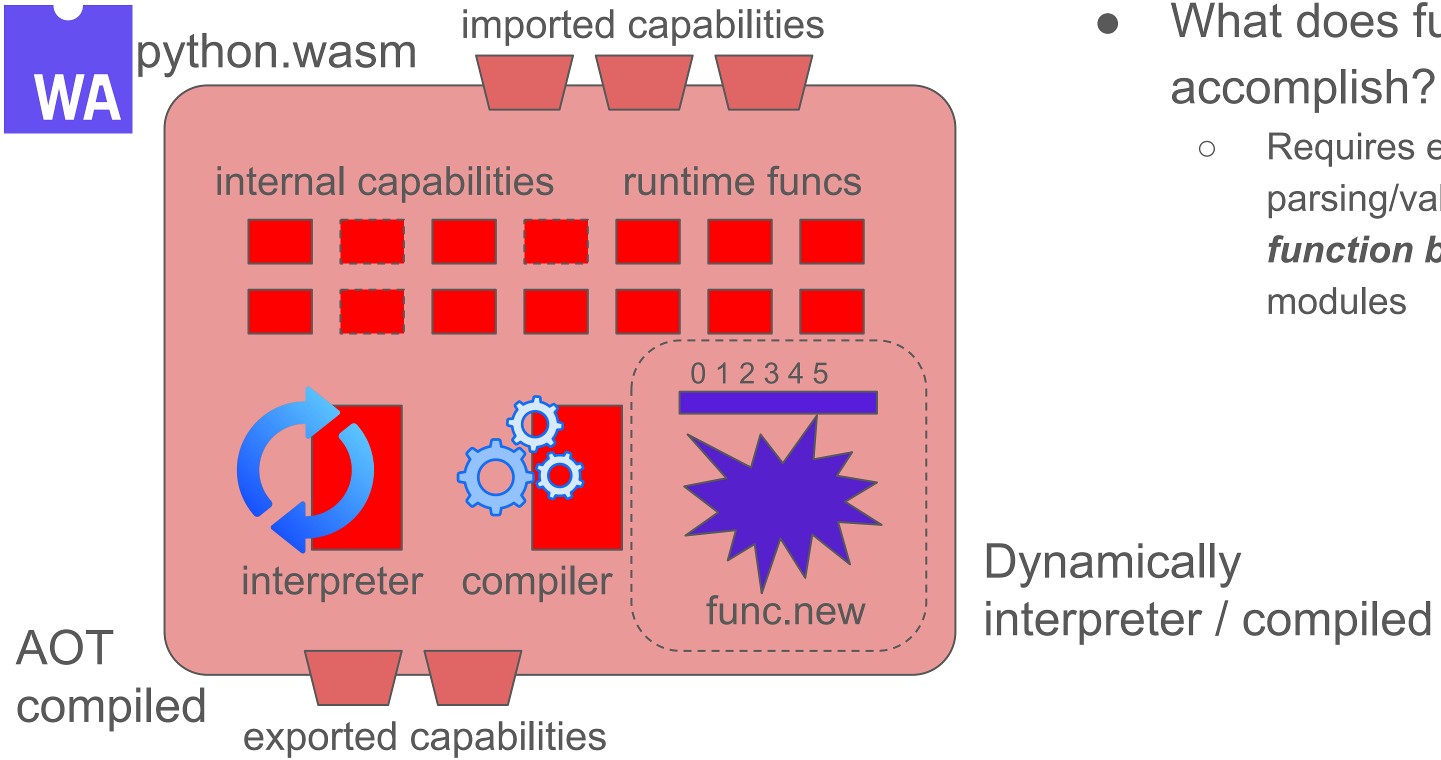
- What does func.new accomplish?
  - No new host capability needed for new code
  - New code has no additional privileges
  - Preserves guest runtime module encapsulation
  - Environment describes exactly what new code can use from the containing module, allowing static reasoning
  - Engine has simplified code validator (not whole module)

Motivating Example - Guest language runtime



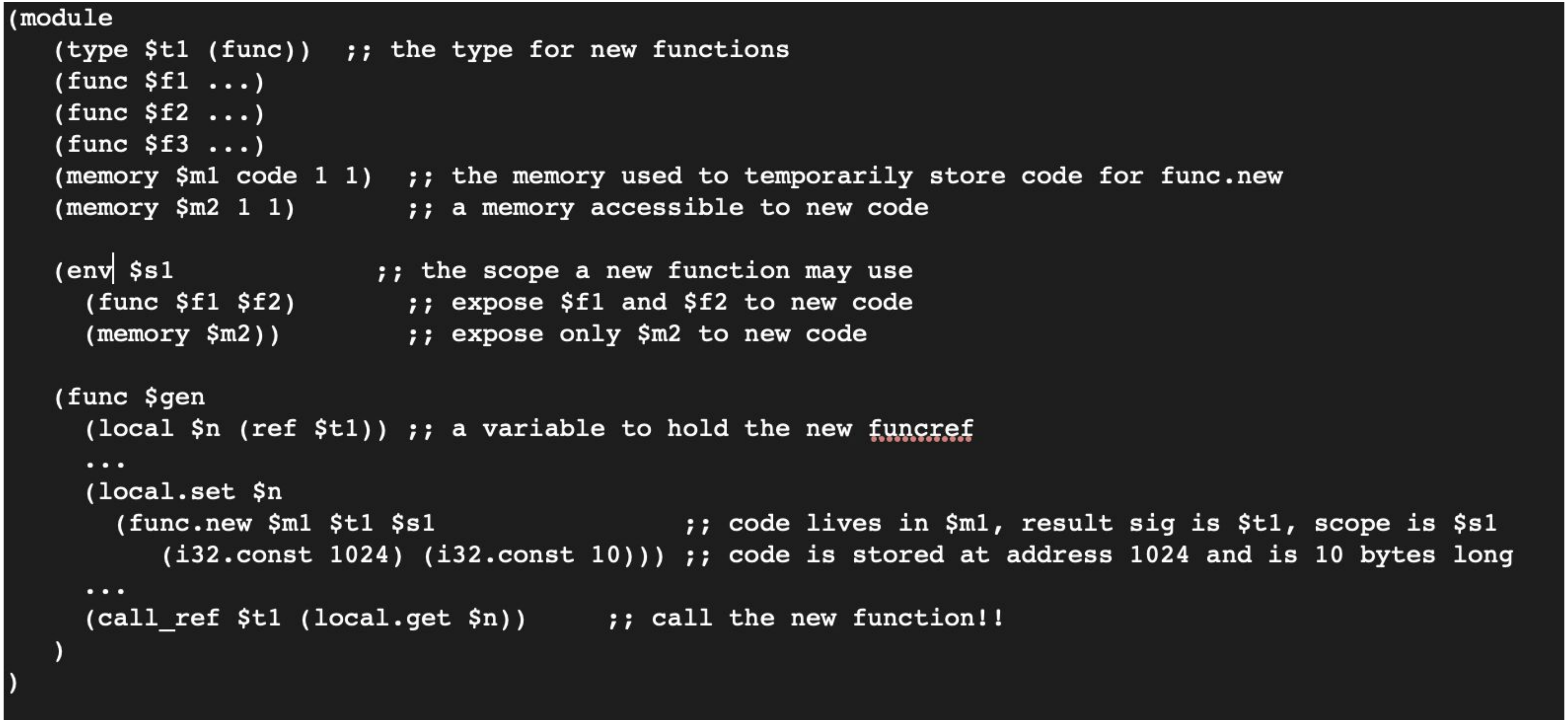
- What does func.new accomplish?
  - Toolchains can still reorganize modules without requiring new code to renumber
  - Allows sound DCE, inlining, module combining, and other transformations

Motivating Example - Guest language runtime



- What does func.new accomplish?
  - Requires engine to support parsing/validating ***only function bodies***, not whole modules

Example .wat Usage



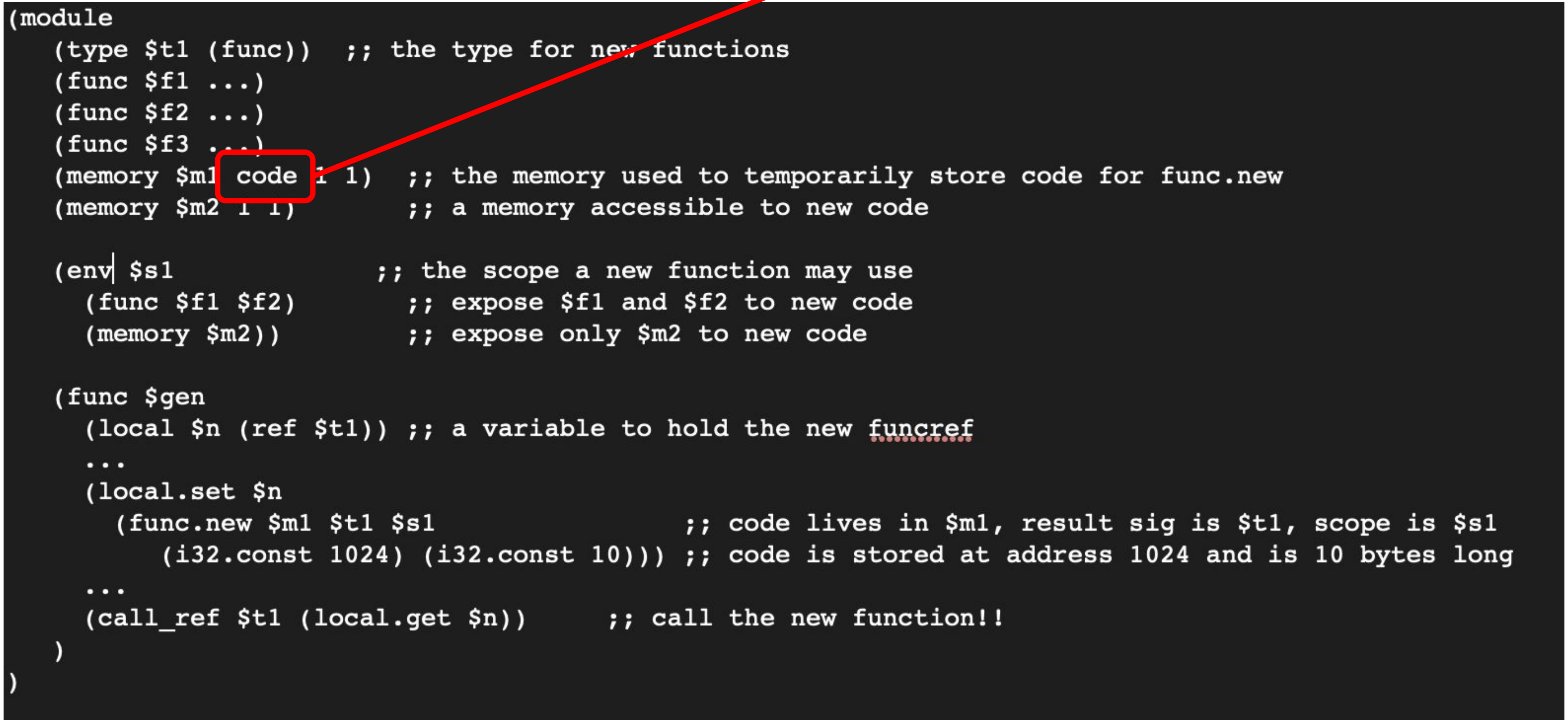
```
(module
   (type $t1 (func))  ;; the type for new functions
   (func $f1 ...)
   (func $f2 ...)
   (func $f3 ...)
   (memory $m1 code 1 1)  ;; the memory used to temporarily store code for func.new
   (memory $m2 1 1)       ;; a memory accessible to new code

   (env $s1             ;; the scope a new function may use
     (func $f1 $f2)       ;; expose $f1 and $f2 to new code
     (memory $m2))        ;; expose only $m2 to new code

   (func $gen
     (local $n (ref $t1)) ;; a variable to hold the new funcref
     ...
     (local.set $n
       (func.new $m1 $t1 $s1                ;; code lives in $m1, result sig is $t1, scope is $s1
          (i32.const 1024) (i32.const 10))) ;; code is stored at address 1024 and is 10 bytes long
     ...
     (call_ref $t1 (local.get $n))     ;; call the new function!!
   )
)
```

Example .wat Usage

Code flag for memory



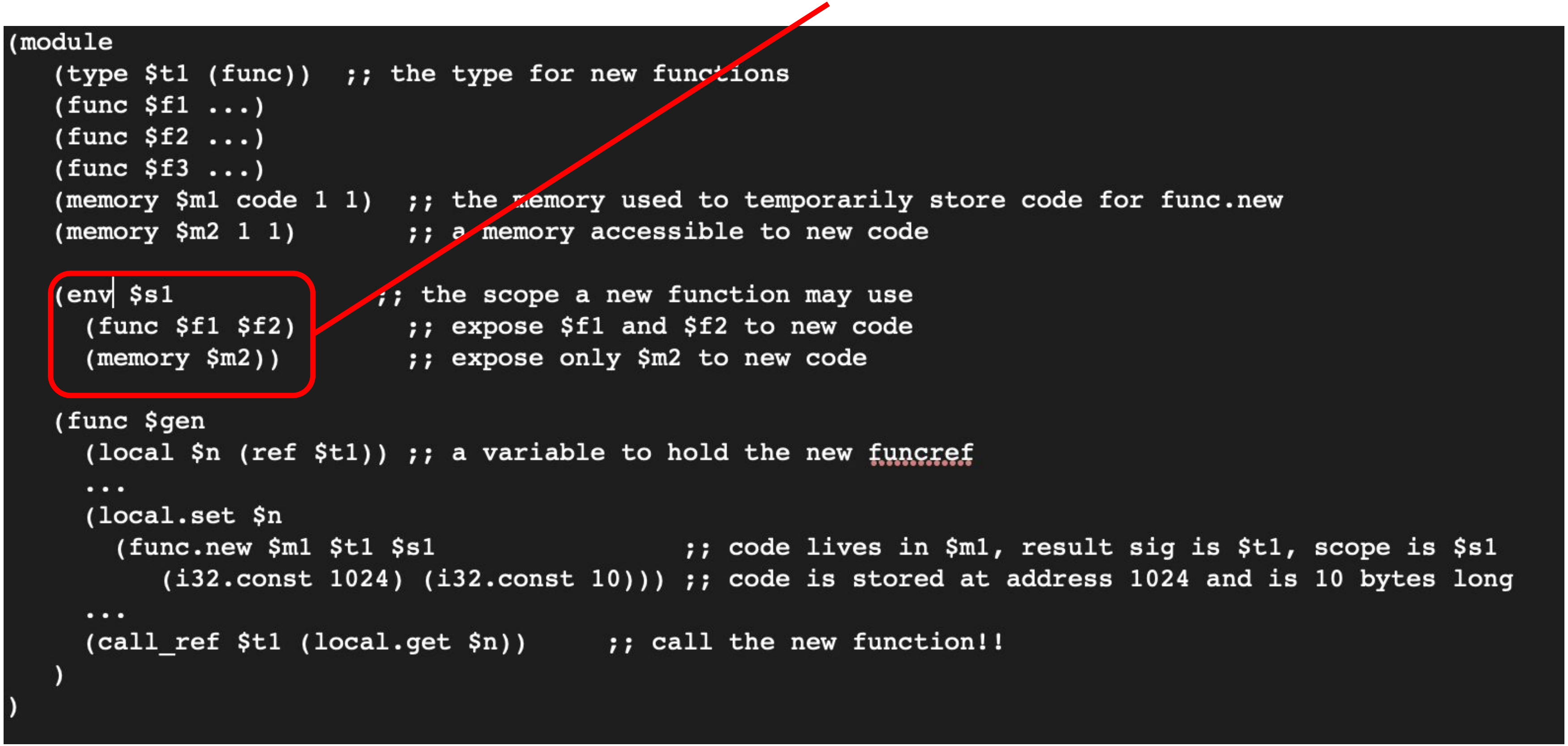
```
(module
   (type $t1 (func))  ;; the type for new functions
   (func $f1 ...)
   (func $f2 ...)
   (func $f3 ...)
   (memory $m1 code 1 1)   ;; the memory used to temporarily store code for func.new
   (memory $m2 1 1)        ;; a memory accessible to new code

   (env $s1              ;; the scope a new function may use
     (func $f1 $f2)        ;; expose $f1 and $f2 to new code
     (memory $m2))         ;; expose only $m2 to new code

   (func $gen
     (local $n (ref $t1)) ;; a variable to hold the new funcref
     ...
     (local.set $n
       (func.new $m1 $t1 $s1                ;; code lives in $m1, result sig is $t1, scope is $s1
          (i32.const 1024) (i32.const 10))) ;; code is stored at address 1024 and is 10 bytes long
     ...
     (call_ref $t1 (local.get $n))     ;; call the new function!!
   )
)
```

Example .wat Usage

Environment for new code



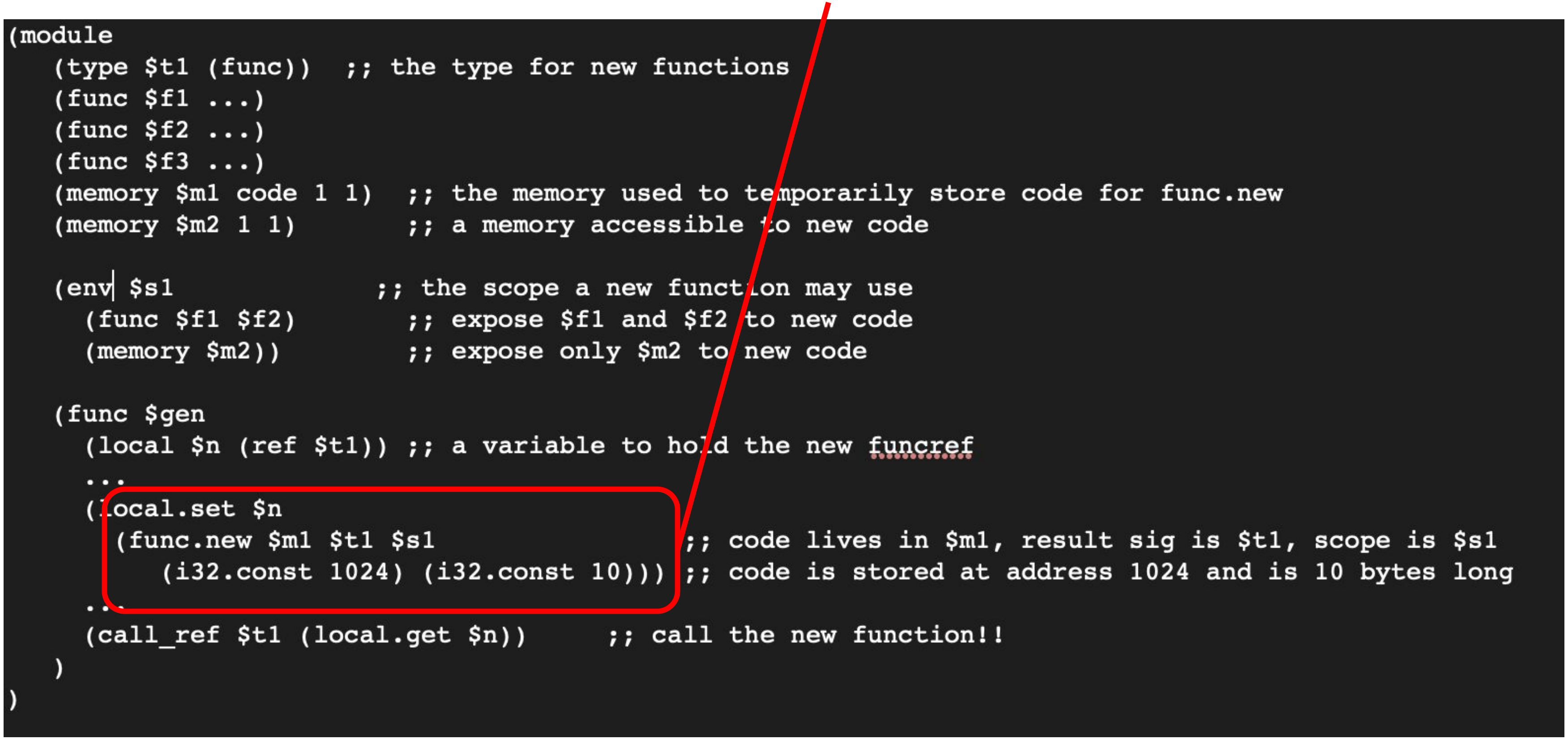
```
(module
   (type $t1 (func))  ;; the type for new functions
   (func $f1 ...)
   (func $f2 ...)
   (func $f3 ...)
   (memory $m1 code 1 1)  ;; the memory used to temporarily store code for func.new
   (memory $m2 1 1)       ;; a memory accessible to new code

   (env $s1             ;; the scope a new function may use
     (func $f1 $f2)       ;; expose $f1 and $f2 to new code
     (memory $m2))        ;; expose only $m2 to new code

   (func $gen
     (local $n (ref $t1)) ;; a variable to hold the new funcref
     ...
     (local.set $n
       (func.new $m1 $t1 $s1                ;; code lives in $m1, result sig is $t1, scope is $s1
          (i32.const 1024) (i32.const 10))) ;; code is stored at address 1024 and is 10 bytes long
     ...
     (call_ref $t1 (local.get $n))     ;; call the new function!!
   )
)
```

Example .wat Usage

func.new bytecode usage



```
(module
   (type $t1 (func))  ;; the type for new functions
   (func $f1 ...)
   (func $f2 ...)
   (func $f3 ...)
   (memory $m1 code 1 1)  ;; the memory used to temporarily store code for func.new
   (memory $m2 1 1)       ;; a memory accessible to new code

   (env $s1             ;; the scope a new function may use
     (func $f1 $f2)       ;; expose $f1 and $f2 to new code
     (memory $m2))        ;; expose only $m2 to new code

   (func $gen
     (local $n (ref $t1)) ;; a variable to hold the new funcref
     ...
     (local.set $n
       (func.new $m1 $t1 $s1                ;; code lives in $m1, result sig is $t1, scope is $s1
          (i32.const 1024) (i32.const 10))) ;; code is stored at address 1024 and is 10 bytes long
     ...
     (call_ref $t1 (local.get $n))     ;; call the new function!!
   )
)
```

Discussion points

- Is this nested modules? (component model)
- Asynchronous compilation
  - Different opcode `func.new_async`?
  - Blocking by default?
  - Non-blocking, blocks on call?
  - `Func.ready` allows polling?
- How to handle failure, e.g. resource exhaustion
- Feature testing
- How to incorporate custom sections (debug names, branch/compilation hints, other annotations)
- Is the `code` flag for memories useful?
- Should we accept Wasm GC array of bytes?
- Multiple functions at a time? (for mutual direct calls)
- What about new GC types? (`type.new`)

Poll for Phase 1

Entry requirements

- There is general interest within the CG in this feature.
- The CG believes the feature is in-scope and will plausibly be workable.

Func.new is All We Need

https://suno.com/song/19e0679a-81a2-419f-8cc7-ff85ac49c23d

[verse]
In the old days
We could just flip a bit
T.L.B. shootdown (OS kernel)
And new instructions were ready
All the baddies came (cash money)
And took over
And that was bad (privilege escalation)

[chorus]
func.new (ooh-ooh)
is all we need
func.new (ooh-ooh)
is all we need
new code
with just the caps we want (no escalation)

[verse]
So hey we did Wah-zum
Now there's no R...C...Es
Code and data izza sep-ah-rut
But sometimes
Interpreters go slow (oh oh oh)
A whole new module is a hassle

[verse]
Inside a module, it's all our show
But the tools rearrange us
So better let them know
What the new code uses

[verse]
So we propose
a new mechanism
for new code
but no funny business