Design Considerations for cont.bind

Frank Emrich

University of Edinburgh

Agenda

- Some background:
  - Compiling stack switching
  - Passing payloads
- Naive implementation of `cont.bind`
- Combining `cont.bind` with more efficient payload passing schemes
- Left-to-right vs. right-to-left binding of `cont.bind`

Background

Use cases (1)

- In general: Giving continuations same type so we can treat them uniformly
- Example: Generator function reading some memory, taking address and length

```
(func $generator (param $addr i32) (param $length i32) …)

(func $consumer (param i32 $addr)
  (local $c (ref $ct0))

  (local.get $addr) ;; first cont.bind arg
  (i32.const 100)   ;; second cont.bind arg
  (cont.new $ct2 (ref.func $generator))
  (cont.bind $ct2 $ct0)
  (local.set $c)

  ($loop
    ;; act on continuation in $c of type $c0:
    ;; either initial value created above or received by handlers on $yield
  )
)
```

```
;; $ct2 ~~ cont [i32 i32] -> []
;; $ct0 ~~ cont [] -> []
(tag $gen (param i32))
```

Use cases (2)

- In general: Giving continuations same type so we can treat them uniformly
- Example: Task scheduling with two different tag return types

  - `(tag $yield)`
    Used by tasks to yield control

  - `(tag $request (result i32))`
    Used by tasks to indicate that they need to be assigned next job to work on (identified by i32 id)

  - `(table $task_queue 0 (ref null $ct)),` with `$ct0 ~~ cont [] -> []`
    Task queue managed by scheduler

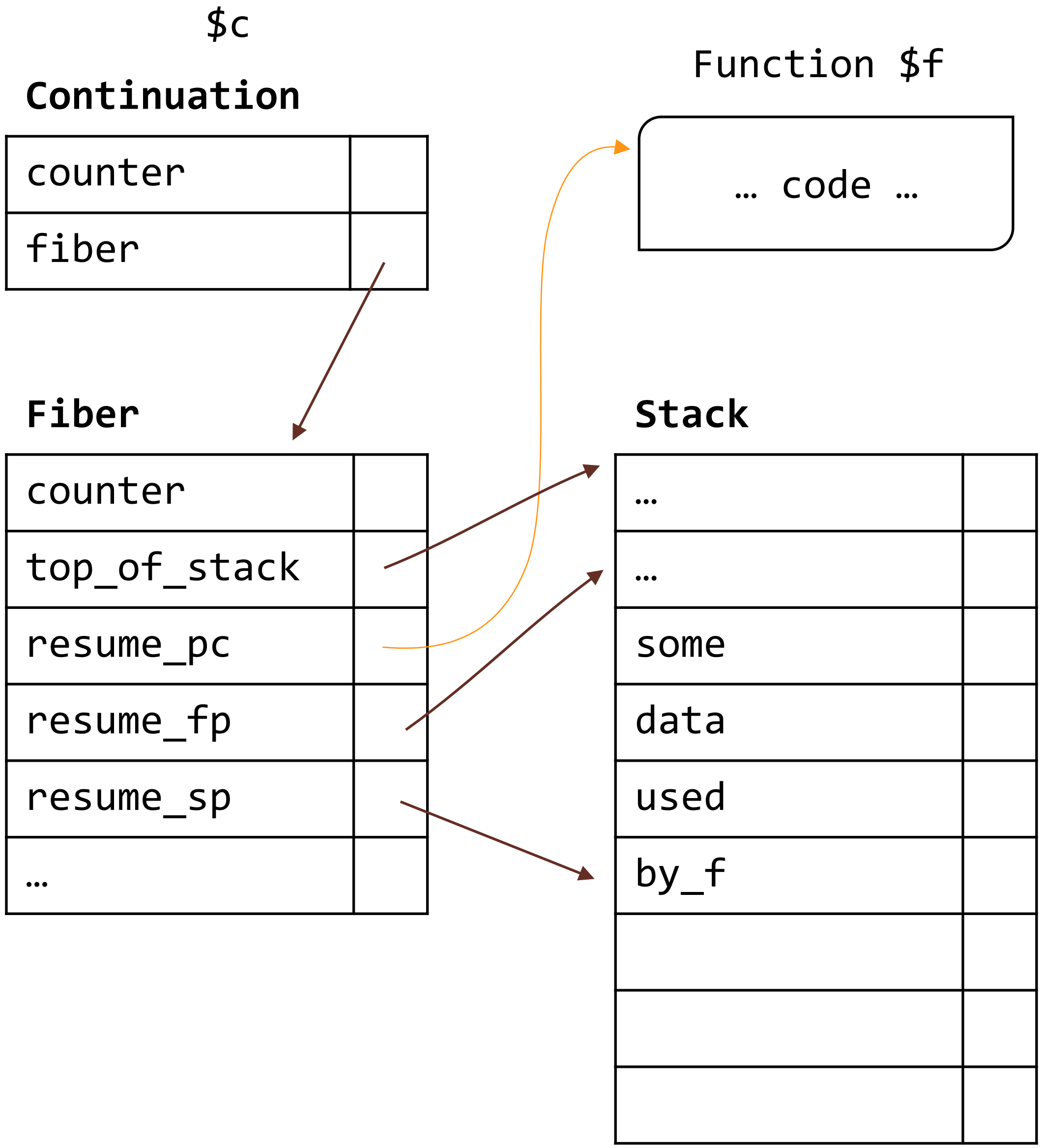
  - Continuations received in `$yield` handler:
    Have type `$ct0`, immediately re-added to `$task_queue`

  - Continuations received `in $request` handler:
    Have type `$ct1 ~~ cont [i32] -> []`. Supply work package id using `cont.bind $ct1 $ct0`, then add resulting continuation to `$task_queue`

Compiling stack switching

- $c is currently executing $f
- What to do on the following?
  `(resume $ct (local.get $c))`

- Update stack chain
- Save current state (GRPs, SP, FP, IP to resume to)
- Handle resume arguments
- `SP := fiber.resume_sp`
- `FP := fiber.resume_fp`
- `JMP fiber.resume_pc`



$c

Continuation

counter

fiber

Function $f

… code …

Fiber

counter

top_of_stack

resume_pc

resume_fp

resume_sp

…

Stack

…

…

some

data

used

by_f

Passing arguments on resume

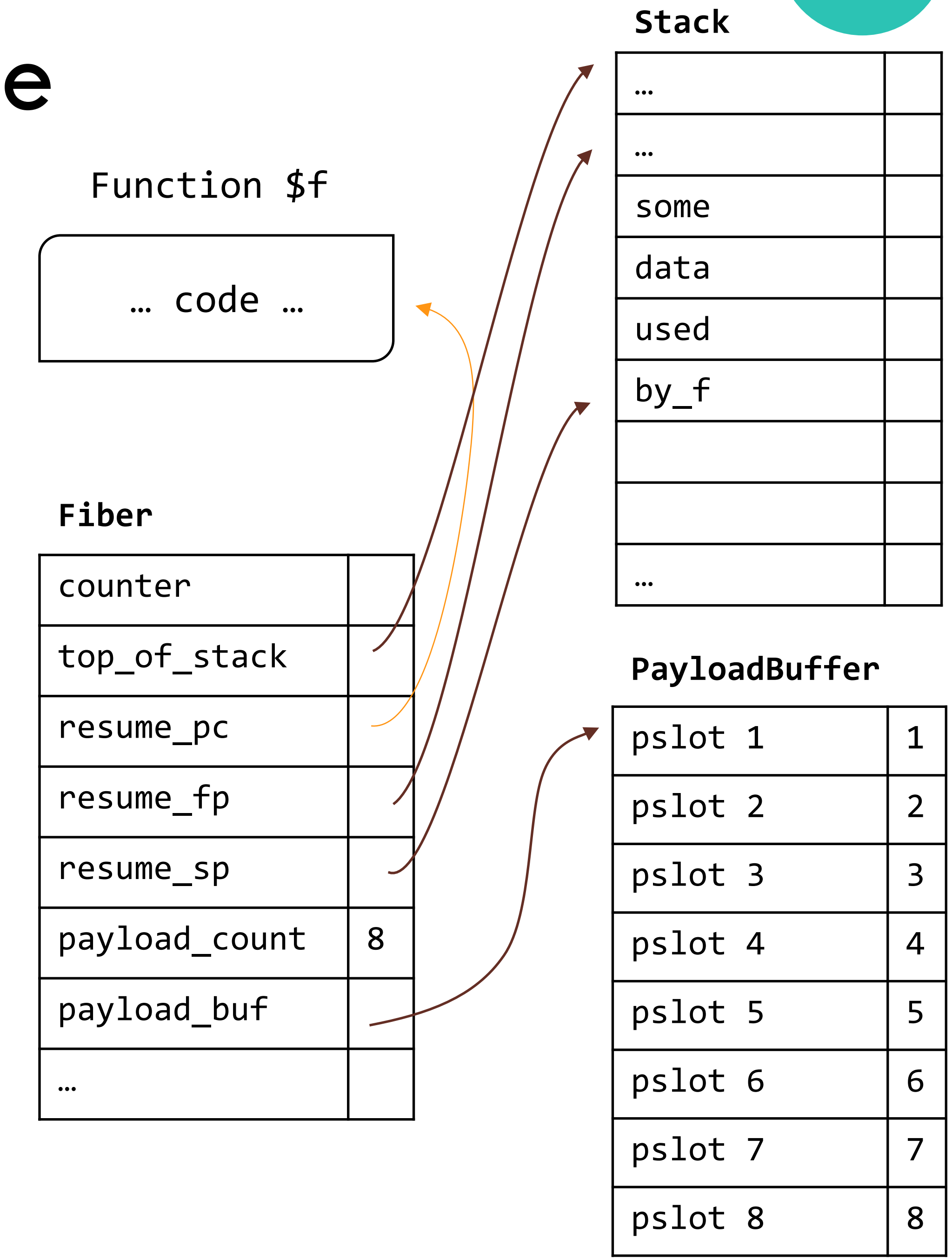
Assumption: no `cont.bind` happened

What to do on the following?

```
;; $ct8~~= cont [i32^8] -> []
(i32.const 1)
(i32.const 2)
 …
(i32.const 7)
(i32.const 8)
(resume $ct8 (local.get $c))
```

Option 1: Write to separate buffer, read by other stack after stack switch

- If continuation is “fresh” (at beginning of function $f), we need extra logic to load arguments into registers as per calling convention



Function $f

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
payload_count	8
payload_buf	
…	

Stack

…	
…	
some	
data	
used	
by_f	
…	

PayloadBuffer

pslot 1	1
pslot 2	2
pslot 3	3
pslot 4	4
pslot 5	5
pslot 6	6
pslot 7	7
pslot 8	8

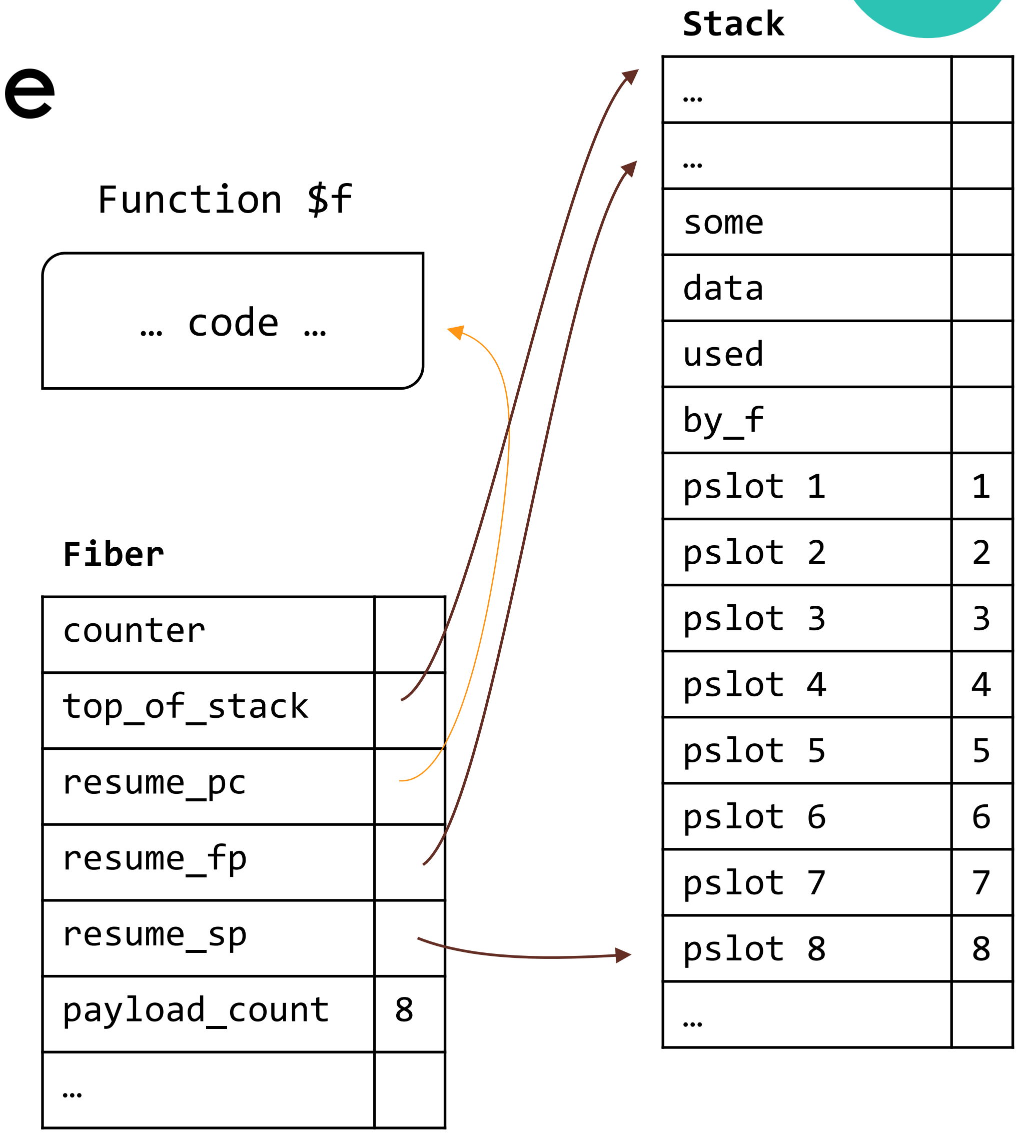
Passing arguments on resume

Assumption: no `cont.bind` happened

Situation: `resume` with i32 args 1 … 8

Option 2: Same, but use actual stack memory for data

- When c is created (by `cont.new, suspend, switch`, …), we know how many slots to reserve)
- Still need to reshuffle if continuation is fresh



Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
payload_count	8
…	

Stack

…	
…	
some	
data	
used	
by_f	
pslot 1	1
pslot 2	2
pslot 3	3
pslot 4	4
pslot 5	5
pslot 6	6
pslot 7	7
pslot 8	8
…	

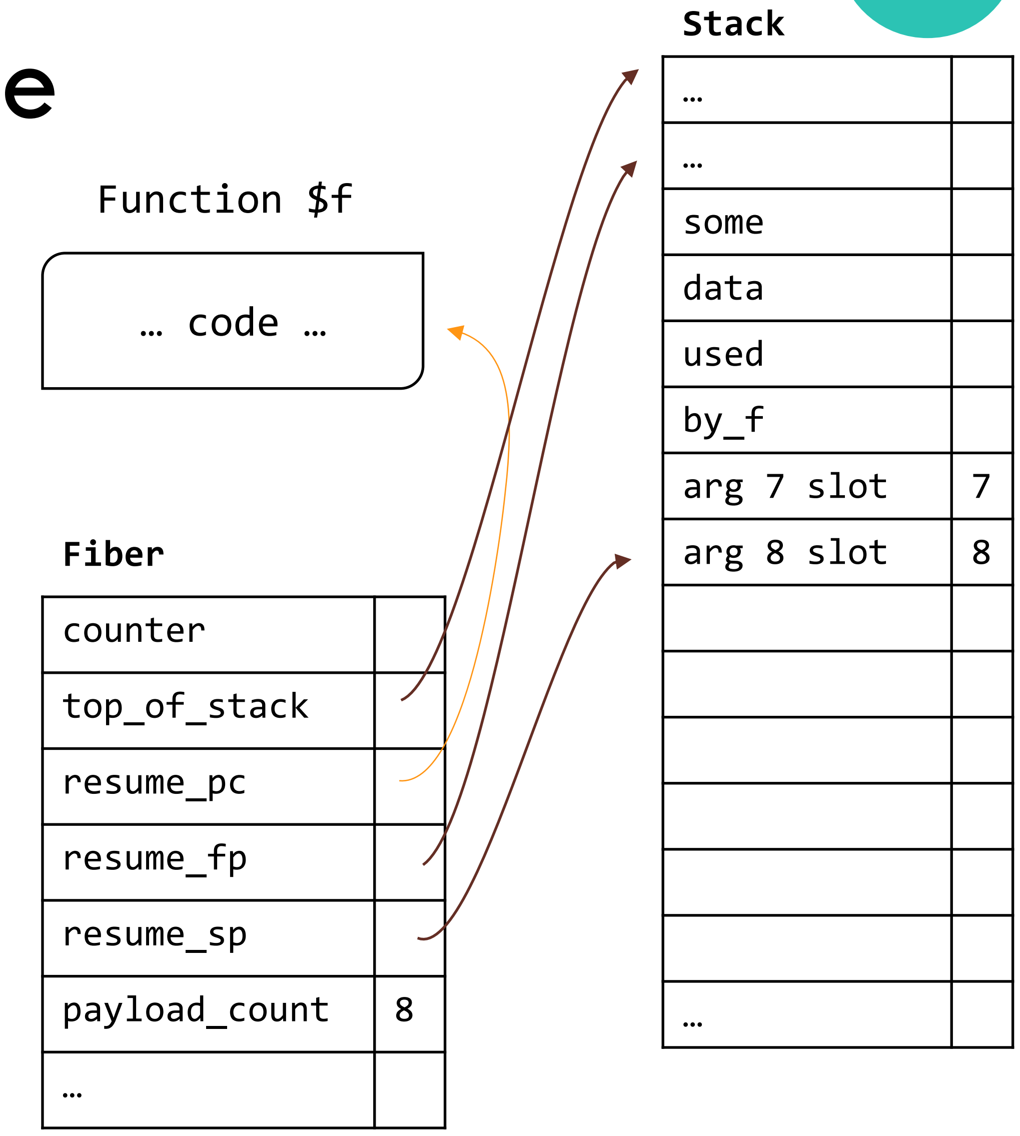
Passing arguments on resume

Assumption: no `cont.bind` happened

Situation: `resume` with i32 args 1 … 8

Option 3: *Always* use system calling convention

- If passing 8 i32, put first 6 in registers, pass remaining two on stack (assuming System V CC)
- Do this regardless of continuation being fresh or not
- Never need to reshuffle if continuation is fresh, everything is exactly in the right place
- We've ignored `cont.bind` so far. How to handle that?



Function $f

… code …

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
payload_count	8
…	

Stack

…	
…	
some	
data	
used	
by_f	
arg 7 slot	7
arg 8 slot	8
…	

RDI: 1, RSI: 2, RDX: 3, RCX: 4, R8: 5, R9: 6

cont.bind: Simple implementation

Everything handled through memory

Initial observations

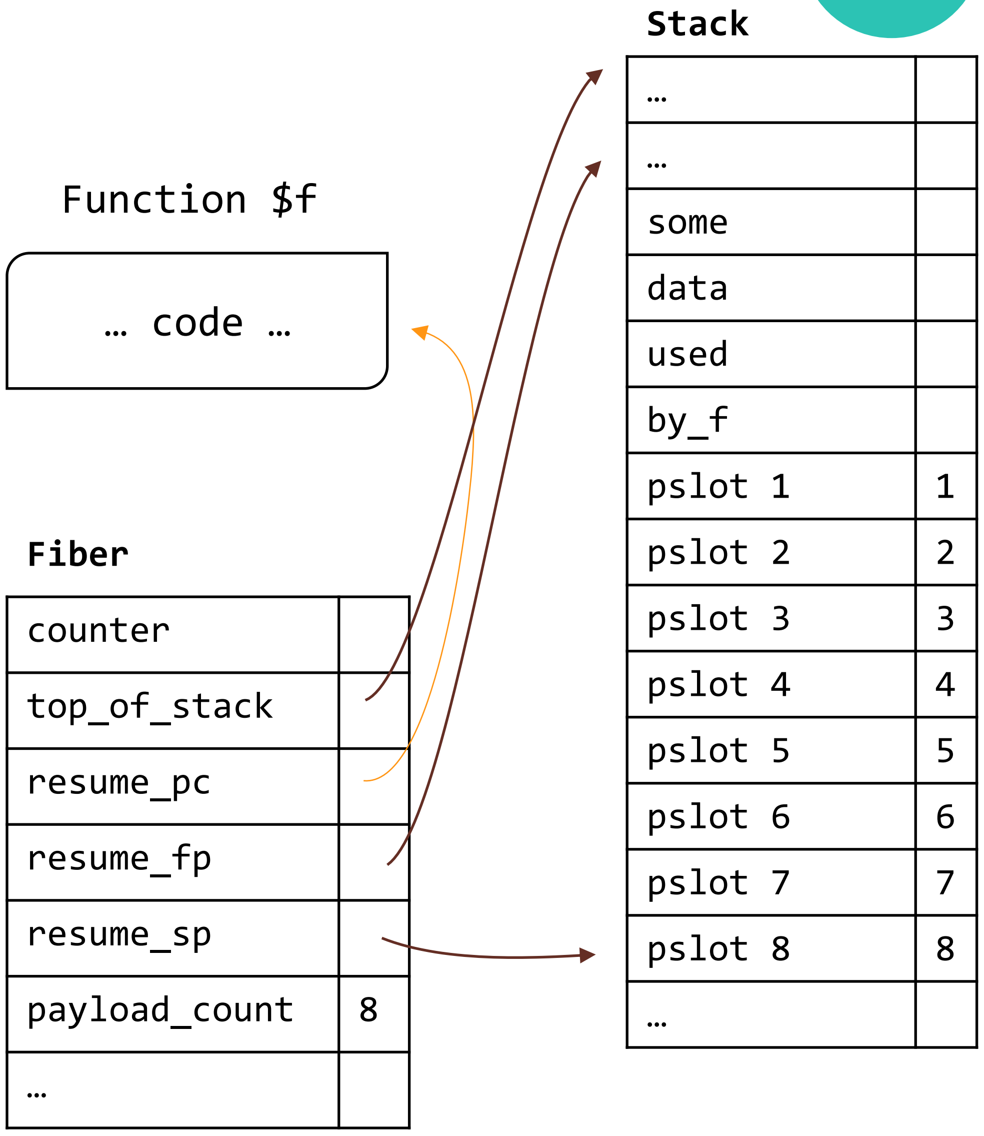
- By default, we should assume that every **cont.bind** argument must be stored somewhere in memory

```
(i32.const 123) ;; cont.bind arg
(cont.new $ct1 (ref.func $f))
(cont.bind $ct1 $ct0)
(local.set $c)

<some 1000 unrelated instructions>

(resume $ct0 (local.get $c))
```

- If all payloads passed using memory ("Option 1 / 2"), we can easily extend that to handle **cont.bind**
- How to handle more efficient handling scheme? ("Option 3")



Function $f

… code …

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
payload_count	8
…	

Stack

…	
…	
some	
data	
used	
by_f	
pslot 1	1
pslot 2	2
pslot 3	3
pslot 4	4
pslot 5	5
pslot 6	6
pslot 7	7
pslot 8	8
…	

The challenge

- Can we achieve the following at the same time?
  1. Use some fixed calling convention to pass payloads using registers (& stack, if needed) ("Option 3")
  2. Absolutely no overhead on `resume` and `switch` if the continuation was not created by `cont.bind`

- Taking this further: Move all potential overhead into `cont.bind` code itself. For everything else, must generate and run same code as *if `cont.bind` didn't exist in the proposal*

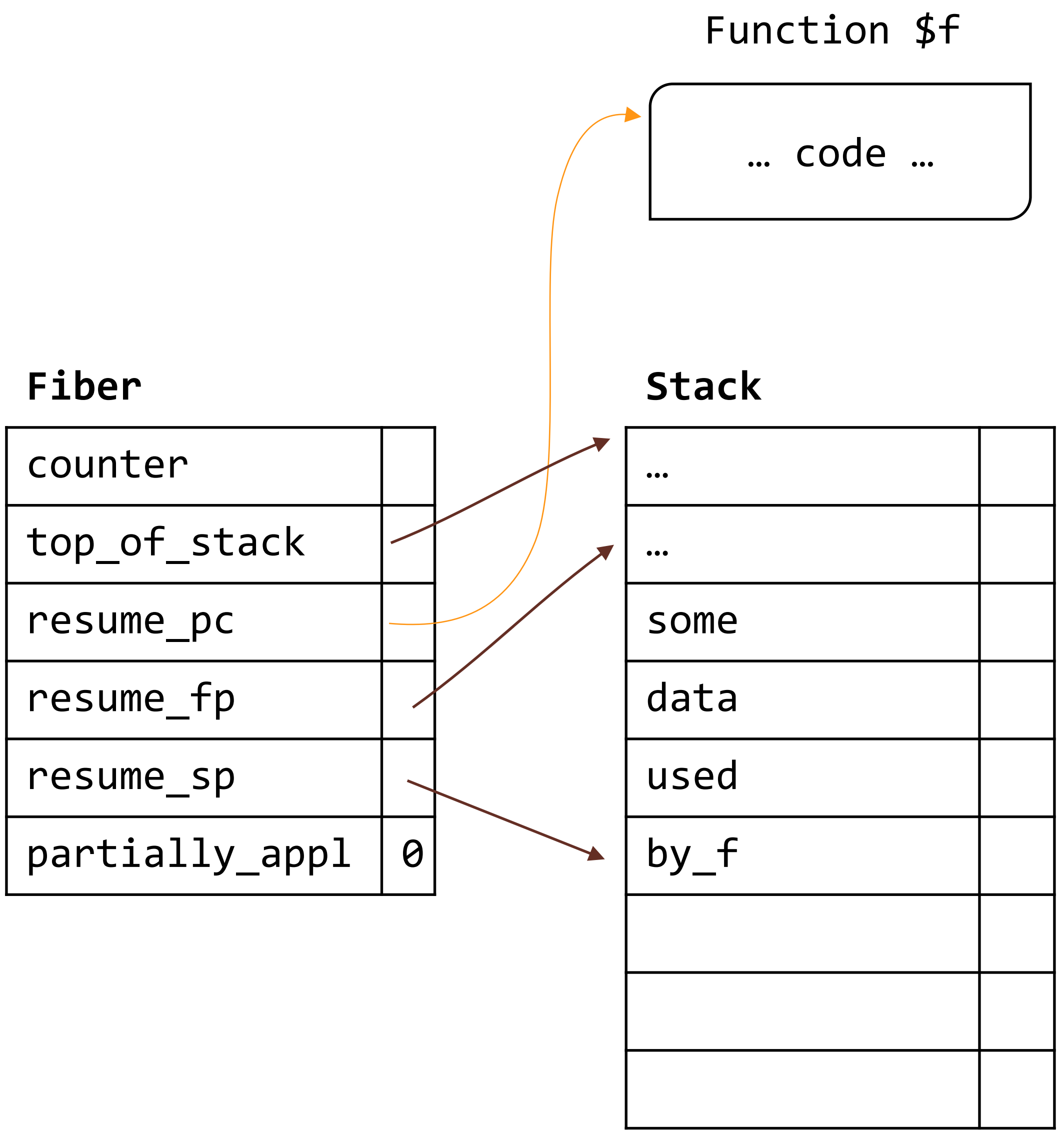
- My claim: Yes! All we need is one boolean flag in each `Fiber`, *never accessed if there was no `cont.bind`*

Revisiting resume

We don’t want this on `resume`:

- Update stack chain
- Save current state (GRPs, SP, FP, IP to resume to)

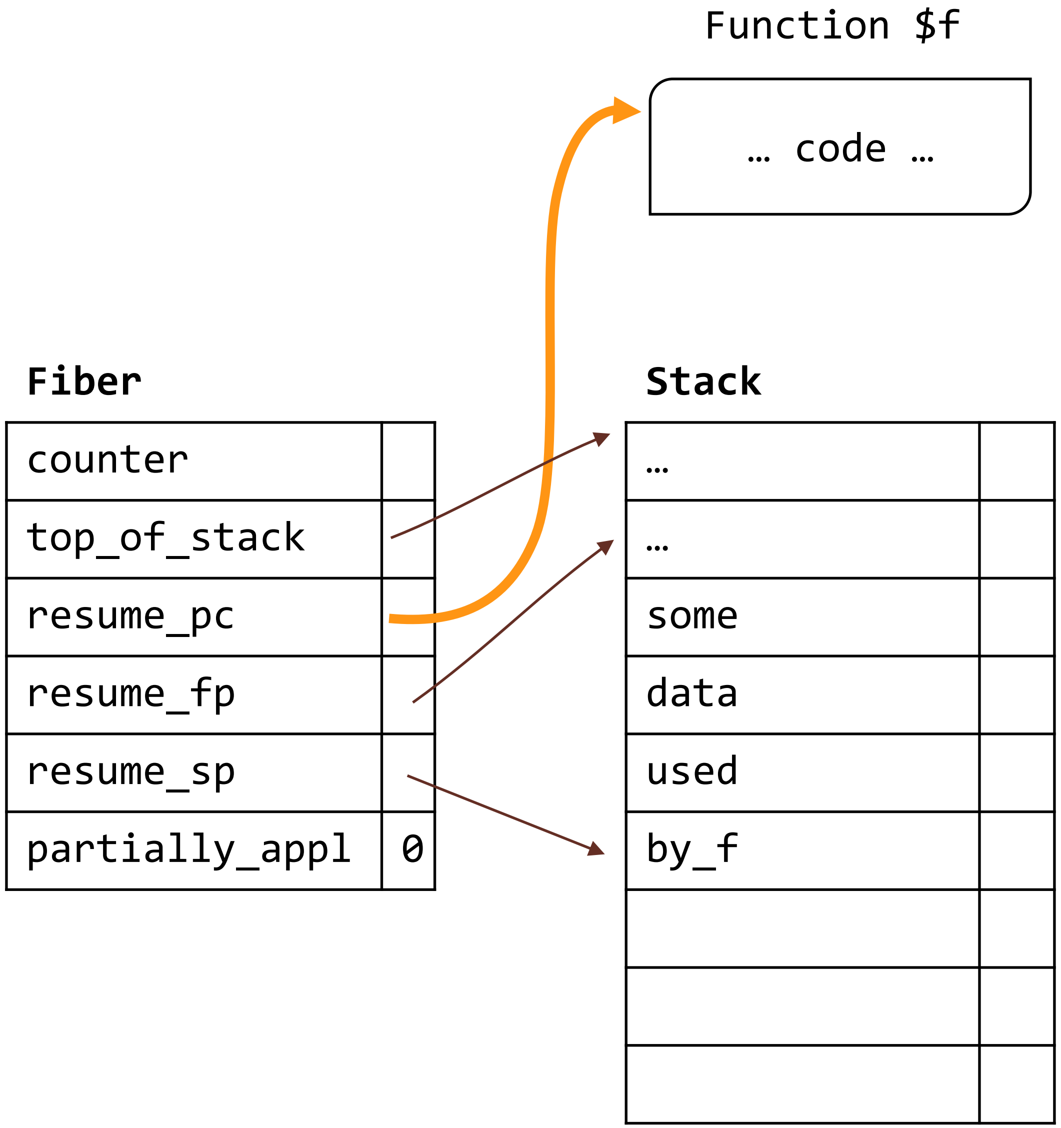
-
  ```
  if fiber.partially_appl {
     // do something
  } else {
     // do something else
  }
  ```
- `SP := fiber.resume_sp`
- `FP := fiber.resume_fp`
- `JMP fiber.resume_pc`



Revisiting resume

We don’t want this on `resume`:

- Update stack chain
- Save current state (GRPs, SP, FP, IP to resume to)
-
```
if fiber.partially_appl {
    // do something
} else {
    // do something else
}
```
- `SP := fiber.resume_sp`
- `FP := fiber.resume_fp`
- **`JMP fiber.resume_pc`**



Function $f

… code …

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
partially_appl	0

Stack

…	
…	
some	
data	
used	
by_f	

cont.bind: Trampoline-based implementation

Idea

- `resume`-ing a continuation resulting from *n* `cont.bind` instructions goes through *n* trampolines before switching to actual continuation

```
;; 5 arguments on Wasm value stack
(local.get $c) ;; c : $ct5
(cont.bind $ct5 $ct4) ;; 3rd trampoline to be executed
(cont.bind $ct4 $ct3) ;; 2nd trampoline to be executed
(cont.bind $ct3 $ct1) ;; 1st trampoline to be executed
(resume $ct1)
```

- Trampolines accumulate arguments:
  - Receive all previous payloads as parameters
  - Load `cont.bind`-applied values from memory
  - Pass on combined values to next stage (trampoline or continuation)

Example (1)

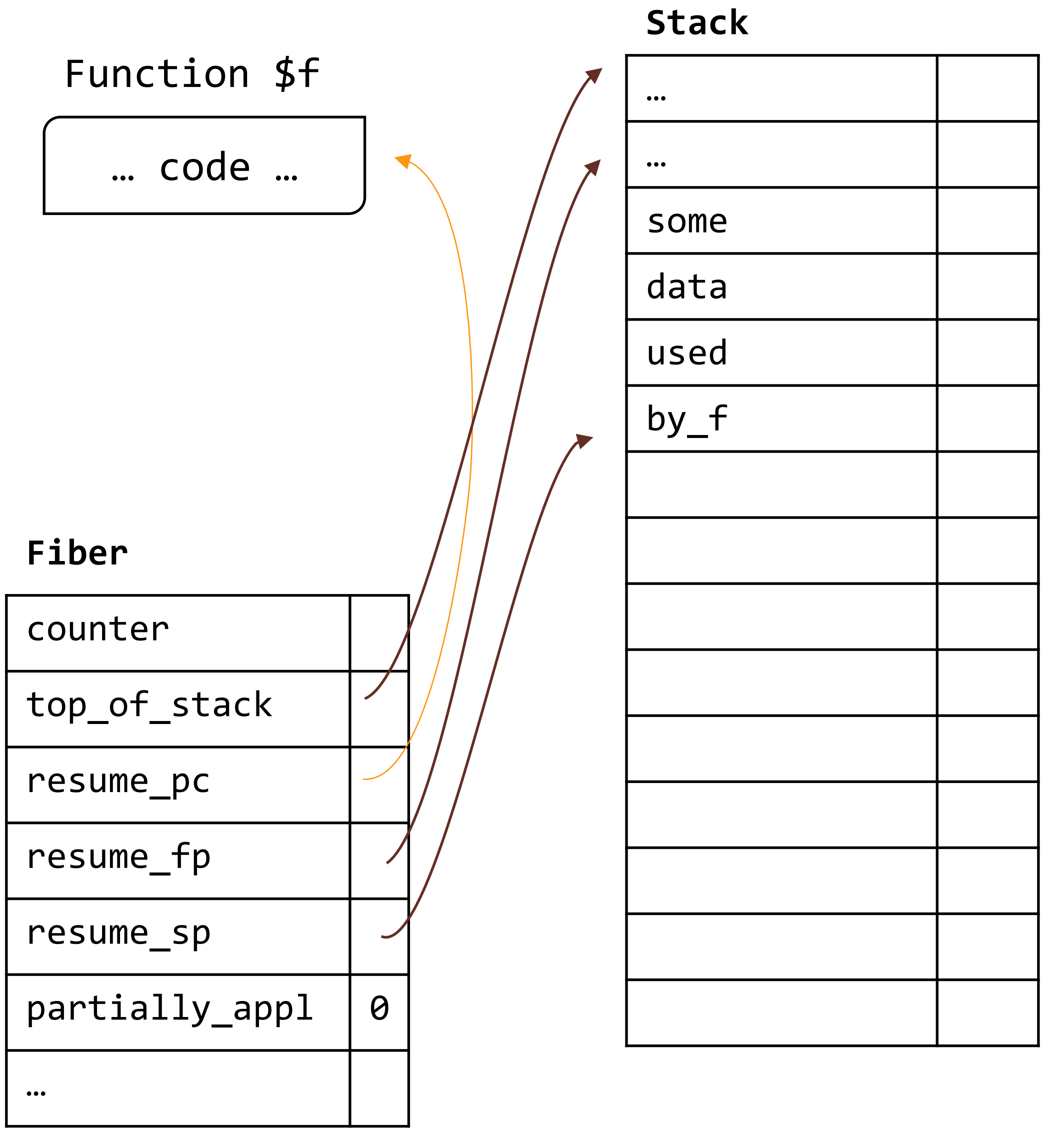
```
;; $ct2 ~~ cont [i64 i32] -> []
;; $ct1 ~~ cont [    i32] -> []


;; $c2 not created by cont.bind

(cont.bind $ct2 $ct1
  (i64.const 1)
  (local.get $c2))
(local.set $c1)

...

(resume $ct1 (i32.const 2) (local.get $c1)
```



Function $f

… code …

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
partially_appl	0
…	

Stack

…	
…	
some	
data	
used	
by_f	

Example (2)

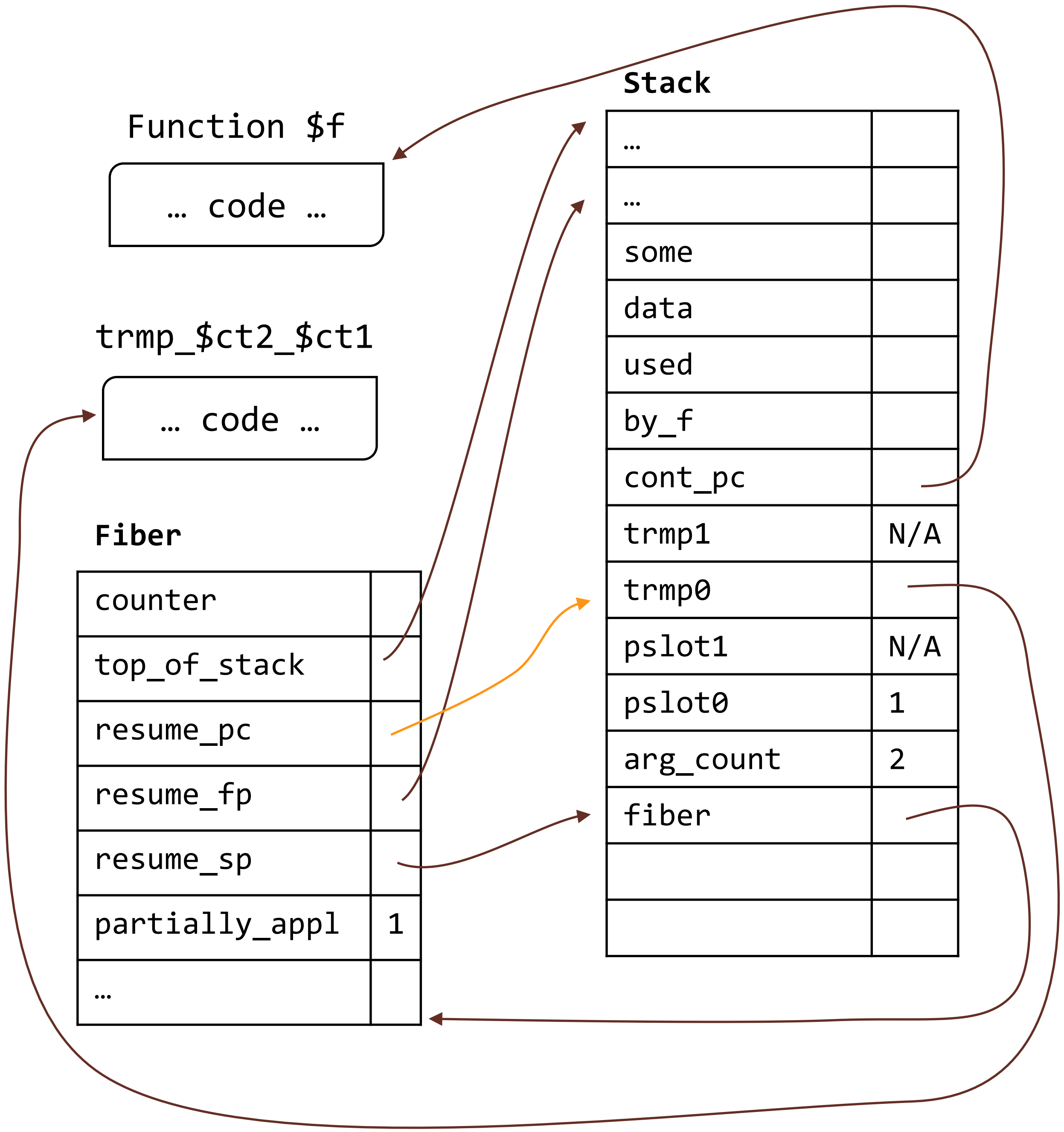
```
;; $ct2 ~~ cont [i64 i32] -> []
;; $ct1 ~~ cont [    i32] -> []


;; $c2 not created by cont.bind

(cont.bind $ct2 $ct1
  (i64.const 1)
  (local.get $c2))
(local.set $c1)

...

(resume $ct1 (i32.const 2) (local.get $c1)
```



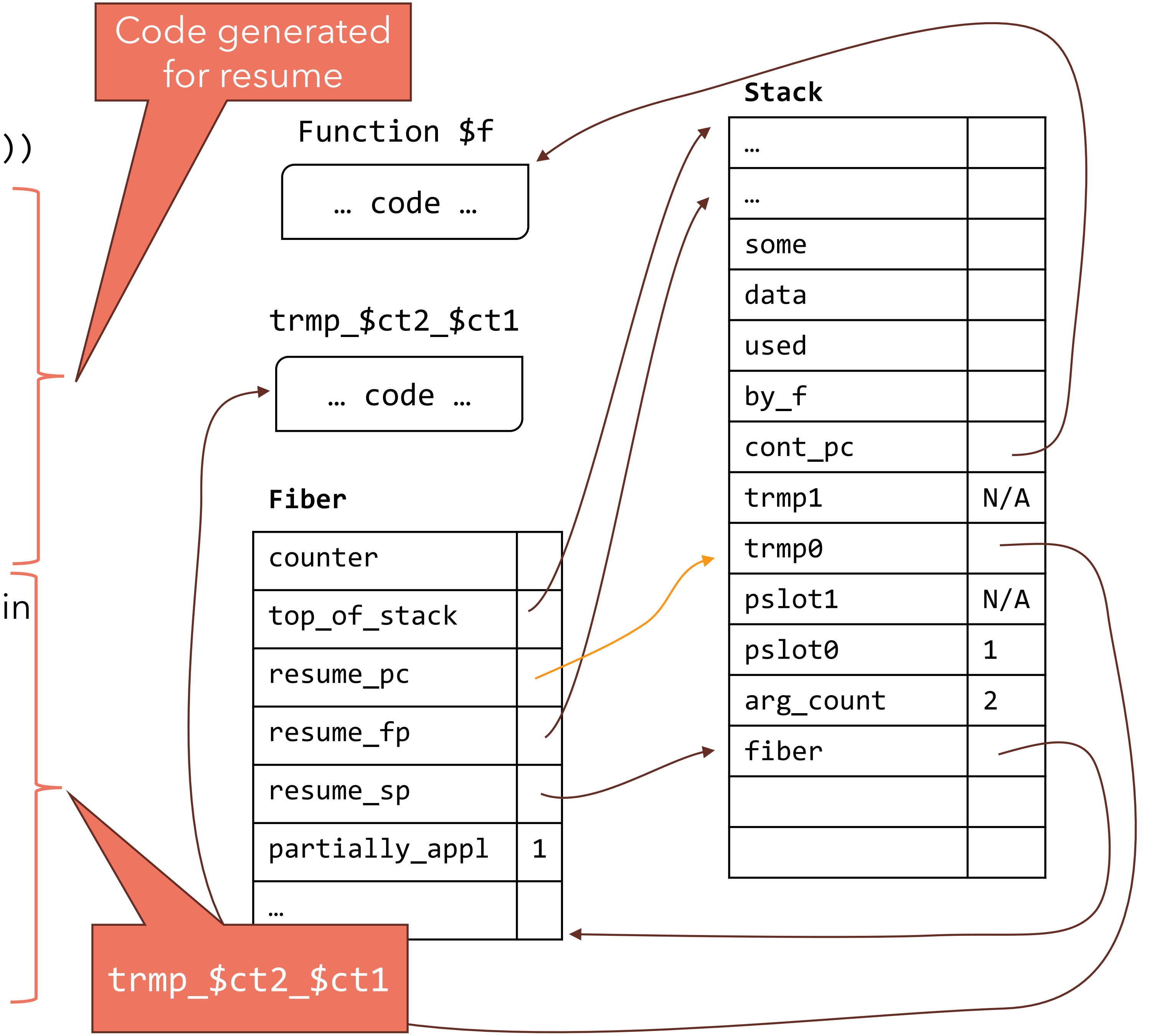
Fiber	
counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
partially_appl	1
…	

Stack	
…	
…	
some	
data	
used	
by_f	
cont_pc	
trmp1	N/A
trmp0	
pslot1	N/A
pslot0	1
arg_count	2
fiber	

Example (3)

```
(resume $ct1 (i32.const 2) (local.get $c1))
```

- Update stack chain
- Save current state (GRPs, SP, FP, IP to resume to)
- Put resume payload (`i32 2`) into first argument register (e.g. RDI)
- SP := fiber.resume_sp
- FP := fiber.resume_fp
- JMP fiber.resume_pc
- Receive previous arg (`i32 2`) as argument in RDI
- load `cont.bind` argument (`i64 1`) from `pslot0`
- Due to no subsequent cont.bind:
  - Remove `cont.bind` area from stack
  - Set up combined arguments: RDI := 1; RSI := 2
  - JMP cont_pc



Code generated for resume

trmp_$ct2_$ct1

Function $f

```
… code …
```

trmp_$ct2_$ct1

```
… code …
```

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
partially_appl	1
…	

Stack

…	
…	
some	
data	
used	
by_f	
cont_pc	
trmp1	N/A
trmp0	
pslot1	N/A
pslot0	1
arg_count	2
fiber	

Trampolines vs cont.bind code

```
(cont.bind $ct_before $ct_after):
$ct_before = cont [t1* t2*] -> [t3*]
$ct_after  = cont [    t2*] -> [t3*]
```

- At `cont.bind`:
  - Potentially: Do some setup on target stack
  - Write values of types `t1*` to memory

- Inside the trampoline
  (specifically compiled for that combination of types):
  - Receive all previous arguments (types `t2*`) as parameters
  - Load `cont.bind` arguments (types `t1*)` from memory
  - Potentially: clean up stack
  - Pass combined sequence (types `t1* t2*`) as arguments on to next stage (continuation or another trampoline) when `JMP`-ing there

Handling stack-passed arguments

Not shown before: Trampolines handling arguments that must eventually be passed on in stack slots per CC

Trampoline for `(cont.bind $ct_before $ct_after)`, where

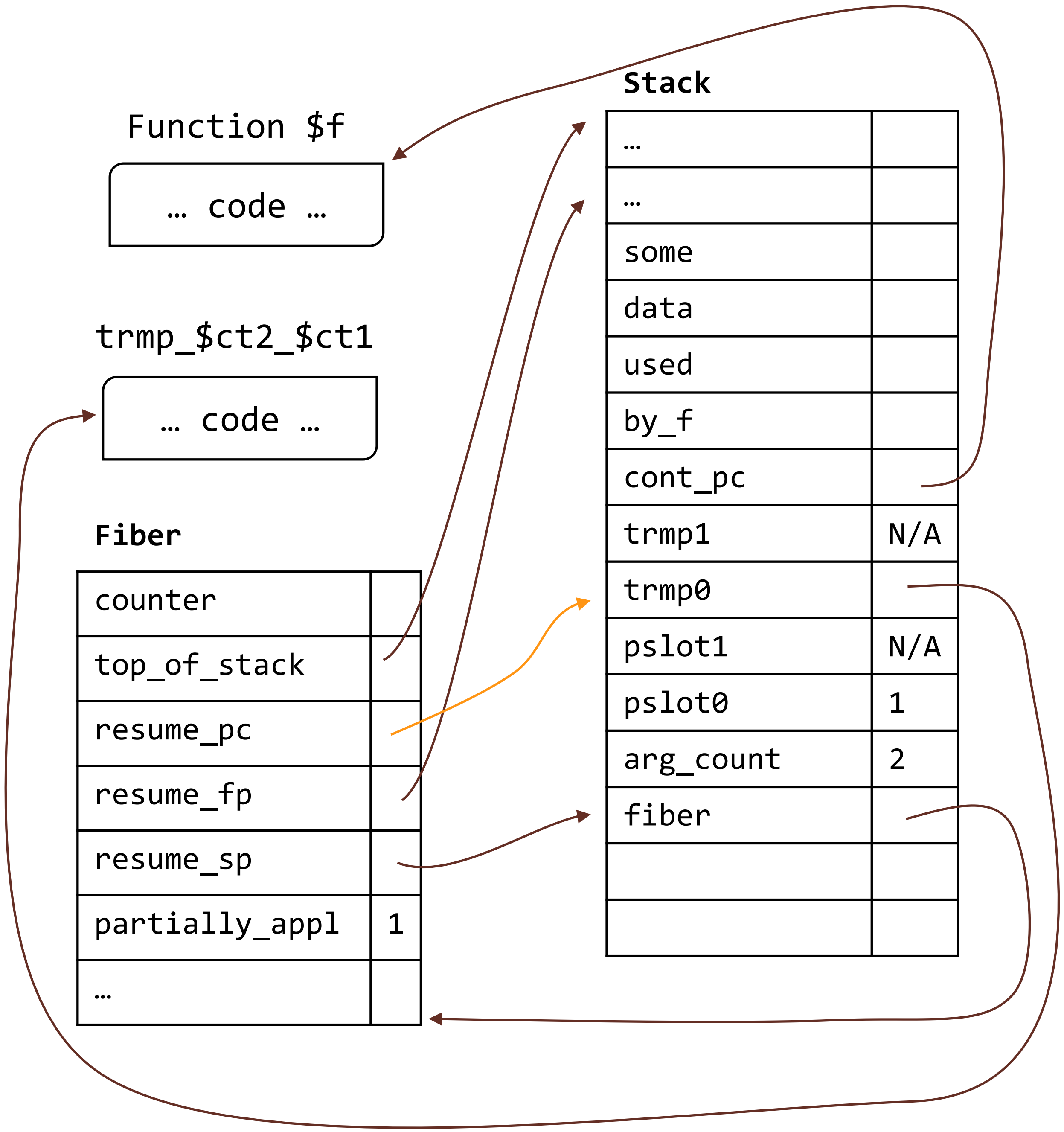
```
$ct_before = cont [t1* t2*] -> [t3*]
$ct_after  = cont [    t2*] -> [t3*]
```

Idea: If `t1* t2*` exceeds what can be passed in registers, start filling stack slots "backwards" starting from end of `t1* t2*`

Just from `t1* t2*`, the trampoline statically knows how many stack slots have already been filled

What about GC? (1)

- Observation: When encountering a continuation, GC must scan any `cont.bind` arguments
- Assumption: Scanning continuations generally works as follows
  - Obtain initial stack map using `fiber.resume_pc`, valid stack segment starting at `fiber.resume_sp`
  - Then walk continuation's stack and its parents' stacks
- What if `fiber.resume_pc` points to a `cont.bind` trampoline?



Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
partially_appl	1
…	

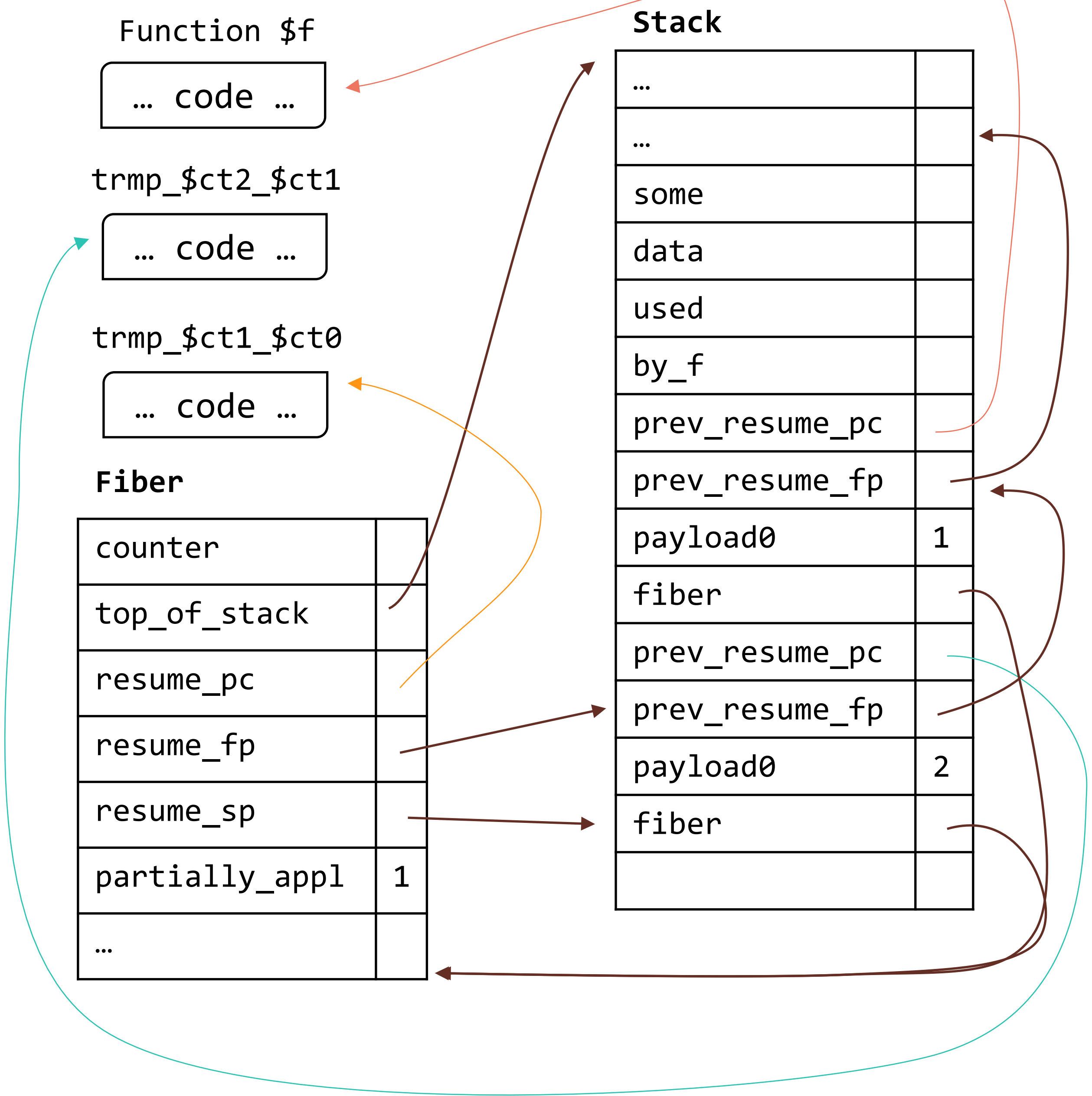
Stack

…	
…	
some	
data	
used	
by_f	
cont_pc	
trmp1	N/A
trmp0	
pslot1	N/A
pslot0	1
arg_count	2
fiber	

What about GC? (2)

- What if `fiber.resume_pc` points to a `cont.bind` trampoline?
- If `cont.bind` can only be applied once: Layout similar to what's shown previously should work
- Otherwise: Make each `cont.bind` add a pseudo stack frame?
  - GC traverses these as always
  - Trampolines are aware of their special nature (e.g., don't `RET` from their frame)

```
(i32.const 2) ;; second cont.bind
(i32.const 1) ;; first cont.bind
(local.get $c)
(cont.bind $ct2 $ct1)
(cont.bind $ct1 $ct0)
```



Function $f

… code …

trmp_$ct2_$ct1

… code …

trmp_$ct1_$ct0

… code …

Fiber

counter	
top_of_stack	
resume_pc	
resume_fp	
resume_sp	
partially_appl	1
…	

Stack

…	
…	
some	
data	
used	
by_f	
prev_resume_pc	
prev_resume_fp	
payload0	1
fiber	
prev_resume_pc	
prev_resume_fp	
payload0	2
fiber	

Right-to-left binding (1)

- So far: `cont.bind` works left-to-right
- What happens if it worked right-to-left?

Left-to-right

```
;; $ct2 ~~ cont [i32 i64] -> []
;; $ct1 ~~ cont [    i64] -> []
;; $ct0 ~~ cont [       ] -> []

(i64.const 2) ;; resume arg
(i32.const 1) ;; cont.bind arg
(local.get c) ;; cont.bind cont
(cont.bind $ct2 $ct1)
(resume $ct1)
```

Right-to-left

```
;; $ct2 ~~ cont [i64 i32] -> []
;; $ct1 ~~ cont [i64    ] -> []
;; $ct0 ~~ cont [       ] -> []

(i64.const 2) ;; resume arg
(i32.const 1) ;; cont.bind arg
(local.get c) ;; cont.bind cont
(cont.bind $ct2 $ct1)
(resume $ct1)
```

Right-to-left binding (2)

- So far: `cont.bind` works left-to-right
- What happens if it worked right-to-left?

Left-to-right

```
;; $ct2 ~~ cont [i32 i64] -> []
;; $ct1 ~~ cont [    i64] -> []
;; $ct0 ~~ cont [       ] -> []

trmp_$ct2_$ct1(prev0 : i64 @ RDI) {
  Load next0 : i32 from memory
  if (is_last_trampoline) {
    RDI := next0
    RSI := prev0
    clean up stack, set RSP, RBP
    JMP continuation_pc
  } else { … }
}
```

Register reshuffle here

Right-to-left

```
;; $ct2 ~~ cont [i64 i32] -> []
;; $ct1 ~~ cont [i64    ] -> []
;; $ct0 ~~ cont [       ] -> []

trmp_$ct2_$ct1(prev0 : i64 @ RDI) {
  Load next0 : i32 from memory
  if (is_last_trampoline) {
    RDI := prev0
    RSI := next0
    clean up stack, set RSP, RBP
    JMP continuation_pc
  } else { … }
}
```

Right-to-left binding (3)

- So far: `cont.bind` works left-to-right
- What happens if it worked right-to-left?

Left-to-right

Filling stack slots:

- Arguments that will eventually be passed in stack slots are passed between trampolines until accumulated arguments exceed what can be passed in registers
- Only then written to corresponding stack slot

Right-to-left

Filling stack slots:

- Each `cont.bind` trampoline statically knows whether a given argument will need to go into stack slot end up in a stack slot, puts it there

This only affects situations where we pass many arguments (e.g. > 6 ints on System V) and used `cont.bind` more than once

Summary (1)

- We want to pass payloads to continuations using same CC as for functions
- Trampoline-based approach can achieve the following:
  - Accumulate payloads previously supplied with `cont.bind`, eventually pass to continuation using chosen CC (in registers & stack slots)
  - **Zero overhead on switch/resume if no `cont.bind` happened!** All runtime overhead lives in code generated for `cont.bind`

- Memory cost:
  - In absence of any `cont.bind`: Potentially a single flag in `Fiber`
  - Per `cont.bind`:
    - At runtime: Occupy stack space
    - Code size: Need to compile type-specific trampolines

Summary (2)

- Two potential approaches towards laying out stack space used by cont.bind

1. “One big chunk”:
    - Shared by all trampolines
    - Created by first cont.bind instruction
    - Removed by last trampoline before handing over to continuation
    - More difficult to make work with GC if we allow more than one cont.bind to occur
    - Makes it easier to share state between trampolines
2. “Pseudo-frames per trampoline”
    - Each cont.bind instruction pushes its own pseudo-frame
    - Each cot.bind trampoline removes its pseudo frame
    - Easier to integrate with GC in presence of multiple cont.bind: The GC just sees a valid sequence of frames with corresponding stack maps.
    - … haven’t thought trough all the details

- Somewhat open question: Which of the two potential memory layouts to use?
    - Make sure that zero overhead introduced into GC

Thank you