Component Model Async Support

WebAssembly CG

May/June, 2022

Outline

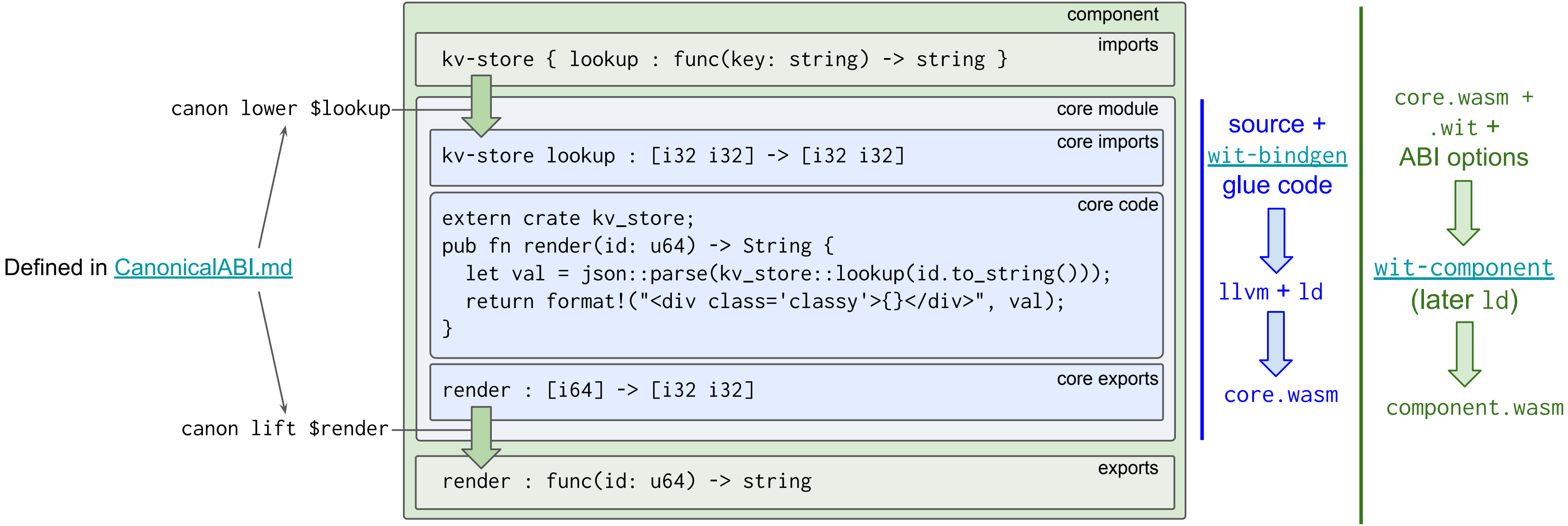
- Motivation
- Background: synchronous Canonical ABI
- Async support
  - `future`
  - Optimization: callback ABI
  - Optimization: eager return
  - Optimization: `stream`
  - Optimization: splicing and skipping streams
- Structured concurrency
  - Task
  - Task tree
  - Task cancellation
  - Task scheduling
- Core WebAssembly stack-switching integration

Caveat: still in flux; feedback welcome

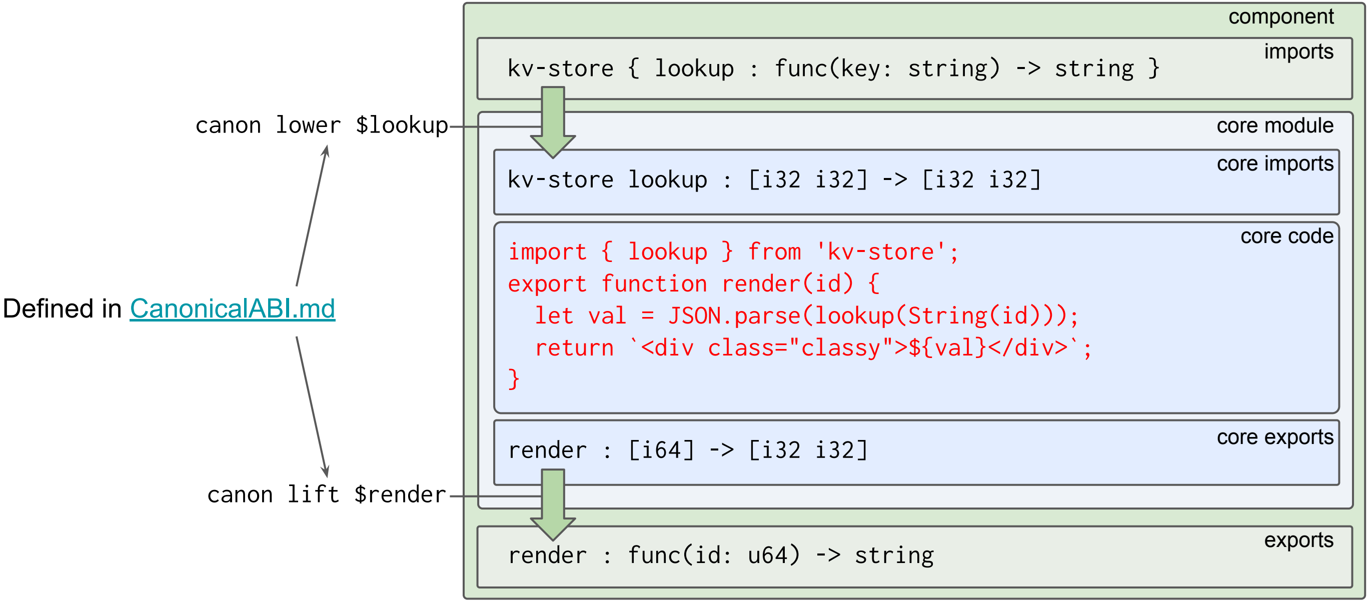
Motivation

- (One slide recap of previously-presented[1,2,3])
- How do we specify async/non-blocking operations in WASI and `wit`?
- Can’t we just add first-class functions / callbacks to `wit`?
  - Cyclic leak problems in non-GC setting (see: Web APIs)
  - Very low-level -- requires manual per-API wrapping to integrate with language concurrency
- Requirements/goals:
  - Virtualizability: async interfaces can be implemented by the host or wasm
  - Efficient I/O implementation when the “other side” is the host (e.g., epoll, io_uring)
  - Ergonomic automatic (`wit-bindgen`) language bindings
  - Support different styles of language-level concurrency (sync, non-blocking, async, coroutine)
  - Built-in backpressure story (not left as an exercise to the developer)
  - Integrated select / timeout / cancellation across independent interfaces (WASI and host-defined)
  - Ability to keep executing after returning a final value
  - “Just because I want async + modularity doesn’t mean I want multi-threading”

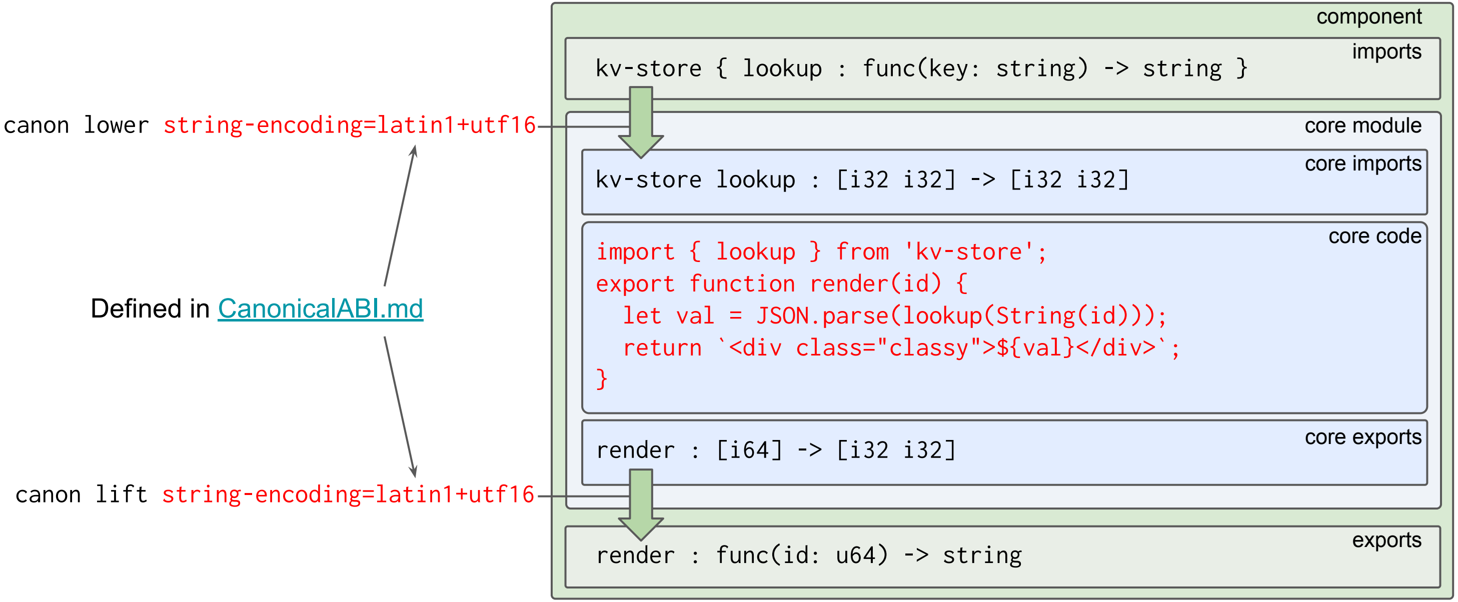
Background: synchronous canonical ABI



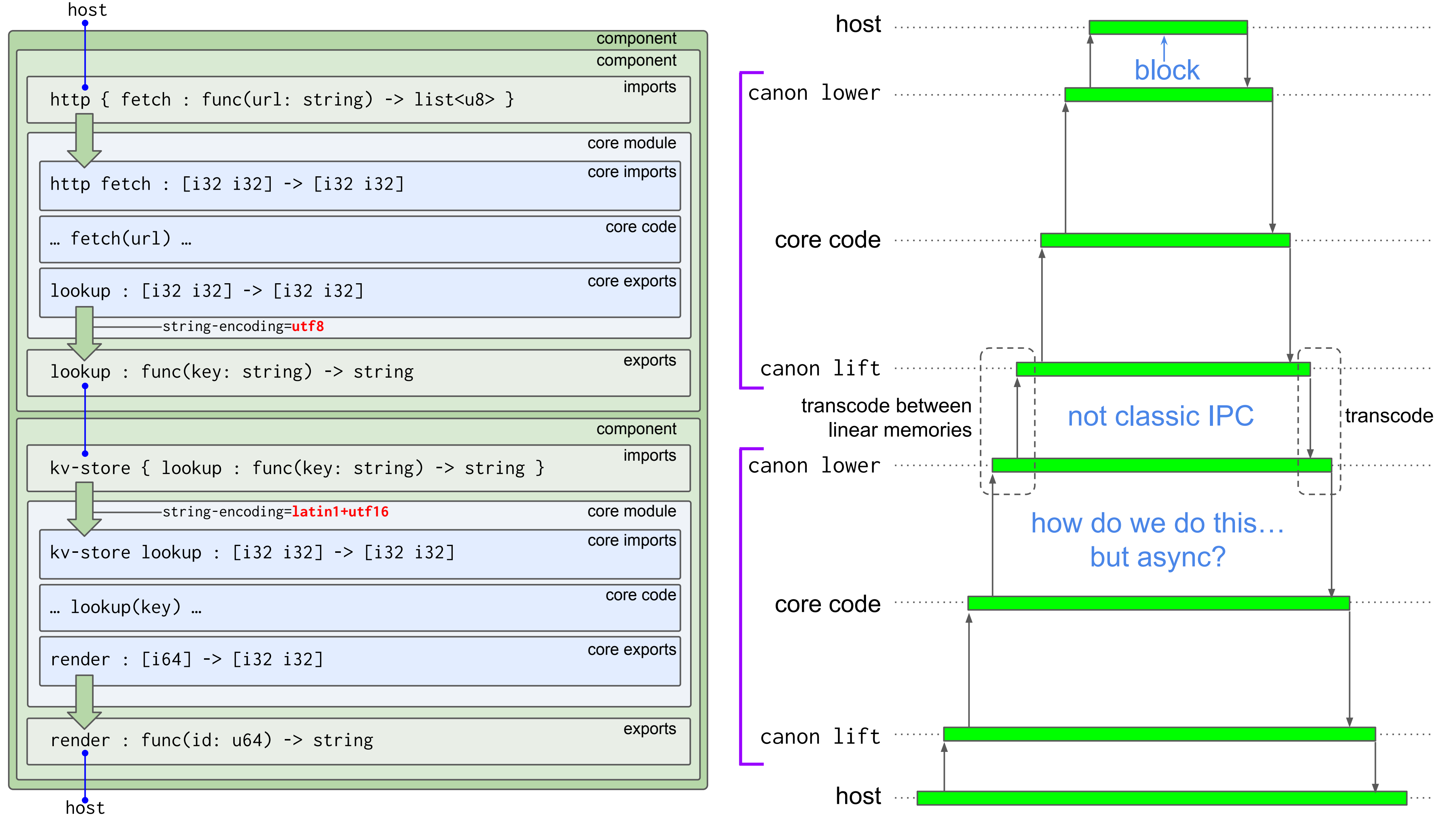
Background: synchronous canonical ABI



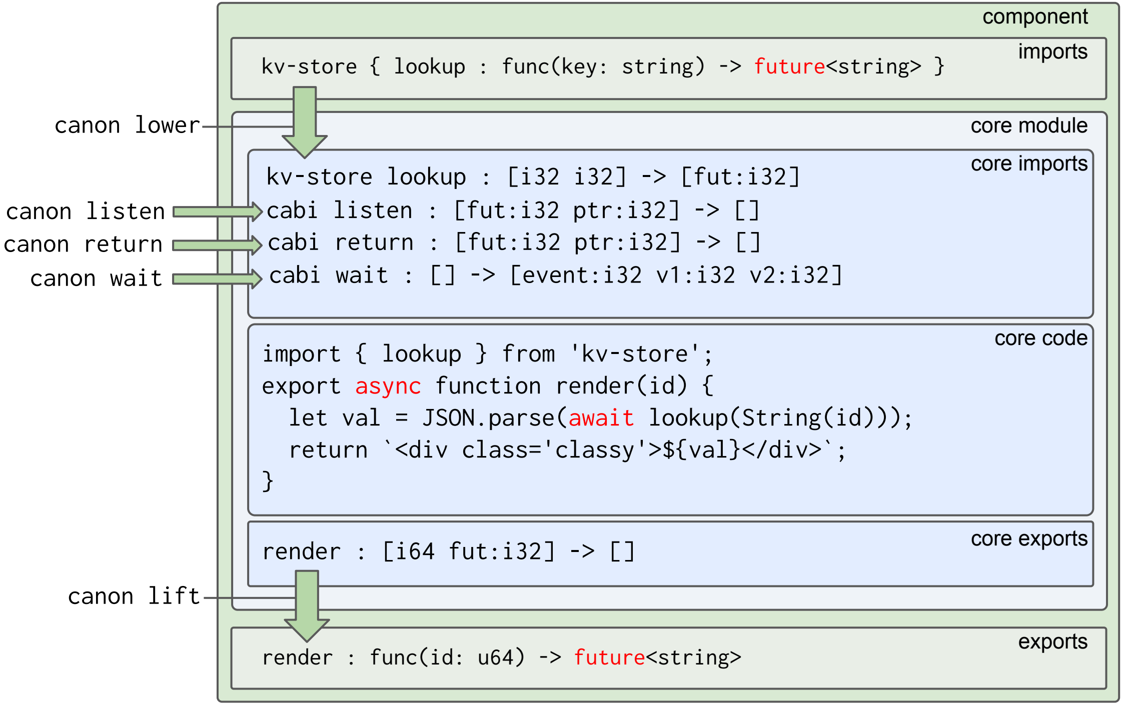
Canonical ABI options



Possible because `canon lift` and `lower` bracket all component entry/exit


host
component
component
imports
http { fetch : func(url: string) -> list<u8> }
core module
core imports
http fetch : [i32 i32] -> [i32 i32]
core code
… fetch(url) …
core exports
lookup : [i32 i32] -> [i32 i32]
string-encoding=utf8
exports
lookup : func(key: string) -> string
component
imports
kv-store { lookup : func(key: string) -> string }
string-encoding=latin1+utf16
core module
core imports
kv-store lookup : [i32 i32] -> [i32 i32]
core code
… lookup(key) …
core exports
render : [i64] -> [i32 i32]
exports
render : func(id: u64) -> string
host
host
block
canon lower
core code
canon lift
transcode between
linear memories
not classic IPC
transcode
canon lower
how do we do this…
but async?
core code
canon lift
host

future



`canon lift`:

- Passed the index of the future for this export call
- The callee must call `return` then `wait` for `return-complete`

`canon return`:

- Non-blocking: *offers* a `T` return value for the given future
- `ptr` must stay valid until the `return-complete` event
- Traps if given the result of `canon lower`

`canon wait`:

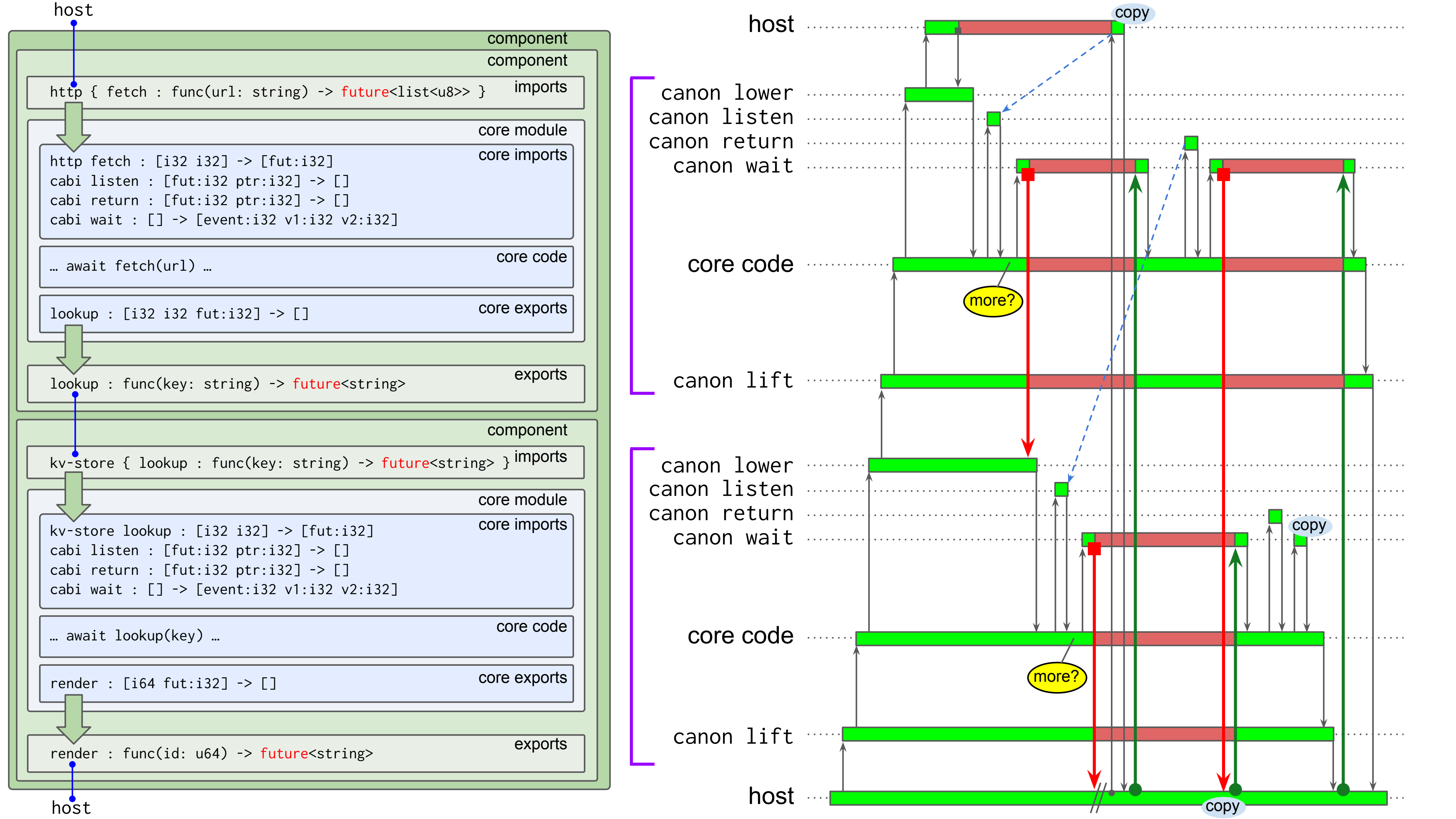
- Blocks until *some* `event` occurs, including:
  - `return-complete` (`v1` is the future index)
  - `returned` (`v1` is the future index)

`canon lower`:

- Returns the index of the future for this import call
- The future is initially in a “not listening” state.

`canon listen`:

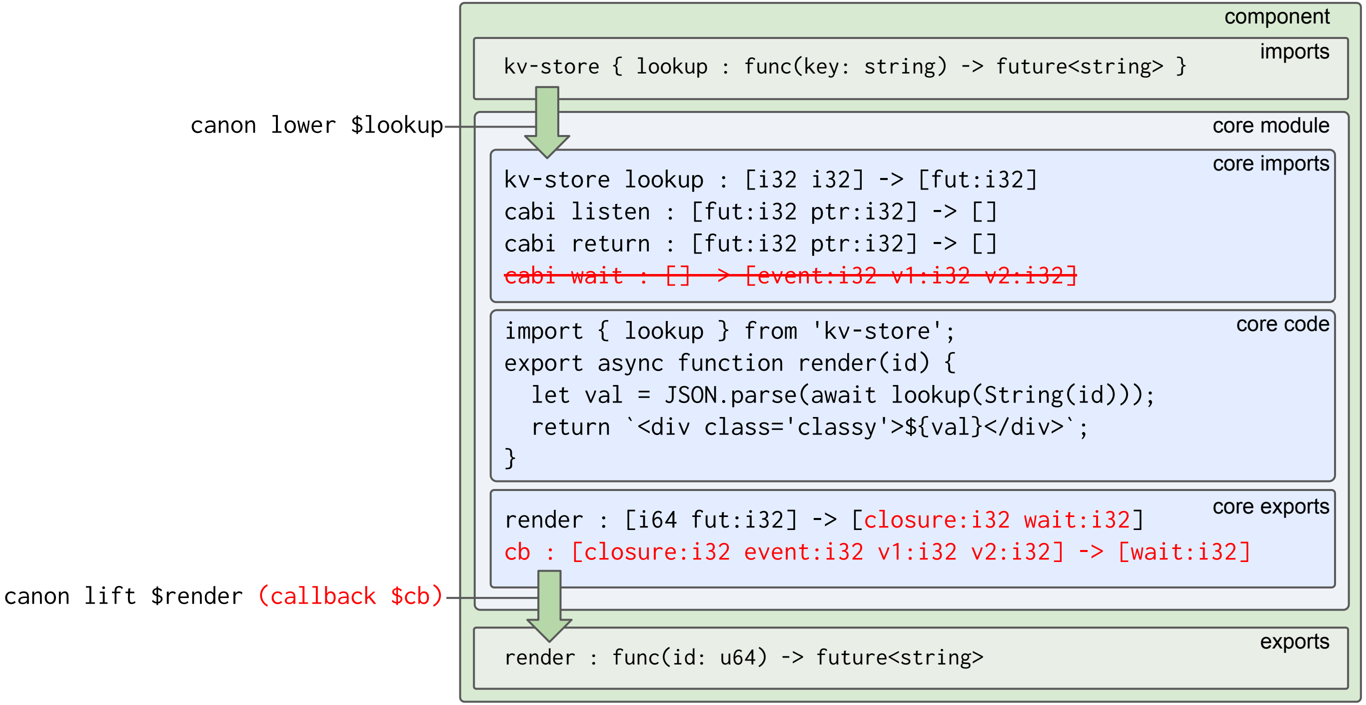
- Non-blocking: *offers* a buffer to receive the future’s value
- `ptr` must stay valid until the `return` event.
- Traps if given the parameter from `canon lift`


host
component
component
imports
http { fetch : func(url: string) -> future<list<u8>> }
core module
core imports
http fetch : [i32 i32] -> [fut:i32]
cabi listen : [fut:i32 ptr:i32] -> []
cabi return : [fut:i32 ptr:i32] -> []
cabi wait : [] -> [event:i32 v1:i32 v2:i32]
core code
… await fetch(url) …
core exports
lookup : [i32 i32 fut:i32] -> []
exports
lookup : func(key: string) -> future<string>
component
imports
kv-store { lookup : func(key: string) -> future<string> }
core module
core imports
kv-store lookup : [i32 i32] -> [fut:i32]
cabi listen : [fut:i32 ptr:i32] -> []
cabi return : [fut:i32 ptr:i32] -> []
cabi wait : [] -> [event:i32 v1:i32 v2:i32]
core code
… await lookup(key) …
core exports
render : [i64 fut:i32] -> []
exports
render : func(id: u64) -> future<string>
host
host
copy
canon lower
canon listen
canon return
canon wait
core code
more?
canon lift
canon lower
canon listen
canon return
canon wait
copy
core code
more?
canon lift
host
copy

Optimization: callback ABI

- For the future/promise/task/async+await family of languages…
  - viz., .NET, JS, Rust
- … `wait` will always be performed at the base of the callstack
  - As part of a runtime-owned event loop designed to integrate with OS event system
- In this setting, full stack-switching is overkill
  - The language compiler/runtime already did all the "hard work" of clearing the native stack
- It would be nice to allow producer toolchains to opt out of stack switching
  - Reap the performance benefits paid for by their async model
- Also, some hosts won't support native stack switching for a while (or ever?)
  - Stack switching can be emulated/polyfilled via asyncify, but it's expensive

Optimization: callback ABI

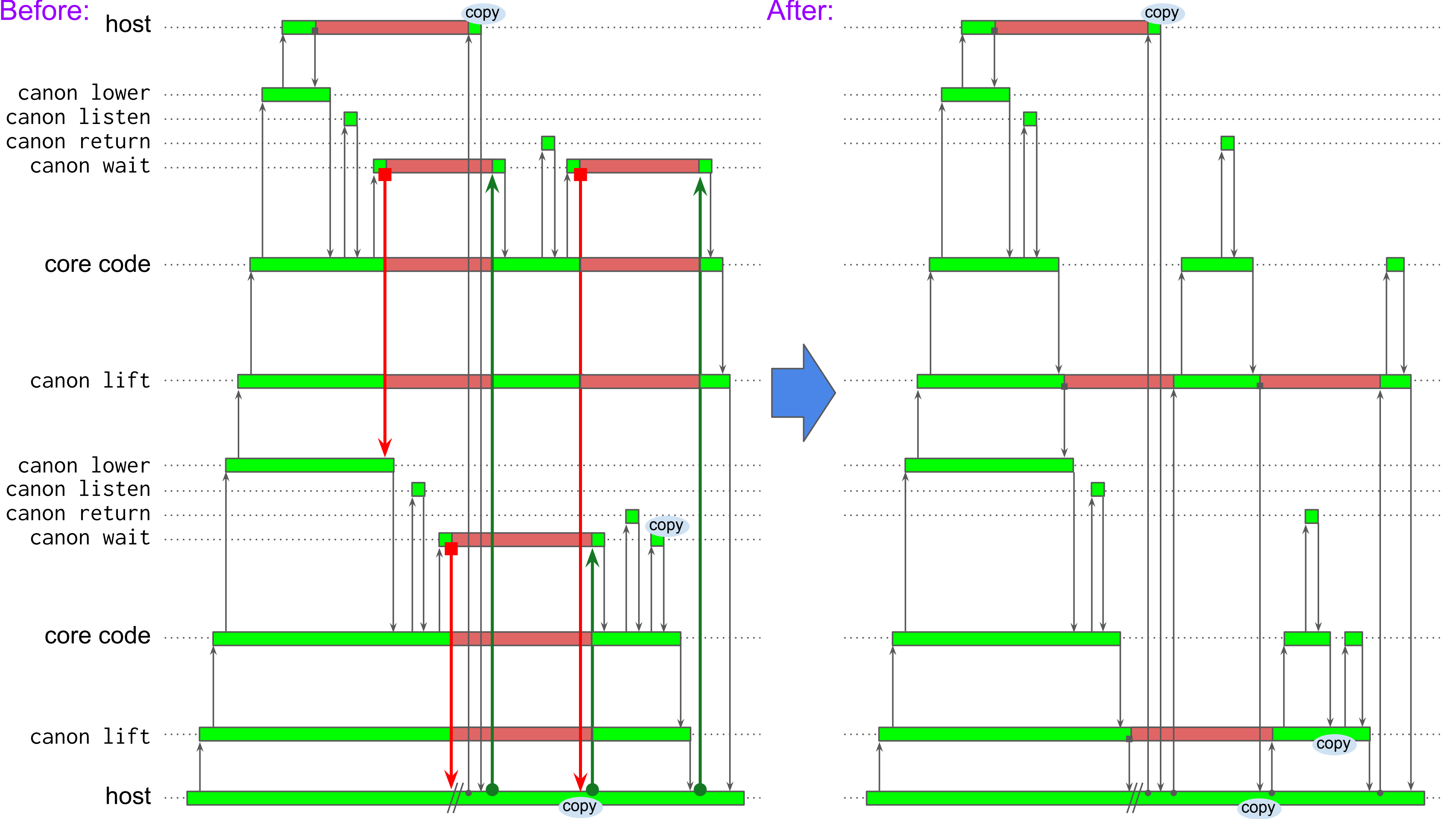


As if:

```
canon-lift (params) {
  let (c, wait) = render(params)
  while (wait) {
    let (event,v1,v2) = wait()
    wait = cb(c, event, v1, v2)
  }
}
```

Notes:

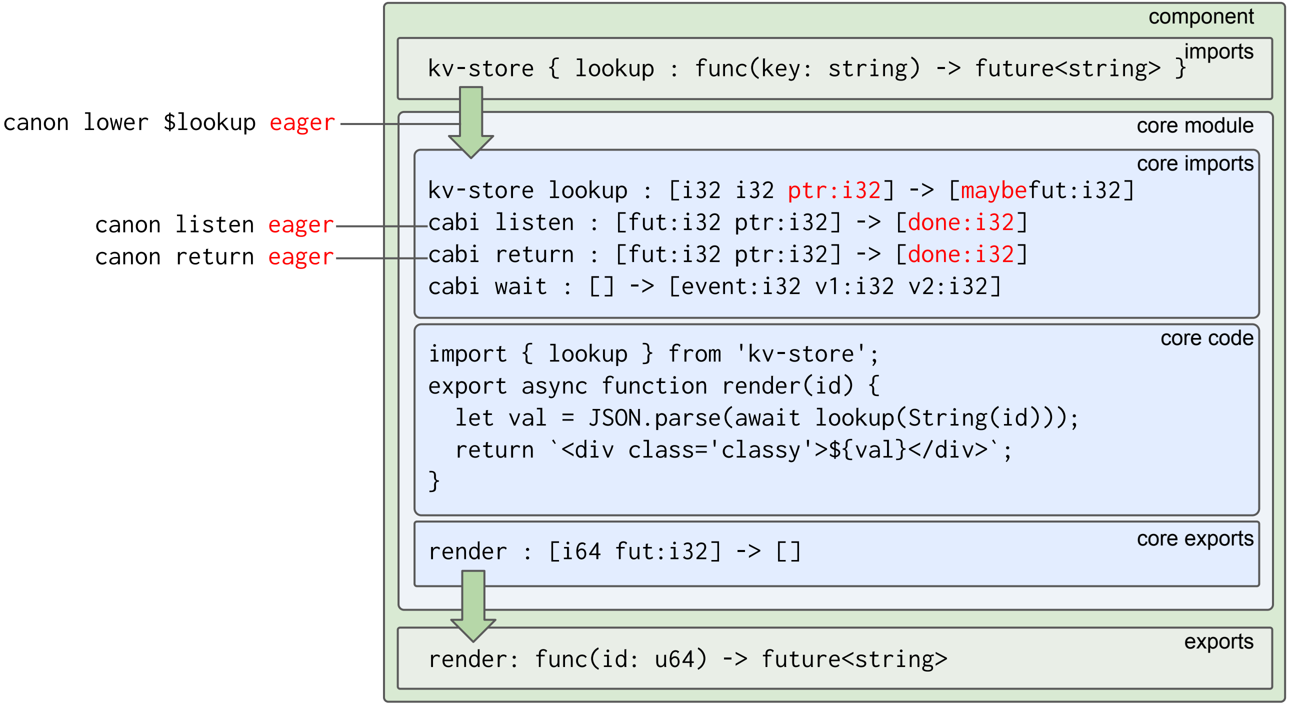
- Encapsulated impl. detail
- Composes with non-callback
- Calling `canon wait` traps

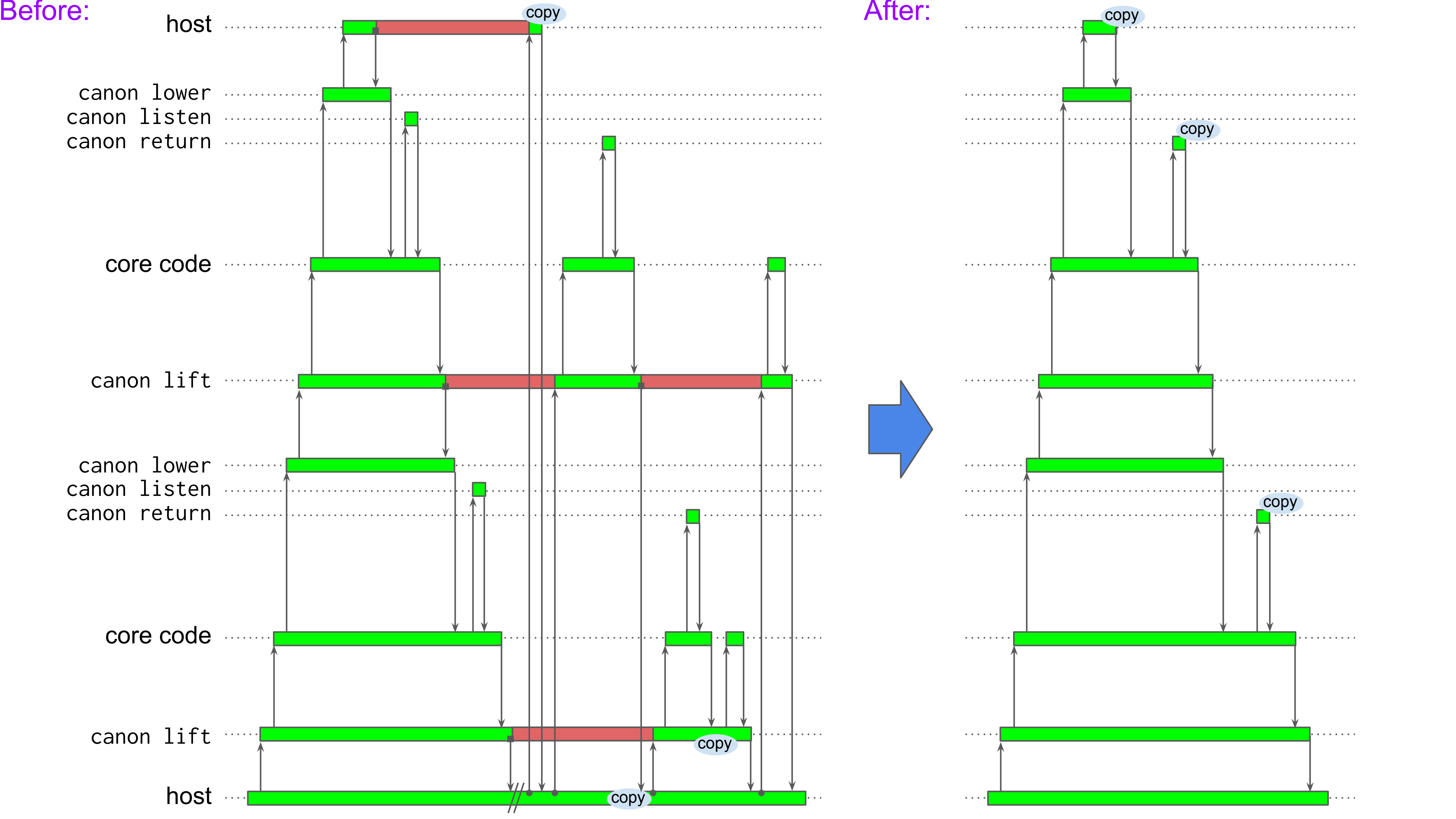

Before:
host
copy
canon lower
canon listen
canon return
canon wait
core code
canon lift
canon lower
canon listen
canon return
canon wait
copy
core code
canon lift
host
copy
After:
copy
copy
copy

Optimization: eager return

- If the result is already available, `future` adds overhead
  - Runtime internal allocations, extra listen / wait calls.
- Some languages allow promises/futures to be returned already-resolved
  - Avoiding a trip through the event loop

Optimization: eager return



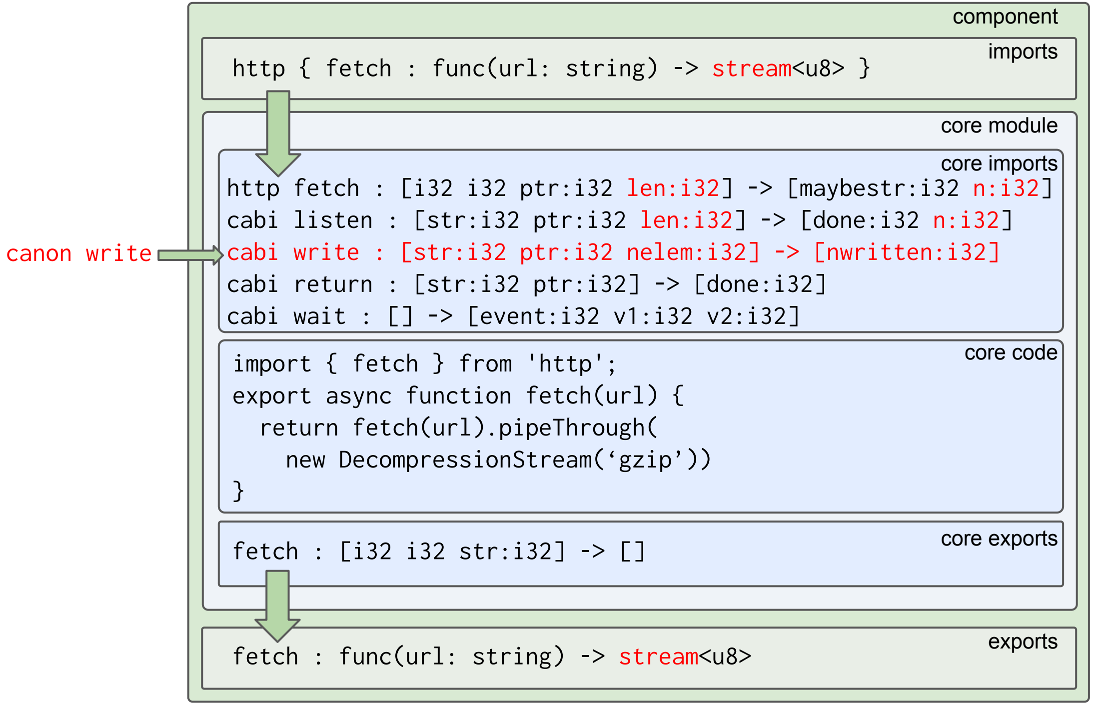


Before:
host
copy
canon lower
canon listen
canon return
core code
canon lift
canon lower
canon listen
canon return
core code
canon lift
copy
host
copy
After:
copy
copy
copy

Optimization: `stream`

- Streams are possible with `stream<T> = future<option<pair<T,stream<T>>>>`
  - (Hand-waving over how we make `stream<T>` recursive…)
- But that's not going to cut it for streams of bytes
  - Need: bulk copies, directly between linear memory (in component-to-component)
  - Don't want to create a completely separate `bytestream` (stream of `vec2` should be fast too!)
- Languages increasingly have a built-in stream primitives
  - Tightly integrated with the rest of the language (syntax, concurrency model, backpressure, …)
  - Want interface types to automatically bind to these stream language primitives.
- So define `stream<T>` as a new interface type constructor
  - Both as an optimization but also for improved language bindings
- Streams also sometimes have a "closing" value distinct from the elements
  - Effectively: `stream<T,U> = future<either<U,pair<T,stream<T,U>>>>`
  - E.g., `main: (stdin:stream<u8>, argv:list<string>) -> stream<u8,expected<_,_>>`
  - `stream<T> = stream<T,unit>`

Optimization: `stream`



`canon lower` (of function returning `stream<T,U>`):
- Additionally takes the byte-length of `ptr`
- `maybestr=0` means `ptr` holds `T*U`, `n = |T*|`.
- `maybestr≠0` means `ptr` holds `T*`, `n = |T*|`.

`canon listen` (on `stream<T,U>`):
- Requires `len > max(sizeof(T),sizeof(U))`
- `ptr` receives `T*` *xor* `U`; must stay valid until `written` event.
- `n=0` means “returned `U`” / `n>0` means “`n` `T`s written”

`canon write` (on `stream<T,U>`):
- Non-blocking: *offers* `nelem` `T` values for the given stream
- Requires `nelem > 0` (progress, symmetry with listen).
- `ptr` must stay valid until `write-complete`

`canon return` (on `stream<T,U>`):
- Traps if `write` in progress.
- Non-blocking: *offers* a `U` return value for the given stream
- Closes the stream (no more writes possible)

`canon wait`:
- Additional events:
  - `write-complete` (`v1` is stream index, `v2` is num written)
  - `written` (`v1` is stream index, `v2` is num written)
    - The stream goes back to the “not listening” state
    - Not listening = **backpressure**

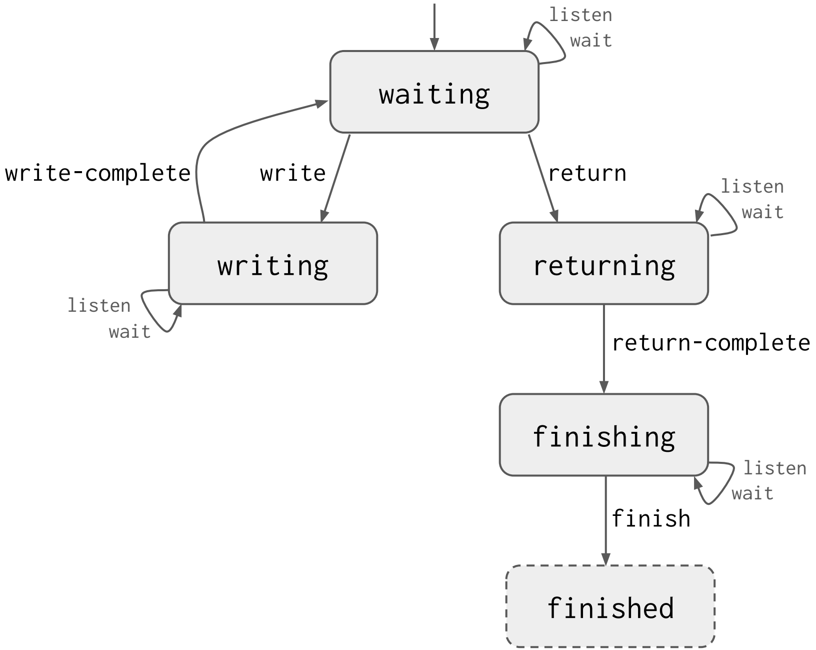
Optimization: splicing and skipping streams

- It's very common to copy big chunks from one stream to another
  - Don't want to have to read into linear memory just to immediately write back out.
- `canon splice : [src-str:i32 dst-str:i32 nelem:i32] -> [done:i32]`
  - Acts like `listen(src-str, buf)` + `write(dst-str, buf)`, but without the `buf`.
  - If `done=0`, must wait for a `write-complete` event (`nwritten <= nelem`).

- `canon forward : [src-str:i32 dst-str:i32] -> []`
  - Like `splice(src-str, dst-str, ∞)`, but caller doesn't have/get to wait on the completion.
  - `dst-str` is immediately removed from table.
  - Also works on futures (analogous to JS rules when a `then()`-function returns a Promise).

- Sometimes we want to ignore a run of elements in a stream
  - Don't want to copy bytes into linear memory just to advance the read offset
- `canon skip : [str:i32 nelem:i32] -> [done:i32]`
  - Acts like `listen`: if `done=0`, must wait for a `written` event.

State machine



A `future<U>` is just a `stream<T,U>` that always writes zero `T`s before returning a `U`.

We can think of `future` and `stream` as two static descriptions of the *dynamic behavior* of a “**task**”...

Tasks

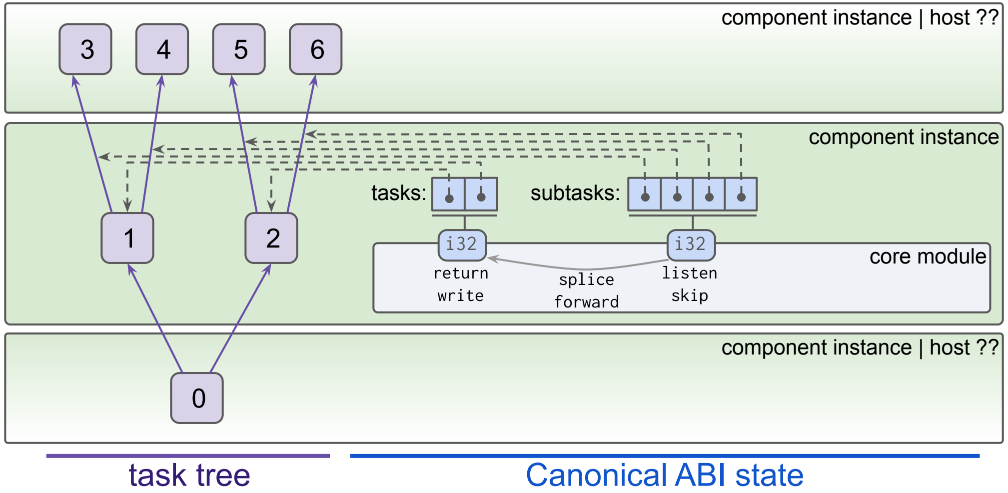
- A **task** is a stateful *resource* representing an asynchronous computation
  - … producing one value (`future`) or a sequence of values (`stream`).
- A task has a **producer state** and a **consumer state**
  - Producer state = { `waiting`, `writing(ptr,n,done?)`, `returning(ptr,done?)`, `finishing`, `finished` }
  - Consumer state = { `listening(ptr,len,done?)`, `not-listening` }
- `canon write`/`return` consult consumer state before updating producer state:
  - If consumer state is `listening`: do the copy, then eager return.
- `canon listen` consults producer state before updating the consumer state:
  - If producer state is `writing`/`returning`: do the copy, then eager return.
- The `done?` boolean sub-state says we still need to *notify* the task
  - … before transitioning back to `waiting`/`returning`/`not-listening`.
- Batch reads/writes by bumping `writing.ptr` / `listening.ptr`
  - … without setting `done?`, so that further reads/writes are possible with the same buffer.

Task trees

- Tasks form a **task tree** with edges from **super**tasks to **sub**tasks.
  - (The natural “parent”/“child” terminology is already used to describe instance nesting.)
- Because of tree-ness, subtasks are **uniquely owned** by their supertask.
  - However, subtask ownership can be **transferred** (as we’ll see next).
- Tasks can be **host-implemented** or **wasm-implemented**.
  - The task tree’s **root** is an ever-present host-implemented task (calling component exports).
  - All other host-implemented tasks are **leaves** (called by component imports).

Canonical ABI representation of tasks vs. subtasks

- The Canonical ABI defines 2 component-instance-local tables:
  - **Task table**: tasks implemented by the containing instance.
  - **Subtask table**: subtask edges whose sources are tasks in the task table.
- The tables are instance-wide (like linear memory)
  - So any core code can `listen`/`write`/`return` to any (sub)task any time.



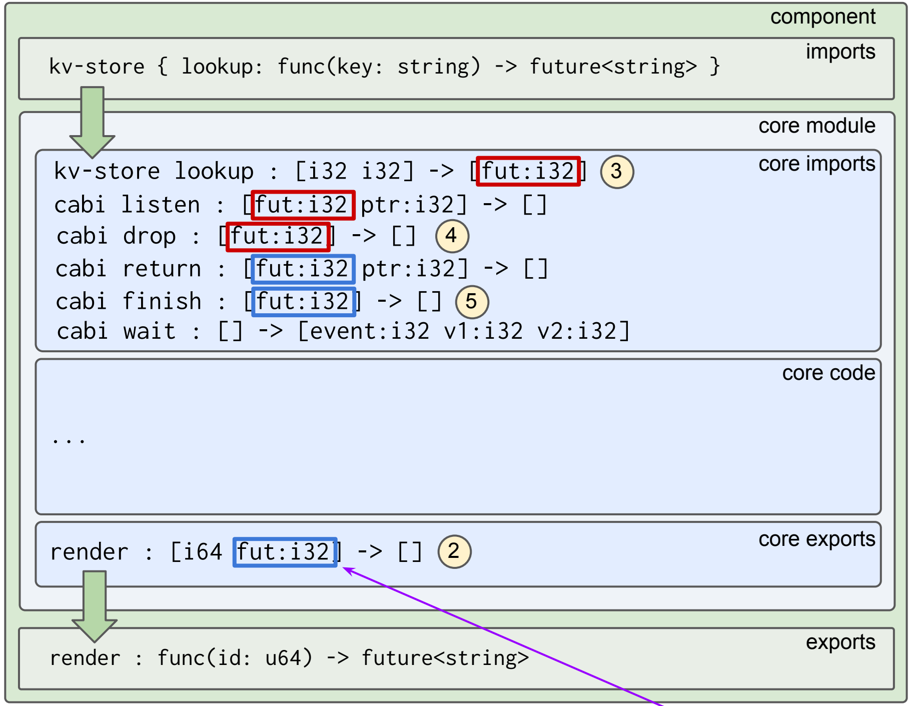
Structured concurrency

- A task tree is effectively the “async version” of a synchronous callstack
  - Asynchronicity means that new stack frames can be created before their sibling frames finish.
    - Hence “tree” not “stack”
    - The same tree structure also shows up when you have lexical closures (callbacks)...
- **Structured concurrency** means making the “async callstack” metaphor hold:
  - Invariant: **subtasks can’t outlive their supertasks**
  - … although the supertask can change over time via explicit ownership transfer (how next)
- Why is this useful?
  - Abstractly, it ensures async callees are an encapsulated implementation detail of async callers
    - … just like with sync calls; all we’re doing is allowing the calls to overlap (= concurrency).
  - Concretely, this enables:
    - Devtools / debugging
    - Tracing (in the “observability” sense)
    - A compositional recursive cancellation story (next next)
- How is this invariant achieved?

Structured concurrency

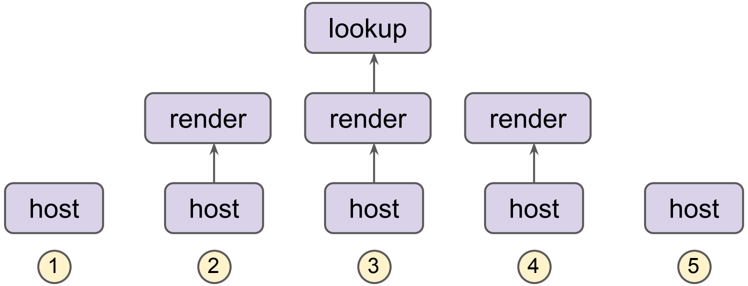
- How exactly does a task "finish"?
  - `canon finish : [task:i32] -> []` (`task` can be a future or stream)
  - **Traps if `task` has any remaining subtasks**
  - The supertask of a finished task receives a `finished` event from `canon wait`.
- When *precisely* is a subtask taken "off the books"?
  - `canon drop : [subtask:i32] -> []` (`subtask` can be a future or stream)
  - **Trap if `subtask` isn't `finished`**
  - Explicit `drop` lets the toolchain control when the task index may be recycled (like an OS handle).
- Putting these together: before calling `canon finish`, a task must…
  - `canon drop` each subtask, which requires…
    - `canon wait`-ing for `finished` from each subtask, which requires…
      - Those subtasks to themselves call `canon finish`, which requires … (recurse)

Original `future` example *redux*



Task table indices

Subtask table indices



Passing the export's returned future as an outparam is a special-case to allow eager return.

General case: dynamically creating tasks

component

core module

core imports

canon task.new $T

```
cabi task.new : [] -> [fut:i32 subtask:i32] 2 2 2
cabi return : [fut:i32 ptr:i32] -> [] 3 (array of subtask i32s)
cabi finish : [fut:i32] -> [] 4 5 5 5
cabi wait : [] -> [event:i32 v1:i32 v2:i32]
```

core code

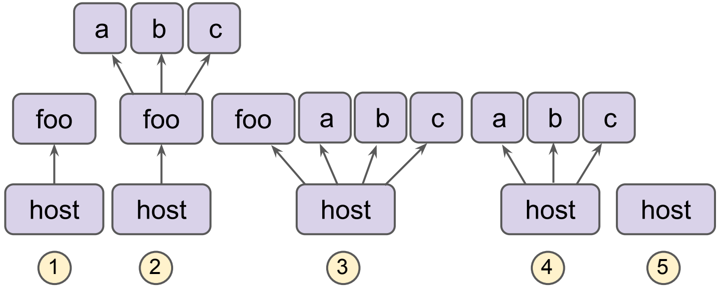
```
export async function foo() {
  return [‘a’, ‘b’, ‘c’].map(c => Promise.resolve(c))
}
```

core exports

```
foo : [fut:i32] -> [] 1
```

exports

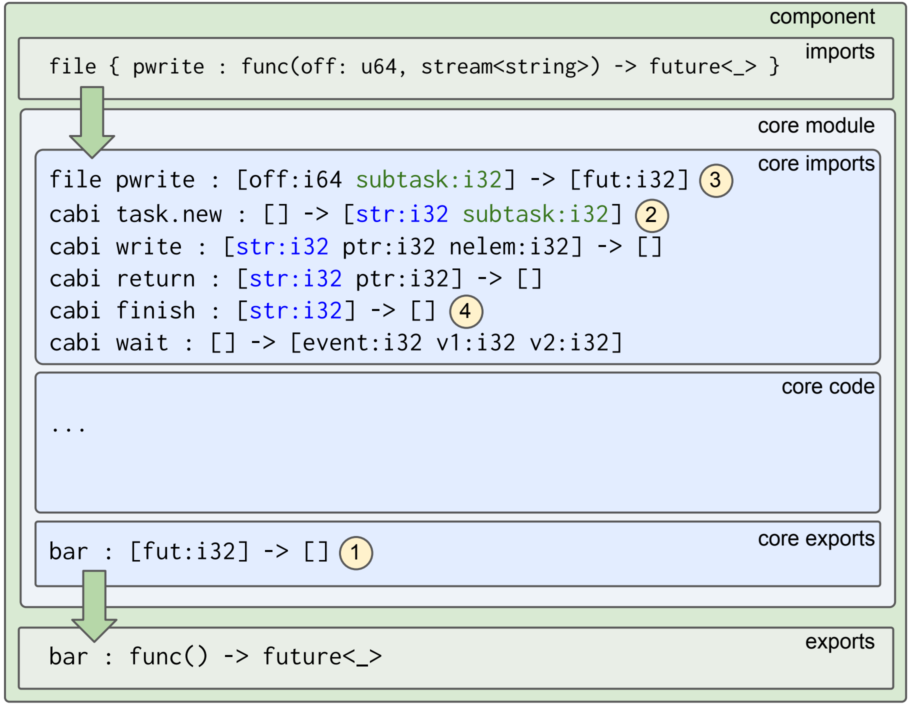
```
foo : func() -> future<list<future<string>>>
```

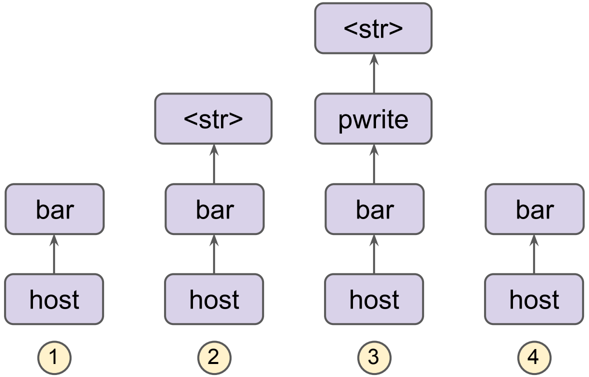


In general: `future<T>/stream<T,U>` can be *arbitrarily nested* in params *and* results

… passed by **ownership transfer** of subtasks from one task to another

Passing tasks as parameters





Note: `bar→pwrite→<str>` interleaves instances, but task tree ensures acyclicy (no leaks).

Passing tasks as parameters (passthrough)

component

imports

```
bar { baz : func(future<string>) -> future<string> }
```

core module

core imports

```
bar baz : [fut-in:i32] -> [fut-out:i32] ③
cabi listen : [fut:i32 ptr:i32] -> []
cabi return : [fut:i32 ptr:i32] -> []
cabi wait : [] -> [event:i32 v1:i32 v2:i32] ④
```

core code

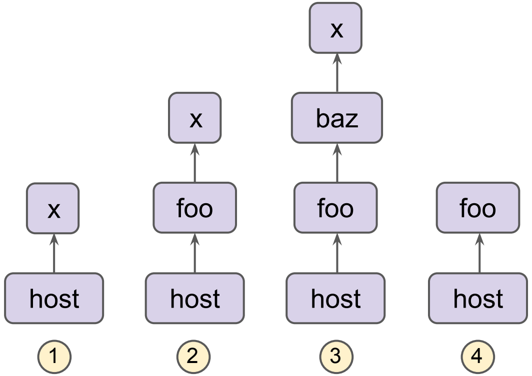
```
import { baz } from 'bar';
export async function foo(x) {
  let p = await baz(x);
  return ‘(‘ + p + ‘)’;
}
```

core exports

```
foo : [fut-in:i32 fut-out:i32] -> [] ②
```

exports

```
foo : func(future<string>) -> future<string>
```

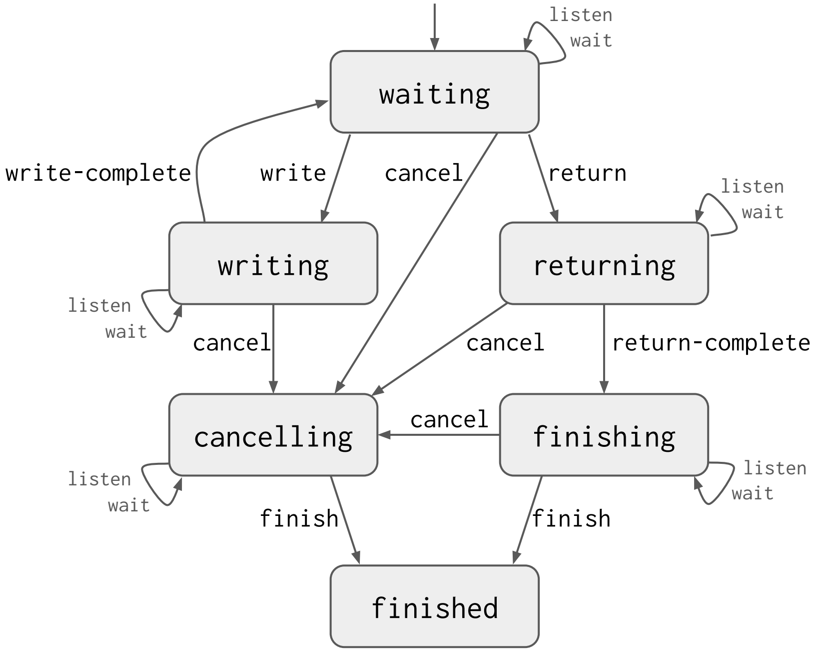


Ownership transfer allows passthrough in all directions

Task cancellation

- What if a supertask starts a subtask but loses interest?
  - E.g., race two network requests, one wins, want to “cancel” the other.
  - `canon cancel : [subtask:i32] -> []`
- Can `cancel` just straight-up *delete* the subtask? No:
  - Its containing instance (and linear memory) lives on, so this might leak / leave in an invalid state
  - Analogous to the usual problem of killing a thread without running destructors
- Can `cancel` *force* a subtask to wrap up “promptly”? No:
  - The subtask may legitimately need to perform some async work as part of its cancellation.
  - E.g., rolling back a transaction or posting some logs or metrics
- Thus, `cancel` must be *cooperative*:
  - Non-cooperative host/guest scenarios need a new “blast zone” feature *anyhow*
  - So: `canon cancel` just delivers a `cancelled` event to the subtask.
  - But the subtask can keep calling imports and waiting before calling `canon finish`.

State machine (with cancellation)



How does this look in the source language?

- JavaScript:
  - If the implementation GCs an unresolved Promise: call `canon cancel`
  - Eager cancellation via [`AbortController`] signal accepted by JS import bindings.
- Rust:
  - If a `Future`'s destructor is called before the future is resolved: call `canon cancel`
- Both: the language runtime implicitly waits for all subtasks to finish
  - … before finishing the `Future`/`Promise` returned by the export.
- But what if I want to *explicitly* wait for `finish` in my source language?
  - Usually I don't care, but I may in advanced scenarios.
  - TBD
  - Maybe the bindings could define a *subclass* of `Promise`/`Future` that exposes the `finish` event?

Core WebAssembly stack-switching integration

- [TODO]
- What happens when an async export is called while one is in-progress?
  - New stack *and* new task
- Rules ensure every stack implements at least one task
  - Created one first task (on export call)
  - Trap if wait after finish last task
- Same “stacks” as the Core WebAssembly stack-switching proposal
  - Core stack-switching adds: cooperative/green pthreads + coroutines
- Updated definition of “a task”:
  - Resource representing async computation producing one or multiple values
  - Mutable state:
    - Producer and consumer state
    - List of subtasks
    - Stack reference

[TODO: multi-async example]

[TODO: stack.new example]

Task scheduling

- The Component Model defines a scheduler loop executed by the host.
- The task tree serves as the scheduler state.
- Initially, the task tree contains a single root node representing the host.
- On each iteration, one of the following may happen (non-deterministically):
    - The host creates a new task to execute an async export (for whatever host-defined reason).
        - E.g., HTTP call, timer fired, UDF invoked, message arrived, …
    - An I/O operation completes, transitioning a host-defined leaf task producer state.
    - One or more values are copied from a subtask to a supertask, updating both tasks' state.
    - Notify a task of an event based on its task (producer|consumer) state.
        - "Notify" means resuming the task's stack's `wait` (at which the stack must be suspended).
    - If none of these apply, the loop blocks (waiting for I/O or a new export call to be triggered).
- Thus, we have *two-level* scheduling:
    - *Inter*-component: language-agnostic via the above scheduling loop.
    - *Intra*-component: language-specific as compiled by the language toolchain.
    - Not surprising: this is similar to OS processes, but without the separate threads.

Canonical ABI Summary

[TODO: complete list of canon definitions added/modified]

TODO

- `park/unpark`
  - How does one stack wait on another stack?
  - This allows deadlocks :-(
    - BUT, they can be reliably detected by the semantics so `park` returns a failure, not hangs.
    - Unavoidable if you can “tee” and “join” (due to resource exhaustion)
- `unlisten/unwrite/unreturn`
  - If the guest code needs to synchronously deallocate a buffer passed to `listen/write/return`
  - … may conflict with efficient (io_uring) I/O host implementation?
- Async-to-sync adapters
  - A sync import can be implemented by an async function if the caller isn’t reentered.
  - Component non-reentrance invariants already enforce this.

Summary

- Proposing a common set of high-level ***concurrency types***
  - The types prescribe a low-level control-flow *protocol* between the two sides of an async call
  - The runtime mediates and enforces the protocol via dynamic checks and the scheduler.
- Should be “bindable” into different languages’ native concurrency support
  - Ergonomically usable without manual hand-written *per-interface* glue code
  - Turning an O(N×M) situation into an O(N+M) situation (N = interfaces, M = languages)
  - (Which is the general goal of the Component Model.)
- Interestingly: **not a “process”/“thread” model** (e.g., CSP, π calculus)
  - No (preemptively-scheduled) threads (instead cooperative scheduling/stack switching)
  - No channels, pipes, message-boxes (instead direct copy + buffering in the wasm + backpressure)
- … but could complement a process/thread model
  - Can compile process-style languages to run ***inside*** a component
  - Can instantiate components ***inside*** the processes of a process model

Next steps

- Use `stream`/`future` in WASI snapshot preview2
  - Just the relevant subset (`lower`+`listen`+`wait`)
  - Using the Canonical ABI to define as a pure Core WebAssembly interface…
  - … so not dependent on the Component Model
- Write up in a PR to the component-model repo
  - Extend the explainer (AST), binary format and CanonicalABI.md
  - Get implementation feedback
- Working with Lucy Amidon and Amal Ahmed on formal semantics
  - Rough idea: define this all in terms of algebraic effects (composable with stack-switching)