Component Model Async Support

WebAssembly CG

May/June, 2022

Outline

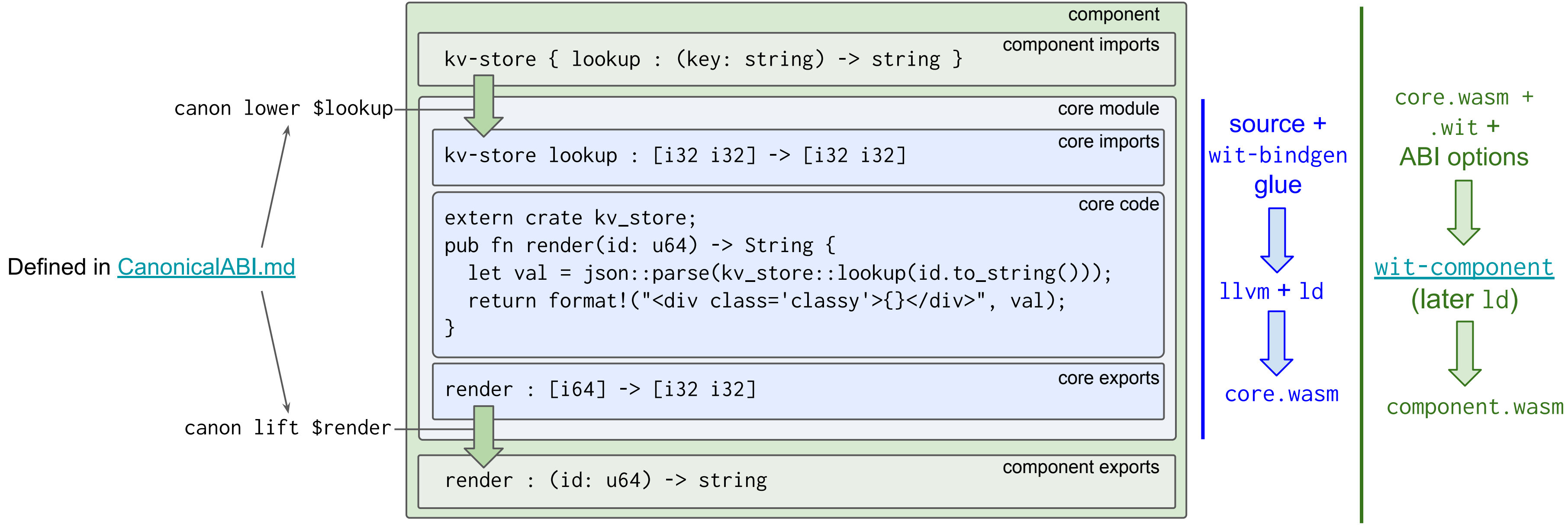
- Motivation
- Background: synchronous Canonical ABI
- Async support
  - `future`
  - Optimization: callback ABI
  - Optimization: eager return
  - Optimization: `stream`
  - Optimization: splicing and skipping streams
- Structured concurrency
  - Task tree
  - Task cancellation
  - Task finishing
 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

Caveat: still in flux; feedback welcome

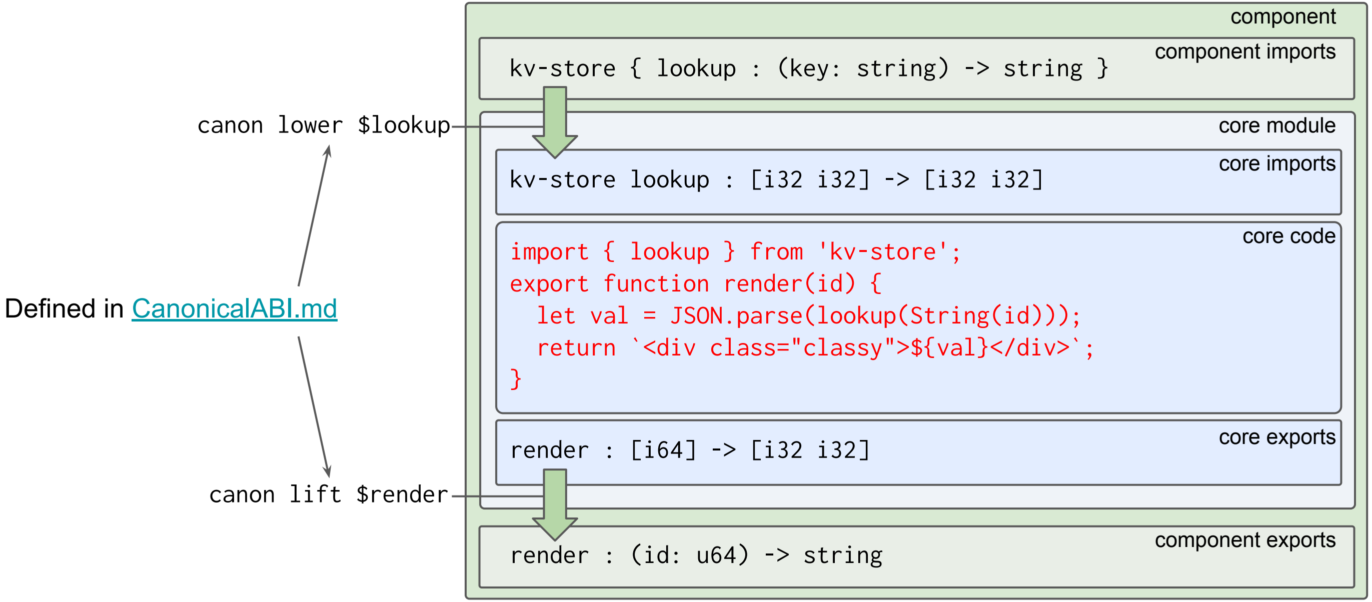
Motivation

- (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:
  - 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

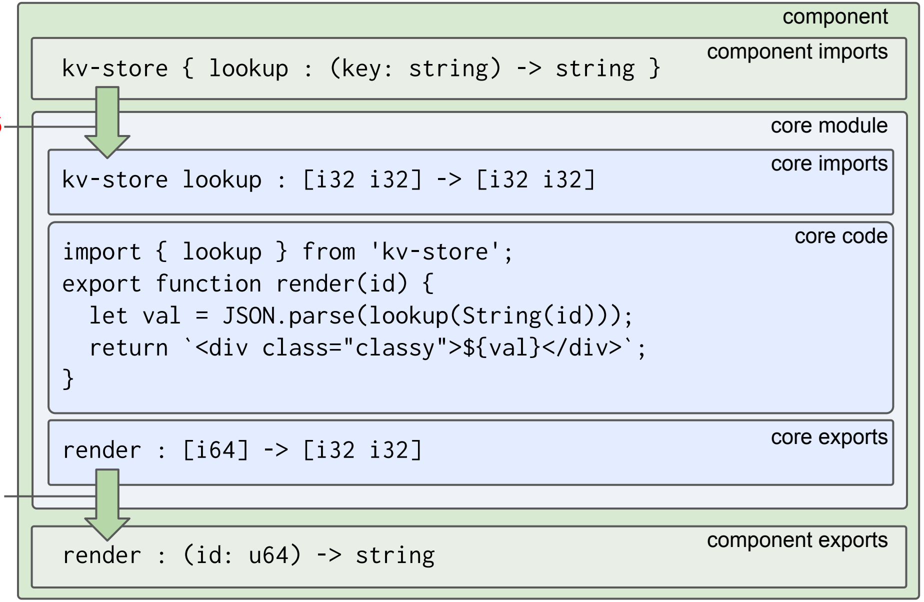


Canonical ABI options

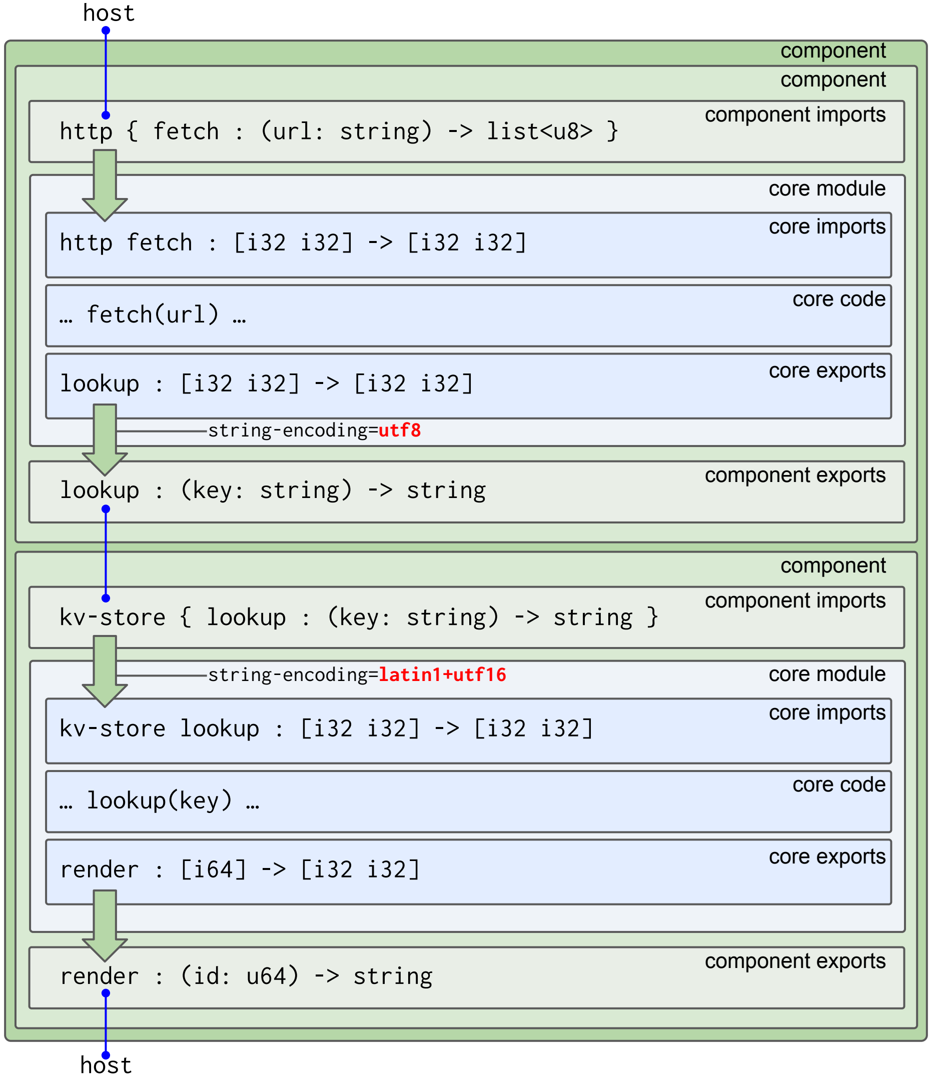
canon lower string-encoding=latin1+utf16

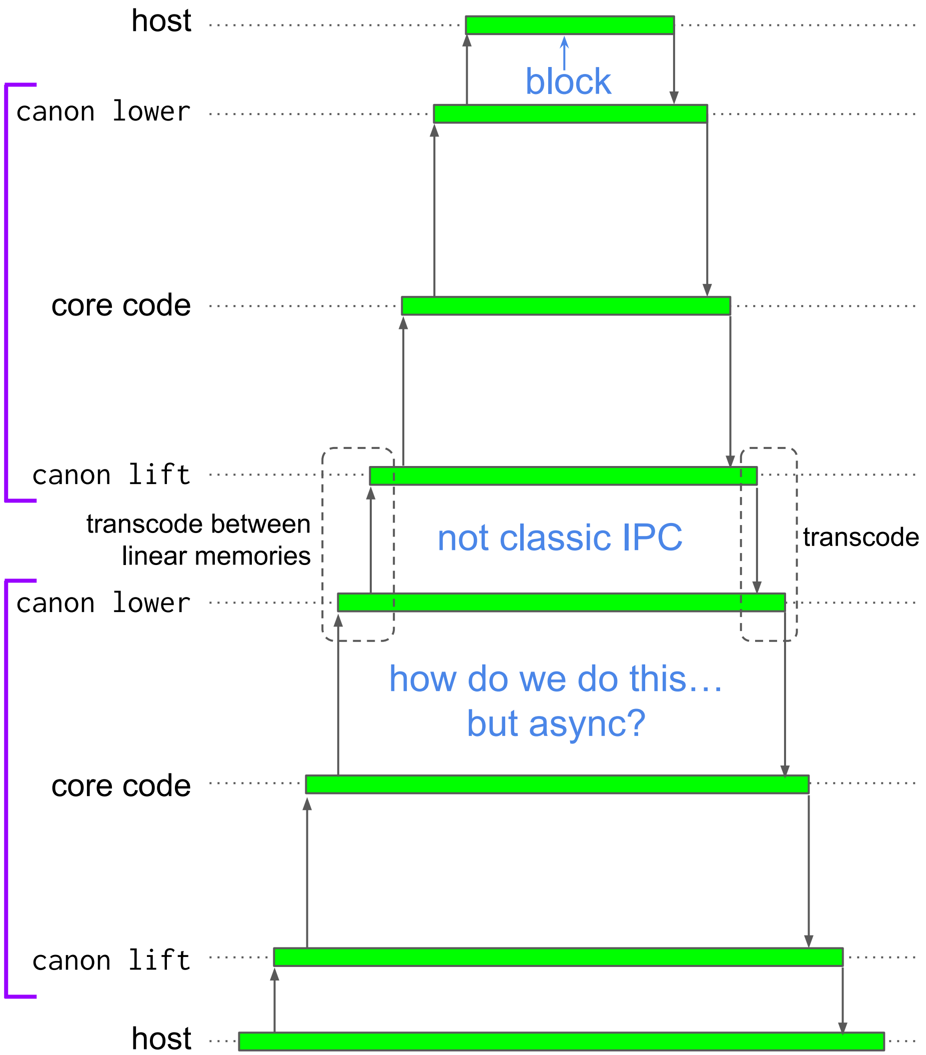
Defined in CanonicalABI.md

canon lift string-encoding=latin1+utf16

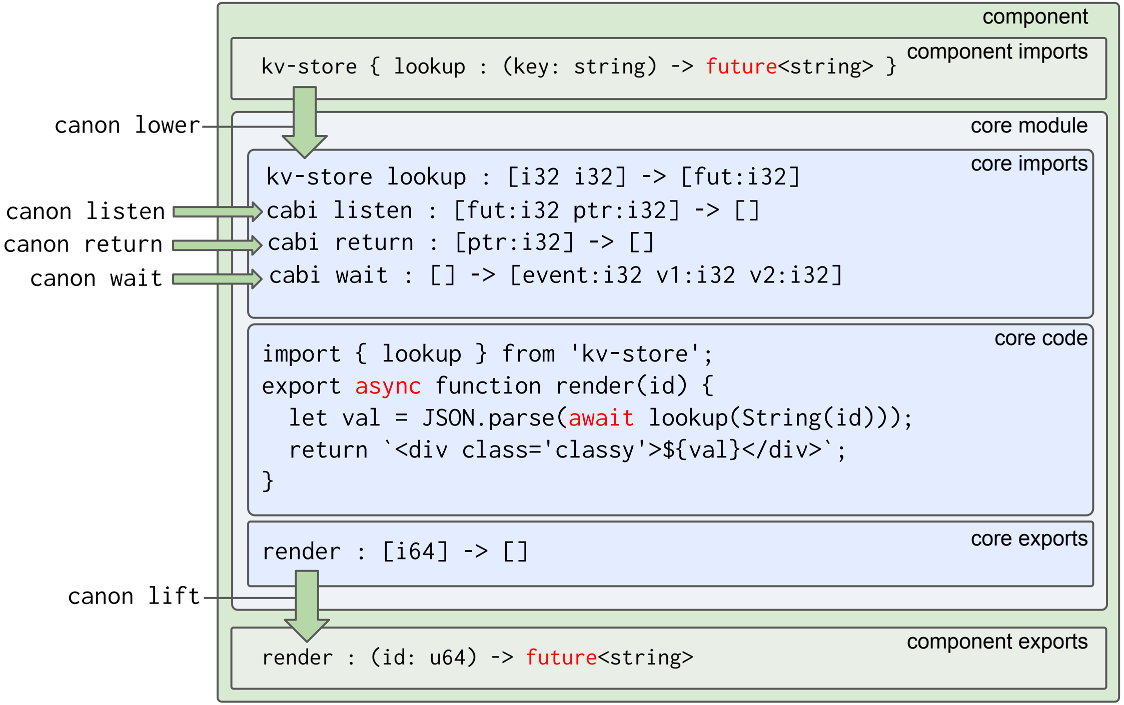


Possible because `canon lift` and `lower` bracket all component entry/exit


host
component
component
component imports
http { fetch : (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
component exports
lookup : (key: string) -> string
component
component imports
kv-store { lookup : (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]
component exports
render : (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 lower`:

- Returns an index into a component-local future table.
- The future is initially in a "not listening" state.

`canon listen`:

- Takes a future index returned by `lower` (et al…)
- Non-blocking: announces *interest* in the future's value.
- `outptr` must stay valid until the `return` event.

`canon return`:

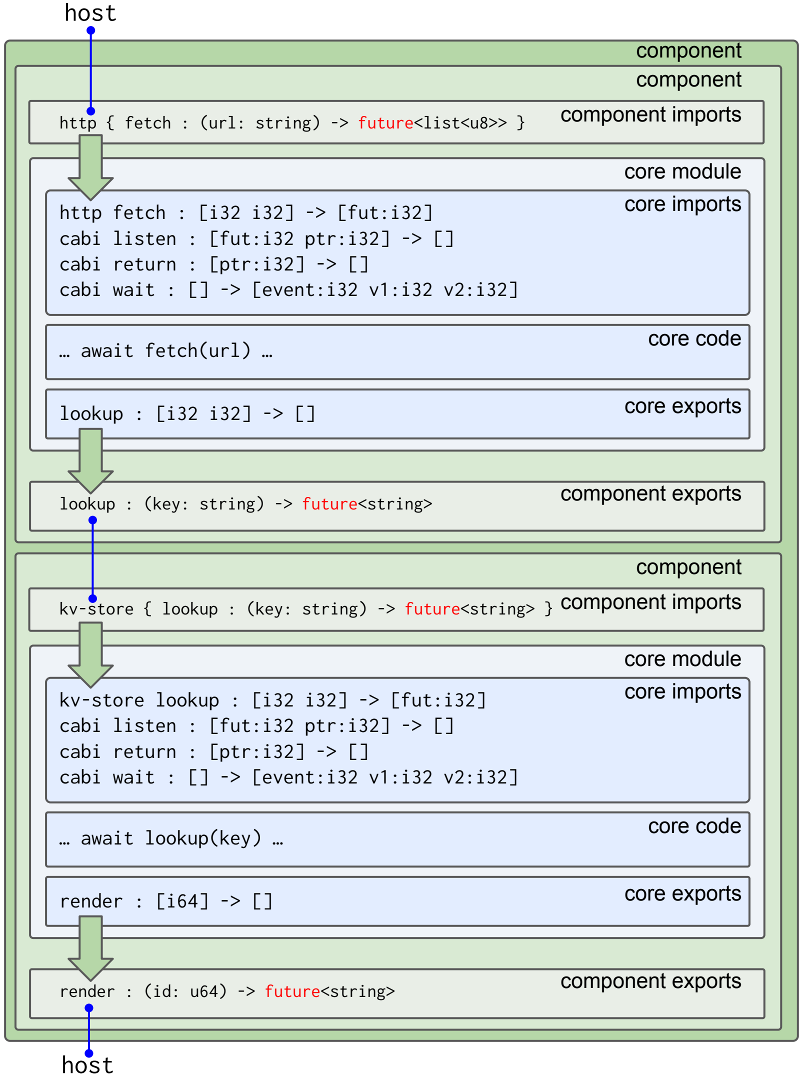
- Non-blocking: *offers* a `T` return value to the async caller.
- `ptr` must stay valid until the `return-complete` event.

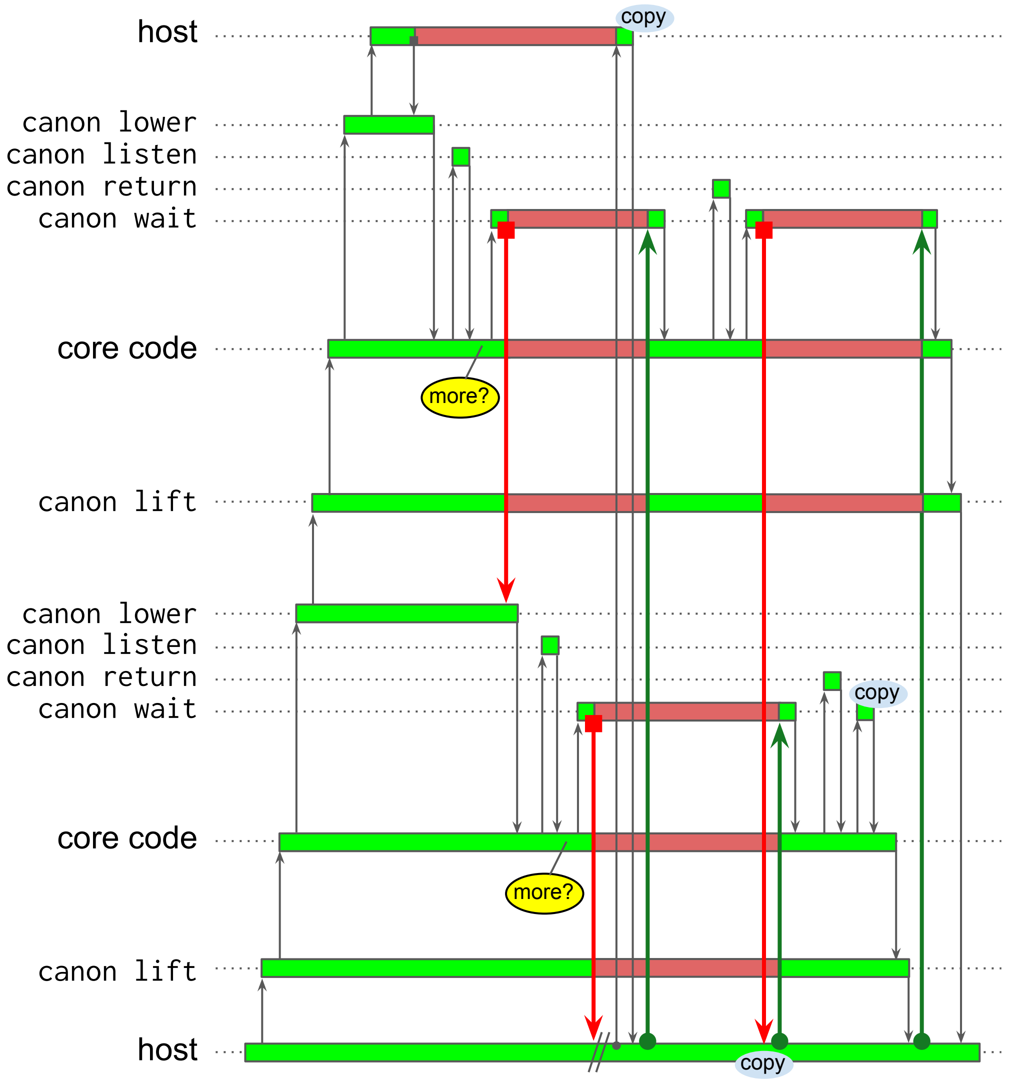
`canon wait`:

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

`canon lift`:

- Traps on return before the `return-complete` event.

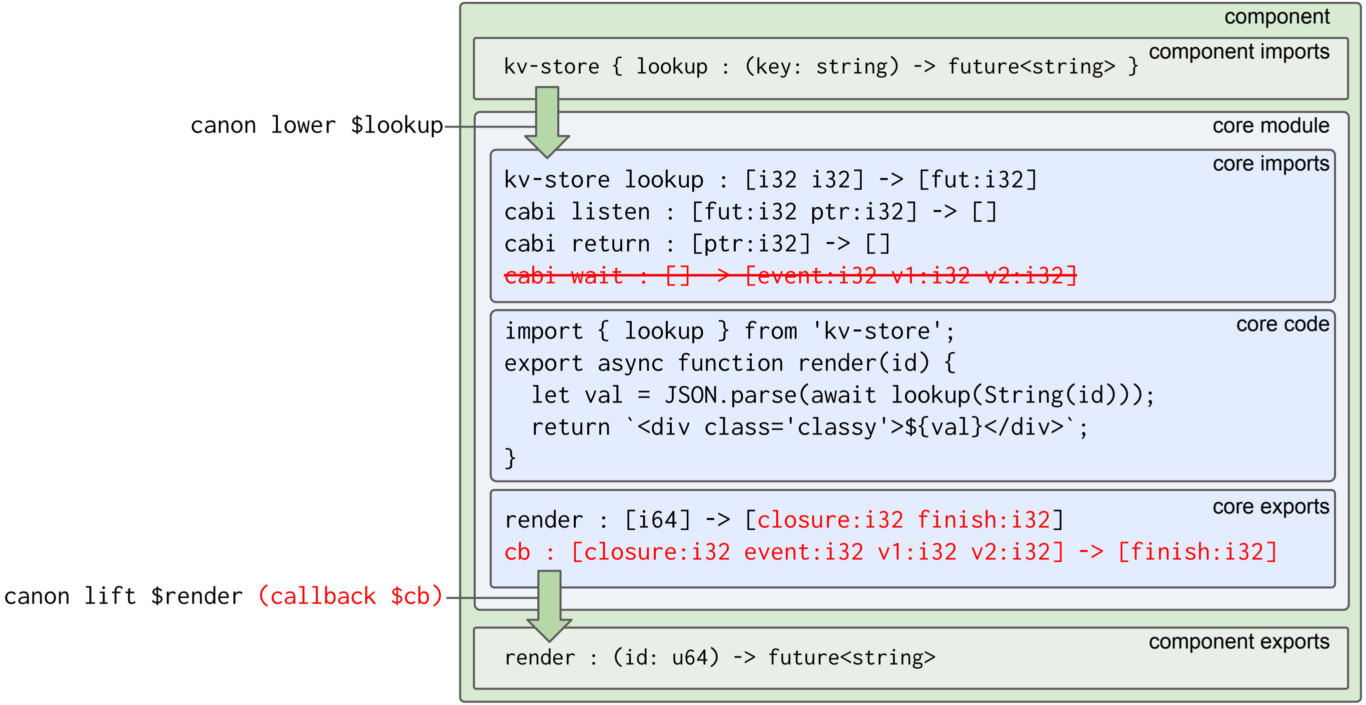

host
component
component
http { fetch : (url: string) -> future<list<u8>> }
component imports
core module
core imports
http fetch : [i32 i32] -> [fut:i32]
cabi listen : [fut:i32 ptr:i32] -> []
cabi return : [ptr:i32] -> []
cabi wait : [] -> [event:i32 v1:i32 v2:i32]
… await fetch(url) …
core code
lookup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

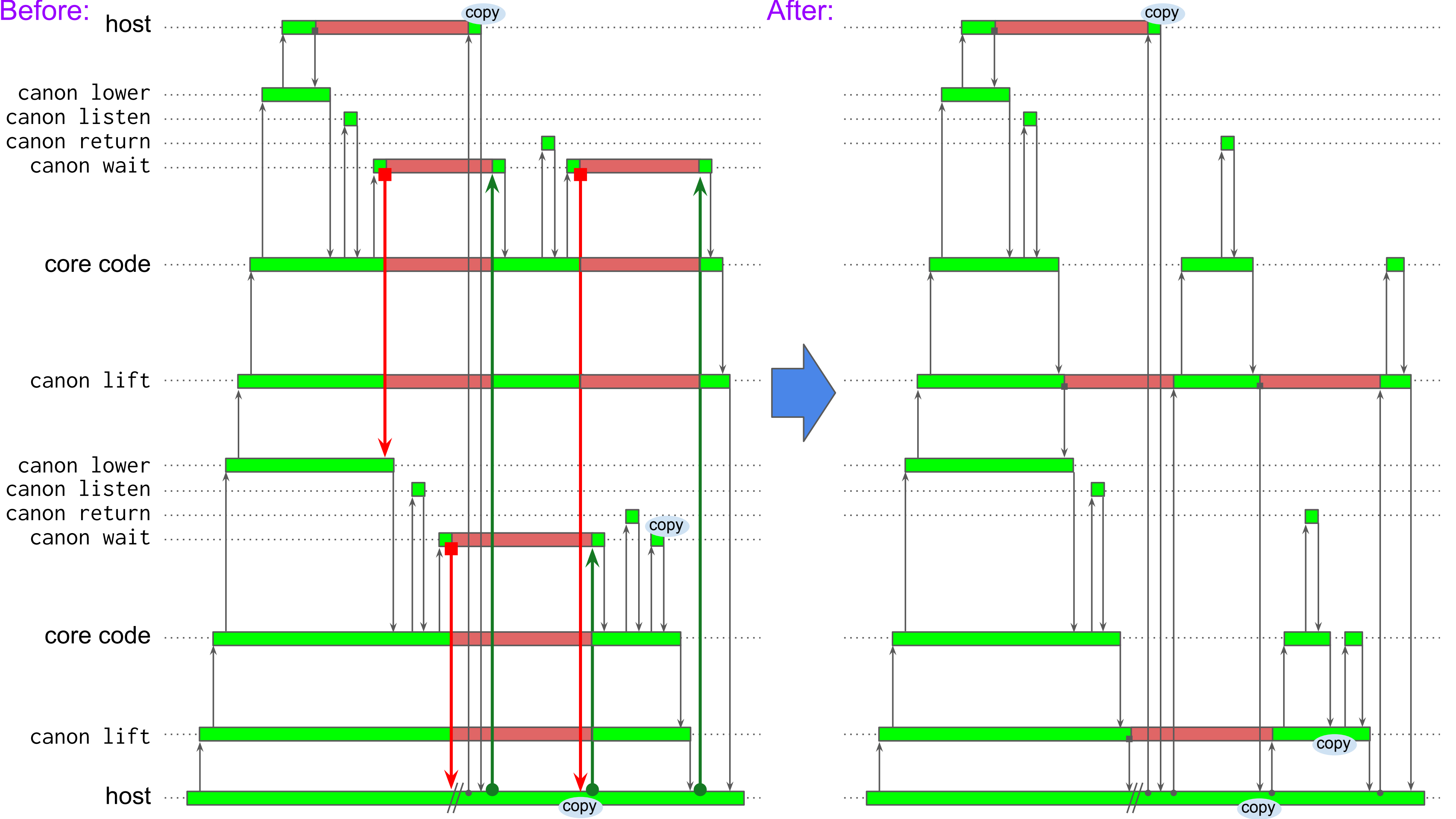


As if:

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

Notes:

- `wait` traps under `(callback $cb)`
- Encapsulated impl. detail
- Composes with non-callback

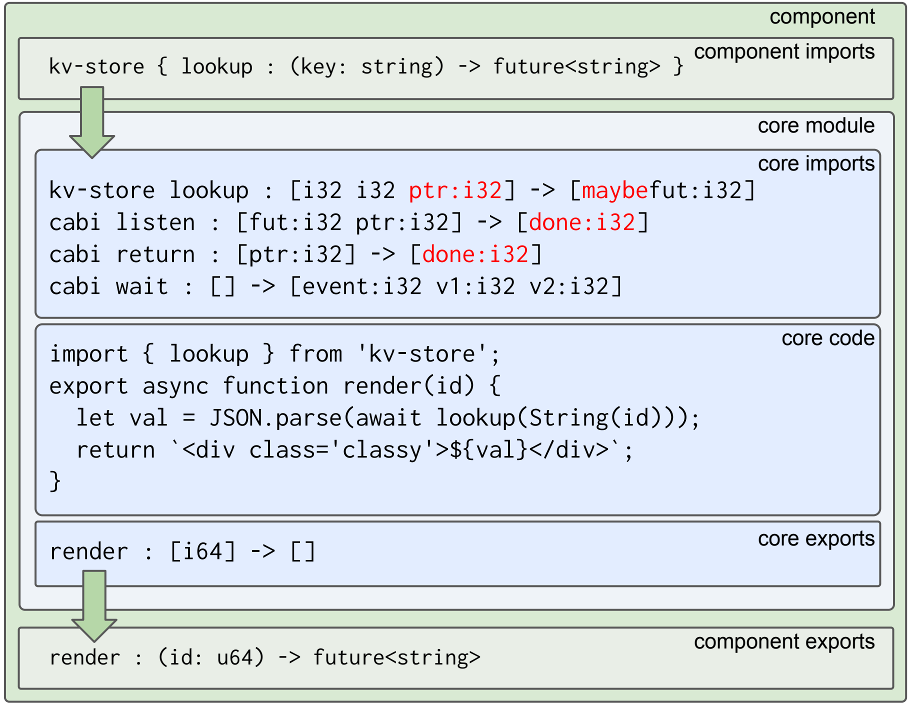


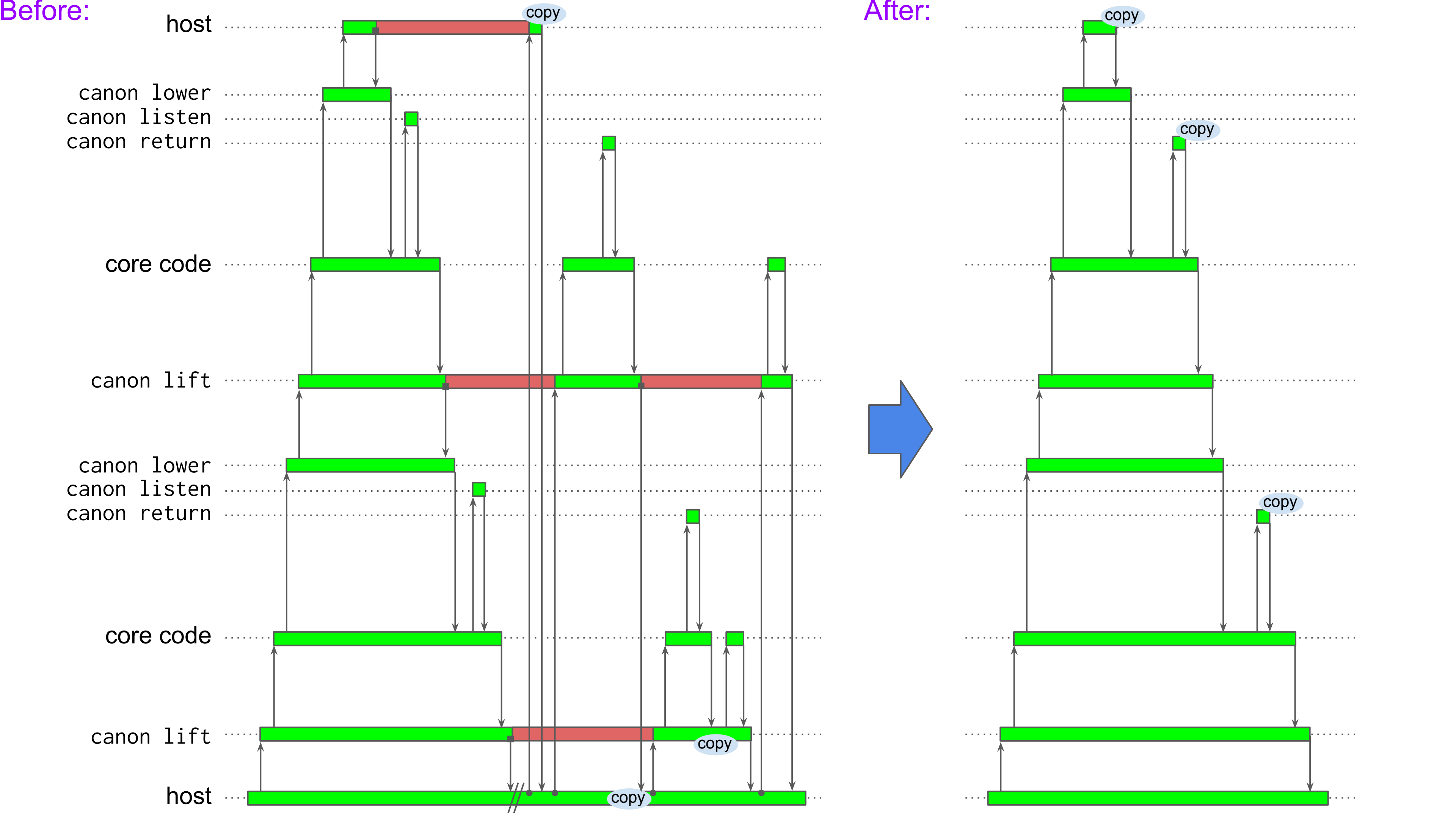
Before:
host
canon lower
canon listen
canon return
canon wait
core code
canon lift
canon lower
canon listen
canon return
canon wait
core code
canon lift
host
copy
copy
copy
After:
copy
copy
copy

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



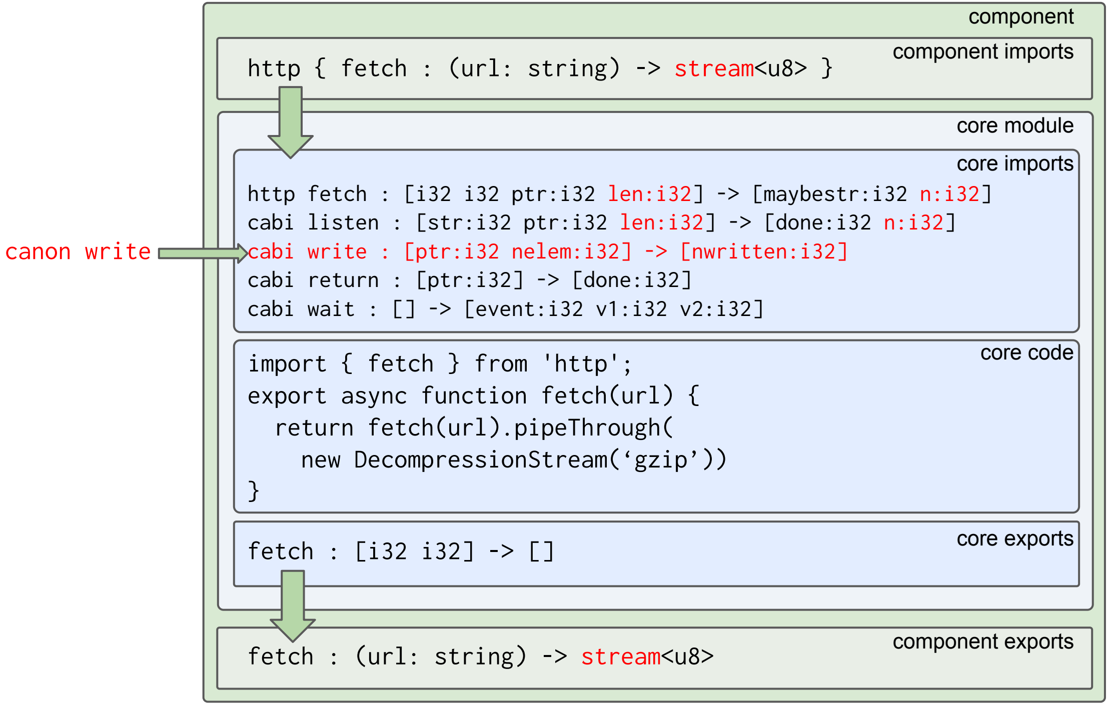


Before:
host
canon lower
canon listen
canon return
core code
canon lift
canon lower
canon listen
canon return
core code
canon lift
host
copy
copy
copy
After:
copy
copy
copy

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)
- Languages also increasingly have a built-in stream primitives
  - Tightly integrated with the rest of the language (syntax, concurrency model, backpressure, …)
- So define `stream` as a separate interface type
  - Both as an optimization but also for improved automatic language bindings
- Streams also sometimes have a "closing" value distinct from the elements
  - Effectively: `stream<T,U> = future<either<pair<T,Stream>,U>>`
  - E.g., `main: (argv:list<string>) -> stream<char,s32>`
  - `stream<T> = stream<T,unit>`

Optimization: `stream`



`canon lift` (of `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` (of `stream<T,U>`):
- Additionally takes the byte-length `ptr`
- `ptr` receives `T*` *xor* `U`.
- `n>0` means "wrote `n` `T`s", `n=0` means "returned `U`"

`canon write` (of `stream<T,U>`):
- Offers `nelem` `T` values to our async caller
- `ptr` must stay valid until `write-complete`

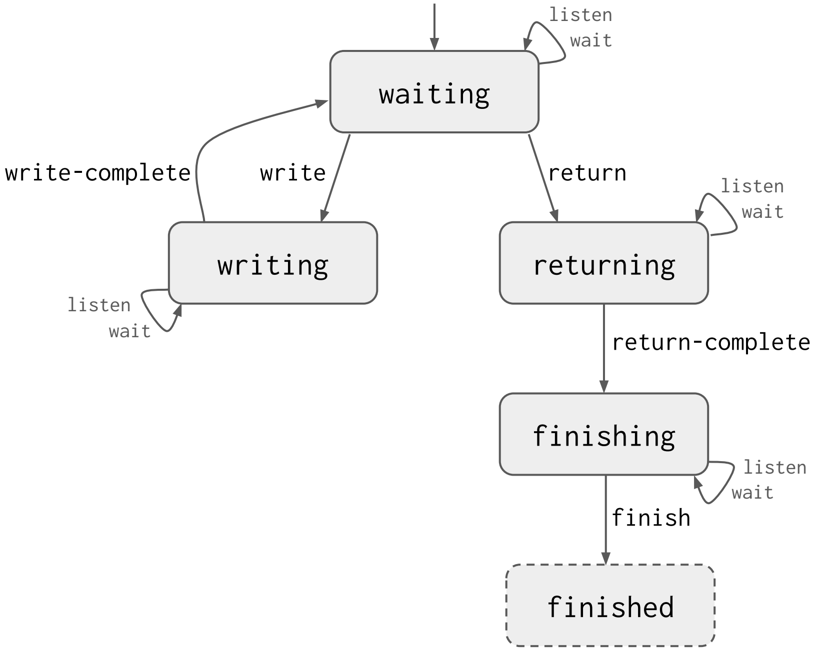
`canon return` (of `stream<T,U>`):
- Traps if `write` in progress.
- Offers a `U` return value to our async caller
- Closes the stream (no more writes possible)

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

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 : [str:i32 nelem:i32] -> [done:i32]`
  - Acts like `listen+write`: if `done=0`, must wait for a `write-complete` event.
- Sometimes we want to slice out just a subset of 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 `write` event.

State machine



A `future<U>` is just a `stream<T,U>` that 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**"...

Definitions

- A **task** is a stateful *resource* representing an asynchronous call
  - … containing a (live or suspended) stack/continuation
  - … producing one (`future`) or multiple (`stream`) values
  - … before performing some final execution to finish up and exit.
- Tasks may be **host-implemented** or **wasm-implemented**
  - … representing both the host async caller and host-implemented async import calls.
- A task also has a **producer state** and a **consumer state**
  - Producer state = { `waiting`, `writing`, `returning`, `finishing`, `finished` }
  - Consumer state = { `listening`, `not-listening` (= backpressure) }
- Tasks depend on each other via **task handles**
  - The Canonical ABI stores task handles in instance-wide tables, referred to by `i32` indices.
- The graph with tasks as nodes and task handles as edges forms a **task tree**.
  - A task handle represents **unique ownership** of a **subtask** by a **supertask**.
  - A supertask **owns** its subtasks, but may **transfer ownership** of them…

Passing tasks as parameters

component

component imports

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

core module

core imports

```
foo : [fut-in:i32] -> [fut-out:i32]
cabi listen : [fut:i32 ptr:i32] -> []
cabi return : [ptr:i32] -> []
cabi wait : [] -> [event:i32 v1:i32 v2:i32]
```

core code

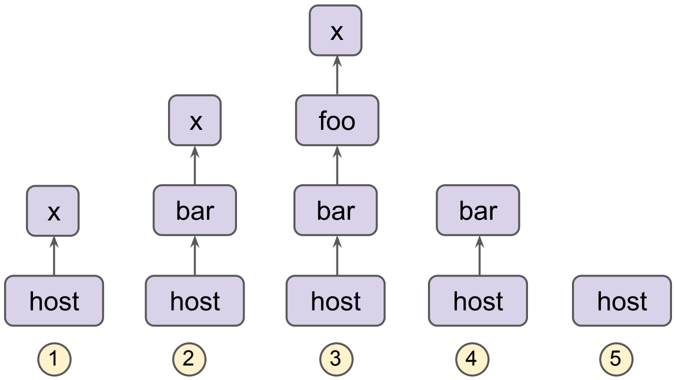
```
import { foo } from 'foo-interface';
export async function bar(x) { 2
  let p = await foo(x);
  return ‘(‘ + p + ‘)’; 4
}
```

core exports

```
bar : [fut:i32] -> []
```

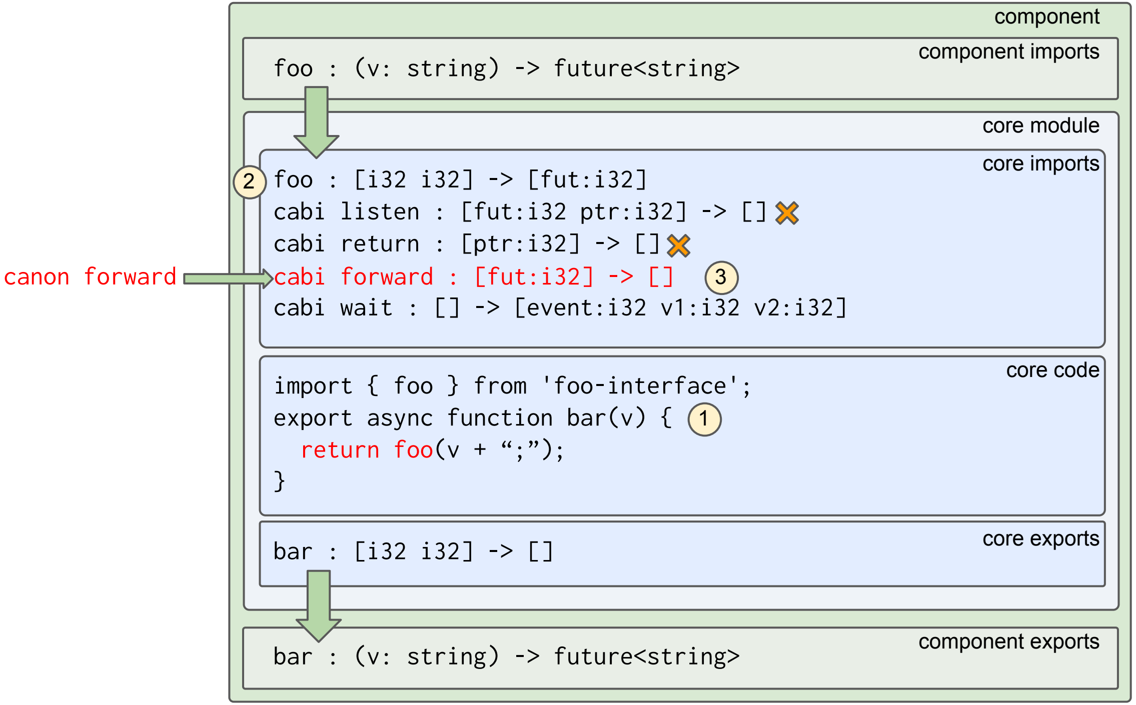
component exports

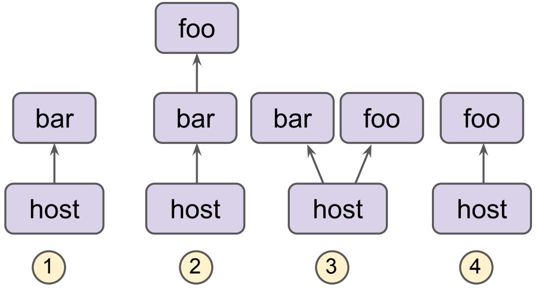
```
bar : (future<string>) -> future<string>
```



Transfer/move semantics preserves tree-ness

Returning tasks (without copying)





Returning new tasks not derived from imports

component

core module

core imports

```
cabi listen : [fut:i32 ptr:i32] -> []
cabi return : [ptr:i32] -> []
cabi new-task : [closure:i32 funcptr:i32] -> [task:i32]
cabi wait : [] -> [event:i32 v1:i32 v2:i32]
```

array of `i32` handle indices (3)

`canon new-task $T` → `cabi new-task` (2)

core code

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

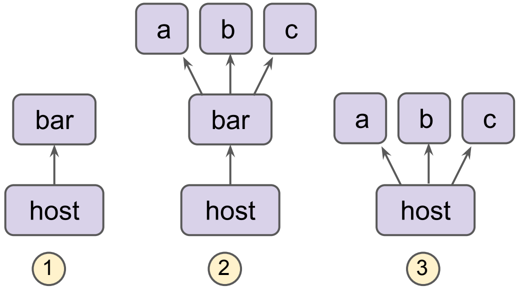
(1)

core exports

```
bar : [] -> [i32 i32]
```

component exports

```
bar : () -> list<future<string>>
```



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

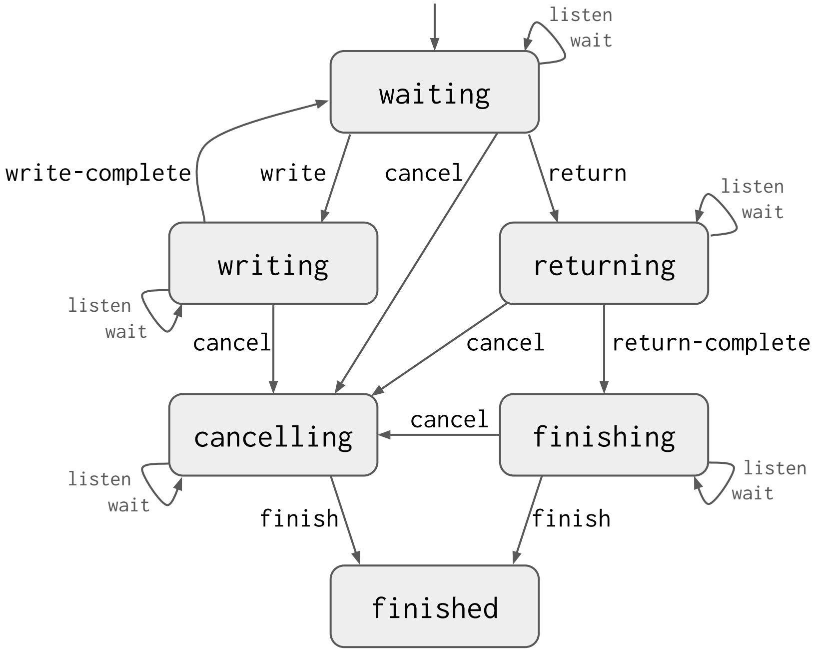
Structured concurrency

- What does "structured concurrency" mean at the component level?
  - **Subtasks can't outlive their supertasks.**
  - … although the supertask can change over time (via **explicit** parameter/result passing).
- How is this achieved?
  - `canon lift` traps if a task attempts to finish while it has any subtasks.
  - It's up to each language toolchain to figure out what this means from a language perspective
    - E.g., maintaining + draining a per-task event loop.
- 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:
    - Cross-component async callstacks in devtools
    - Automatic "tracing" (in the "observability" sense)
    - A *compositional* recursive cancellation story…

Task cancellation

- What does “cancellation” mean?
  - **The supertask has lost interest**; the result value(s) will not be consumed.
  - Also, the supertask would like the subtask to wrap it up and finish.
- Can we 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 we *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, cancellation must be *cooperative*:
  - Non-cooperative host/guest scenarios need a new “blast zone” feature *anyhow*
- Canonical ABI additions:
  - `canon cancel : [subtask:i32] -> []`
  - The subtask receives a “`cancelled`” event from `canon wait`.
  - The supertask later receives a “`finish`” event from `canon wait` after `subtask` finishes.

State machine (with cancellation)



Task finishing

- How does a task finish?
  - Normal ABI: return from the export (empty return value)
  - Callback ABI: return the "`finish`" code from the export or callback
  - Structured concurrency traps if a supertask tries to finish with any remaining subtasks.
- Q: when *precisely* is a subtask taken "off the books"?
  - A: Only when the supertask calls:
    - `canon drop : [subtask:i32] -> []`
  - This lets the toolchain control when the task index may be recycled (like closing an fd).
- Thus: supertasks must explicitly `drop` all their subtasks before finishing.
- What happens if you `drop` a subtask that isn't `finished`?
  - We can't just kill it (leak/corruption) or let it keep running (structured concurrency)
  - So: trap
- Due to uniqueness, a dropped task can be eagerly destroyed.

How does cancellation 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?

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 to `writing`/`returning`/`finished`.
  - A `writing`/`returning` subtask copies a value to a listening supertask; both sides receive events.
  - A supertask is notified that one of its subtasks is `finished`.
  - 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.

Not covered here, but (maybe) part of the proposal

- Optimization: batched reads and writes
  - Multiple writes can write into a single listen buffer.
  - Multiple listens can read from a single write buffer.
- `notify`
  - How to implement a “tee” and handle the slow-reader/fast-writer case
  - This allows deadlocks :-(
    - BUT, they can be reliably detected by the semantics to produce an error return code.
    - 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 async I/O host implementation?

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 stack switching / algebraic effects)
    - 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)