A Wasm + stack switching backend for Links

By Loïc CHEVALIER

ENS Ulm

June 30, 2025

Outline

Outline

Links

The Links language

Links

The Links language:

- Functional

Links

The Links language:

- Functional
- Supports parametric polymorphism

Links

The Links language:

- Functional
- Supports parametric polymorphism
- Effects

Links

The Links language:

- Functional
- Supports parametric polymorphism
- Multishot effects

Links

The Links language:

- Functional
- Supports parametric polymorphism
- Multishot effects with *shallow and deep* handlers

Links

The Links language:

- Functional
- Supports parametric polymorphism
- Multishot effects with *shallow and deep* handlers
- Actors

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions:

- Imperative (assembly-like), stack-based

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions:

- Imperative (assembly-like), stack-based
- Monomorphic

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions:

- Imperative (assembly-like), stack-based
- Monomorphic, but with some top types and runtime casts

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions:

- Imperative (assembly-like), stack-based
- Monomorphic, but with some top types and runtime casts
- Single-shot effects (tags)

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions:

- Imperative (assembly-like), stack-based
- Monomorphic, but with some top types and runtime casts
- Single-shot effects (tags) with *sheep* handlers (`resume`)

WebAssembly + stack switching

The Wasm language with the GC and stack switching extensions:

- Imperative (assembly-like), stack-based
- Monomorphic, but with some top types and runtime casts
- Single-shot effects (tags) with *sheep* handlers (`resume`)
- No actor

Source: Links
Target: Wasm: structured instructions

Source: ~~Links~~ *Links IR*
Target: Wasm: structured instructions

Source: Links IR: A-normal form
Target: Wasm: structured instructions
How to translate from one to the other?
A-normal form: term is a let-value or a let-apply-values or a value
Links IR: bindings followed by a tail computation: slight generalization
Structured instructions: nested blocks of instructions

Outline

Constants

Simple programs first

Constants

Simple programs first

Simplest program: integer constant

Constants

Simple programs first

Simplest program: integer constant

```
0
```

Constants

Result: `main` function storing the return value somewhere

Constants

Result: `main` function storing the return value somewhere

Target Wasm:

```
(module
  (global (mut i64) (i64.const 0))
  (func (export "main")
    (i64.const 0)
    (global.set 0)
  )
)
```

Constants

Result: `main` function storing the return value somewhere
Target Wasm:

```
(module
  (global (mut i64) (i64.const 0))
  (func (export "main")
    (i64.const 0)
    (global.set 0)
  )
)
```

Also, print the return value

Constants

Result: `main` function storing the return value somewhere
Target Wasm:

```
(module
  (global (mut i64) (i64.const 0))
  (func (export "main")
    (i64.const 0)
    (global.set 0)
  )
)
```

~~Also, print the return value~~ The output logic will be hidden in these slides

Variables

Next level: add variable support

Variables

Next level: add variable support

```
var v = 0;
v
```

Variables

Target Wasm: use local variables as well

```
(module
  (global (mut i64) (i64.const 0))
  (func (export "main") (local $v i64)
    (i64.const 0)
    (local.set $v)
    (local.get $v)
    (global.set 0)
  )
)
```

Variables

The Links IR has UID instead of variable names
Problem: how to define the correspondance from Links variable ID to Wasm variable ID?

Variables

The Links IR has UID instead of variable names
Problem: how to define the correspondance from Links variable ID to Wasm variable ID?
Solution: add an intermediary IR: WasmIR

Variables

The Links IR has UID instead of variable names
Problem: how to define the correspondance from Links variable ID to Wasm variable ID?
Solution: add an intermediary IR: WasmIR

```
{ mod_body =
    ([Assign ((Local StorVariable, TInt, 0l),
              EConstInt 0L)],
     EVariable (Local StorVariable,TInt, 0l));
  _ }
```

Variables

Problem: how to define the correspondance Links ID to Wasm ID?
Solution: add an intermediary IR: WasmIR

```
{ mod_body =
    ([Assign ((Global,                TInt, 0l),
              EConstInt 0L)],
     EVariable (Global,               TInt, 0l));
  _ }
```

Actually, store main variables in global variables; also used: local closure storage (for functions), see later slides

Functions

Next level: add function support

Functions

Next level: add function support

```
fun id(x) { x }
id(0)
```

Functions

```
(module
  (global $output (mut i64) (i64.const 0))
  (func $id (param ($x i64)(ref null struct))
    (local.get $x)
  )
  (func (export "main") (type 0)
    (i64.const 0)
    (ref.null none)
    (call $id)
    (global.set $output)
  )
)
```

Function references

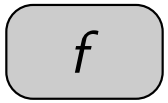
Links functions are first class

Function references

Links functions are first class
Wasm functions are not, but we can create reference to them

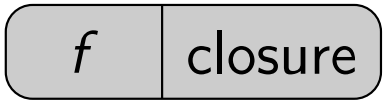
Function references

Links functions are first class
Wasm functions are not, but we can create reference to them



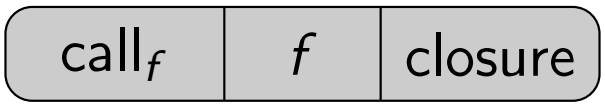
Function references

Links functions are first class
Wasm functions are not, but we can create reference to them
Also, add a closure (see next part)



Function references

Links functions are first class
Wasm functions are not, but we can create reference to them
Also, add a closure (see next part)
However, due to polymorphism issues, a "closed function" is actually a triplet:



Closures

Functions can be nested in the source language:

```
sig f : (Int) ~> (Int) ~> Int
fun f(x) {
  sig g : (Int) ~> Int
  fun g(y) { x + y }
  g
}
f(1)(2)
```

Closures

Functions can be nested in the source language:

```
sig f : (Int) ~> (Int) ~> Int
fun f(x) {
  sig g : (Int) ~> Int
  fun g(y) { x + y }
  g
}
f(1)(2)
```

Closures

They are extracted to the global scope in the Links IR:

```
([IR.Fun { fn_binder = { id = 1029; _ };
    fn_closure = Some ({ id = 1031; _ };
    fn_params = [{ id = 1030; _ }];
    fn_body = ([],
      IR.Return IR.ApplyPure (id_add,
        [IR.Project (1031, "1028");
         IR.Variable 1030])); _ };
  ...)
```

Polymorphism

There is polymorphism in Links
There is no polymorphism in Wasm

Polymorphism

There is polymorphism in Links
There is no polymorphism in Wasm
Solution: use the GC extension: the `eqref` type
Cast `structs` and `arrays` (`TString`) into `ref null eq` implicitely, box `i64`s (`TInt`) and `f64`s (`TFloat`) into a struct, and cast `i32`s (`TBool`) into `i31`s

Polymorphism

There is polymorphism in Links
There is no polymorphism in Wasm
Solution: use the GC extension: the `eqref` type
Cast `structs` and `arrays` (`TString`) into `ref null eq` implicitely, box `i64`s (`TInt`) and `f64`s (`TFloat`) into a struct, and cast `i32`s (`TBool`) into `i31`s
To call closed functions, call call_f (first element of the triplet) with the boxed arguments then the closure (third element) and f (second element), then unbox the result as needed

Polymorphism

There is polymorphism in Links
There is no polymorphism in Wasm
Solution: use the GC extension: the `eqref` type
Cast `structs` and `arrays` (`TString`) into `ref null eq` implicitely, box `i64`s (`TInt`) and `f64`s (`TFloat`) into a struct, and cast `i32`s (`TBool`) into `i31`s
To call closed functions, call call_f (first element of the triplet) with the boxed arguments then the closure (third element) and f (second element), then unbox the result as needed
Note that closed functions are therefore always boxed; elements of records are also always boxed.

Overview

Effect: generalization of exception
Two kinds of effect handlers in Links: deep and shallow

Overview

Effect: generalization of exception
Two kinds of effect handlers in Links: deep and shallow
Most used: deep handlers; implement this first

Overview

Effect: generalization of exception
Two kinds of effect handlers in Links: deep and shallow
Most used: deep handlers; implement this first

WasmFX: we have `suspend` and `resume`; `resume` installs suspension handlers

Deep handlers

```
fun loop(i, s) {
  if (i == 0) {s}
  else {
    do Op(i);
    loop(i - 1, s)
  }
}

fun run(n, s) {
  handle (loop(n, s)) {
    case <Op(x) => k> -> var y = k(()); mod(abs(x - 503 * y + 37), 1009)
  }
}

fun step(n, l, s) {
  if (l == 0) {s}
  else { step(n, l - 1, run(n, s)) }
}

fun repeat(n) { step(n, 1000, 0) }
repeat(1000)
```

Deep handlers

```
fun loop(i, s) {
  if (i == 0) {s}
  else {
    do Op(i);
    loop(i - 1, s)
  }
}

fun run(n, s) {
  handle (loop(n, s)) {
    case <Op(x) => k> -> var y = k(()); mod(abs(x - 503 * y + 37), 1009)
  }
}

fun step(n, l, s) {
  if (l == 0) {s}
  else { step(n, l - 1, run(n, s)) }
}

fun repeat(n) { step(n, 1000, 0) }
repeat(1000)
```

Handlers can also ignore the continuation (exceptions)

Deep handlers

Different handler mechanism between Links and Wasm

Deep handlers

Different handler mechanism between Links and Wasm
A Wasm suspension has a fixed type, defined when declaring the tag; a Links continuation is identified by its name, and the typing rules ensure the correctness.

Deep handlers

Different handler mechanism between Links and Wasm
A Wasm suspension has a fixed type, defined when declaring the tag; a Links continuation is identified by its name, and the typing rules ensure the correctness.
Solution: use boxed polymorphic types to transfer values, and immediately unbox them

Deep handlers

Different handler mechanism between Links and Wasm
A Wasm suspension has a fixed type, defined when declaring the tag; a Links continuation is identified by its name, and the typing rules ensure the correctness.
Solution: use boxed polymorphic types to transfer values, and immediately unbox them
A Wasm continuation starts from a function; there is no Links continuation. Catching an effect requires `suspending` from within a continuation in Wasm, not in Links.

Deep handlers

Extract the handlee to a new function:

```
(func $handlee (param $abs_closure eqref)
                (local $closure (ref 8))
                (result i64)
  (local.get $abs_closure)
  (ref.cast (ref 8))
  (local.tee $closure) (struct.get 8 $n)
  (local.get $closure) (struct.get 8 $s)
  (ref.null none)
  (return_call $loop)
)
```

Deep handlers

Extract the handler to a new function as well:

```
(func $handler (param $handlee (ref 10))
               (param $handlee_closure eqref)
               (param $handler_closure structref)
               (local $x i64)
               (local $y i64)
               (result i64)
  (local.get $handlee_closure)
  (local.get $handlee)
  (block (type 12) (resume 10 (on 0 0)) (return))
  (local.set $handlee)  ;; case <Op ... => k>
  (local.set $x)        ;; case <Op(x) ...>
  (local.get $handlee)
  (ref.null none)
  (local.get $handler_closure)
  (call $handler)       ;; k(())
  (local.set $y)        ;; var y = ...
  (local.get $x)
  (i64.const 503) (local.get $y) (i64.mul)
  (i64.sub)
  (i64.const 37) (i64.add)
  (ref.null none) (call $abs)
  (i64.const 1_009) (i64.rem_s)
)
```

Deep handlers

First, support for exceptions:

```
handle ({do Err}) { case <Err => k> -> () }
```

Deep handlers

First, support for exceptions:

```
handle ({do Err}) { case <Err => k> -> () }
```

Generate the handlee/handler functions, call the handler with the handlee, ignore the new continuation if we catch the `Err` effect

Deep handlers

Next, support the use of the continuation directly in the handling:

```
handle ({do Get}) { case <Get => k> -> k(1) }
```

Deep handlers

Next, support the use of the continuation directly in the handling:

```
handle ({do Get}) { case <Get => k> -> k(1) }
```

Gets translated similarly with the two auxiliary functions
Ultimately, calls the handler function again with the new continuation

Deep handlers

Next, support the use of the continuation directly in the handling:
`handle ({do Get}) { case <Get => k> -> k(1) }`

Gets translated similarly with the two auxiliary functions
Ultimately, calls the handler function again with the new continuation
Doesn't support multishot handlers, only zeroshot/singleshot ones: cannot have `case <Get => k> -> concat(k(0), k(1))`
Limitation due to WasmFX; effort by Elouan to use CPS in pure Wasm + GC to support multishot handlers

Deep handlers

Finally, support the use of the continuation indirectly:

```
handle ({do Set(5) + 1}) {
  case <Set(x) => k> -> fun(_) { k(x)(x) }
  case v -> fun(x) { v - x }
}
```

Deep handlers

Finally, support the use of the continuation indirectly:

```
handle ({do Set(5) + 1}) {
  case <Set(x) => k> -> fun(_) { k(x)(x) }
  case v -> fun(x) { v - x }
}
```

Gets translated similarly with the two auxiliary functions

Deep handlers

Finally, support the use of the continuation indirectly:

```
handle ({do Set(5) + 1}) {
  case <Set(x) => k> -> fun(_) { k(x)(x) }
  case v -> fun(x) { v - x }
}
```

Gets translated similarly with the two auxiliary functions

Ultimately, requires a third auxiliary function calling the handler with `k`

Deep handlers

One more feature
Typical global state implementation with handlers:

```
fun state(f)(st0) {
  (handle (f()) {
    case <Get => k> ->     fun(s) { k(s )(s) }
    case <Set(s) => k> -> fun(_) { k(())(s) }
    case v ->             fun(_) { v }
  })(st0)
}
```

Deep handlers

One more feature
Instead, optimize the handler:

```
fun state(f)(st0) {
  handle (f()) (s <- st0) {
    case <Get => k> ->                    k(s , s)
    case <Set(s) => k> ->                 k((), s)
    case v ->                             v
  }
}
```

Shallow handlers

Shallow handlers are simpler than deep handlers: extract the handlee, then put the handler inline; the subsequent continuations have empty suspension handlers

Shallow handlers

Shallow handlers are simpler than deep handlers: extract the handlee, then put the handler inline; the subsequent continuations have empty suspension handlers

Syntax:

```
shallowhandle ({do Foo}) {
  case <Foo => k> -> 0
}
```

Actor

Actor: lightweight thread system

Actor

Actor: lightweight thread system
Every thread has a *mailbox*

Actor

Actor: lightweight thread system
Every thread has a *mailbox*
Threads can send messages to any thread; threads can receive messages from their own mailbox only

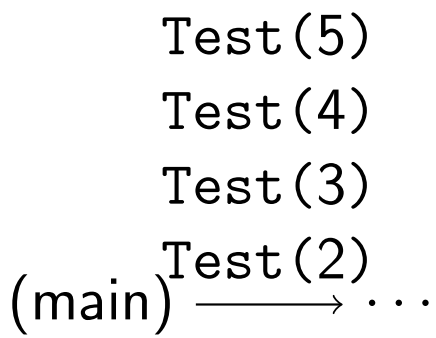
Example: Eratosthenes' sieve

$$(\text{main}) \longrightarrow \cdots$$

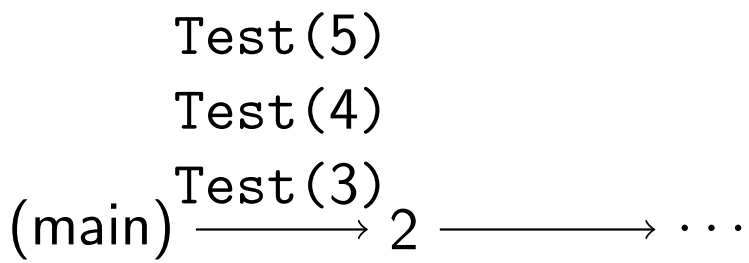
Example: Eratosthenes' sieve

$$(\text{main}) \xrightarrow{\texttt{Test(2)}} \cdots$$

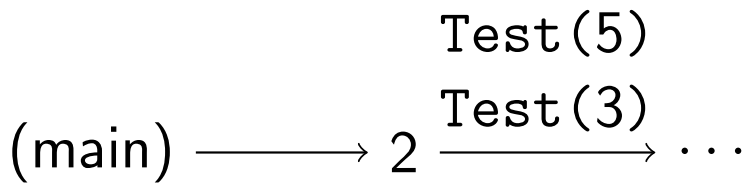
Example: Eratosthenes' sieve



Example: Eratosthenes' sieve

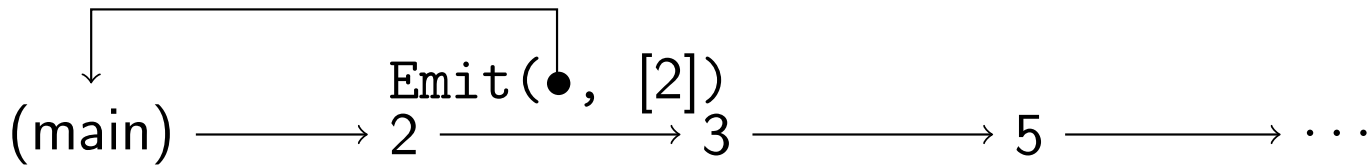


Example: Eratosthenes' sieve

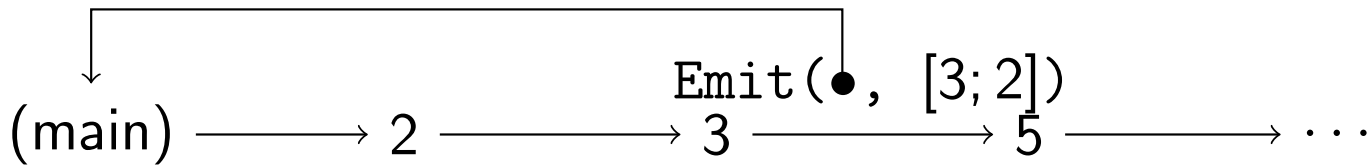


Example: Eratosthenes' sieve

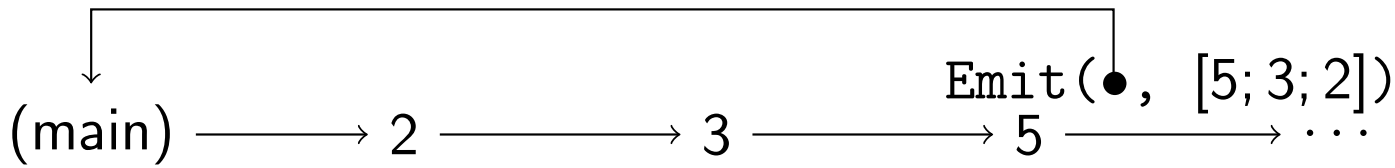
Example: Eratosthenes' sieve



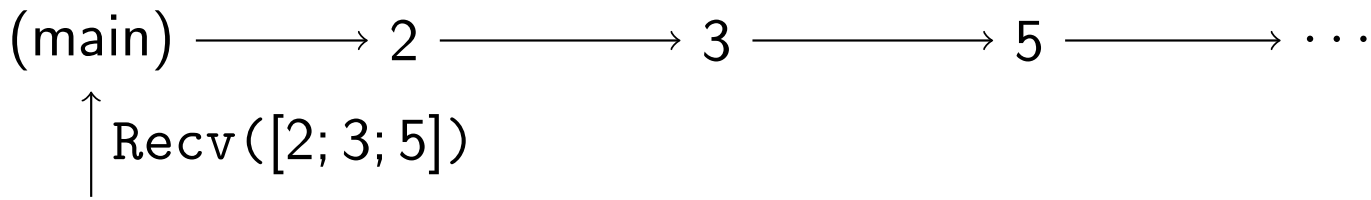
Example: Eratosthenes' sieve



Example: Eratosthenes' sieve



Example: Eratosthenes' sieve



Example: Eratosthenes' sieve

```
fun new_proc() { spawn {
  fun loop(n, sub) {
    receive {
      case Emit(p, acc) -> sub ! Emit(p, n :: acc)
      case Test(k) -> if (mod(k, n) != 0) sub ! Test(k)
    };
    loop(n, sub)
  }
  receive {
    case Emit(p, acc) -> p ! Recv(reverse(acc))
    case Test(n) -> loop(n, new_proc())
  }
}}

var root_proc = new_proc();
fun loop(k) {
  if (k > 100) spawnWait {
    root_proc ! Emit(self(), []);
    receive { case Recv(v) -> v }
  } else {
    root_proc ! Test(k);
    loop(k + 1)
  }
}
loop(2)
```

Actor

Detect whether any actor-related functions are used: `Send`, `recv`, `spawn`, `spawnWait`

Actor

Detect whether any actor-related functions are used: `Send`, `recv`, `spawn`, `spawnWait`
Also, added two actor-specific flags: `wasm_yield_is_switch` and `wasm_prefer_globals`

Actor effects

Possible actions:

Actor effects

Possible actions:

- Send a message

Actor effects

Possible actions:

- Send a message
- Receive a message

Actor effects

Possible actions:

- Send a message
- Receive a message
- Spawn a new actor

Actor effects

Possible actions:

- Send a message
- Receive a message
- Spawn a new actor
- Wait on an actor to finish

Actor effects

Possible actions:

- Send a message
- Receive a message
- Spawn a new actor
- Wait on an actor to finish
- Exit the program

Actor effects

Possible actions:

- Send a message
- Receive a message
- Spawn a new actor
- Wait on an actor to finish
- Exit the program
- Get the active actor (`self`)

Actor effects

Possible actions:

- Send a message
- Receive a message
- Spawn a new actor
- Wait on an actor to finish
- Exit the program
- Get the active actor (`self`)
- Yield

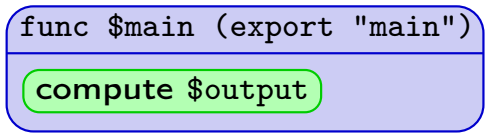
Actor effects

Possible actions:

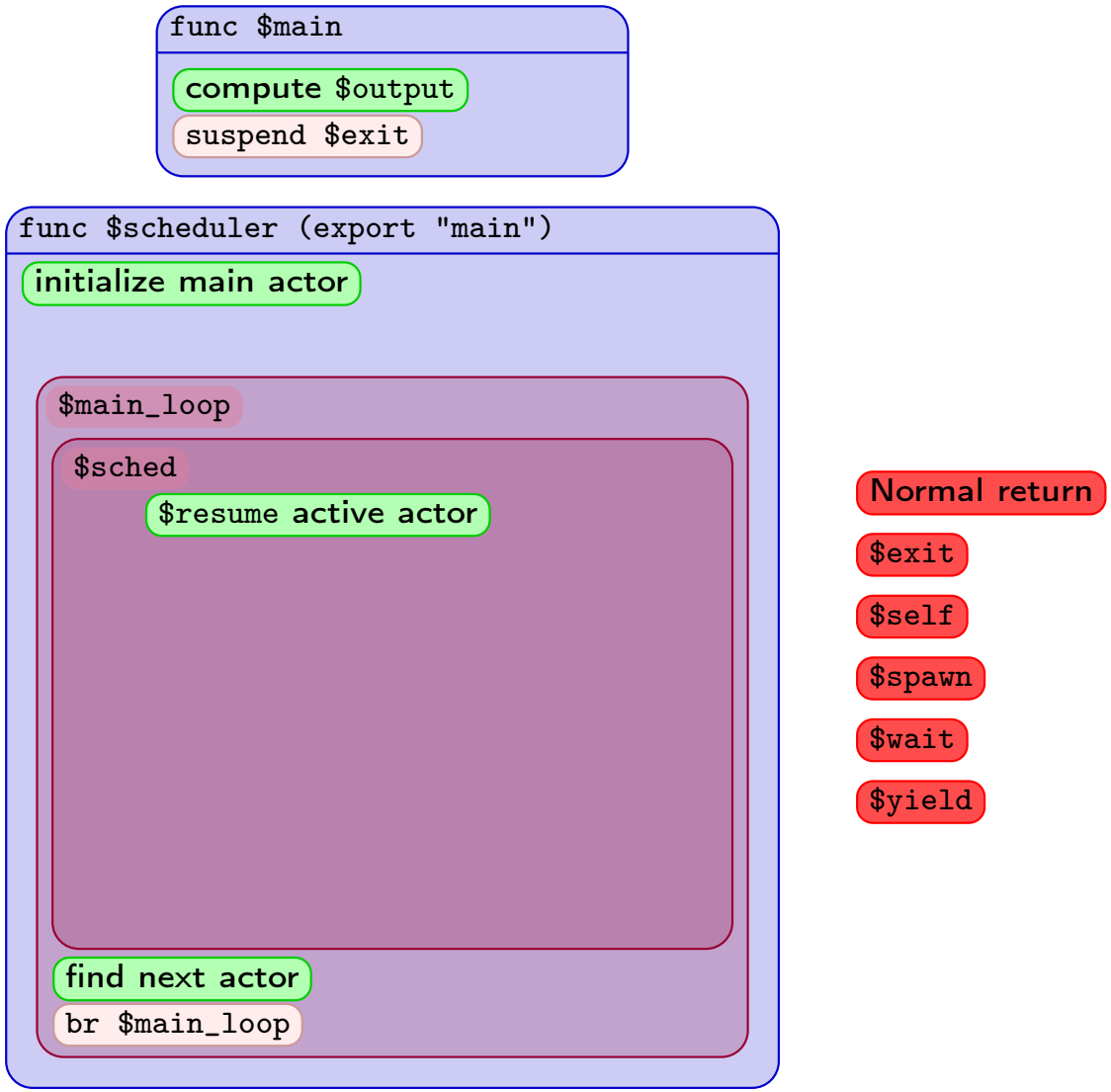
- Send a message
- Receive a message
- Spawn a new actor
- Wait on an actor to finish
- Exit the program
- Get the active actor (`self`)
- Yield

Explicit yields added in the code to enable concurrent multithreading in the source language

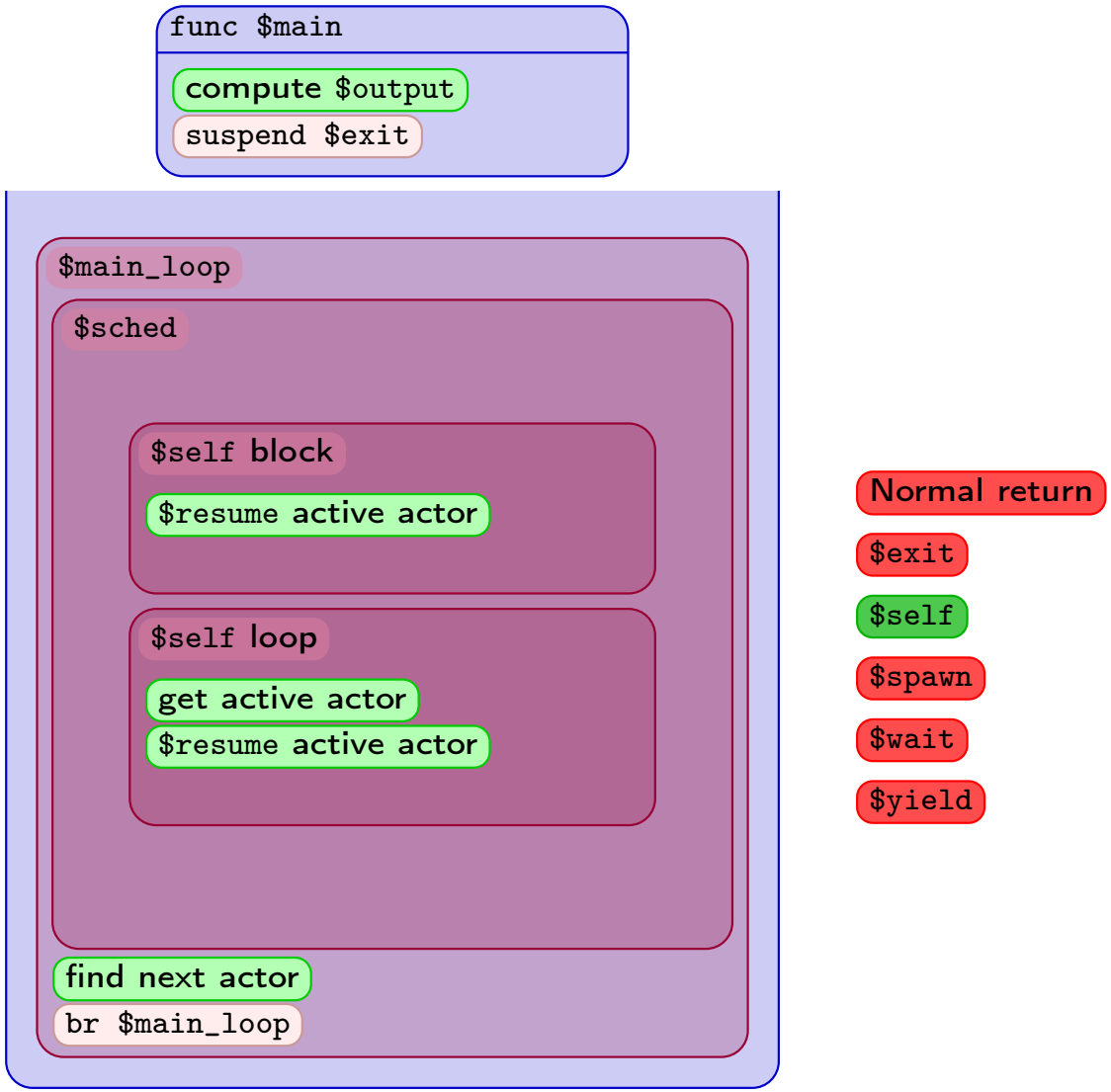
Actor logic



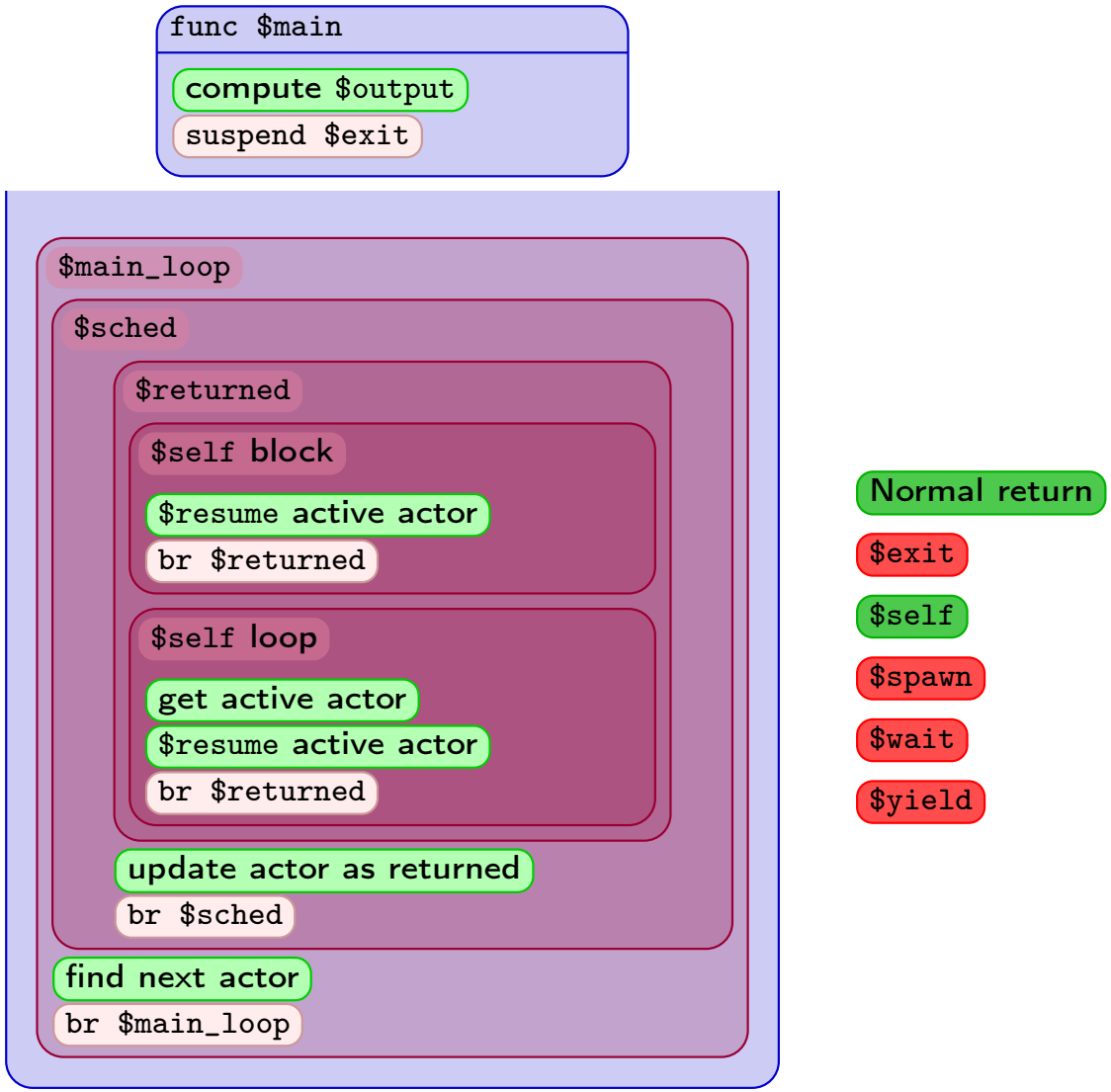
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$main_loop
$sched
$resume active actor
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

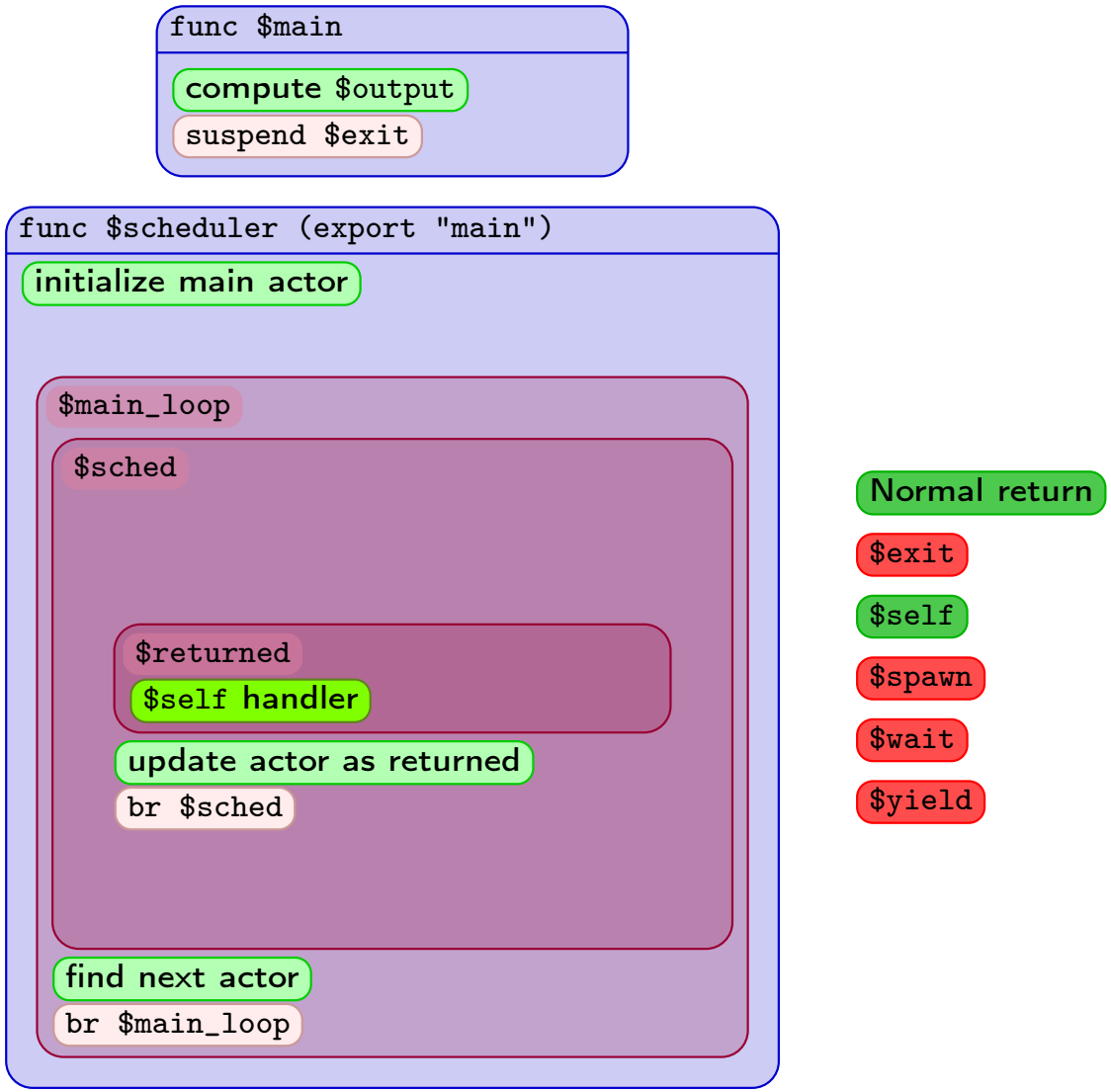
Actor logic


func $main
compute $output
suspend $exit
$main_loop
$sched
$self block
$resume active actor
$self loop
get active actor
$resume active actor
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

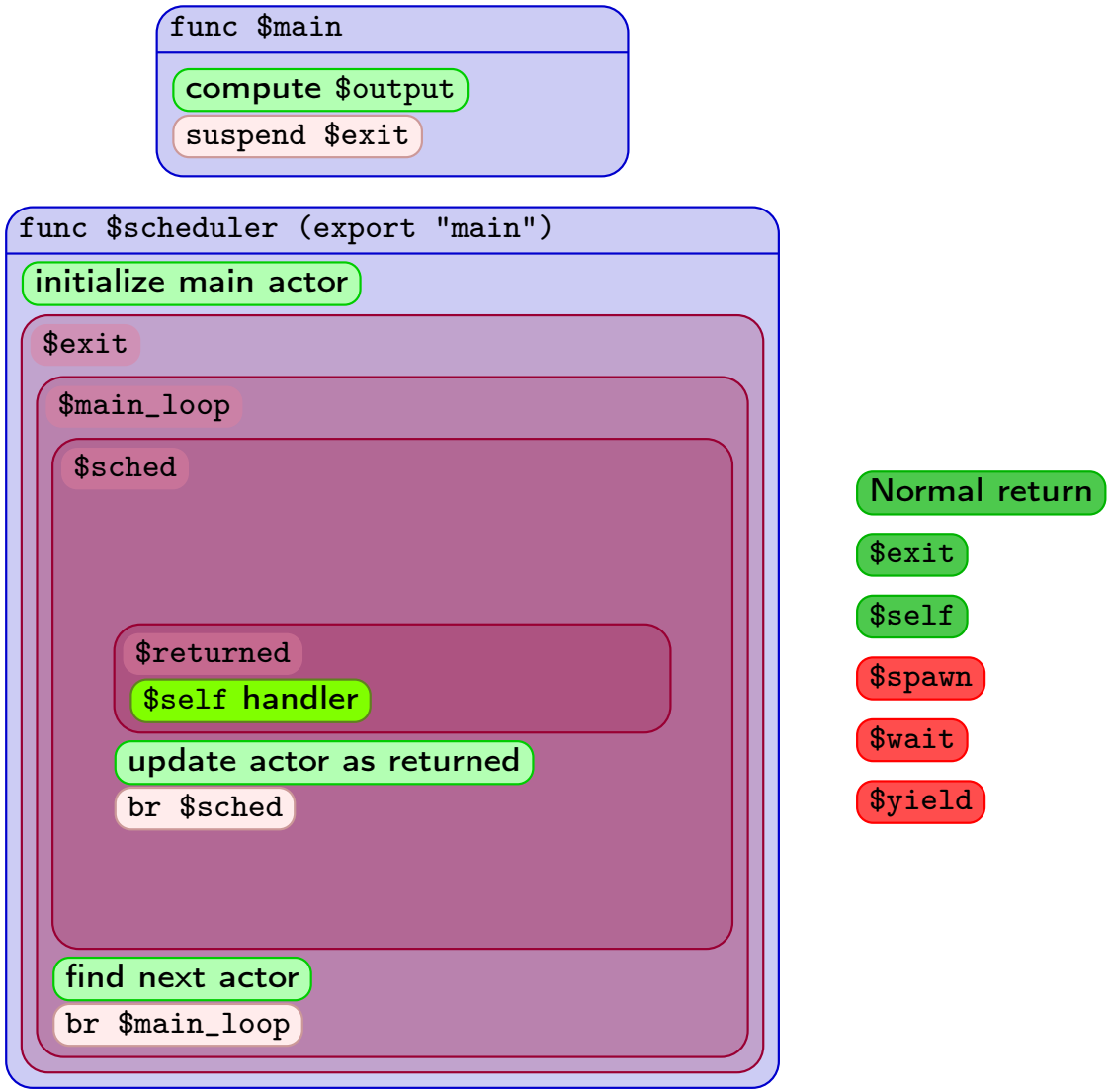
Actor logic


func $main
compute $output
suspend $exit
$main_loop
$sched
$returned
$self block
$resume active actor
br $returned
$self loop
get active actor
$resume active actor
br $returned
update actor as returned
br $sched
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

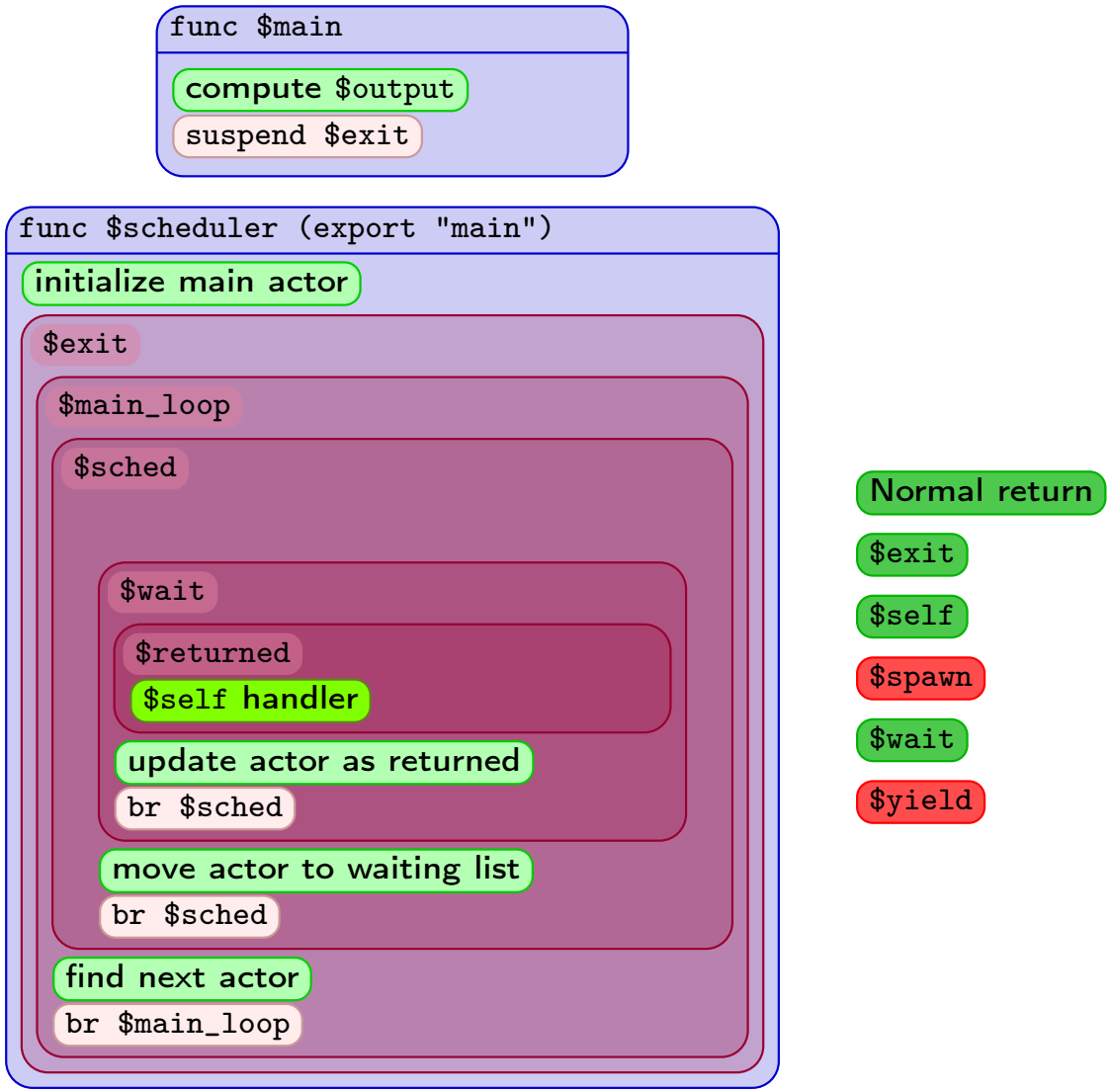
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$main_loop
$sched
$returned
$self handler
update actor as returned
br $sched
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

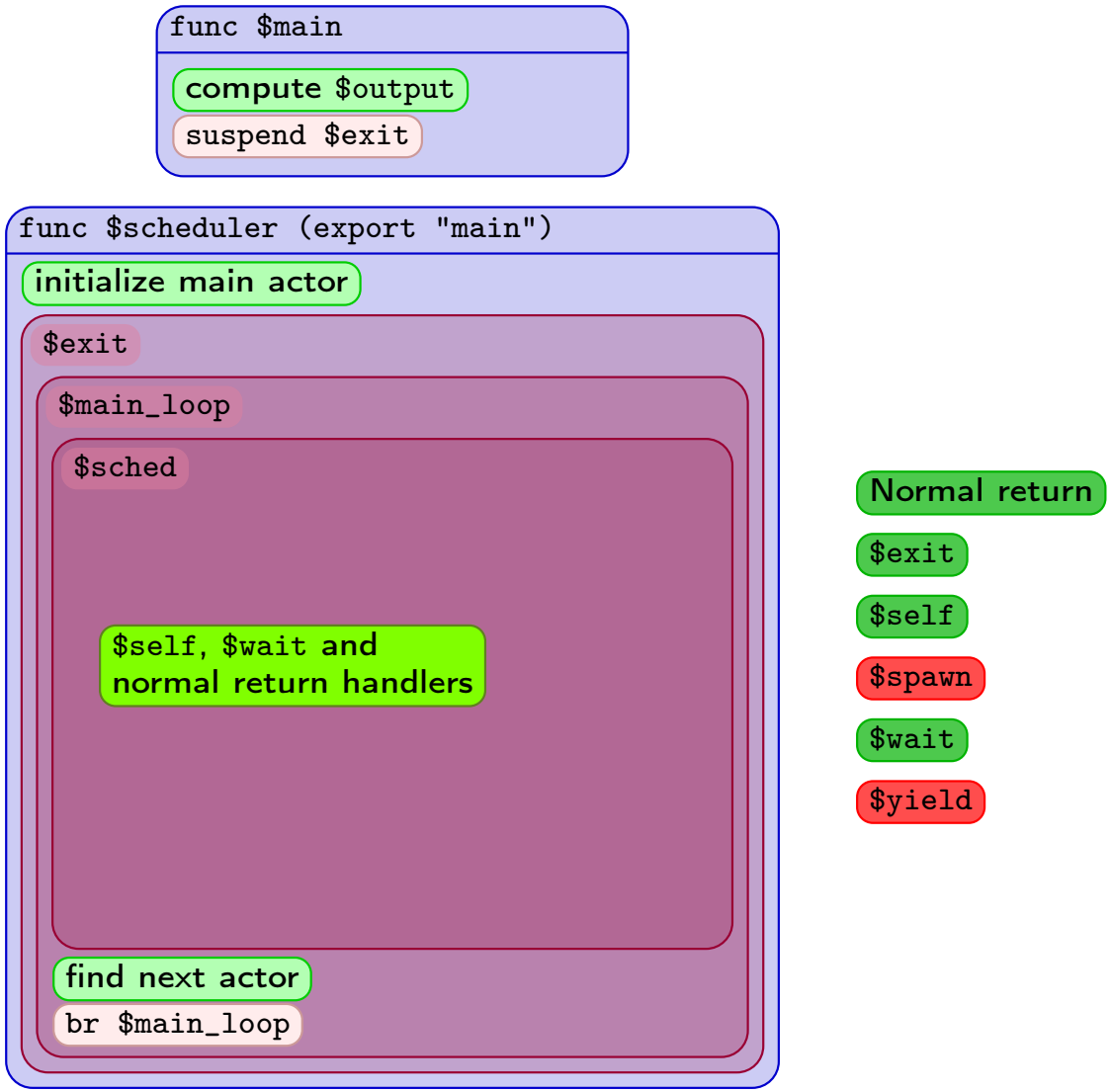
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$exit
$main_loop
$sched
$returned
$self handler
update actor as returned
br $sched
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

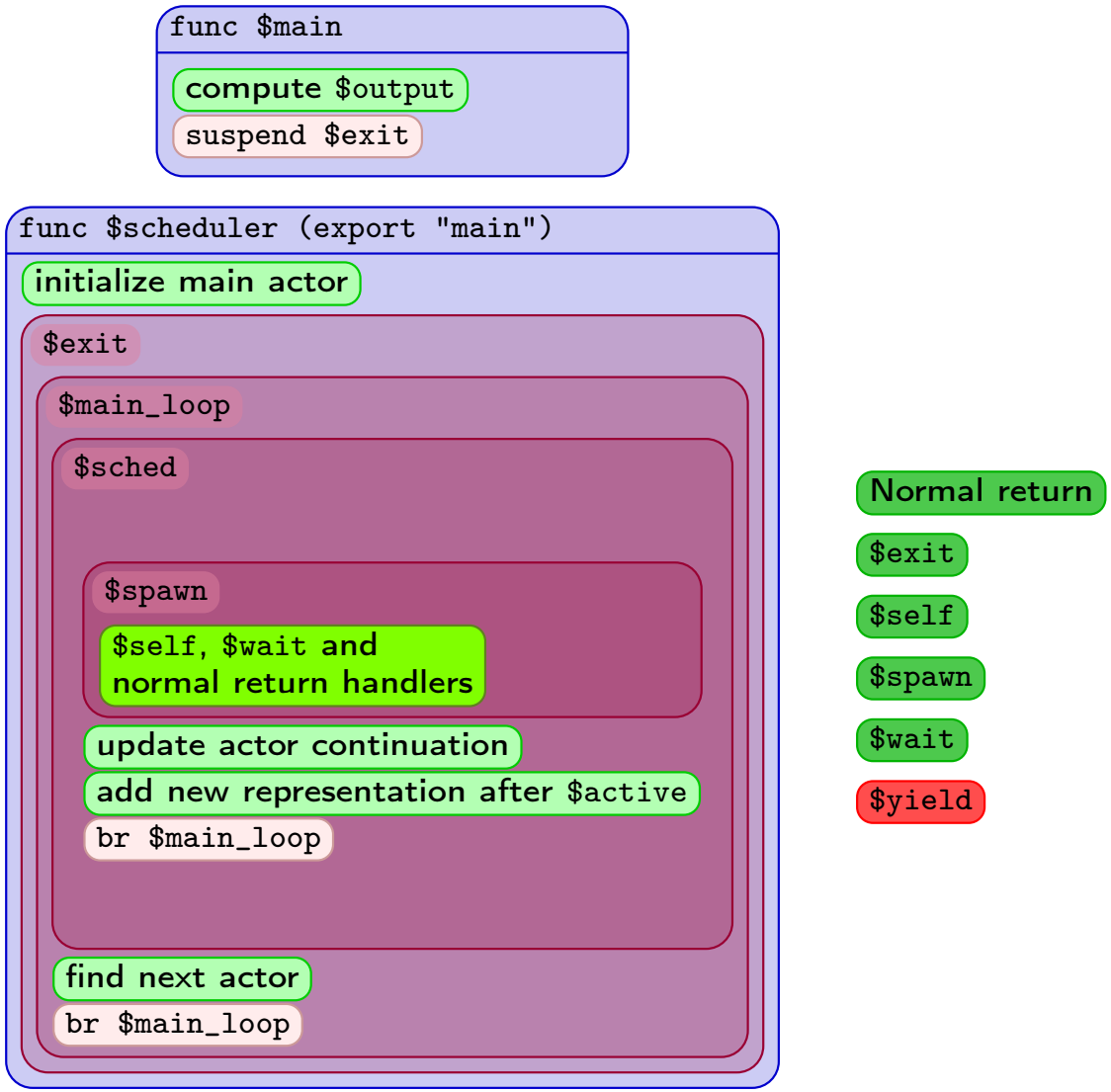
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$exit
$main_loop
$sched
$wait
$returned
$self handler
update actor as returned
br $sched
move actor to waiting list
br $sched
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

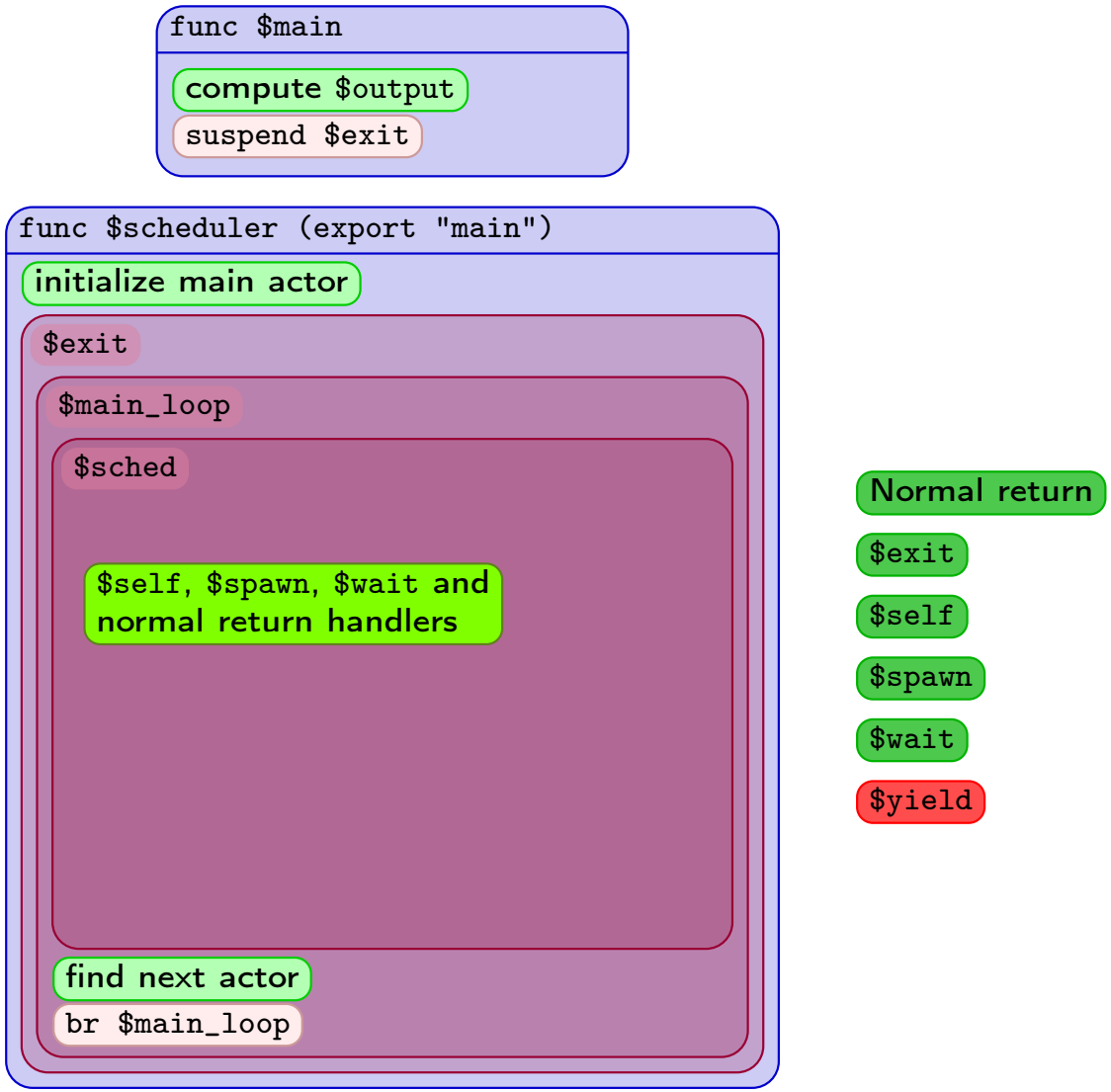
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$exit
$main_loop
$sched
$self, $wait and
normal return handlers
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

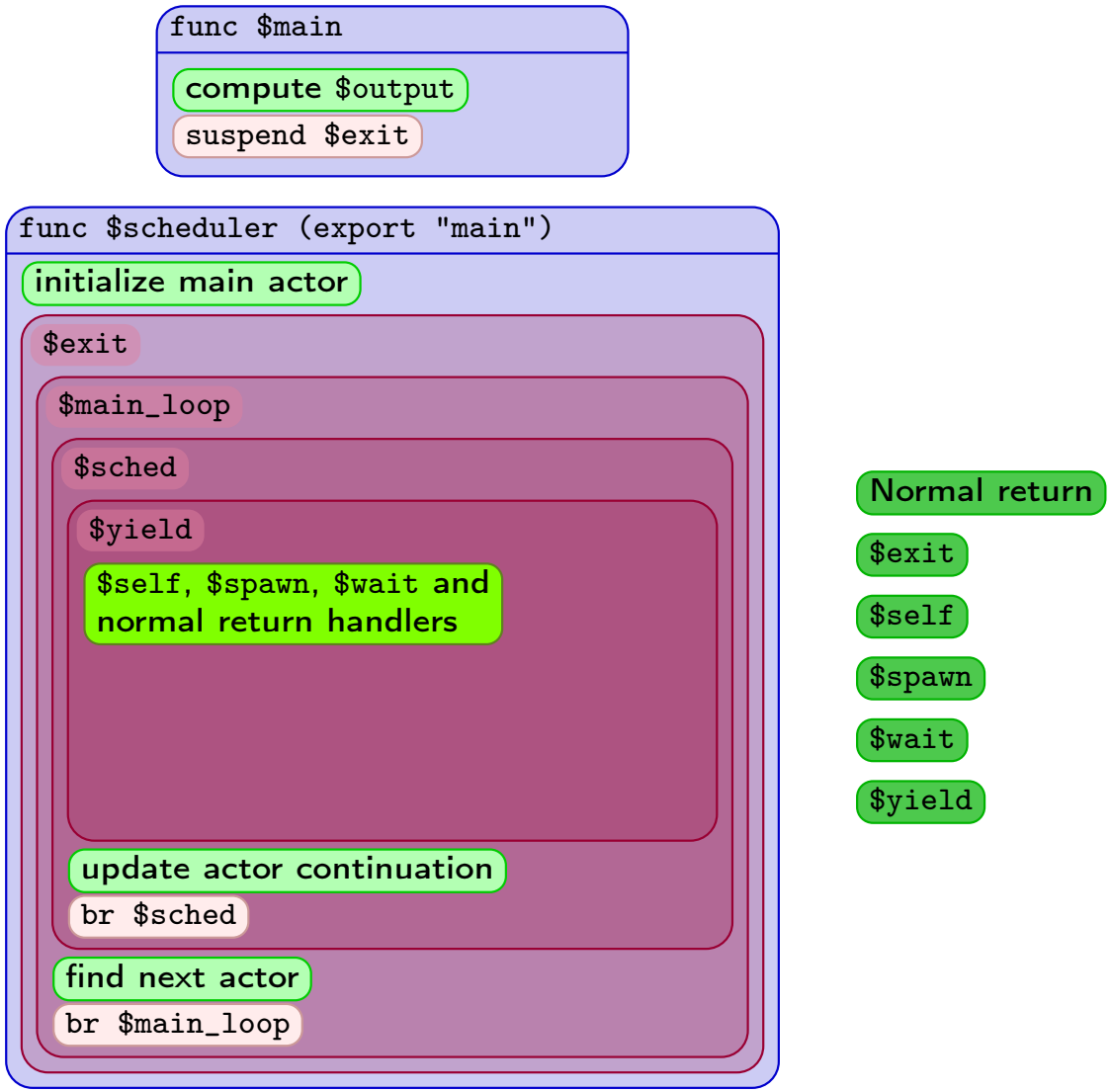
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$exit
$main_loop
$sched
$spawn
$self, $wait and
normal return handlers
update actor continuation
add new representation after $active
br $main_loop
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

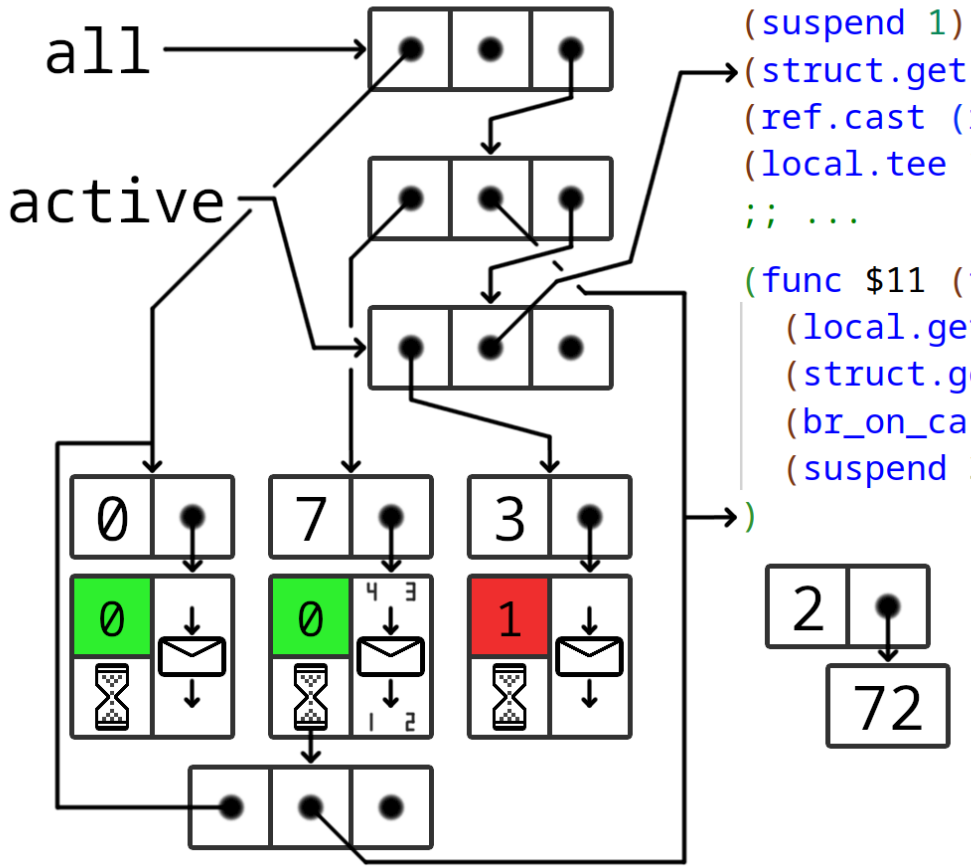
Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$exit
$main_loop
$sched
$self, $spawn, $wait and normal return handlers
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

Actor logic


func $main
compute $output
suspend $exit
func $scheduler (export "main")
initialize main actor
$exit
$main_loop
$sched
$yield
$self, $spawn, $wait and normal return handlers
update actor continuation
br $sched
find next actor
br $main_loop
Normal return
$exit
$self
$spawn
$wait
$yield

Actor logic




```
(suspend 1)
(struct.get 6 1)
(ref.cast (ref 12))
(local.tee 1)
;; ...

(func $11 (type 33)
  (local.get 0)
  (struct.get 6 1)
  (br_on_cast_fail 0 (ref null eq) (ref 12))
  (suspend 3)
)
```

Actor effects

- Send a message: find the actor representation, push the message to the mailbox

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox or block until a message is sent

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox or block until a message is sent
- Spawn a new actor: create a new actor representation, add it after `active`

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox or block until a message is sent
- Spawn a new actor: create a new actor representation, add it after `active`
- Wait on an actor to finish: `suspend` to the scheduler, push the continuation to the waiting list

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox or block until a message is sent
- Spawn a new actor: create a new actor representation, add it after `active`
- Wait on an actor to finish: `suspend` to the scheduler, push the continuation to the waiting list
- Exit: `suspend` to the scheduler with the `$exit` tag, exiting the main loop

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox or block until a message is sent
- Spawn a new actor: create a new actor representation, add it after `active`
- Wait on an actor to finish: `suspend` to the scheduler, push the continuation to the waiting list
- Exit: `suspend` to the scheduler with the `$exit` tag, exiting the main loop
- Get the active actor: `suspend` to the scheduler, get the active actor representation from `active`

Actor effects

- Send a message: find the actor representation, push the message to the mailbox
- Receive a message: get `self()`, pop a message from the mailbox or block until a message is sent
- Spawn a new actor: create a new actor representation, add it after `active`
- Wait on an actor to finish: `suspend` to the scheduler, push the continuation to the waiting list
- Exit: `suspend` to the scheduler with the `$exit` tag, exiting the main loop
- Get the active actor: `suspend` to the scheduler, get the active actor representation from `active`
- Yield: `suspend` to the scheduler

Actor effects optimizations

Optimizations: if `active` is available globally, no need for `spawn` or `self` effects

Actor effects optimizations

Optimizations: if `active` is available globally, no need for `spawn` or `self` effects: `wasm_prefer_globals`

Actor effects optimizations

Optimizations: if `active` is available globally, no need for `spawn` or `self` effects: `wasm_prefer_globals`

If `active` and `all` are available globally, we can schedule everywhere

Actor effects optimizations

Optimizations: if `active` is available globally, no need for `spawn` or `self` effects: `wasm_prefer_globals`

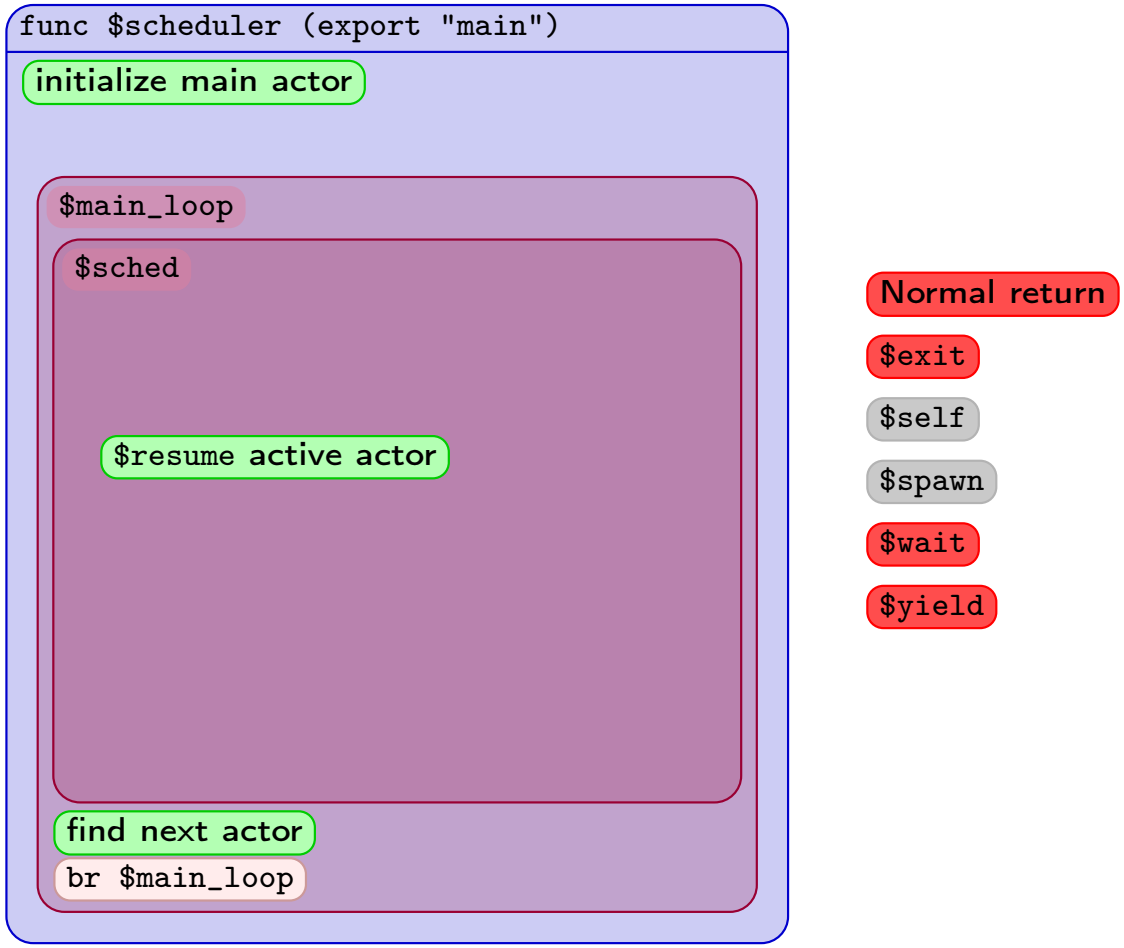
If `active` and `all` are available globally, we can schedule everywhere `switch`: instead of suspending then reinstalling the same handler, context switch to the other continuation directly

Actor effects optimizations

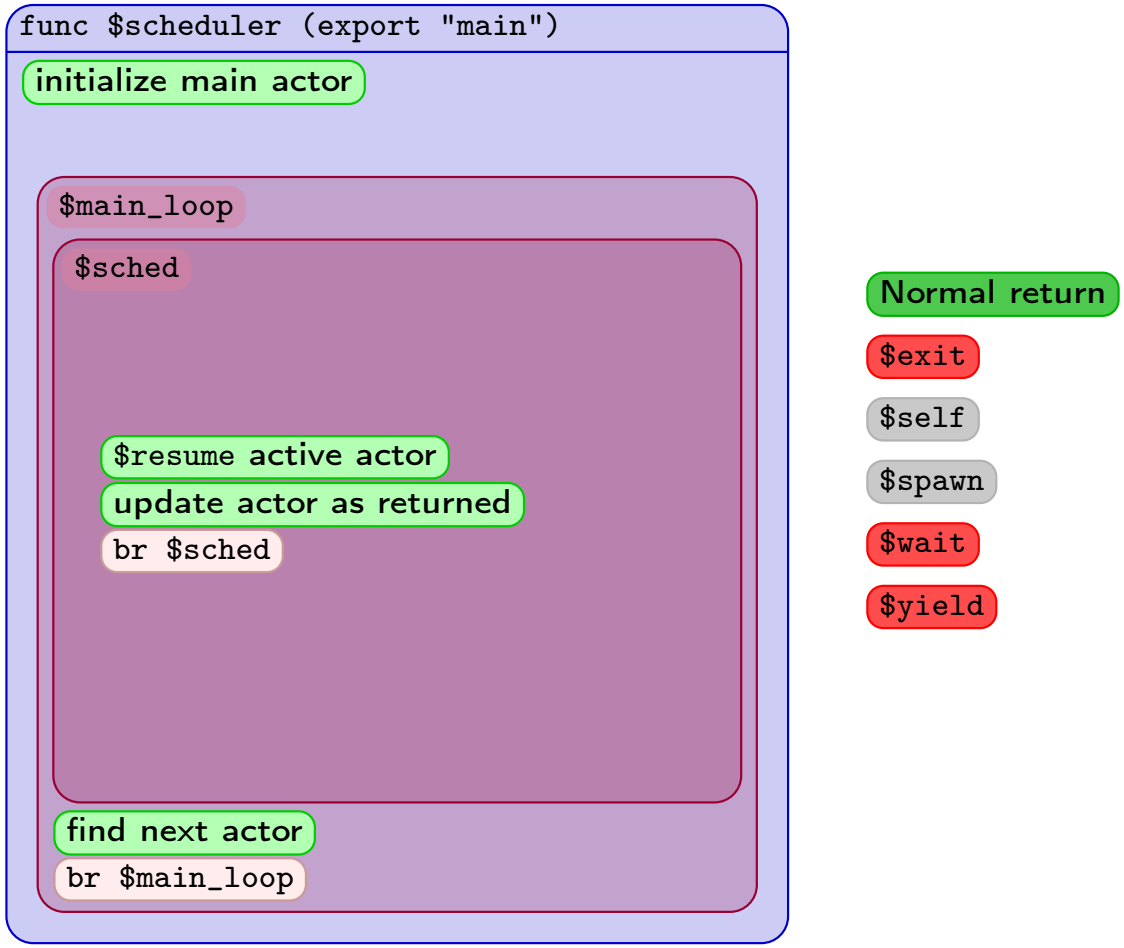
Optimizations: if `active` is available globally, no need for `spawn` or `self` effects: `wasm_prefer_globals`

If `active` and `all` are available globally, we can schedule everywhere `switch`: instead of suspending then reinstalling the same handler, context switch to the other continuation directly
`wasm_yield_is_switch`: We can try to use this instead of `suspend` for `yield`: move the AAL, CAAP and scheduling to the global scope, then call that function and `switch` to the next actor instead of raising an effect to the scheduler function

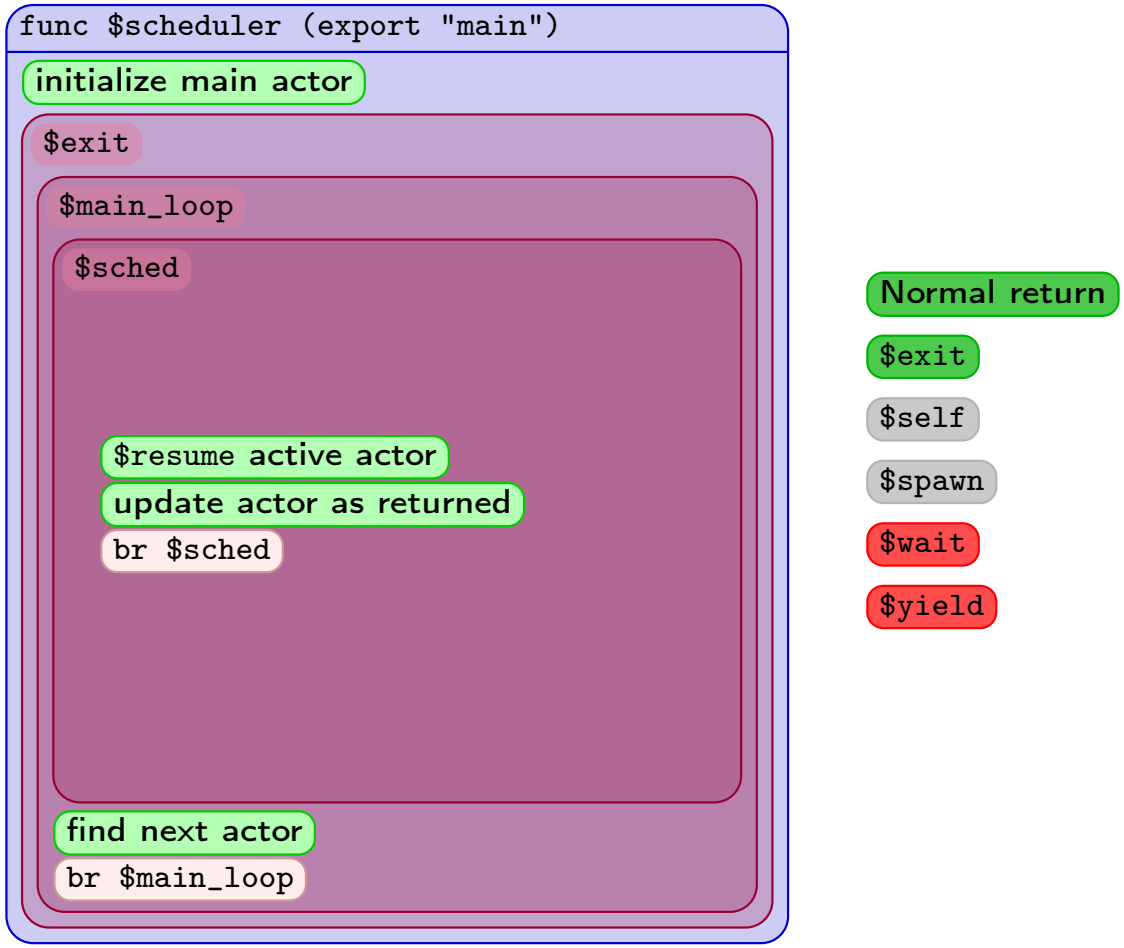
Actor logic (prefer globals)



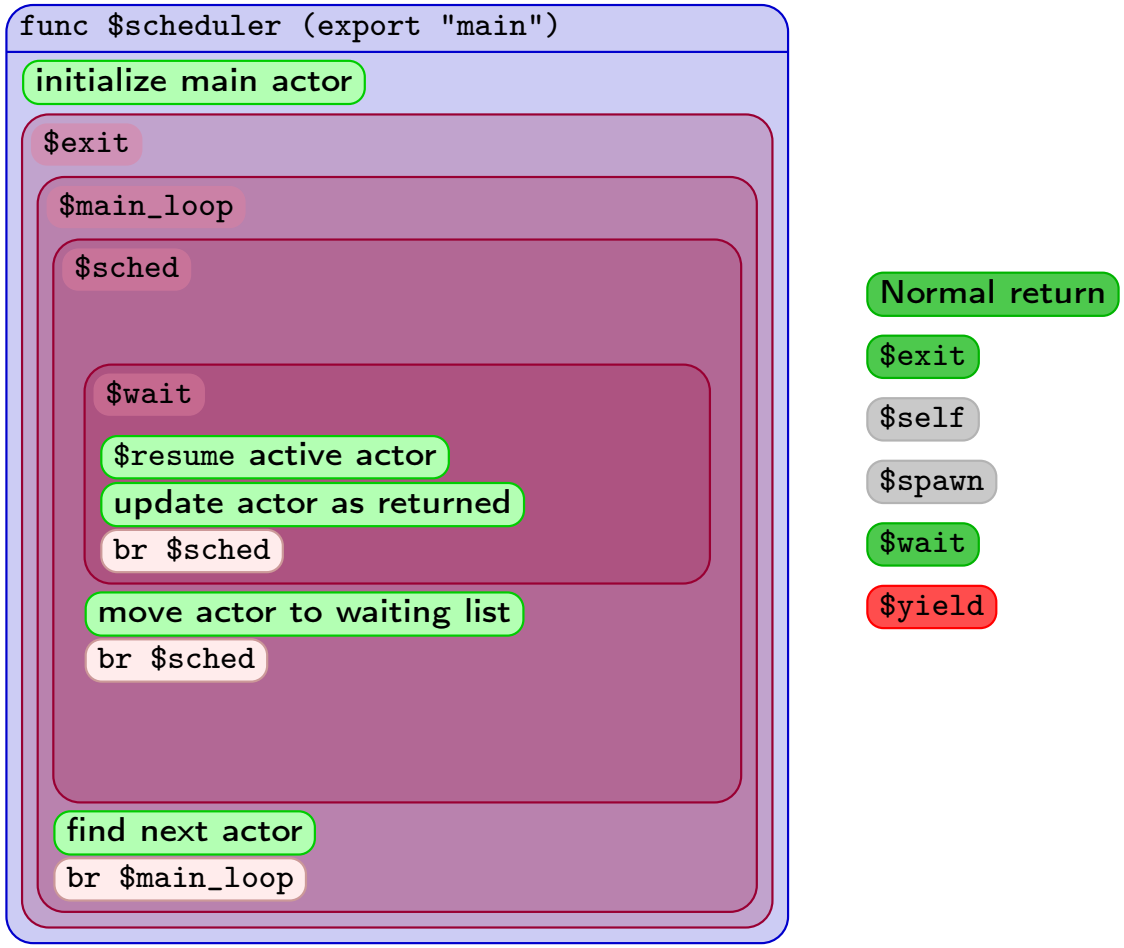
Actor logic (prefer globals)



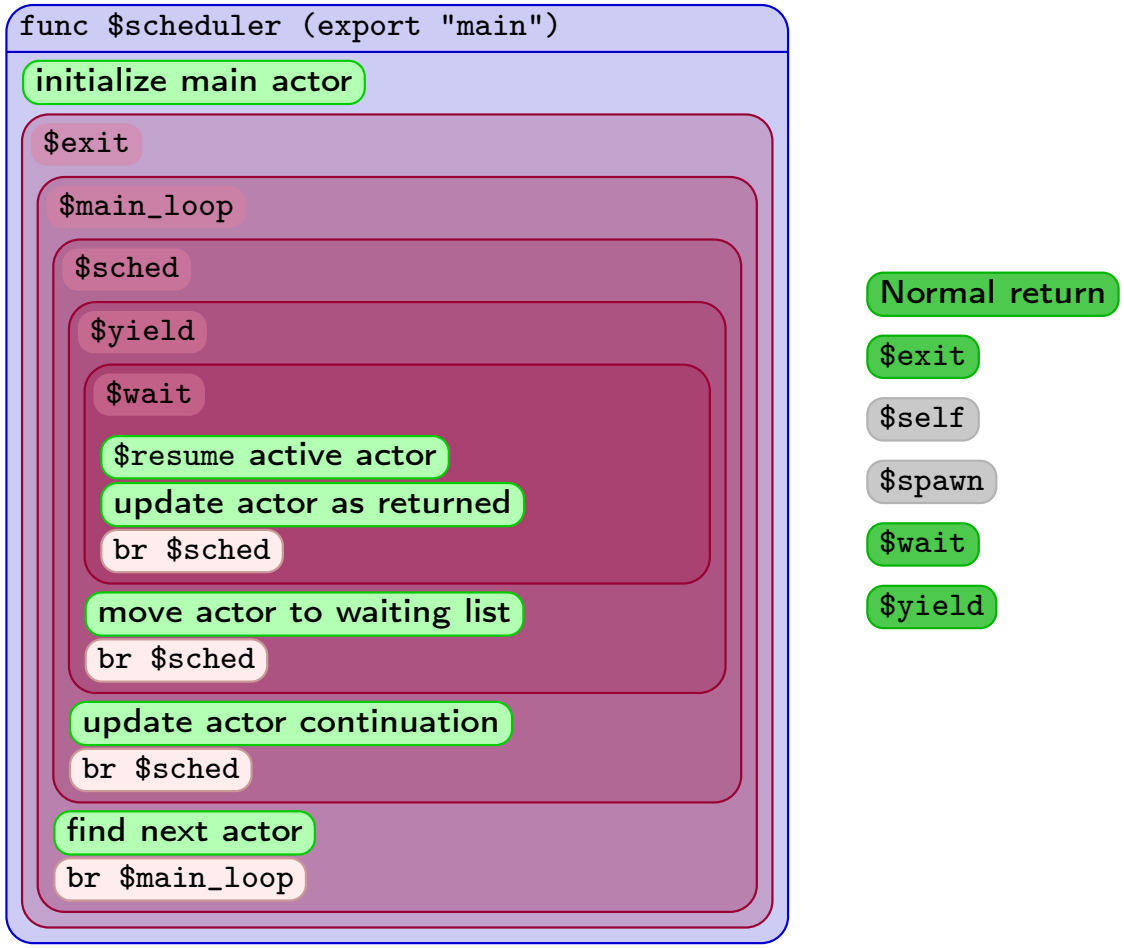
Actor logic (prefer globals)



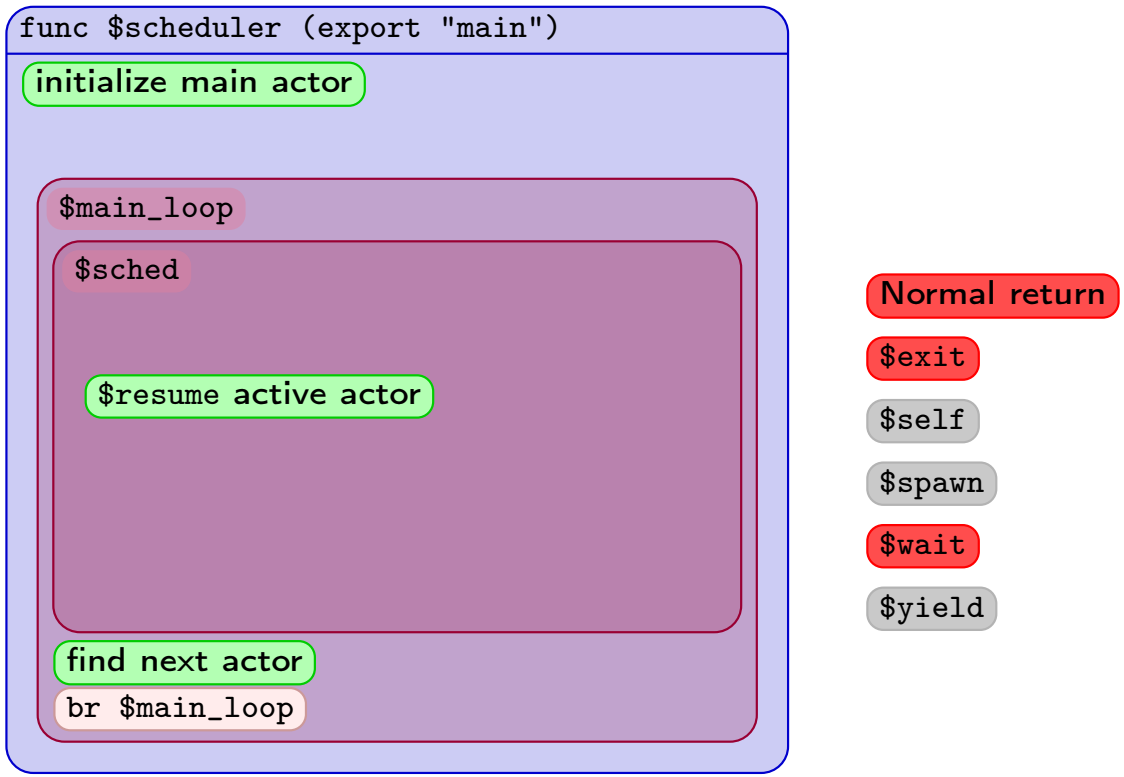
Actor logic (prefer globals)



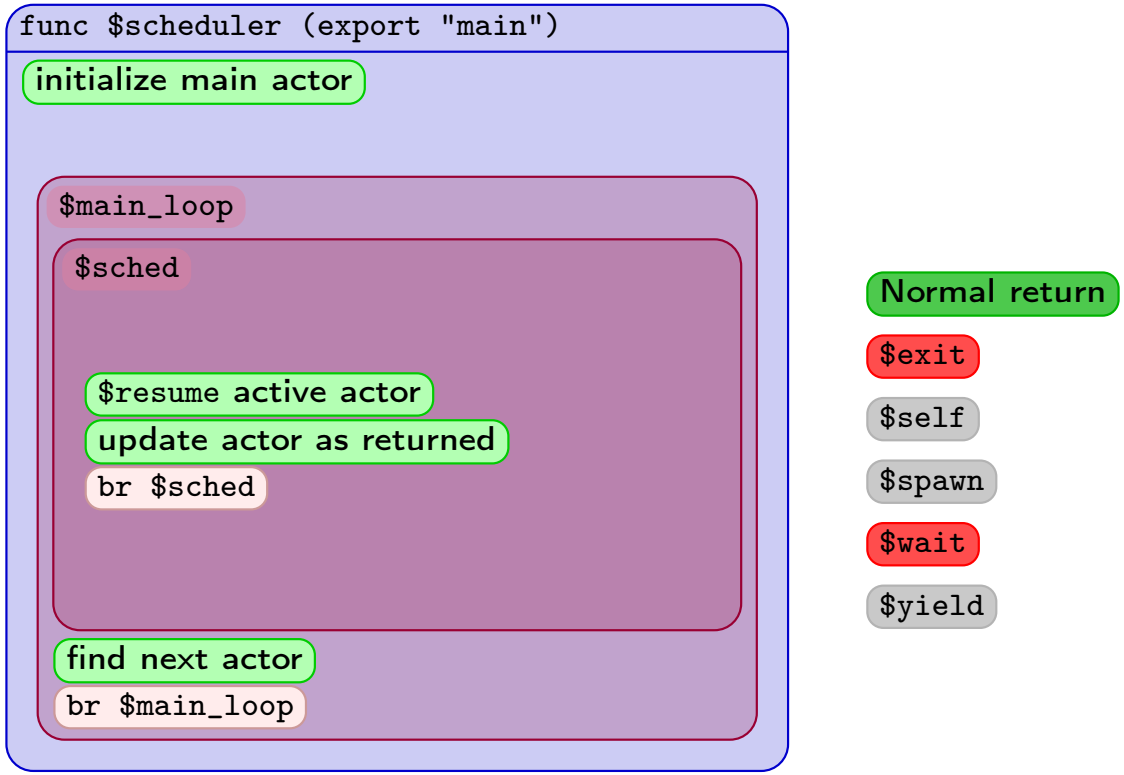
Actor logic (prefer globals)



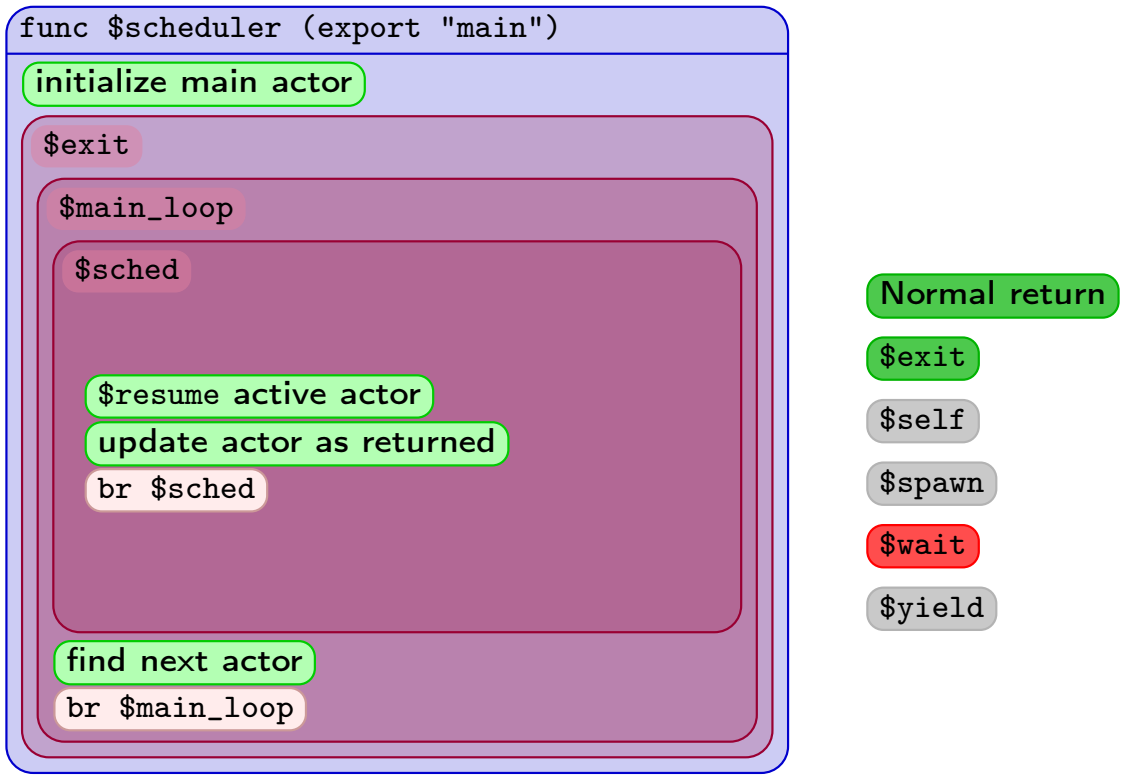
Actor logic (prefer `switch`)



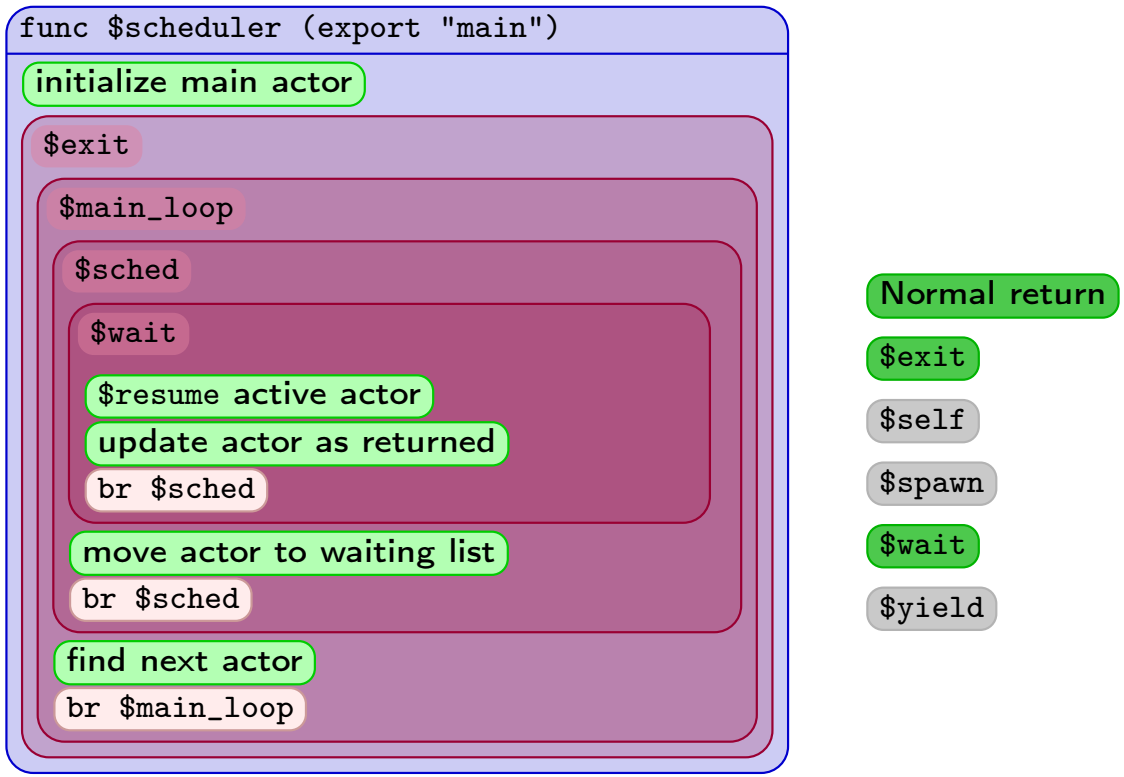
Actor logic (prefer `switch`)



Actor logic (prefer `switch`)



Actor logic (prefer `switch`)



Angel actors

Angel actors: block process termination

Angel actors

Angel actors: block process termination
`spawnAngel`: like `spawn`, but add the new process to a global angel list
Main function: after computing `$output`, wait for all angels to finish; only then `suspend $exit`

Some numbers

Test	LoC	LoC (Wat)	Size (bytes, Wasm)
Deep handlers example	21	207	601
(with output)		264	733
(force actors)		403	1141
(+ output)		460	1272
Actors example	34	718	2047
(with output)		807	2257

References I

[1] Sam Lindley and James Cheney. "Row-based effect types for database integration". In: *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*. TLDI '12. Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 2012, pp. 91–102. ISBN: 9781450311205. DOI: 10.1145/2103786.2103798. URL: https://doi.org/10.1145/2103786.2103798.

[2] Frank Emrich et al. "FreezeML: Complete and Easy Type Inference for First-Class Polymorphism". In: *CoRR* abs/2004.00396 (2020). arXiv: 2004.00396. URL: https://arxiv.org/abs/2004.00396.

[3] *The Links Programming Language*. URL: https://links-lang.org/.

[4] *WasmFX: Effect Handlers for WebAssembly*. URL: https://wasmfx.dev/.

[5] *WebAssembly*. URL: https://webassembly.org/.