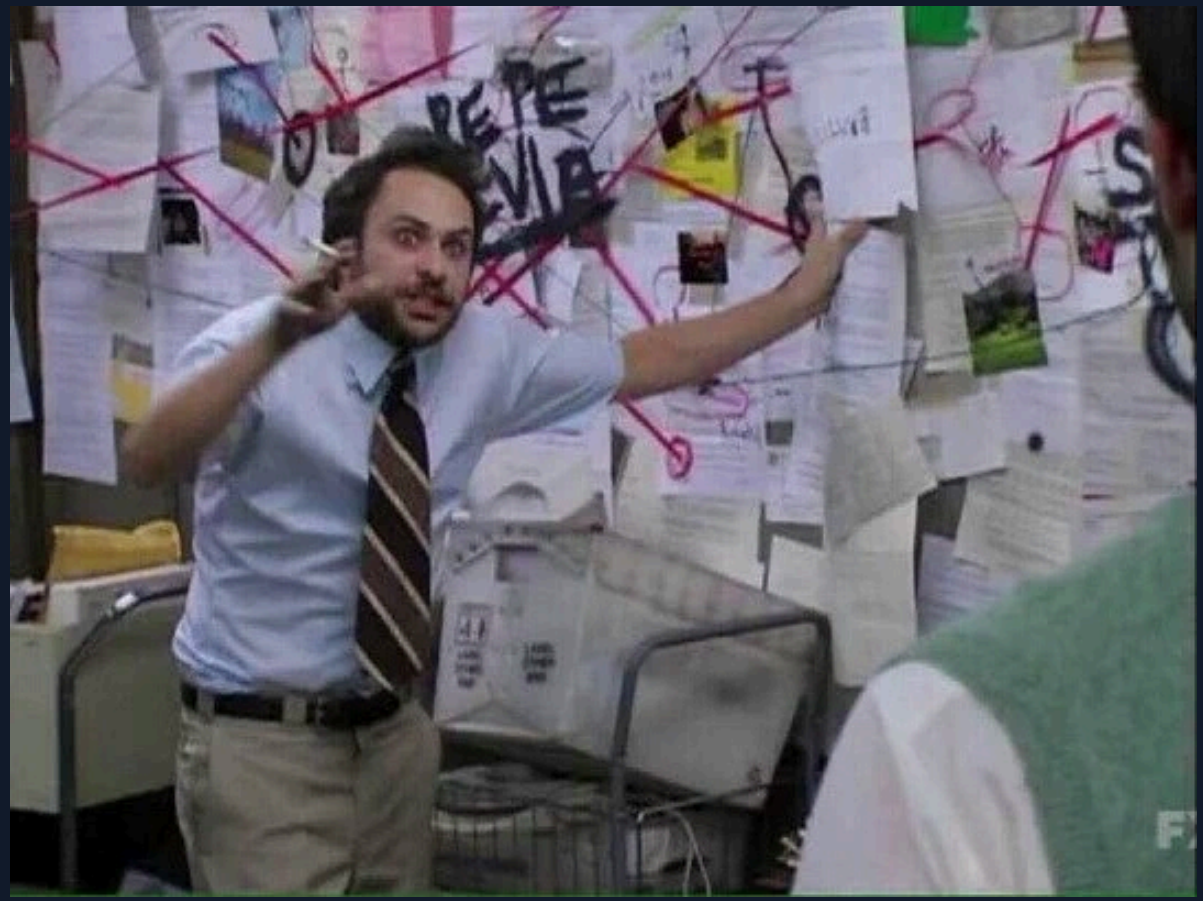
Trampolining Nix

How I nearly lost my mind
fighting the evaluator

Mika Bohinen - NixCon 2026

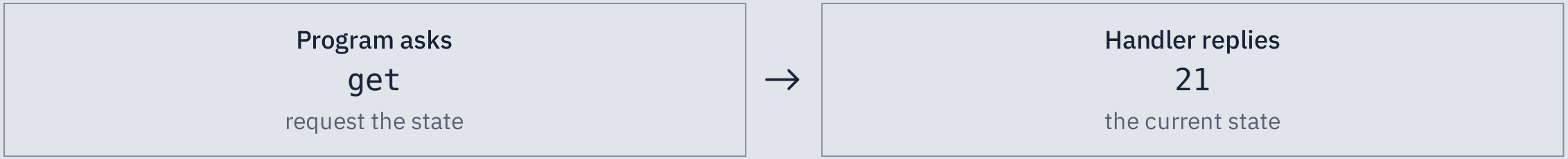


The program asks for state; the handler supplies it.

An effect is a request whose meaning is supplied by a handler

```
get
```

An operation and the handler that answers it

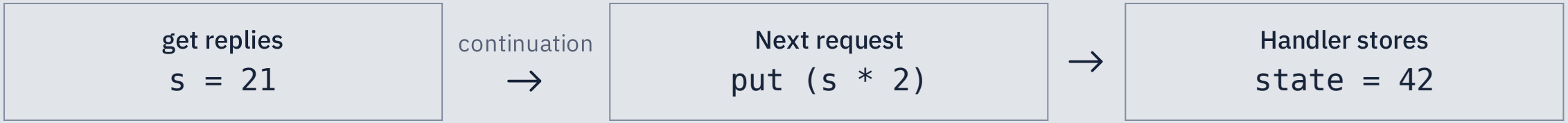


bind passes that reply to the rest of the program.

A continuation: the function to run after get replies

```
s: put (s * 2)
```

The continuation turns the reply into the next request



The program changes the state, then returns the original value.

Read state, store twice the value, return the original

```
bind get (s:
  bind (put (s * 2)) (_:
    pure s))
```

Follow the requests and replies

The state ends at 42; the program returns 21.

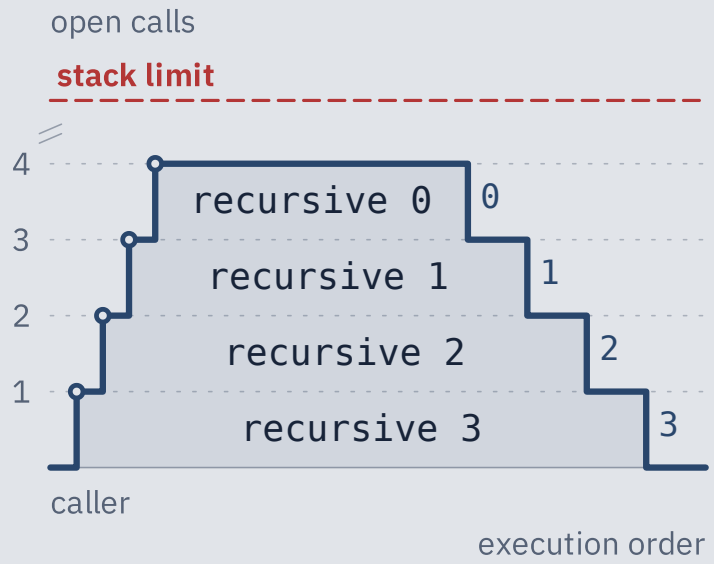
nix-effects / README.md

Each recursive return still owes an addition.

Trace: recursive 3. Recorded run: n = 100000.

```
recursive = count:
  if count == 0 then 0
  else 1 + recursive (count - 1);
```

recursive 3 nests four calls before any addition

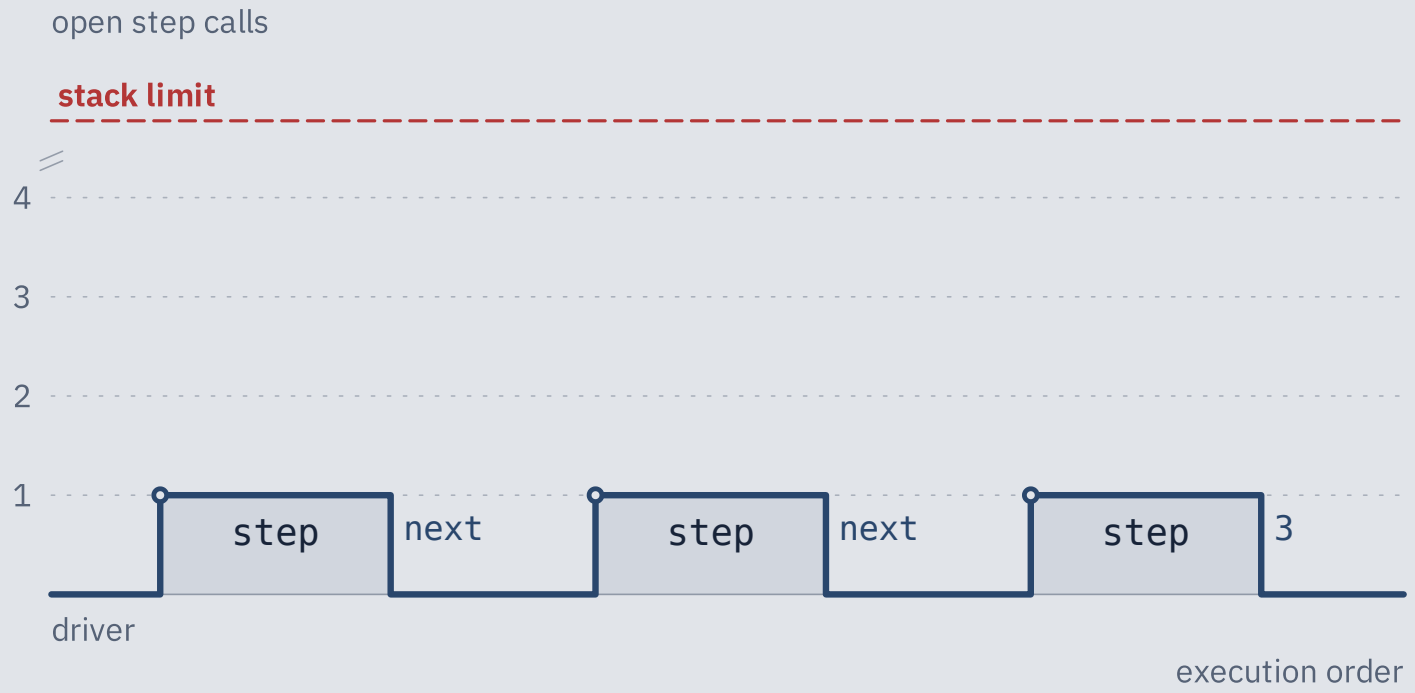


Illustrative trace; spacing shows order, not elapsed time.

recursive n	error: stack overflow (possible infinite recursion)
Recorded result	

More steps. The same nesting depth.

Every step gives control back to the driver



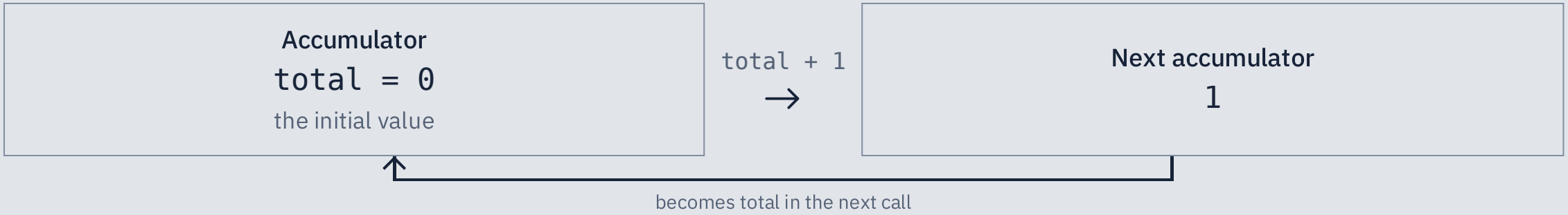
Same scale as the recursive trace: more steps, but the depth stays at one.

A fold passes the running total to this function.

One addition per input

```
total: _: total + 1
```

One call, starting from zero



A strict scalar fold already solves this fixed counter.

scalarFold, with n = 100000

```
builtins.foldl'
  (total: _: total + 1)
  0
  (builtins.genList (i: i) n)
```

scalarFold	100000

Recorded result

The operator chooses a successor state or ends the traversal.

genericClosure owns the loop; operator chooses the next work

The result lists every visited record. Keys prevent revisiting the same identity.

Start with one record: an identity and a running total.

genericClosure input; total stands in for the handler's state

```
startSet = [{ key = 0; total = 0; }];
```

One state, two jobs

key = 0	**total = 0**
Identity used by the worklist	Payload used by our counter

Build the successor from the current record, item.

Inside the operator: the next total, then the successor record

```
next = { total = item.total + 1; };
{ key = item.key + 1; inherit (next) total; }
```

The first call to operator

Current item

`key = 0`

`total = 0`

the seed

operator →

Successor

`key = item.key + 1`

`inherit (next) total`

no addition has run yet

Return no successors when the key reaches the limit.

Operator excerpt; the else branch returns the successor in a list

```
operator = item:
  if item.key >= n then [] else
```

The operator returns a list

Condition		operator returns	Effect
`item.key < n`	→	`[ successor ]`	one more state
`item.key >= n`	→	`[]`	this branch stops

n = 100000. The seed has key 0; the last visited record has key n.

naive names the returned list, including the initial state.

Combine the seed and operator we just built

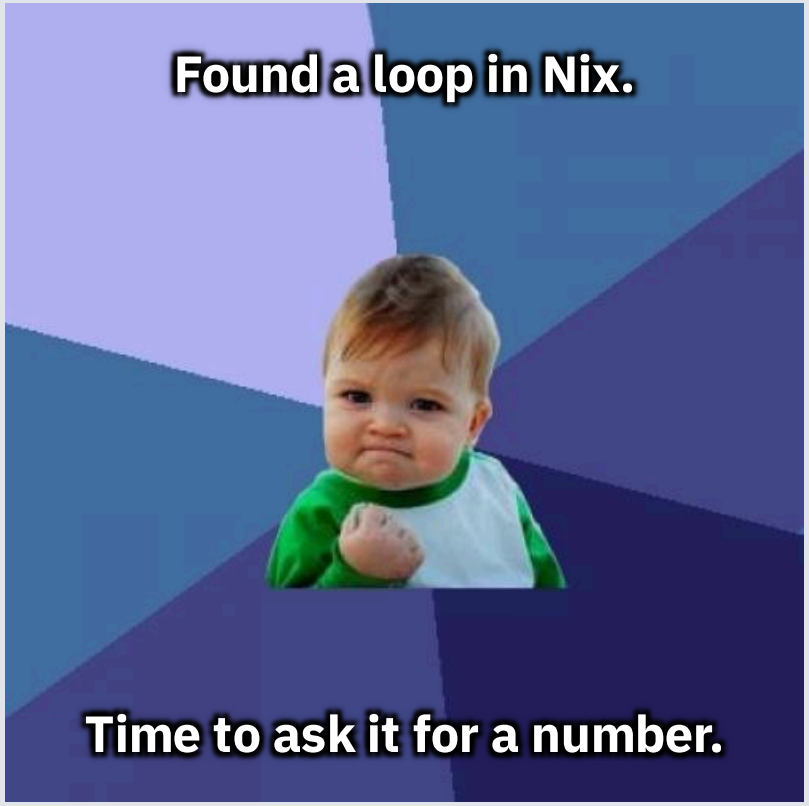
```
naive = builtins.genericClosure {
  inherit startSet operator;
};
```

The returned list is named naive

First element	Second element	...	Last element
`key = 0`	`key = 1`		`key = n`
the seed			selected by last naive

`builtins.length naive` and `(last naive).total` are the next two questions.

Surely the hard part is over.



The same program succeeds or overflows depending on what I ask for.

naive is the list of states returned by my first attempt at a counter.

How many states were visited?

```
builtins.length naive
```

```
100001
```

Recorded result

What is the final total?

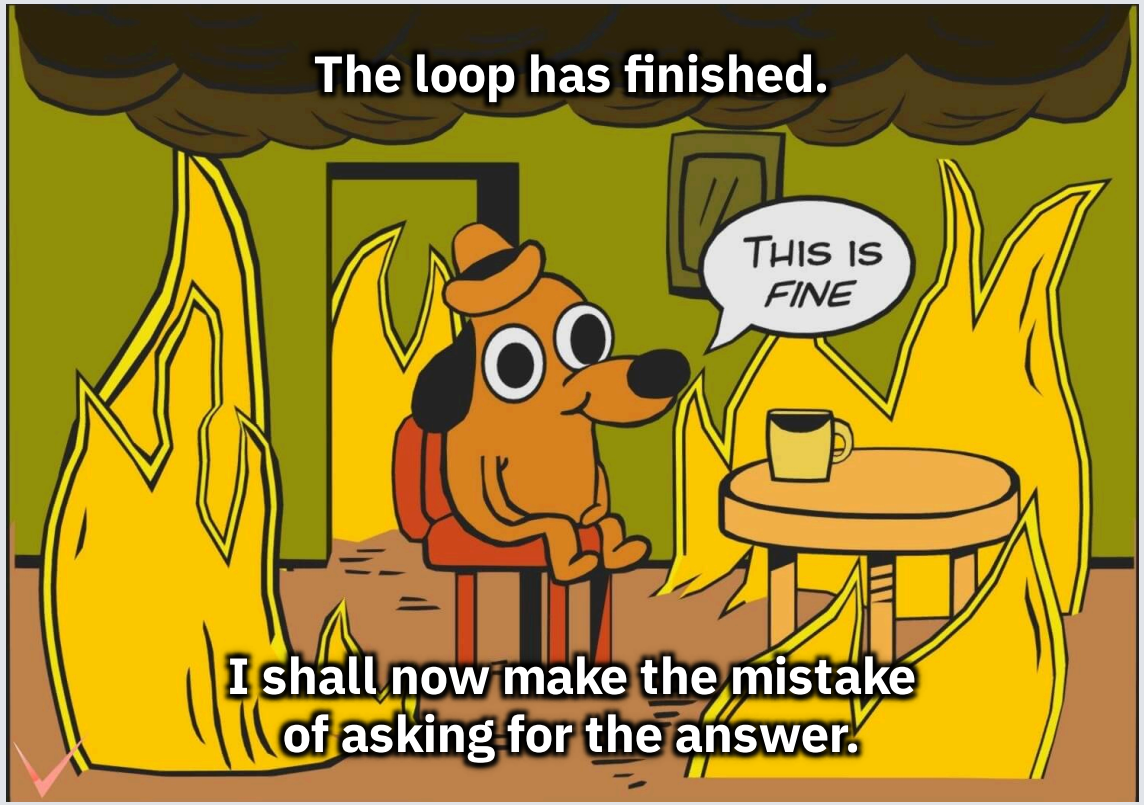
```
(last naive).total
```

```
error: stack overflow (possible
infinite recursion)
```

Recorded result

Counting the visited states tells us nothing about whether their totals have been computed.

I had a rather generous definition of finished.

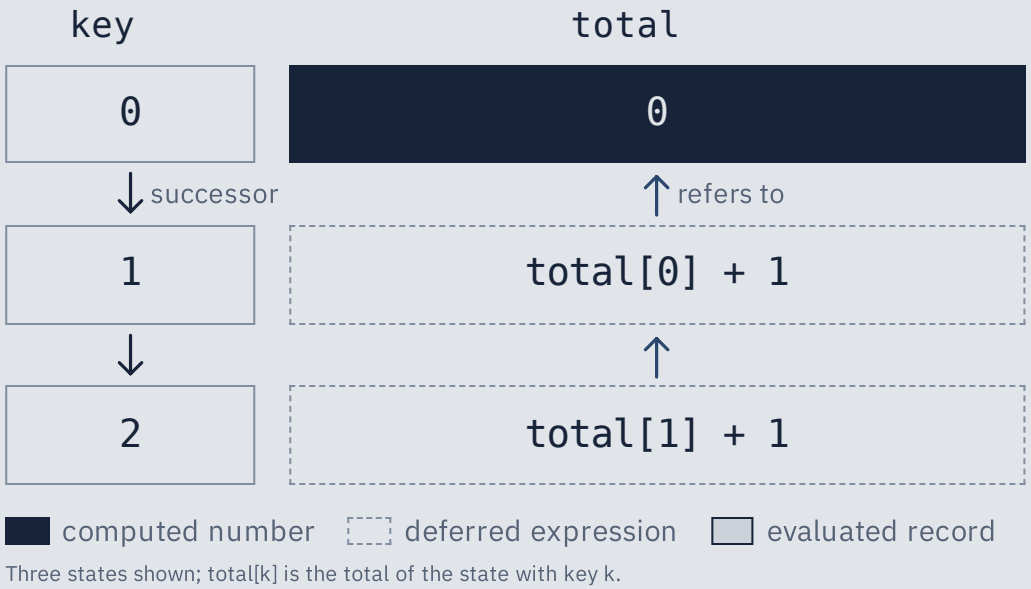


The key and the total have separate demand paths.

The two field expressions, extracted from the operator

```
next.total = item.total + 1;
key = item.key + 1;
```

Keys move forward; each total refers back



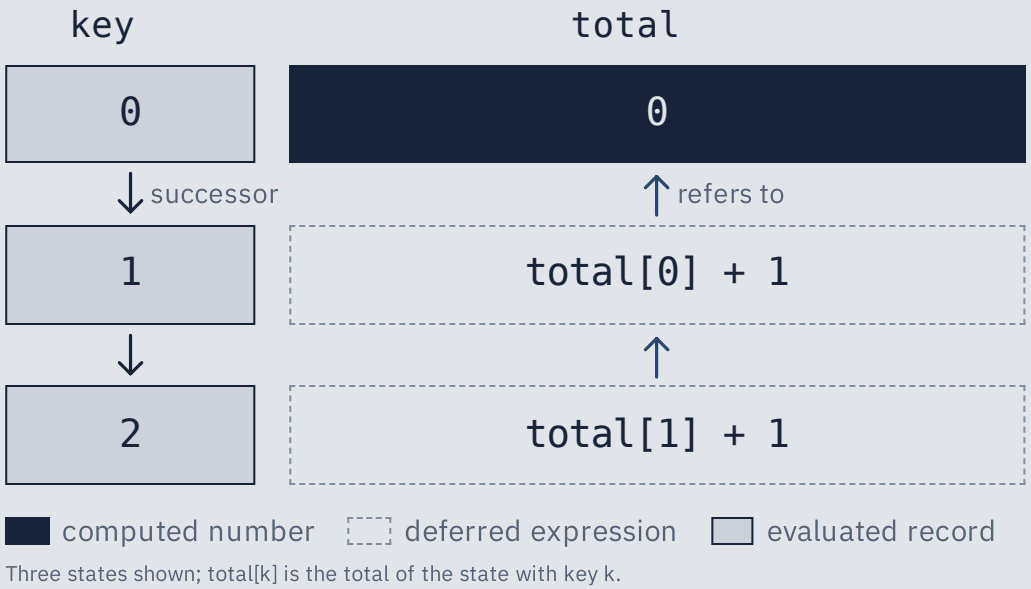
Three states shown; total[k] is the total of the state with key k.

Counting visited states does not demand their totals.

The two field expressions, extracted from the operator

```
next.total = item.total + 1;
key = item.key + 1;
```

Visiting evaluates each record and its key



Three states shown; total[k] is the total of the state with key k.

builtins.length naive	100001

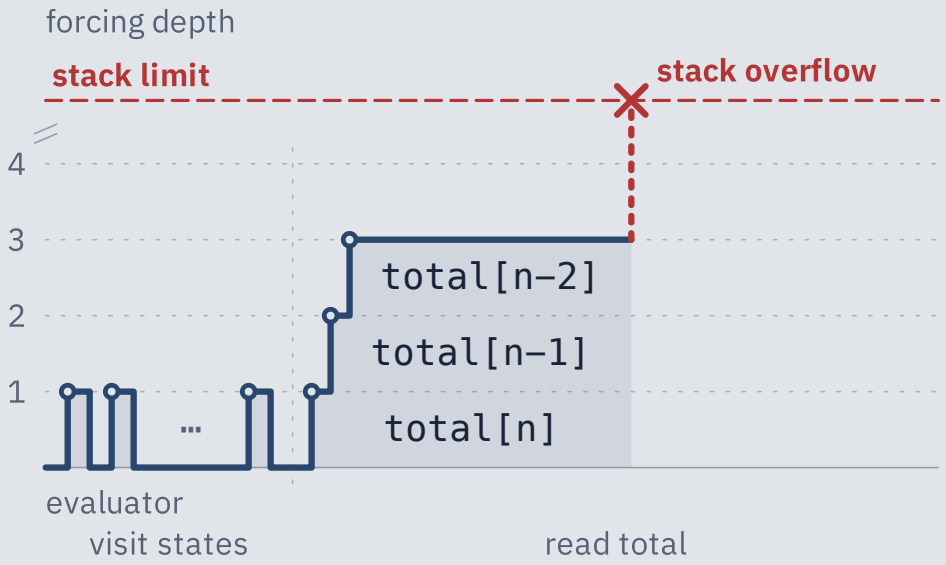
Recorded result

The final total depends on earlier deferred additions.

The two field expressions, extracted from the operator

```
next.total = item.total + 1;
key = item.key + 1;
```

Reading the final total climbs through every deferred addition



Illustrative trace for n steps; the recorded run overflows at n = 100000.

```
(last naive).total                                  error: stack overflow (possible infinite recursion)
```

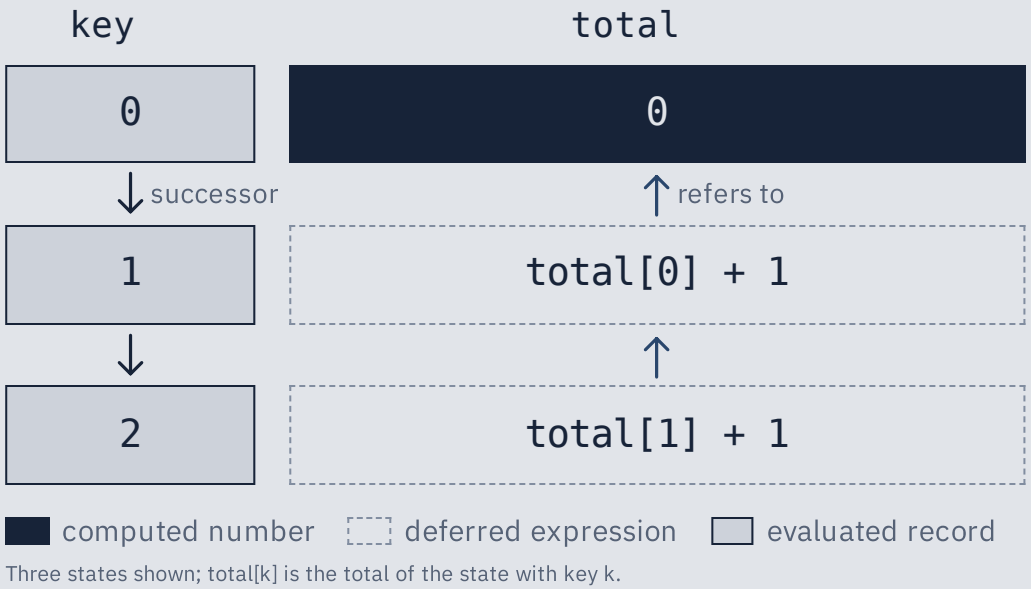
Recorded result

Forcing the record leaves its total field deferred.

shallow: the same traversal, forcing next

```
next.total = item.total + 1;
key = builtins.seq next
  (item.key + 1);
```

seq next evaluates the record, not its total



Three states shown; total[k] is the total of the state with key k.

```
(last shallow).total                    error: stack overflow (possible infinite recursion)
```

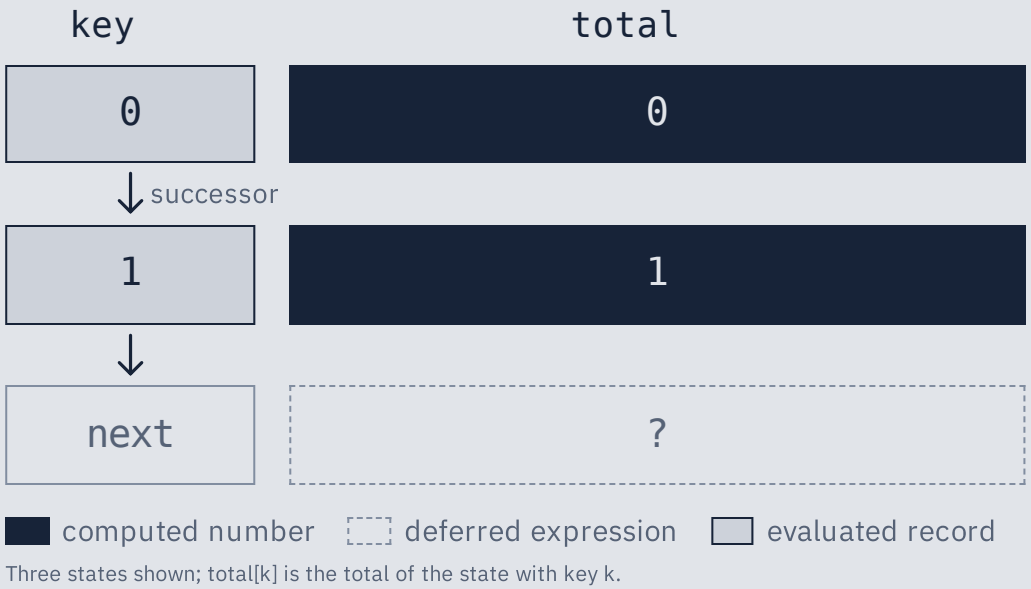
Recorded result

The next key becomes available after next.total has been computed.

The two field expressions, extracted from the operator

```
next.total = item.total + 1;
key = builtins.seq next.total
  (item.key + 1);
```

First transition: compute 1, then expose key 1



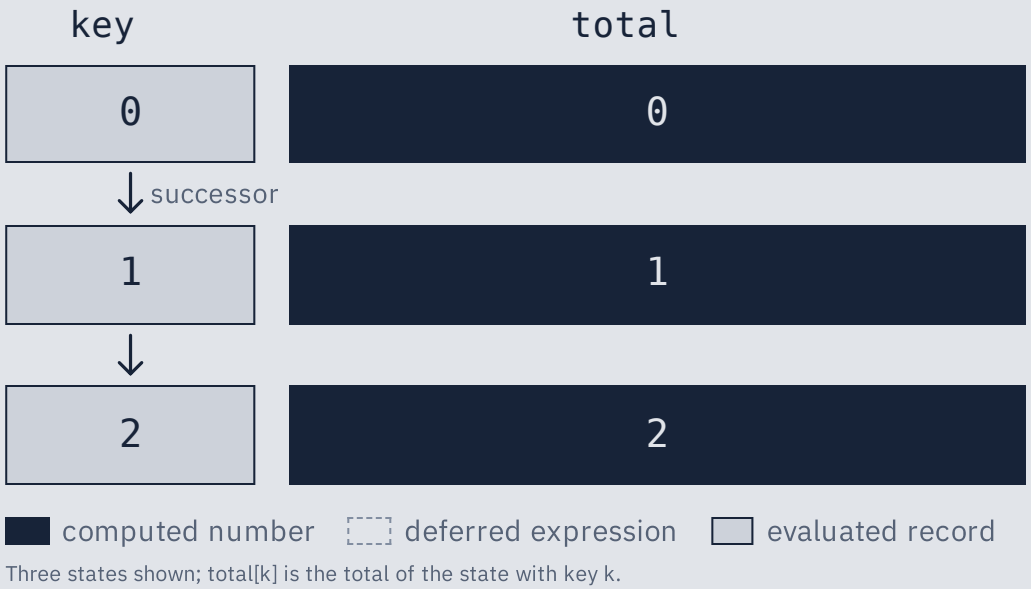
Three states shown; total[k] is the total of the state with key k.

The following step receives a total that has already been computed.

The two field expressions, extracted from the operator

```
next.total = item.total + 1;
key = builtins.seq next.total
  (item.key + 1);
```

Each transition carries a computed number



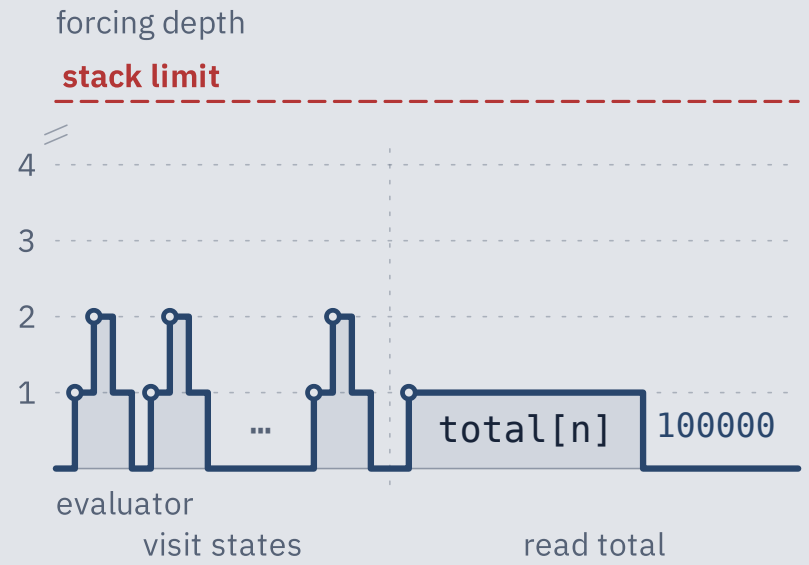
Three states shown; total[k] is the total of the state with key k.

Asking for the final total now succeeds with the same input and limits.

targeted: the same traversal, forcing next.total

```
next.total = item.total + 1;
key = builtins.seq next.total
  (item.key + 1);
```

Each step computes its total before the next begins



Illustrative trace for n steps; the recorded run returns 100000.

```
(last targeted).total                                    100000
```

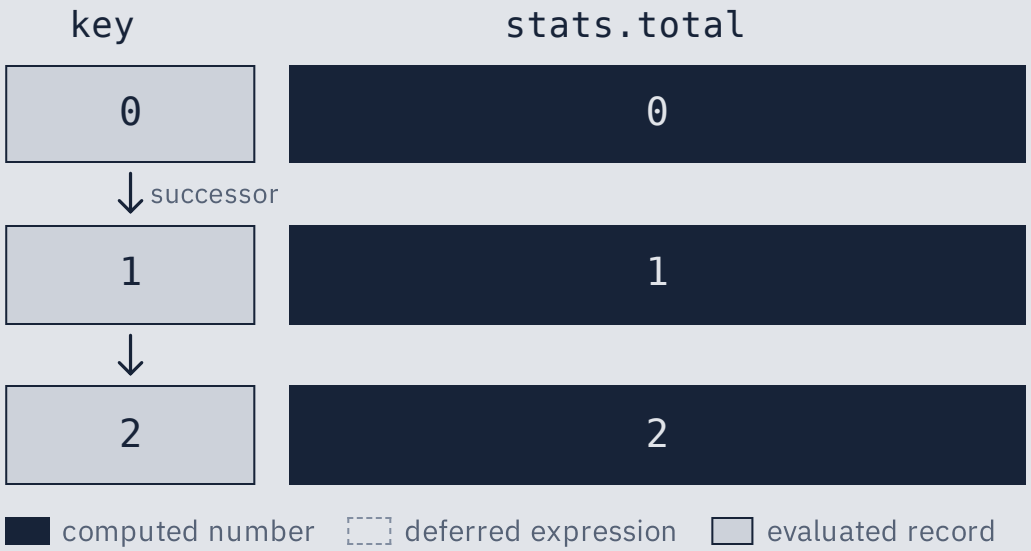
Recorded result

Computing next.stats.total at each step breaks the nested dependency chain.

nestedTargetedResult: final total after forcing stats.total

```
next.stats.total = item.stats.total + 1;
key = builtins.seq next.stats.total
  (item.key + 1);
```

Each transition carries a computed number



Three states shown; stats.total[k] is the stats.total of the state with key k.

nestedTargetedResult	100000

Recorded result

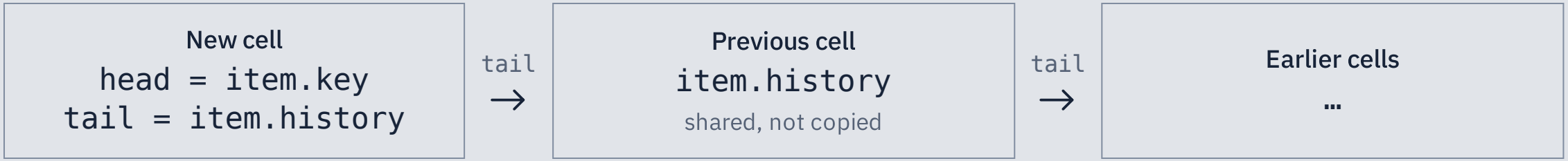
Why not force the whole state?

Each new history cell points to the old history.

The new history field inside next

```
history = {
  head = item.key;
  tail = item.history;
};
```

The new cell retains the existing chain



Extending adds one cell. Deep traversal walks every link again.

Forcing a record, a field, or the whole structure does different work.

next now carries a history

```
next = {
  total = item.total + 1;
  history = { head = item.key; tail = item.history; };
};
```

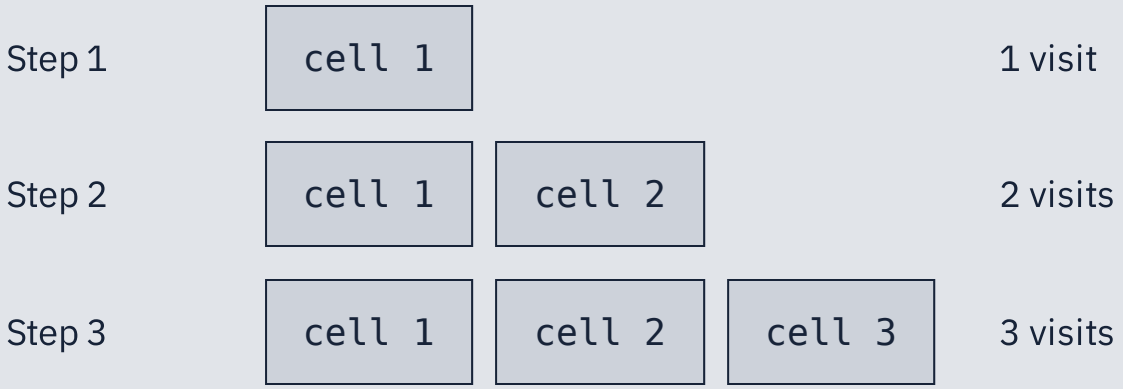
The same state, three demand boundaries

Demand	Evaluates	Still deferred
`seq next`	the record	total, history
`seq next.total`	total	history
`deepSeq next`	total and every history cell	nothing

Forcing a function value does not execute its body.

Repeated deep traversal revisits the growing prefix.

A traversal of the whole retained chain after every extension




```
1 + 2 + 3 = 6 visits
```

n steps cost n(n + 1) / 2 visits: about 5 billion at n = 100000.

This interpreter deliberately demands its next state.

nix-effects interpreter

```
k = builtins.deepSeq newState (step.key + 1);
```

nix-effects / src/trampoline.nix

The forcing policy should follow the state you carry.

Handler state

```
deepSeq newState
```

A deliberately broad policy for carried state.

nix-effects / src/tc/eval/core.nix

Concrete syntax walk

```
key = item.key + 1;
```

A different workload avoids unnecessary traversal.

This function remembers which function to call first.

step wraps previous in a closure

```
f0 = x: x;
step = previous: x: previous x + 1;
```

Constructing a function that remembers previous

Call
`step f0`

returns →

A new function
`x: previous x + 1`
remembers previous = f0

Each new function remembers the one before it.

Three wrappers; no argument has been supplied to f3

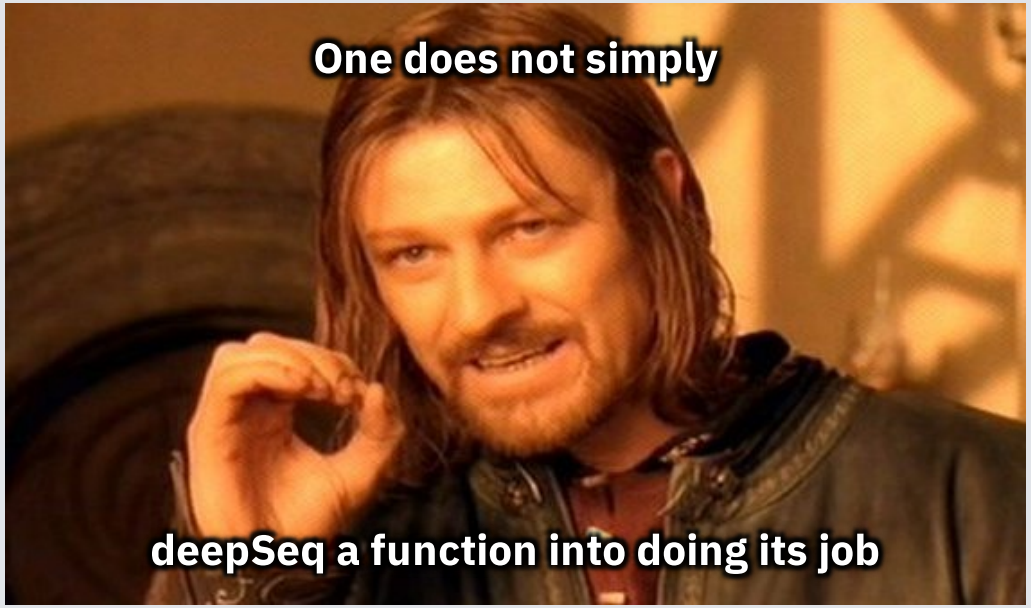
```
f1 = step f0;
f2 = step f1;
f3 = step f2;
```

Arrows are captured references, not calls

A function value can be ready while its body still has work.

closureChain: the same construction, repeated 100000 times

```
builtins.deepSeq closureChain true
```




```
builtins.deepSeq closureChain true                    true
```

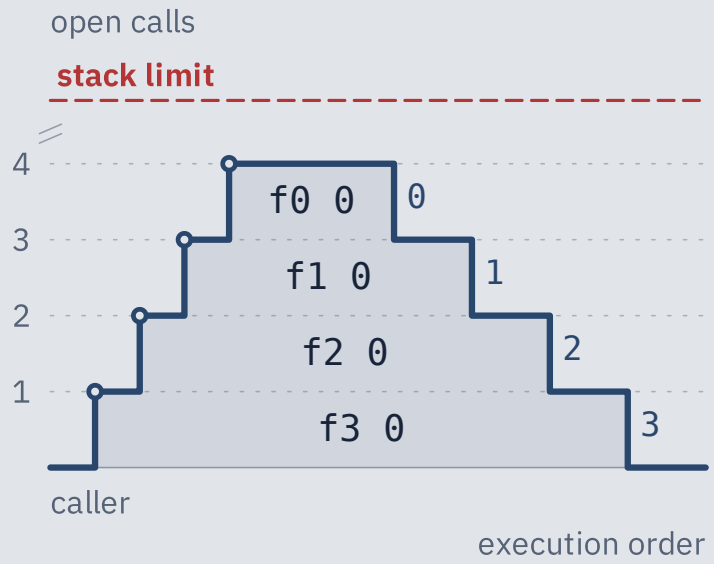
Recorded result

Applying the function enters the remembered calls.

Run the larger construction

```
closureChain 0
```

f3 0 nests four calls before any addition

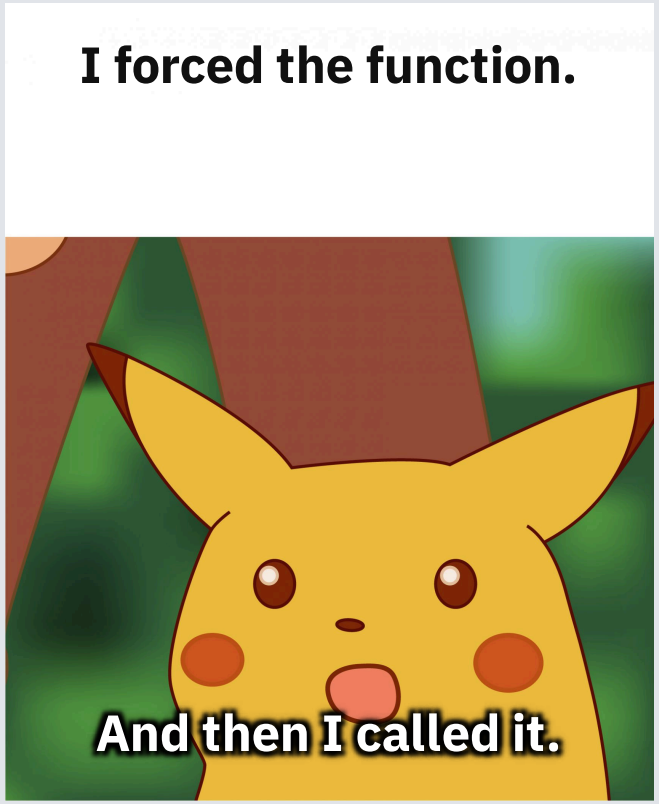


Illustrative trace; spacing shows order, not elapsed time.

```
closureChain 0                                  error: stack overflow (possible infinite recursion)
```

Recorded result

The recursion has moved inside the callback.



These pending additions can be represented as data.

Preserve the pending operation and its order

Function body

```
x: previous x + 1
```

previous is the remembered function. The pending operation is + 1.

Explicit pending operation

```
{ tag = "add"; amount = 1; }
```

Store the operation for the dispatcher to perform. Written add 1 below.

Where did previous go? Its links become the order of the frame list.

First: f1's addition	Then: f2's addition	Last: f3's addition
`add 1`	`add 1`	`add 1`

Defunctionalization

The dispatcher reads one frame and performs its operation.

apply: the dispatcher for add frames

```
apply = value: frame:
  if frame.tag == "add"
  then value + frame.amount
  else throw "Unknown frame";
```

Hand trace: one frame

Current value	→	apply reads the frame	→	New value
`0`		`tag = "add"` `amount = 1`		`1` value + frame.amount

The dispatcher performs the same three additions.

frames holds the three add records

```
builtins.foldl' apply 0 frames
```

The same computation, one explicit operation at a time


```
f3 0                          3
```

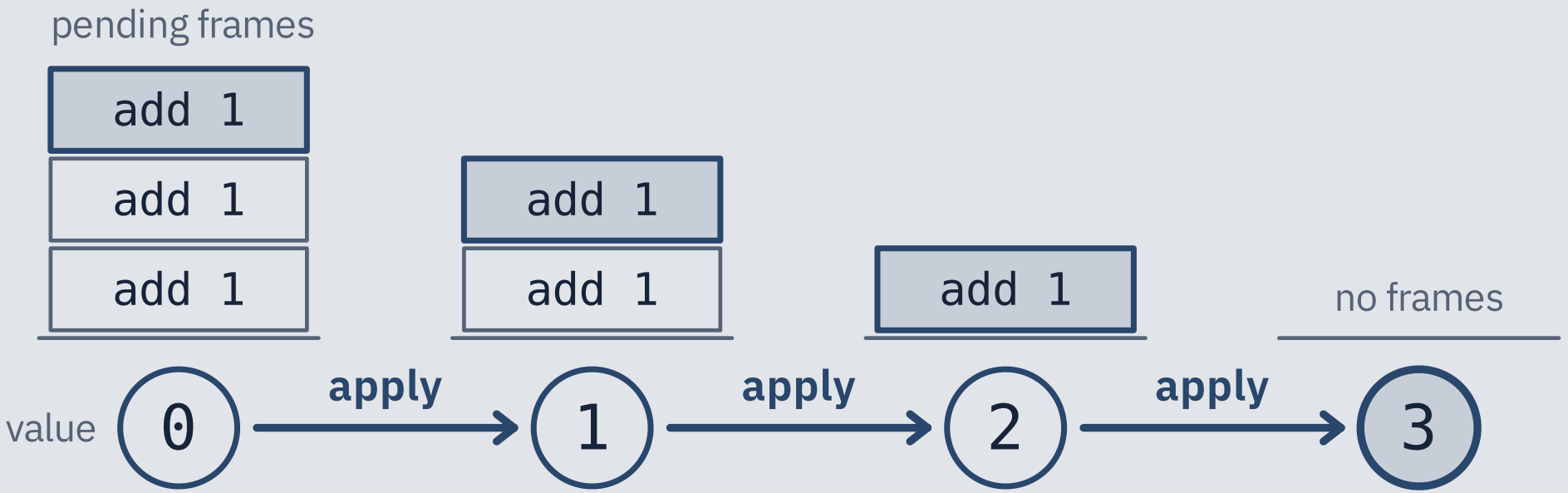
Recorded result

```
builtins.foldl' apply 0       3
frames
```

Recorded result

A machine makes its pending operations inspectable.

Each transition consumes one frame and carries the rest forward



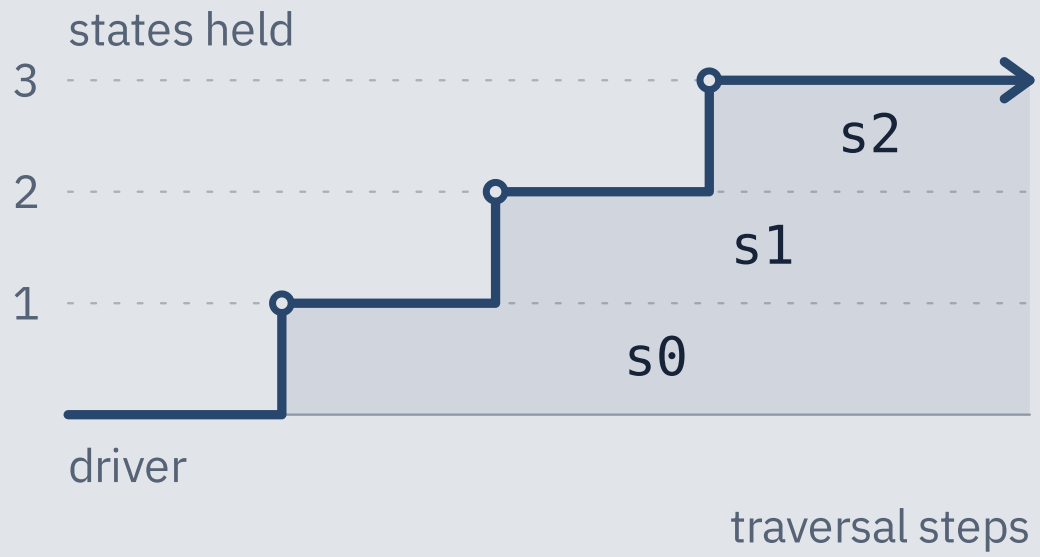
Each transition consumes the top frame; the remaining frames keep their order.

nix-effects / src/tc/eval/machine.nix

This has become a disproportionate response to addition.

genericClosure keeps every visited state reachable from its result.

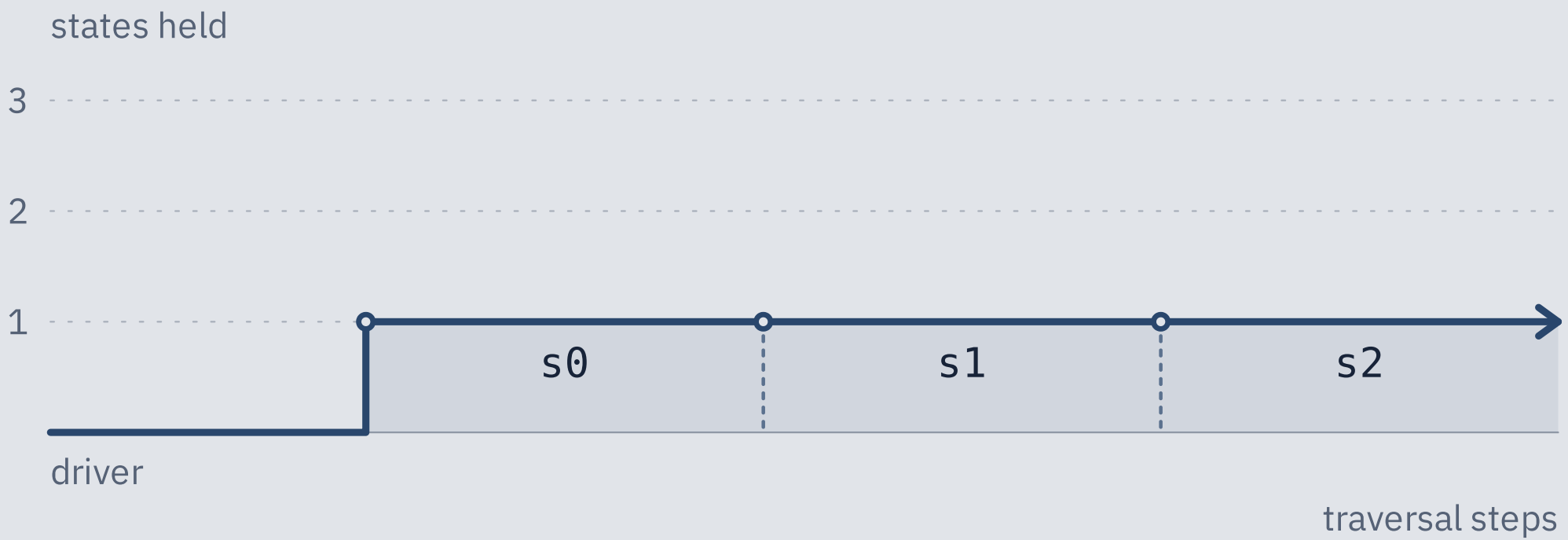
The visited list holds every state until return



Counts states the driver references; references inside a state are not shown.

Keeping only the current state removes this history root.

The driver holds one state at a time



Counts states the driver references; references inside a state are not shown.

nix-effects / src/tc/eval/machine.nix

A native loop can run our steps without worklist keys or history.

The borrowed machinery	What the loop needs
Nested driver calls	Return from one step before calling the next
A fold needs a list of inputs	Let each step choose whether to continue
Worklist keys and visited history	Carry the current state; return the final result

The step still chooses which fields to compute and which calls to make.

The step returns either another state or the final value.

Proposed step protocol; nextState constructs one successor

```
step = s:
  if s.i == 100000
  then [ false s.total ]
  else [ true (nextState s) ];
```

Each reply gives control back to the native driver

step returns		The native driver
`[ true nextState ]`	→	calls step with nextState
`[ false value ]`	→	returns value as the result

hsjobeki / NixOS/nix #14553 - open, unmerged - checked 22 September 2026

Our step still decides when to compute the carried total.

The same targeted forcing policy, now inside nextState

```
nextState = s:
  let total = s.total + 1;
  in builtins.seq total {
    i = s.i + 1; inherit total;
  };
```

The payload policy stays in our program

Current state

`i = 0`
`total = 0`

→

seq computes first

`total = s.total + 1`

before the record is returned

→

Next state

`i = 1`
`total = 1`

total is already a number

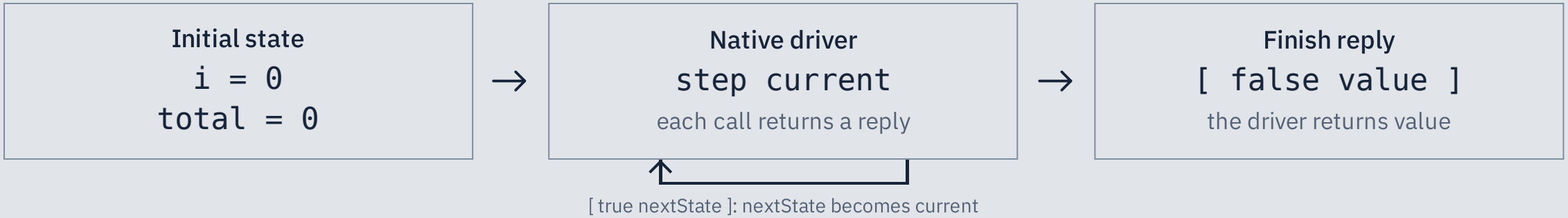
hsjobeki / NixOS/nix #14553 - open, unmerged - checked 22 September 2026

The builtin supplies the loop we kept borrowing.

The step and payload policy have already been defined

```
builtins.trampoline step { i = 0; total = 0; }
```

The native driver replaces our borrowed worklist



No driver-owned list of every state. No keys for deduplication.

hsjobeki / NixOS/nix #14553 - open, unmerged - checked 22 September 2026

Keep the loop small, stack-safe, and interruptible.

Case	What I want the contract to say
The step says finish	No further step is called
A million steps	The driver stack does not grow with them
Only the final state is needed	No driver-owned collection of every state
The loop never says finish	An interrupt still stops it

The step chooses which fields to force. Recursive callbacks still need explicit steps.

After all that, I would quite like a loop.

Run the next step without growing the call stack.

nixcon2026.bohinen.no

Examples - captured evidence - reading - PDFs

Discuss builtins.trampoline: NixOS/nix #14553 / Prototype review / Probe results

sternenseemann: trampolining Nix (2022)