Expositor: Scriptable, Time-travel Debugging with First-class Traces

Work in progress

Khoo Yit Phang, **Michael Hicks**, Jeffrey Foster

University of Maryland, College Park

Debugging

- *"...we talk a lot about finding bugs, but really, [Firefox's] bottleneck is not finding bugs but fixing [them]..."* —Robert O'Callahan

- Debugging = scientific method in action
  - Programmer makes observations
  - Proposes a hypothesis about the cause of the failure
  - Uses hypothesis to make predictions
  - Tests predictions with experiments
  - Victory? Then fix bug. Else repeat.

Tools for understanding a bug

- *"[In debugging,] understanding how the failure came to be...requires by far the most time and other resources"* —Andreas Zeller

- Many tools/techniques in use
  - Interactive debuggers (e.g., gdb)
  - Print statements
  - Profilers, visualizers, etc.
  - Record-replay executions (a.k.a., time travel)
    - VMware Replay, UndoDB, rudimentary support in gdb, OCaml debugger, ...

Scriptable debugging

- Make observations, test hypotheses etc. by writing programs over program executions
  - Benefit: automate tedious repetition
  - Hopefully also:
    - reuse scripts on different programs (or parts of a program)
    - compose old scripts into new ones,
    - build sophisticated tools (e.g., visualizations) ...
- Problem: scripts tend to be brittle, hard to reuse
- Solution: make scripts lazy and functional!

Expositor

- Expositor presents the programmer with a first-class abstraction for an *execution trace*
  - A sequence of snapshots, one per program event
- Programmers write scripts that are a composition of maps, filters, scans, and other combinators on executions
- For efficiency, Expositor builds on top of time travel debugging, and uses laziness liberally
  - Materialize events when you need them

Counting function calls in GDB

```
foo = gdb.Breakpoint("foo")
count = 0; more = True;

def stop_handler(evt):
  if isinstance(evt, gdb.BreakpointEvent) \
      and foo in evt.breakpoints
    global count; count += 1

def exit_handler(evt):
  global more; more = False
```

```
gdb.events.stop.connect(stop_handler)
gdb.events.exited.connect(exit_handler)
gdb.execute("start")
while more:
  gdb.execute("continue")

gdb.events.exited.disconnect
(exit_handler)
gdb.events.stop.disconnect
(stop_handler)
foo.delete()
## count contains the total count
```

Classic callback-style programming

- Scripts hard to understand, compose, reuse
  - Control flow is not "straight-line" but smeared across handlers in possibly many different scripts
  - Effects conflict
    - One script implemented by setting/disabling breakpoints on particular calls may conflict with composed script that attempts to count all calls
    - Name clashes on global variables, event names

Counting function calls in Expositor

```
1 foo = the_execution.breakpoints("foo")
2 count = len(foo)
```

- Easier to reuse and compose scripts
  - len(foo.filter(*lambda* s: p))
- Control-flow is "straight line" — list processing
- Evaluation is lazy
  - After line 1, no computation has been done
  - Line 2 call to len forces computation
    - Sets the breakpoints and runs the program

Expositor API: Execution

class Execution	*has singleton instance* the_execution
get_at(i)	*snapshot at time* i
breakpoints(fname)	*snapshots of calls to* fname
syscalls(sname)	*snapshots of syscalls* sname
watchpoints(vname, rwflags)	*r/w's to var* vname
all_calls()	*snapshots at all function calls*
all_returns()	*snapshots at all function returns*
cont()	*continue interactive execution*
get_time()	*get time of paused execution*

Traces

class Trace — *derived initially from* the_execution

len() — *called by len(trace)*

iter() — *called by* ***for*** *item* ***in*** *trace*

get_at(t), get_before(t), get_after(t)

get items at, just before, or just after time t

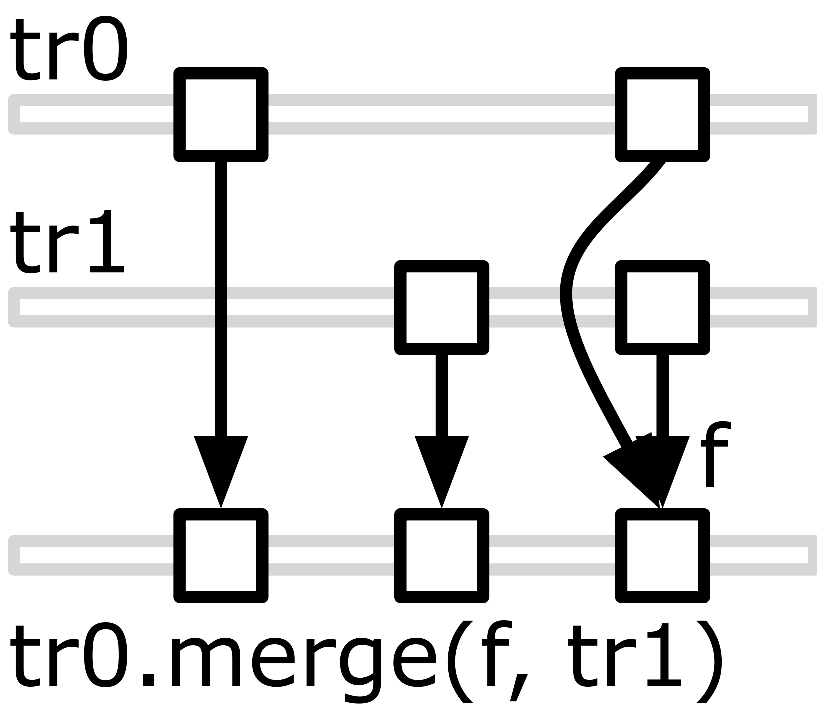
filter(p), map(f) — *usual filter or map of trace*

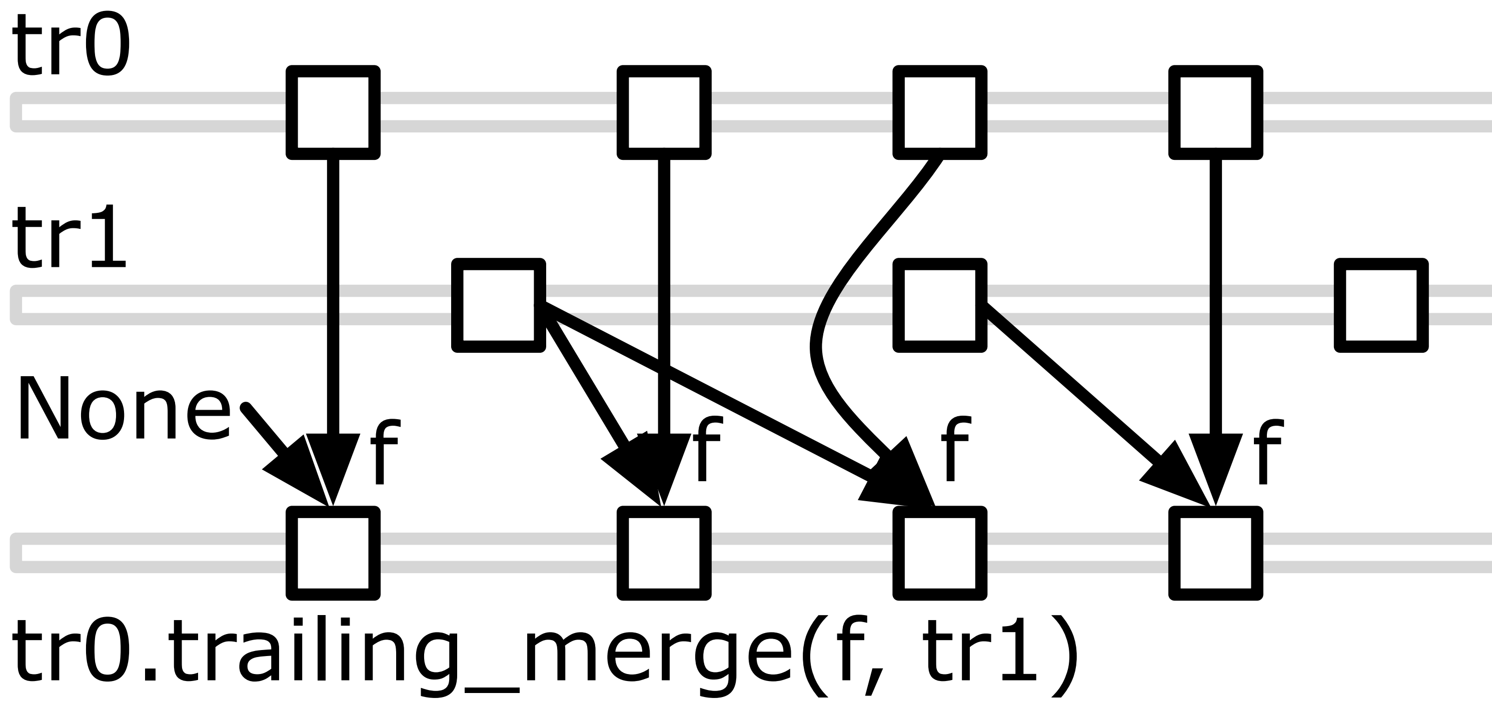
slice(t0,t1) — *subtrace in time interval [t0,t1]*

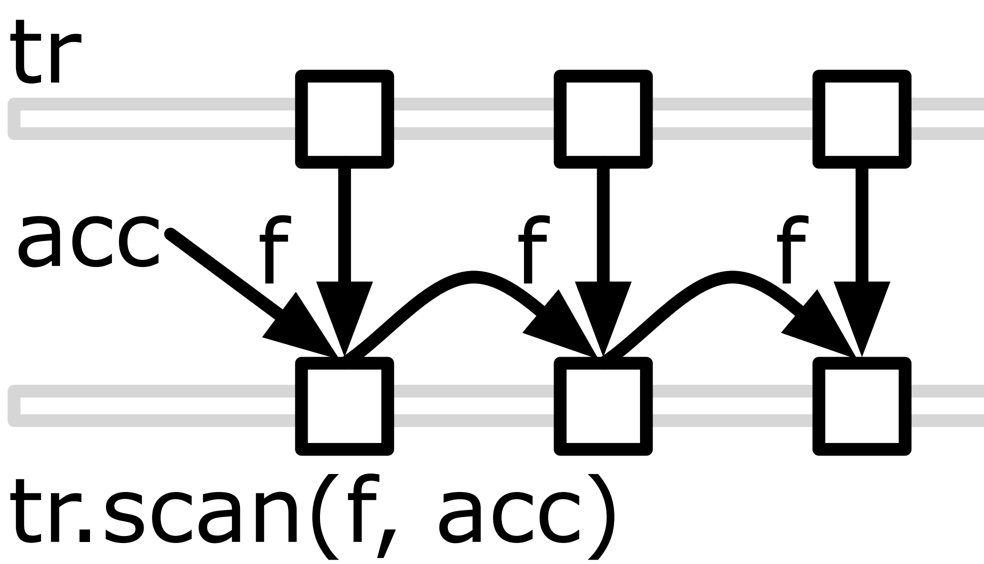
merge(f,tr) — *merge with trace tr*

scan(f,acc) — *like fold*

Merge and scan







Example: finding stack corruption

```
calls = the_execution.all_calls()
rets = the_execution.all_returns()
calls_rets = calls.merge(None,rets)
shadow_stacks = calls_rets.map(lambda s: s.retaddrs())
```

Example: finding stack corruption

```
def find_corrupted(snap, opt_shadow):
  if opt_shadow.force() is not None:
    for x,y in zip(snap.read_retaddrs(), opt_shadow.force()):
      if x != y:
        return x
  return None

corrupted_addrs = calls_rets \
  .trailing_merge(find_corrupted, shadow_stacks) \
  .filter(lambda x: x is not None)
```

Running it

```
% expositor tinyhttpd
(expositor) python-interactive
>> the_execution.cont()
httpd running on port 47055
Now I pwn your computer
^C
>> corrupted_addrs = stack_corruption()
>> t = the_execution.get_time()
>> last_corrupt = corrupted_addrs.get_before(t)
>> bad_writes = the_execution.wps(last_corrupt.value,rw=WRITE)
>> last_bad_write = bad_writes.get_before(last_corrupt.time)
```

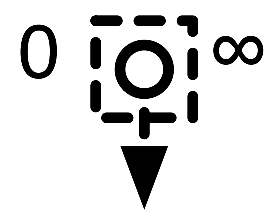
```
% ./exploit.py 47055
Trying port 47055
pwning...
```

Implementing lazy traces

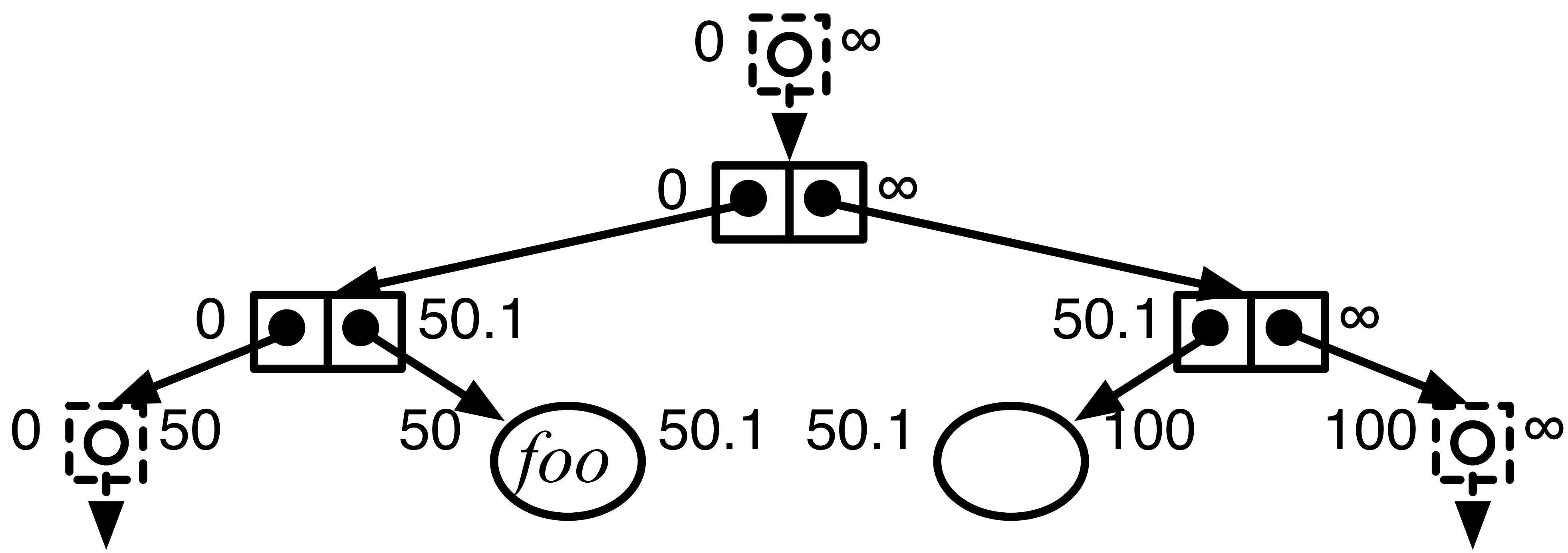
- Builds on top of time-travel debugger, UndoDB
  - Adds to gdb: *go to time t*, and *run backward*
- Expositor goal: minimize demand for snapshots
  - Typical script: run (full speed) for a while, then interact with the execution (as per previous example)
- Lazy traces implemented as interval trees, materialized on demand

Basic idea

- foo = the_execution.breakpoints("foo")



- foo.get_before(100)   *# event at time 50*



Lazy datastructures needed

- Problem: time-travel efficiency thwarted when computing with datastructures

```
sets = add_elems_to_a_set(tr)
is_member(x, sets.get_before(t))
```

  - Must perform full execution to get sets.get_before(t), even if x was added just before time t

- Solution: *lazy* EditHAMT
  - EditHAMT = Editable Hash Array Map Trie

EditHAMT class

class edithamt

find(k) *return most recent value for k, or* None

find_multi(k) *return iterator for all values for k*

 # the following class methods are purely functional

addkeyvalue(lazy_eh,k,v) *add (k → v) to lazy_eh*

remove(lazy_eh,k) *remove all (k → v) from lazy_eh*

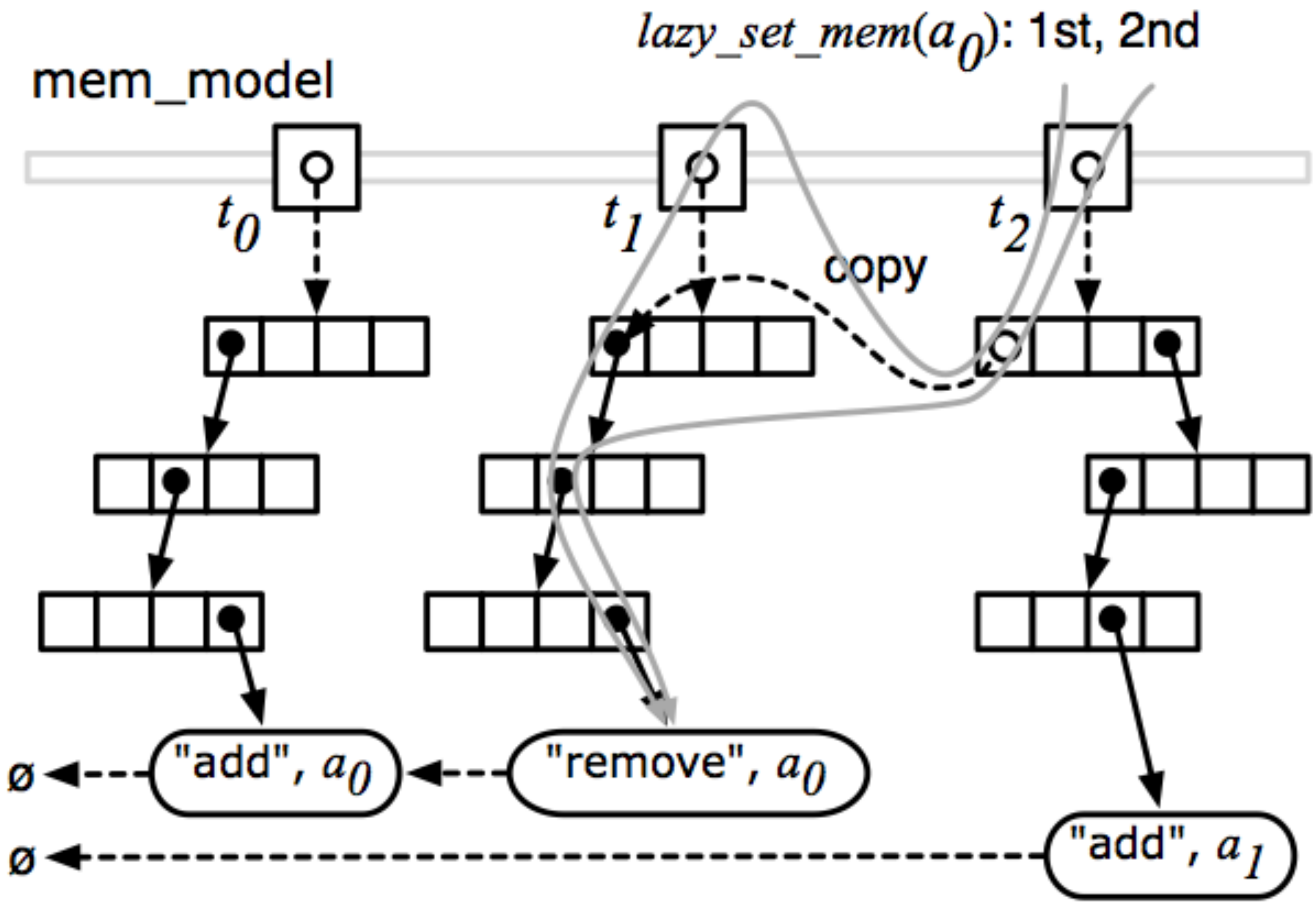
concat(lazy_eh1, lazy_eh2)

return lazy_eh1 + lazy_eh2

EditHAMT: rough construction

- Hash Array Mapped Trie (HAMT)
  - Hash keys to fixed-width integers
- Values in HAMT are (lazy) edit lists
  - Tabulate additions/removals for each key
- Persistent and lazy
  - Each update shares structure with parent, without forcing it to be fully computed
- Requests to find keys force necessary computation

A lazy set as a EditHAMT



Implementation status

- Basic implementation working
  - Performing detailed performance experiments now
  - One area of concern: balancing space/time tradeoff
- Performed one significant case study: debug Firefox memory leak
  - Cause: a combination of a timing bug and a data race
  - First submitted “fix” did not actually correct the bug
  - Wrote a custom happens-before race detector as a script using EditHAMTs

Continuing work

- Adding support for self-adjusting computation
  - foo = the_execution.slice(10,20)
  - bar = script_on(foo)
  - update foo to be the_execution.slice(**5**,20)
    - want bar to update automatically
- Expand API
  - Tied to C-style compilation/calling conventions
  - Support for manipulating snapshots is primitive
- Visualization for traces

Notable related work

- Functional reactive programming
  - MzTake for Racket applies FrTime to debugging Java
  - Here, time always marches forward (eagerly), making it hard to "interact" with an execution
- Record-replay strategies; simple query languages
  - Amber: Snapshot entire executions
  - UndoDB, VMware, OCaml: snapshot + logging
- PTQL, PQL: query languages over executions
  - Implemented as dynamic instrumentation

Summary

- Expositor introduces scripting on execution traces
  - Purely functional. Supports compositionality
- Built on top of time-travel debugging
  - Use laziness for efficiency
  - Could build on top of full captures, too