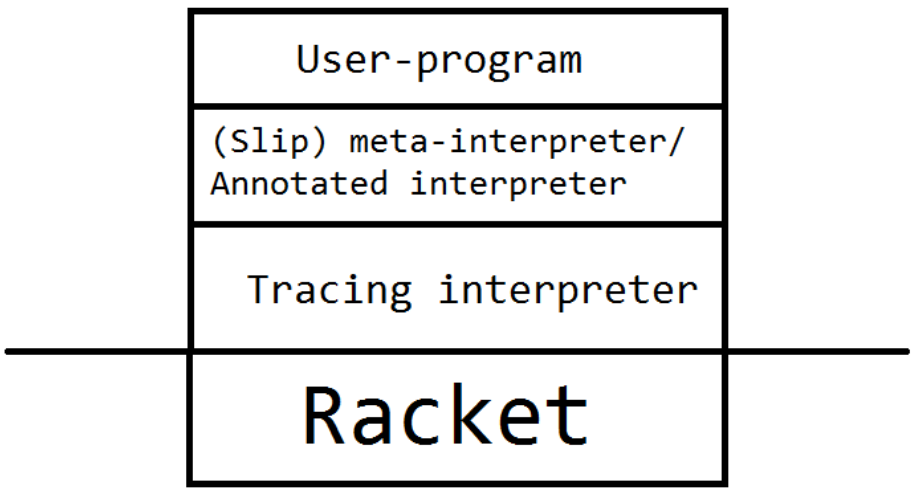
Vrije Universiteit Brussel

Meta-tracing JIT compilation

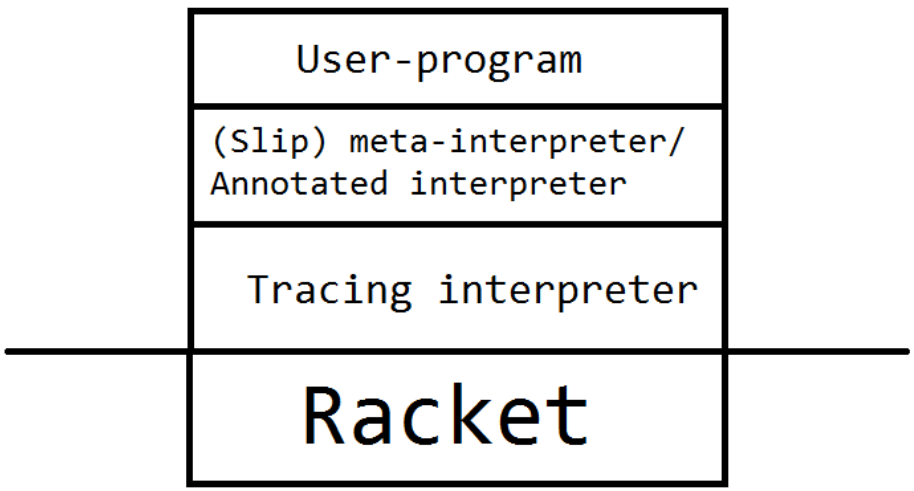
Maarten Vandercammen

Some terminology

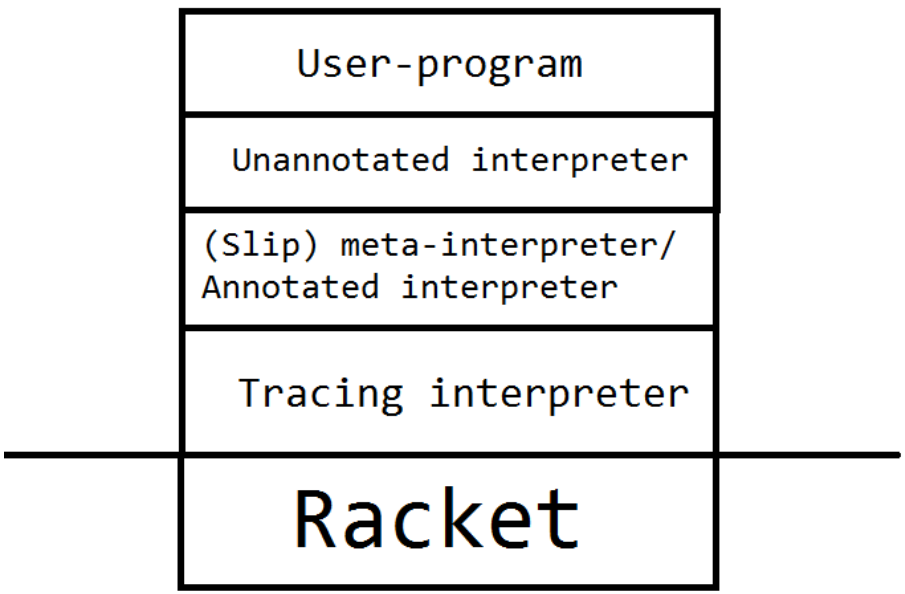


Regular meta-tracing

Some terminology



Regular meta-tracing



Nested meta-tracing

Regular interpretation

CK-based register machine

```
(define ρ #f) ; env

(define σ #f) ; store

(define θ #f) ; non-kont stack

(define v #f) ; general-purpose register

(struct ev (e κ) #:transparent)

(struct ko (ϕ κ) #:transparent)
```

Register manipulation

```
(save-val)
(restore-val)
(save-vals i)
(restore-vals i)
(save-all-vals)

(save-env)
(restore-env)
(set-env ρ*)

(alloc-var x)
(set-var x)
(lookup-var x)

(create-closure x es)
(literal-value e)
(quote-value e)
(apply-native i)

(push-continuation ϕ)
(pop-continuation)
```

Step

step = manipulate registers + return new CK state

```
; one evaluation step
(define new-state (step state))

(define (step state)
  (match state
    ((ev `(and ,e . ,es) κ)
     (execute/trace `(push-continuation ,(andk es)))
     (ev e (cons (andk es) κ)))
    ((ev (? symbol? x) (cons φ κ))
     (execute/trace `(lookup-var ',x) ; manipulate registers
                    `(pop-continuation))
     (ko φ κ)) ; return new state
     ...)
```

Step*

```
; complete evaluation
(define result (step* state))

(define (step* state)
  (match state
    ((ko (haltk) _) ; evaluation finished
     v)
    (_
     (let ((new-state (step state)))
       (step* new-state)))))
```

Closures

```
(struct clo (λ ρ) #:transparent)
(struct lam (x es) #:transparent)

(create-closure x es)

(clo-equal? clo1 clo2)
```

Tracing

Annotations

```
(can-start-loop label debug-info)

(can-close-loop label)
```

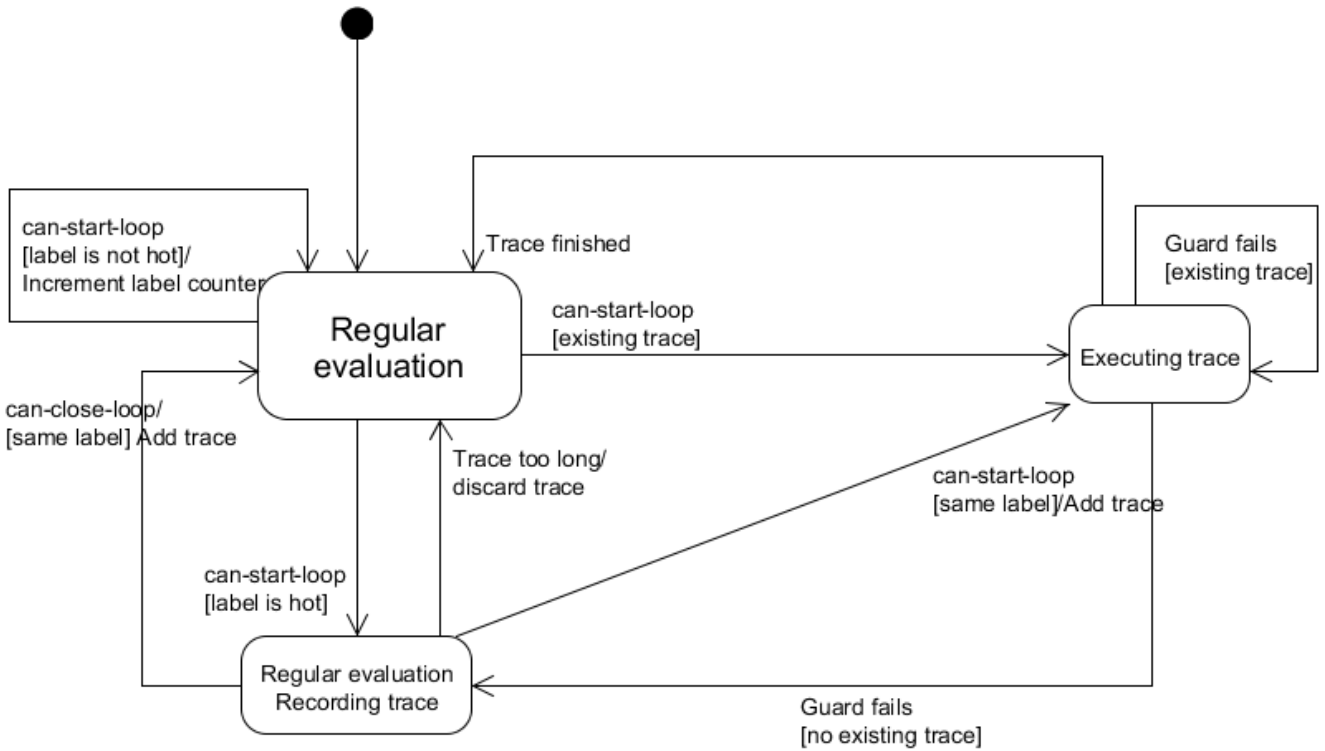
Annotations

```
(define (close parameters expressions closure-name)
    (define lexical-environment environment)
    (define (closure . arguments)
      (define dynamic-environment environment)
      ; function call starts here
      (can-start-loop expressions closure-name)
      (set! environment lexical-environment)
      (bind-parameters parameters arguments)
      (let* ((value (evaluate-sequence
                     expressions)))
        (set! environment dynamic-environment)
        ; function call ends here
        (can-close-loop expressions)
        value))
    closure)
```

Handling annotations

```
(define (step* state)
    (match state
      ((ko (haltk) _)
       v)
      ; evaluate annotations in step* instead of step
      ; annotations might not lead to recursive call to
          step*
      ((ko (can-close-loopk) (cons φ κ))
       (handle-can-close-loop-annotation v (ko φ κ)))
      ((ko (can-start-loopk '() debug-info) (cons φ κ))
       (handle-can-start-loop-annotation v debug-info (ko φ
           κ)))
      (_
       (let ((new-state (step state)))
         (step* new-state)))))
```

Overview


can-start-loop
[label is not hot]/
Increment label counter
Trace finished
Guard fails
[existing trace]
Regular
evaluation
can-start-loop
[existing trace]
Executing trace
can-close-loop/
[same label] Add trace
Trace too long/
discard trace
can-start-loop
[same label]/Add trace
can-start-loop
[label is hot]
Regular evaluation
Recording trace
Guard fails
[no existing trace]

Guards

```
(define (loop n)
  (display n) (newline)
  (if (< n 0)
      (display "Stopped!")
      (loop (- n 1))))
```

Guards

```
(if bool a b) -> guard-false/guard-true
(cond ...) -> guard-false/guard-true
(f ...) -> guard-same-closure
(apply f (list ...)) -> guard-same-nr-of-args

(define (guard-true ...)
  (unless v
          ; do something
          ))
```

Guards

Guards identified with id's

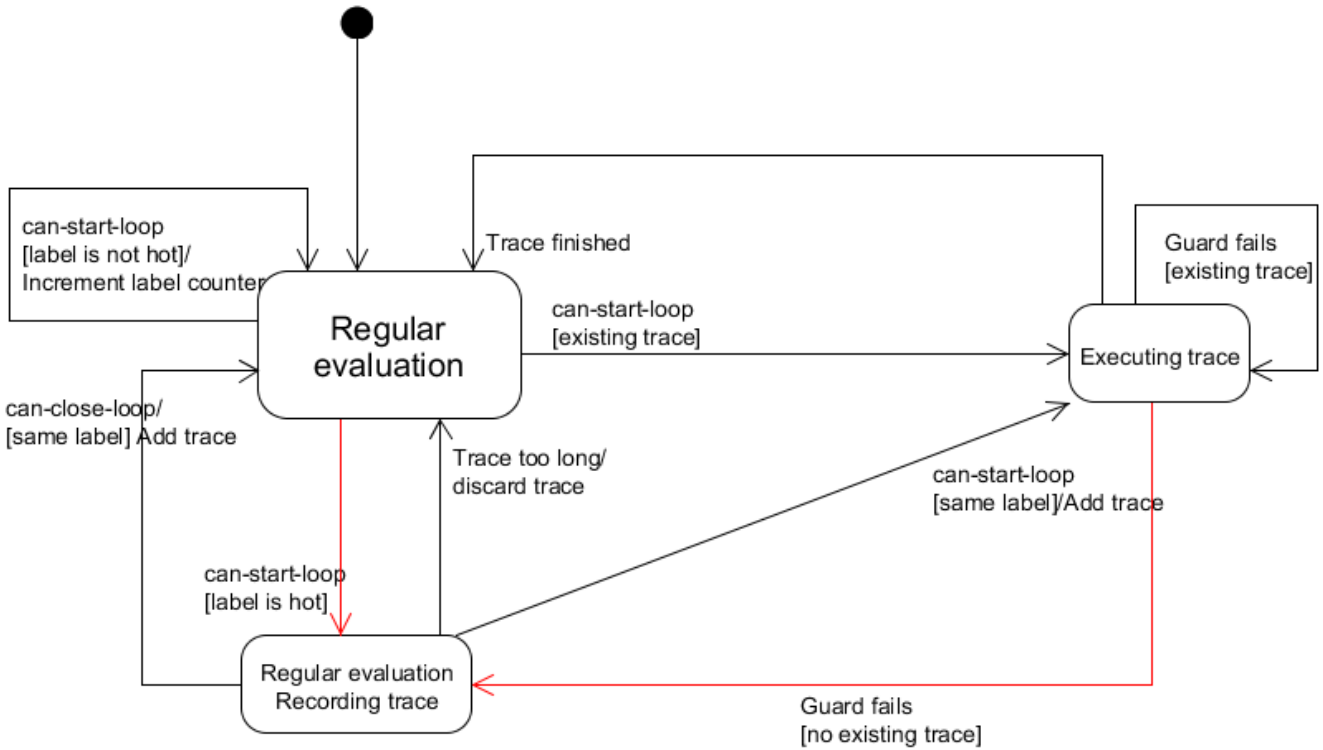
```
(define (step state)
  (match state
    ((ko (ifk e1 e2) κ)
       (execute/trace `(restore-env))
       (let ((new-guard-id (inc-guard-id!)))
         (if v
             (begin (execute/trace `(guard-true
                 ,new-guard-id))
                    (ev e1 κ))
             (begin (execute/trace `(guard-false
                 ,new-guard-id))
                    (ev (car e2) κ)))))
```

Bookkeeping

```
(struct tracer-context (is-tracing?
                        trace-key
                        times-label-encountered-while-tracing
                        current-trace-length
                        labels-encountered
                        trace-nodes
                        trace-nodes-dictionary
                        labels-executing
                        closing-function
                        merges-cf-function
                        guards-dictionary) #:transparent
                            #:mutable)

(define GLOBAL_TRACER_CONTEXT (new-tracer-context))
```

Two traces

Label traces and guard traces



(can-start-loop)

```
(define (handle-can-start-loop-annotation label debug-info state)
  ; Continue regular interpretation with the given state.
  (define (continue-with-state)
    (execute/trace `(pop-continuation))
    (step* state))
  ; Trace hot?
  (define (can-start-tracing-label?)
    (>= (get-times-label-encountered label) TRACING_THRESHOLD))
  (cond ((is-tracing-label? label)
         (stop-tracing! #t)
        (let ((new-state (execute-label-trace-with-label label)))
         (step* new-state)))
        ((label-trace-exists? label)
         ; Execute trace
         (let* ((label-trace (get-label-trace label))
                ((new-state (execute-label-trace-with-label label)))
           (step* new-state)))
        ((and (not (is-tracing?)) (can-start-tracing-label?))
         (start-tracing-label! label debug-info)
         (continue-with-state))
        ; Increase 'hotness' counter of label
        (else
         (inc-times-label-encountered! label)
         (continue-with-state))))
```

Starting tracing

```
; Starting tracing = doing some bookkeeping!
(define (start-tracing-guard! guard-id old-trace-key)
    (clear-trace!)
    (set-tracer-context-is-tracing?! GLOBAL_TRACER_CONTEXT
        #t)
    (set-tracer-context-trace-key! GLOBAL_TRACER_CONTEXT
        (make-guard-trace-key (trace-key-label
        old-trace-key)
```

```
  (define (start-tracing-label! label debug-info)
    (clear-trace!)
    (set-tracer-context-is-tracing?! GLOBAL_TRACER_CONTEXT
        #t)
    (set-tracer-context-trace-key! GLOBAL_TRACER_CONTEXT
        (make-label-trace-key label debug-info)))
```

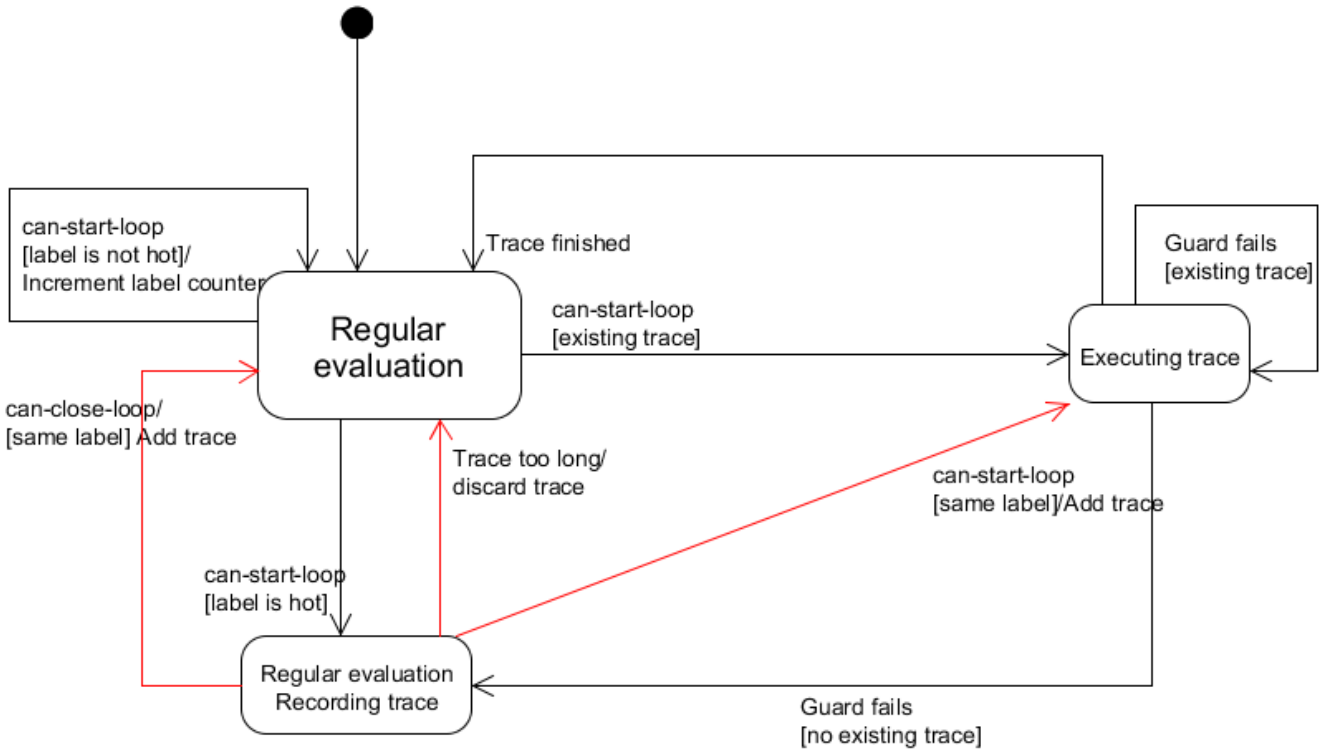
Starting tracing

```
; Keep track of what you're tracing
(struct trace-key (label id) #:transparent)

; Store id of parent label that caused guard
    failure
(struct guard-trace-key trace-key
    (parent-label-trace-id) #:transparent)

(struct label-trace-key trace-key (debug-info)
    #:transparent)
```

Recording operations

```
(define τ '())

(define (append-trace! ms)
  (let ((new-instructions-length (length ms)))
    (set! τ (append (reverse ms) τ))
    (add-trace-length!
       new-instructions-length))))

(define (execute/trace . ms)
  (when (is-tracing?)
    (append-trace! ms))
  (eval-instructions ms))
```

Stopping tracing



Trace too long

```
; Final version
(define (append-trace! ms)
  (let ((new-instructions-length (length ms)))
    (set! τ (append (reverse ms) τ))
    (add-trace-length! new-instructions-length)
    (when (max-trace-length-reached?)
      (handle-max-trace-length-reached))))

(define (handle-max-trace-length-reached)
  ; Stop tracing and discard the trace
  (stop-tracing-abnormal!))
```

Looping vs non-looping

```
; No recursion ->
    no looping
(define (f x)
  (+ x 1))
```

Ends with
(can-close-loop)

Looping vs non-looping

```
; No recursion ->
    no looping
(define (f x)
  (+ x 1))
```

Ends with (can-close-loop)

```
; Recursion ->
    looping
(define (f x)
  (f x))
```

Ends with (can-start-loop label debug-info)

Looping vs non-looping

```
(define (handle-can-close-loop-annotation label state)
    (when (is-tracing-label? label)
      ; #f = label does not loop
      (stop-tracing! #f))
    (execute/trace `(pop-continuation))
    (step* state))

(define (handle-can-start-loop-annotation label
   debug-info state)
    (cond ((is-tracing-label? label)
           ; #t = label does loop
           (stop-tracing! #t)
           (let ((new-state
                  (execute-label-trace-with-label
                   label)))
             (step* new-state))))
```

Looping vs non-looping

Labels:

```
; Recursion ->
    looping
(letrec ((loop
    (lambda ()
    ; trace
        instructions
    (loop))))
  (loop))
```

Looping vs non-looping

Labels:

```
; Recursion ->
    looping
(letrec ((loop
    (lambda ()
    ; trace
        instructions
    (loop))))
  (loop))
```

```
; No recursion -> no
    looping
(letrec ((non-loop
    (lambda ()
    ; trace
        instructions
    )))
  (non-loop))
```

Looping vs non-looping

Guards:

```
(define (f)
  (if (= (random 2) 0)
      (begin (display 0)
             (f))
      (begin (display 1)
             (f))))
```

Looping vs non-looping

Guards:

```
; Recursion -> looping
(letrec ((non-loop
    (lambda ()
    ; trace instructions
    )))
  (non-loop)
  (execute-label-trace-with-id
      parent-label-trace-id))
```

Looping vs non-looping

Guards:

```
; Recursion -> looping
(letrec ((non-loop
    (lambda ()
    ; trace instructions
    )))
  (non-loop)
  (execute-label-trace-with-id
      parent-label-trace-id))
```

```
; Recursion -> looping
(letrec ((non-loop
    (lambda ()
    ; trace instructions
    )))
  (non-loop))
```

Stop tracing normally

Scheme-style state pattern:
first-class functions

```
(define (start-tracing-guard! guard-id old-trace-key)
  ...
  (set-tracer-context-closing-function!
      GLOBAL_TRACER_CONTEXT
      (make-stop-tracing-guard-function guard-id))
  ...)

(define (start-tracing-label! label debug-info)
  ...
  (set-tracer-context-closing-function!
      GLOBAL_TRACER_CONTEXT
      (make-stop-tracing-label-function))
  ...)
```

Stop tracing normally

Closing functions: transform trace correctly and add trace

```
(define (stop-tracing-label! trace looping?)
  (let* ((trace-key (tracer-context-trace-key
      GLOBAL_TRACER_CONTEXT))
         (transformed-trace (transform-and-optimize-trace
             trace (make-transform-label-trace-function
             looping?))))
    (add-label-trace! trace-key transformed-trace
        looping?)))
```

Stop tracing normally

Closing functions: transform trace correctly and add trace

```
(define (stop-tracing-label! trace looping?)
  (let* ((trace-key (tracer-context-trace-key
      GLOBAL_TRACER_CONTEXT))
         (transformed-trace (transform-and-optimize-trace
             trace (make-transform-label-trace-function
             looping?))))
    (add-label-trace! trace-key transformed-trace
        looping?)))

(define (make-transform-label-trace-function looping?)
  (if looping?
      transform-label-trace-looping
      transform-trace-non-looping))
```

Transforming traces

```
(define (transform-trace-non-looping trace)
  `(letrec ((non-loop ,(append '(lambda ()) trace)))
     (non-loop)))

(define (transform-label-trace-looping trace)
  `(letrec ((loop ,(append '(lambda ()) trace
     '((loop)))))
     (loop)))
```

Stop tracing normally

```
(define (stop-tracing! looping?)
    (let ((stop-tracing-function
           (tracer-context-closing-function
           GLOBAL_TRACER_CONTEXT)))
      (stop-tracing-function (reverse τ) looping?)
      (stop-tracing-normal!)))

  (define (stop-tracing-normal!)
    (stop-tracing-bookkeeping!))

(define (stop-tracing-bookkeeping!)
    (set-tracer-context-is-tracing?! GLOBAL_TRACER_CONTEXT
        #f)
    (set-tracer-context-trace-key! GLOBAL_TRACER_CONTEXT #f)
    (set-tracer-context-closing-function!
        GLOBAL_TRACER_CONTEXT #f)
    (clear-trace!))
```

Intermezzo

Trace representation

```
(struct trace-node (trace-key
                    trace
                    (executions #:mutable)))

(struct tracer-context (...
                        trace-nodes
                        trace-nodes-dictionary
                        guards-dictionary
                        ...)
```

Trace execution

Trace execution = call eval on trace

Trace execution

Trace execution = call eval on trace

Actually... for-each eval on trace operations

Trace execution

```
(define (execute-guard-trace guard-id)
    (let* ((guard-trace (get-guard-trace guard-id))
           (trace (trace-node-trace guard-trace)))
      ; Benchmarking
      (add-execution! guard-trace)
      (execute/trace `(let ()
                        (let* ((state (execute-trace ',trace))) ; Actually
                              execute the trace
                          ; Don't mind this!
                          (bootstrap-to-evaluator state))))))

  (define (execute-label-trace-with-trace-node label-trace-node)
    (let ((trace (trace-node-trace label-trace-node)))
      ; Benchmarking
      (add-execution! label-trace-node)
      (execute/trace `(let ()
                        (push-label-trace-executing! ,label-trace-node)
                        (let ((state (execute-trace ',trace))) ; Actually
                              execute the trace
                          (pop-label-trace-executing!)
                          ; Don't mind this!
                          state)))))
```

Trace execution

```
(define (execute-label-trace-with-id label-trace-id)
        ; find trace
  (let ((label-trace-node (find (tracer-context-trace-nodes-dictionary
        GLOBAL_TRACER_CONTEXT) label-trace-id)))
    ; execute trace
    (execute-label-trace-with-trace-node label-trace-node)))

(define (execute-label-trace-with-label label)
        ; find trace
  (let ((label-trace-node (get-label-trace label)))
    ; execute trace
    (execute-label-trace-with-trace-node label-trace-node)))
```

Trace execution

Identify label trace being executed

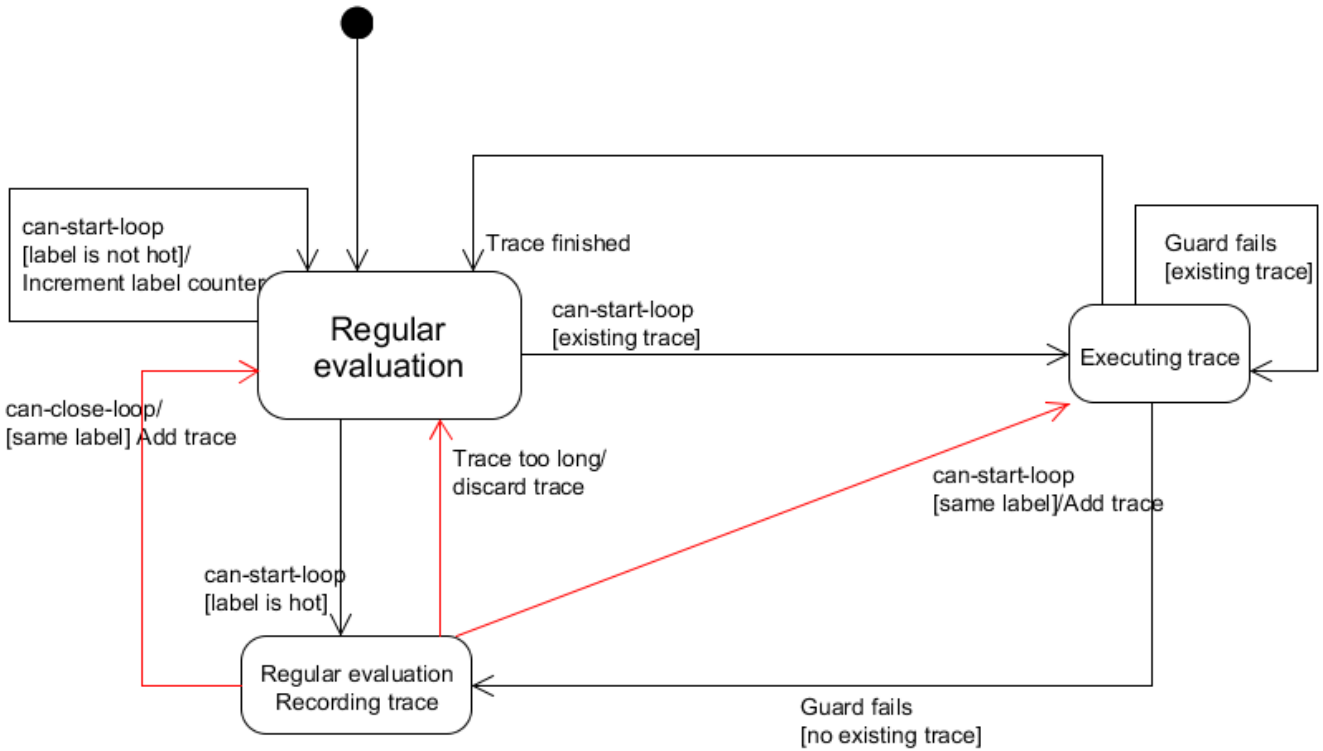
```
(push-label-trace-executing! label-trace-node)
(pop-label-trace-executing!)

(struct tracer-context (...
                        labels-executing
                        ...)
```

Trace execution

Why a stack? Trace jumping!

```
(define (f x)
  (+ x 10))

(define (loop)
  (f 1)
  ; other stuff
  (loop))
```

Stopping trace execution



Stopping trace execution

Two problems:

- Stop trace execution
- Continue with regular interpretation

Continue interpretation

Bootstrap interpreter with CK state $\rightarrow$
Reconstruct expected continuation

Operations on K invisible in trace $\rightarrow$
K gets lost during trace execution

Continue interpretation

Bootstrap interpreter with CK state $\rightarrow$
Reconstruct expected continuation

Operations on K invisible in trace $\rightarrow$
K gets lost during trace execution
Use register for K

Continue interpretation

Bootstrap interpreter with CK state $\rightarrow$
Reconstruct expected continuation

Operations on K invisible in trace $\rightarrow$
K gets lost during trace execution
Use register for K

C? Later

τ-κ

```
(define τ-κ '()) ;continuation stack

(push-continuation φ)
(pop-continuation)

(define (step state)
  (match state
    ((ev (? symbol? x) (cons φ κ))
     (execute/trace `(lookup-var ',x)
                    `(pop-continuation))
     (ko φ κ))
```

Continue interpretation

```
(define (transform-trace-non-looping trace)
    `(letrec ((non-loop ,(append '(lambda () trace)))
       (non-loop)
       (let ((new-state (ko (car φ κ) (cdr φ κ))))
         (remove-continuation)
         new-state)))
```

Non-looping traces

```
(define (handle-can-start-loop-annotation label debug-info
    state)
  (cond ((label-trace-exists? label)
         (let* ((label-trace (get-label-trace label))
                (new-state (execute-label-trace-with-label
                    label)))
           (step* new-state))))
```

Non-looping traces

```
(define (execute-guard-trace guard-id)
    (let* ((guard-trace (get-guard-trace guard-id))
           (trace (trace-node-trace guard-trace)))
      ; Benchmarking
      (add-execution! guard-trace)
      (execute/trace `(let ()
                        (let* ((state (execute-trace ',trace))) ; Actually
                              execute the trace
                          (bootstrap-to-evaluator state))))))

  (define (execute-label-trace-with-trace-node label-trace-node)
    (let ((trace (trace-node-trace label-trace-node)))
      ; Benchmarking
      (add-execution! label-trace-node)
      (execute/trace `(let ()
                        (push-label-trace-executing! ,label-trace-node)
                        (let ((state (execute-trace ',trace))) ; Actually
                              execute the trace
                          (pop-label-trace-executing!)
                          state)))))
```

Looping traces

Ending looping traces = only through guard-failure

```
(define (guard-failed guard-id state)
   (cond ((guard-trace-exists? guard-id)
          (execute-guard-trace guard-id))
         ((not (is-tracing?))
          ; Start tracing guard
          ; Get label being executed!
          (let ((trace-key-executing (get-label-trace-executing-trace-key)))
            (start-tracing-guard! guard-id trace-key-executing)
            (bootstrap-to-evaluator state)))
         (else
          ; Already tracing something else:
          ; stop tracing that and start tracing this guard!
          (let ((trace-key-executing (get-label-trace-executing-trace-key)))
            (switch-to-trace-guard! guard-id trace-key-executing)
            (bootstrap-to-evaluator state)))))
```

State?

Reconstruct C in CK?

```
(define (guard-true guard-id e)
  (unless v
    (guard-failed-with-ev guard-id e))))

(define (guard-failed-with-ev guard-id e)
  (guard-failed guard-id (ev e τ-κ)))

(define (step state)
  (match state
   ((ko (ifk e1 e2) κ)
    ...
    (if v
        (begin (execute/trace `(guard-true
                ,new-guard-id ',e2))
               (ev e1 κ))
        ...))
```

Stopping trace execution

```
(eval '((save-env)
        (lookup-var 'x)
        ...
        (guard-true 123 ...)
        (apply-native '+ 3)
        ...))
```

Stopping trace execution

```
(eval '((save-env)
        (lookup-var 'x)
        ...
        (guard-true 123 ...)
        (apply-native '+ 3)
        ...))
```

Call/cc!

Call/cc

```
(define GLOBAL_CONTINUATION #f)

(define (set-global-continuation! k)
  (set! GLOBAL_CONTINUATION k))

(define (call-global-continuation v)
  (GLOBAL_CONTINUATION v))

(define (run s)
  (apply step* (list (let ((v (call/cc (lambda (k)
                                         (set-global-continuation!
                                             k)
                                         s))))
                       ; Start regular interpretation -> no
                           trace executions
                       (flush-label-traces-executing!)
                       v))))
```

Call/cc

```
(define (bootstrap-to-evaluator state)
  (call-global-continuation state))

(define (execute-guard-trace guard-id)
    (let* ((guard-trace (get-guard-trace guard-id))
           (trace (trace-node-trace guard-trace)))
      ; Benchmarking
      (add-execution! guard-trace)
      (execute/trace `(let ()
                        (let* ((state (execute-trace ',trace))) ; Actually
                              execute the trace
                          (bootstrap-to-evaluator state))))))

  (define (execute-label-trace-with-trace-node label-trace-node)
    (let ((trace (trace-node-trace label-trace-node)))
      ; Benchmarking
      (add-execution! label-trace-node)
      (execute/trace `(let ()
                        (push-label-trace-executing! ,label-trace-node)
                        (let ((state (execute-trace ',trace))) ; Actually
                              execute the trace
                          (pop-label-trace-executing!)
                          ; No bootstrapping!!!
                          state)))))
```

Trace execution

Why no bootstrapping? Trace jumping!

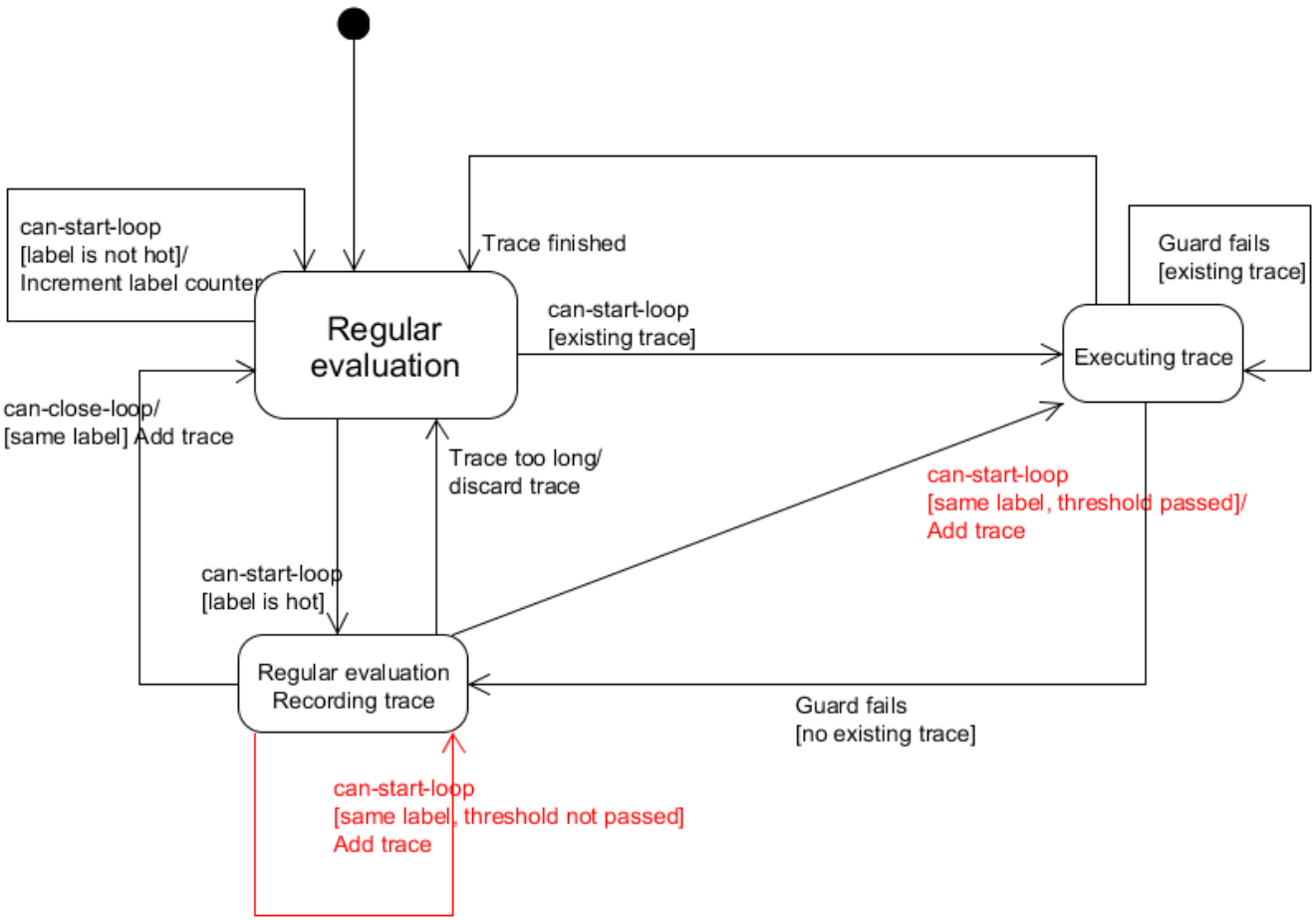
```
(define (f x)
  (+ x 10))

(define (loop)
  (f 1)
  ; other stuff
  (loop))
```

Trace execution

And remember

```
(define (handle-can-start-loop-annotation label
    debug-info state)
  (cond ((label-trace-exists? label)
         (let* ((label-trace (get-label-trace label))
                (new-state
                    (execute-label-trace-with-label
                    label)))
           ; When evaluating non-looping trace no
              need for bootstrapping
           (step* new-state)))
```

True vs false loops



True vs false loops

True loops = functions which have recursed at least x times

```
(struct label-trace trace-node ((loops? #:mutable)))

(define (handle-can-start-loop-annotation label debug-info
    state)
  (cond ((label-trace-exists? label)
         (let ((label-trace (get-label-trace label)))
           (if (and (is-tracing?) (label-trace-loops?
               label-trace))
               ; Record and jump to existing trace
               (let ((new-state
                   (execute-label-trace-with-label label)))
                 (step* new-state))
               ; Ignore existing trace, inline
               (continue-with-state))))
```

Fin!