

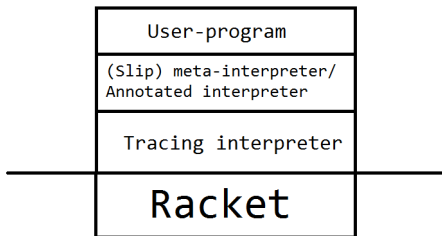


Vrije Universiteit Brussel

# Meta-tracing JIT compilation

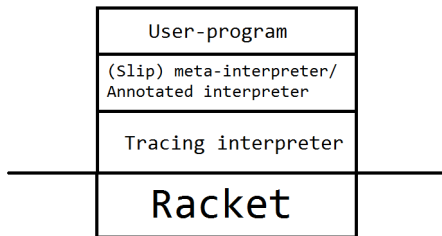
Maarten Vandercammen

## Some terminology

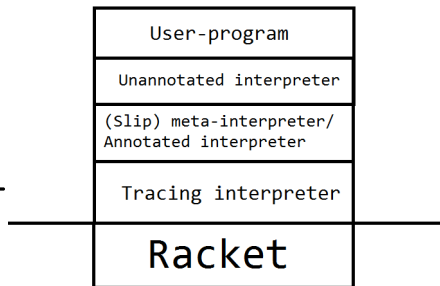


Regular meta-tracing

## Some terminology



Regular meta-tracing



Nested meta-tracing

# Regular interpretation

# CK-based register machine

```
(define  $\rho$  #f) ; env  
  
(define  $\sigma$  #f) ; store  
  
(define  $\theta$  #f) ; non-kont stack  
  
(define v #f) ; general-purpose register  
  
(struct ev (e  $\kappa$ ) #:transparent)  
  
(struct ko ( $\phi$   $\kappa$ ) #:transparent)
```

# Register manipulation

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

(save-env)
(restore-env)
(set-env  $\rho^*$ )

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

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

(push-continuation  $\phi$ )
(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)  $\kappa$ )
     (execute/trace '(push-continuation ,(andk es)))
     (ev e (cons (andk es)  $\kappa$ )))
    ((ev (? symbol? x) (cons  $\phi$   $\kappa$ ))
     (execute/trace '(lookup-var ',x) ; manipulate registers
                     '(pop-continuation))
     (ko  $\phi$   $\kappa$ )) ; 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 ( $\lambda$   $\rho$ ) #: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)
```

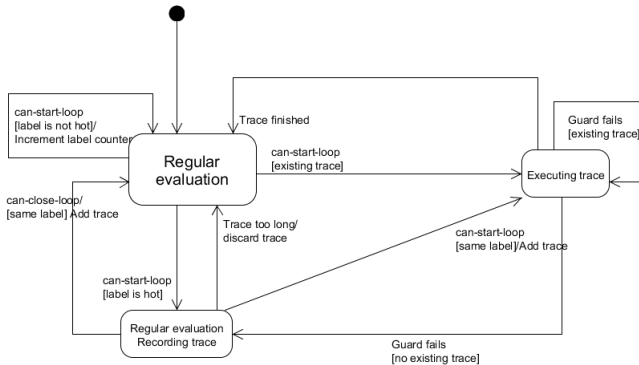
# Annotations

```
(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  $\phi$   $\kappa$ ))
     (handle-can-close-loop-annotation v (ko  $\phi$   $\kappa$ )))
    ((ko (can-start-loopk '() debug-info) (cons  $\phi$   $\kappa$ ))
     (handle-can-start-loop-annotation v debug-info (ko  $\phi$ 
                                                          $\kappa$ )))
    (_
     (let ((new-state (step state)))
       (step* new-state)))))
```

# Overview



# 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

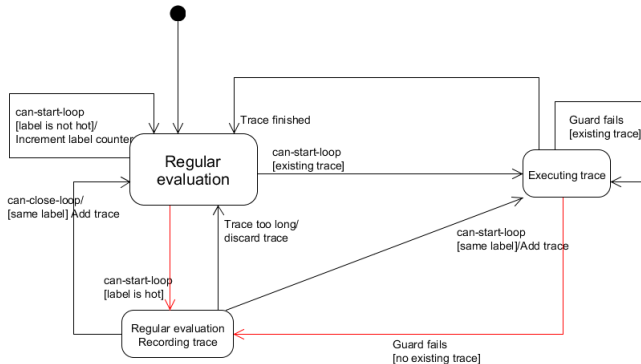
```
(define (step state)
  (match state
    ((ko (ifk e1 e2)  $\kappa$ )
     (execute/trace '(restore-env))
     (let ((new-guard-id (inc-guard-id!)))
      (if v
         (begin (execute/trace '(guard-true
                                ,new-guard-id))
                  (ev e1  $\kappa$ ))
      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

# 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)
```

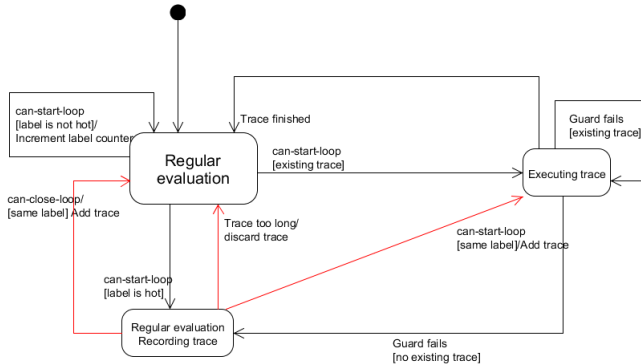
## Recording operations

```
(define  $\tau$  '())

(define (append-trace! ms)
  (let ((new-instructions-length (length ms)))
    (set!  $\tau$  (append (reverse ms)  $\tau$ ))
    (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!  $\tau$  (append (reverse ms)  $\tau$ ))
    (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  $\tau$ ) 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
     ; Actually execute the trace
     '(let* ((state (execute-trace ',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)
        ; Actually execute the trace
        (let ((state (execute-trace ',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!)
(top-label-trace-executing)

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

(define (guard-failed guard-id state)
  (cond ((not (is-tracing?))
         (let ((trace-key-executing
                (get-label-trace-executing-trace-key)))
           ...)))
```

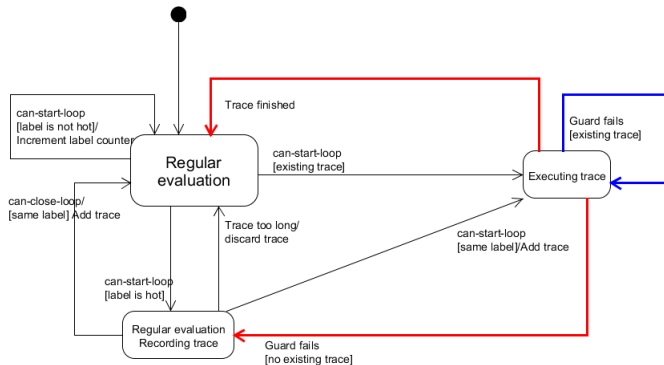
# Trace execution

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

# Restart interpretation

Bootstrap interpreter with CK state →  
Reconstruct expected continuation



# Restart interpretation

Bootstrap interpreter with CK state →  
Reconstruct expected continuation

Operations on K invisible in trace →  
K gets lost during trace execution

# Restart interpretation

Bootstrap interpreter with CK state →  
Reconstruct expected continuation

Operations on K invisible in trace →  
K gets lost during trace execution  
Use register for K

# Restart interpretation

Bootstrap interpreter with CK state →  
Reconstruct expected continuation

Operations on K invisible in trace →  
K gets lost during trace execution  
Use register for K

C? Later

```
(define  $\tau\text{-}\kappa$  '()) ;continuation stack

(push-continuation  $\phi$ )
(pop-continuation)

(define (step state)
  (match state
    ((ev (? symbol? x) (cons  $\phi$   $\kappa$ ))
     (execute/trace '(lookup-var ',x)
                     '(pop-continuation))
     (ko  $\phi$   $\kappa$ ))
```

# Restart interpretation

```
(define (transform-trace-non-looping trace)
  '(letrec ((non-loop ,(append '(lambda ()) trace)))
    (non-loop)
      ; Use  $\tau$ - $\kappa$  to reconstruct state
    (let ((new-state (ko (car  $\tau$ - $\kappa$ ) (cdr  $\tau$ - $\kappa$ ))))
      (pop-continuation)
      new-state)))
```

# Non-looping label-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))))

(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)
        ; Actually execute the trace
        (let ((state (execute-trace ',trace)))
          (pop-label-trace-executing!)
          state)))))
```

# Non-looping guard-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 ()
         ; Actually execute the trace
         (let* ((state (execute-trace ',trace)))
           ; Actually incorrect...
           (bootstrap-to-evaluator 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  $\tau$ - $\kappa$ )))

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

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

# 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 ()
        ; Actually execute the trace
        (let* ((state (execute-trace ',trace)))
          ; Incorrect...
          (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)
        ; Actually execute the trace
        (let ((state (execute-trace ',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))
               ; Also safe without trace jumping
               (new-state
                (execute-label-trace-with-label
                 label)))
          ; When evaluating non-looping trace no
          ; need for bootstrapping
          (step* new-state)))
```

# Trace execution

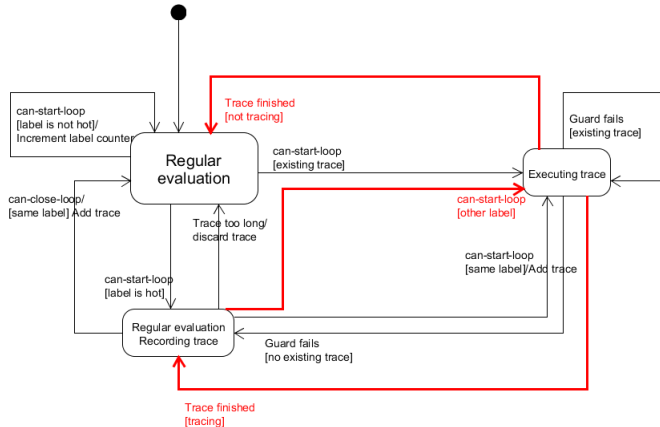
Why are guard-trace executions incorrect?

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

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



# Trace jumping



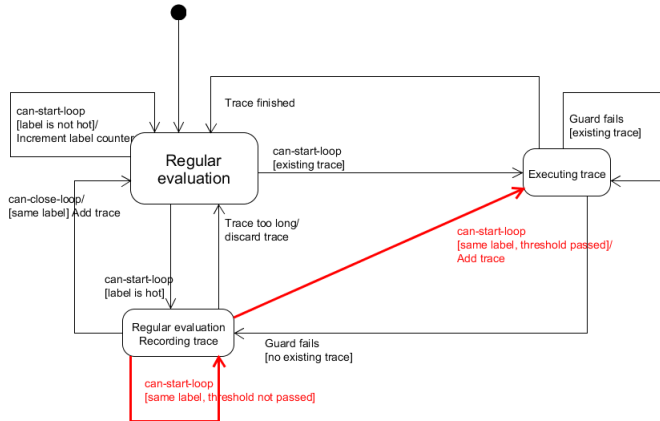
# Trace jumping

## Remember

```
(define (handle-can-start-loop-annotation label debug-info state)
  (cond ; Don't check whether already tracing
        ((label-trace-exists? label)
         ; Execute trace
         (let* ((label-trace (get-label-trace label))
                ((new-state (execute-label-trace-with-label label)))
                ...))))

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

# True vs false loops



# True vs false loops

True loops = functions which have recursed at least x times

```
(struct tracer-context (...
    times-label-encountered-while-tracing
    ...))

(define (handle-can-start-loop-annotation-reg label debug-info state)
  (cond ((is-tracing-label? tracer-context label)
    (check-stop-tracing-label label state))))

(define (check-stop-tracing-label tracer-context label state)
  (define (do-stop-tracing!)
    (stop-tracing! tracer-context #t)
    (let ((new-state (execute-label-trace-with-label label)))
      (step* new-state)))
  (define (do-continue-tracing)
    (execute/trace '(pop-continuation))
    (step* state))
  (inc-times-label-encountered-while-tracing!)
  (if (times-label-encountered-greater-than-threshold?)
    (do-stop-tracing!)
    (do-continue-tracing)))
```

# 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 (or (not (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!