

A Formal Foundation for Trace-Based JIT Compilation

Vandercammen, Maarten; Nicolay, Jens; De Roover, Coen

Publication date:
2015

License:
Unspecified

Document Version:
Accepted author manuscript

[Link to publication](#)

Citation for published version (APA):
Vandercammen, M., Nicolay, J., & De Roover, C. (2015). *A Formal Foundation for Trace-Based JIT Compilation*. Poster session presented at SPLASH 2015, Pittsburgh, United States.
<https://soft.vub.ac.be/Publications/2015/vub-soft-tr-15-17.pdf>

Copyright

No part of this publication may be reproduced or transmitted in any form, without the prior written permission of the author(s) or other rights holders to whom publication rights have been transferred, unless permitted by a license attached to the publication (a Creative Commons license or other), or unless exceptions to copyright law apply.

Take down policy

If you believe that this document infringes your copyright or other rights, please contact openaccess@vub.be, with details of the nature of the infringement. We will investigate the claim and if justified, we will take the appropriate steps.

A Formal Foundation for Trace-Based JIT Compilation

Maarten Vandercammen

Jens Nicolay

Coen De Roover

Software Languages Lab, Vrije Universiteit Brussel, Belgium

Problem: design formal framework for trace-based JIT compilation

Trace-based JIT Compilation

```
(define (fac n)
  (if (< n 2)
      1
      (* n
         (fac (- n 1)))))
```

```
(label 'fac-loop)
(literal-value 2)
(save-val)
(lookup-var 'n)
(save-val)
(lookup-var '<)
(apply-native 2)
(guard-false)
(literal-value 1)
(save-val)
(lookup-var 'n)
(save-val)
(lookup-var '-')
(apply-native 2)
...
(goto 'fac-loop)
```

Guo and Palsberg (2011)
Disegna et al. (2014)

Tracing tied to specific execution model

Language semantics (interpreter)

Research goal

Traced language semantics (TJIT Compiler)

Recording/Executing traces

Handling guard failures

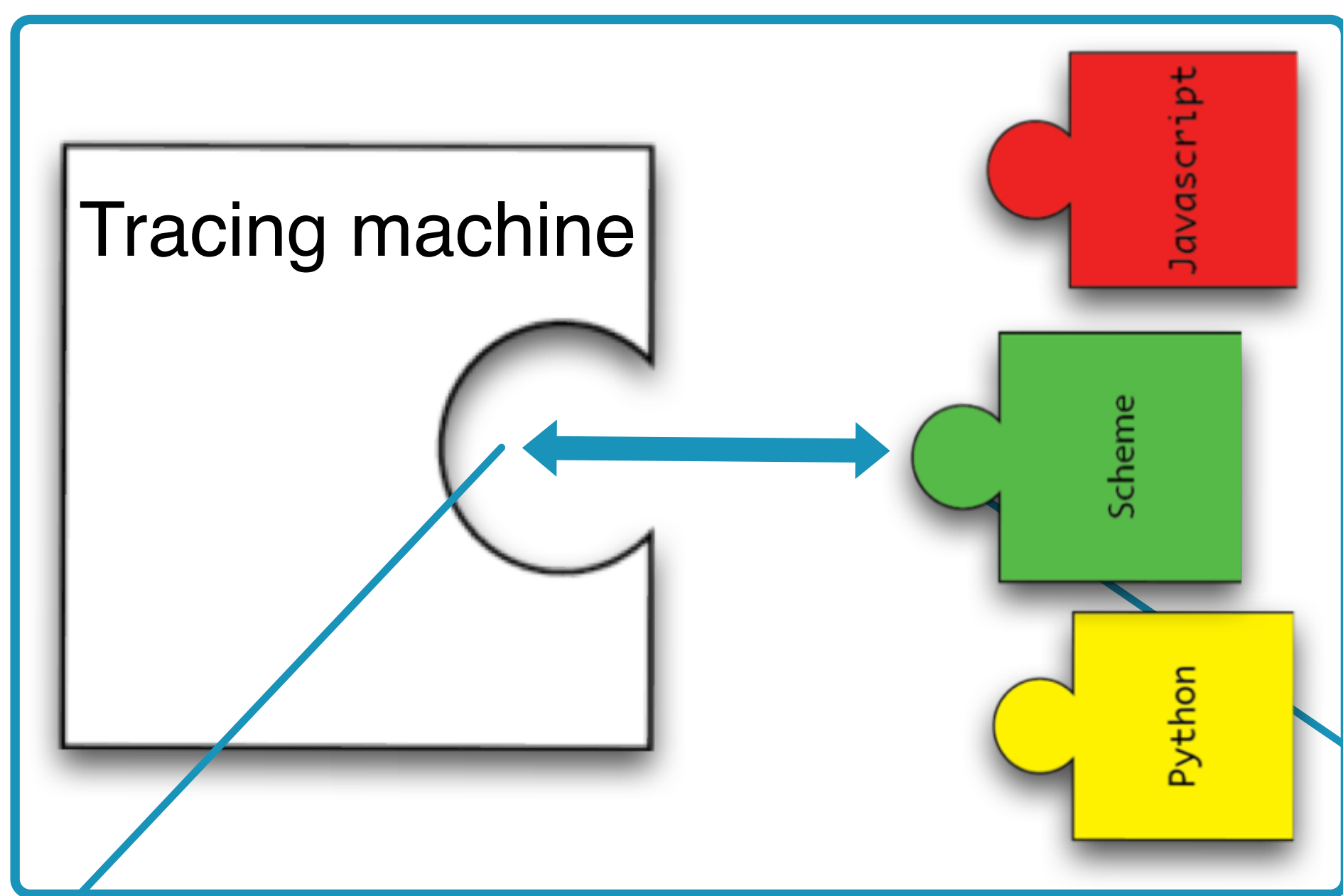
Dynamic analysis hook

From Local optimisations on traces

Common framework for trace and non-trace execution

Towards Global optimisations between traces

Approach: separate tracer and interpreter by creating an explicit interface



Tracer/Interpreter interface

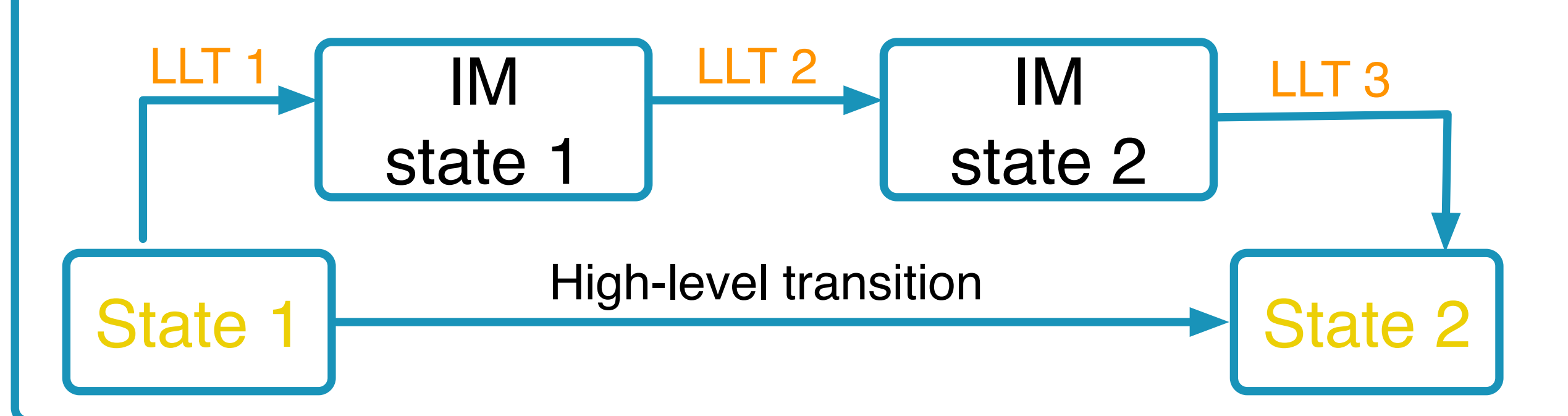
Interpreter as state machine

optimize: Trace → Trace

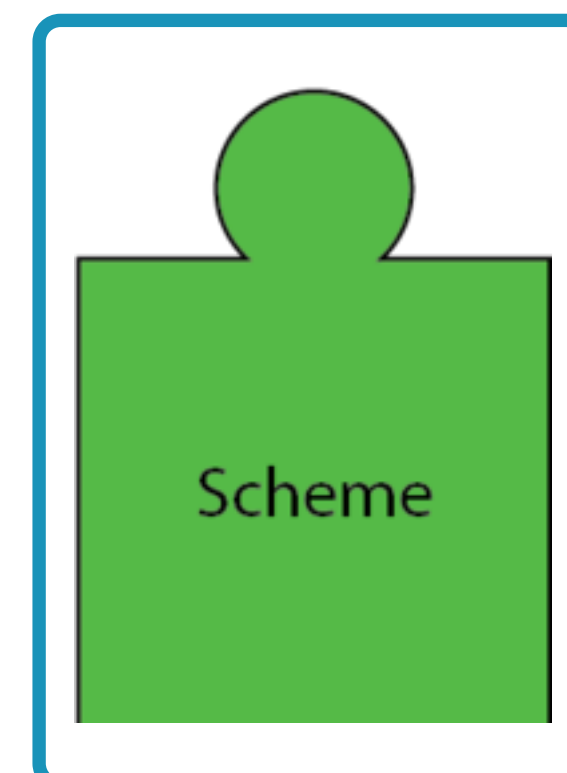
restart: RestartPoint X ProgramState → ProgramState

step: ProgramState → InterpreterReturn

<ProgramState, LLT's, Loop signal>

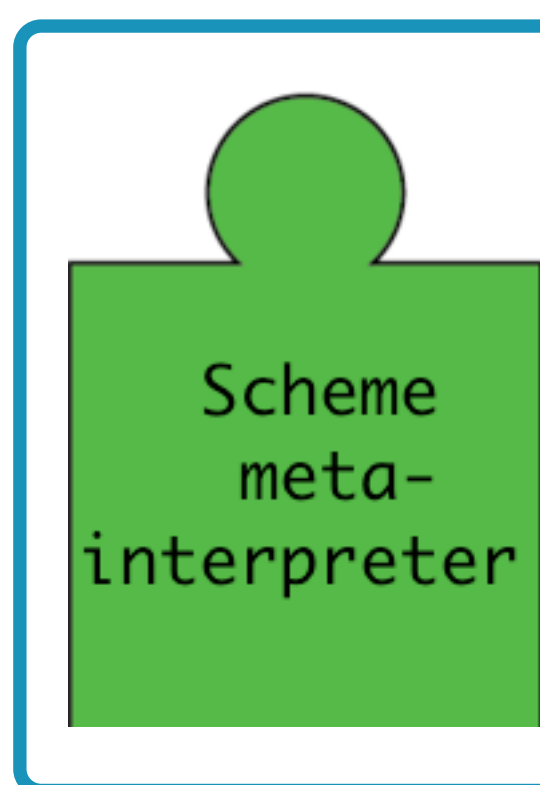


Experiment: apply transformation on existing interpreters



```
(match program-state
  ...
  ((ifk condition consequent alternative) ρ σ κ))
  (if condition
    (interpreter-return
     program-state
     (ev consequent)
     #f
     (restore-env)
     (pop-continuation)
     (guard-true alternative)))
  (interpreter-return
   program-state
   (ev alternative)
   #f
   (restore-env)
   (pop-continuation)
   (guard-false consequent))))
...)
```

Minimal changes to implementation required



Meta-tracing compiler

User program

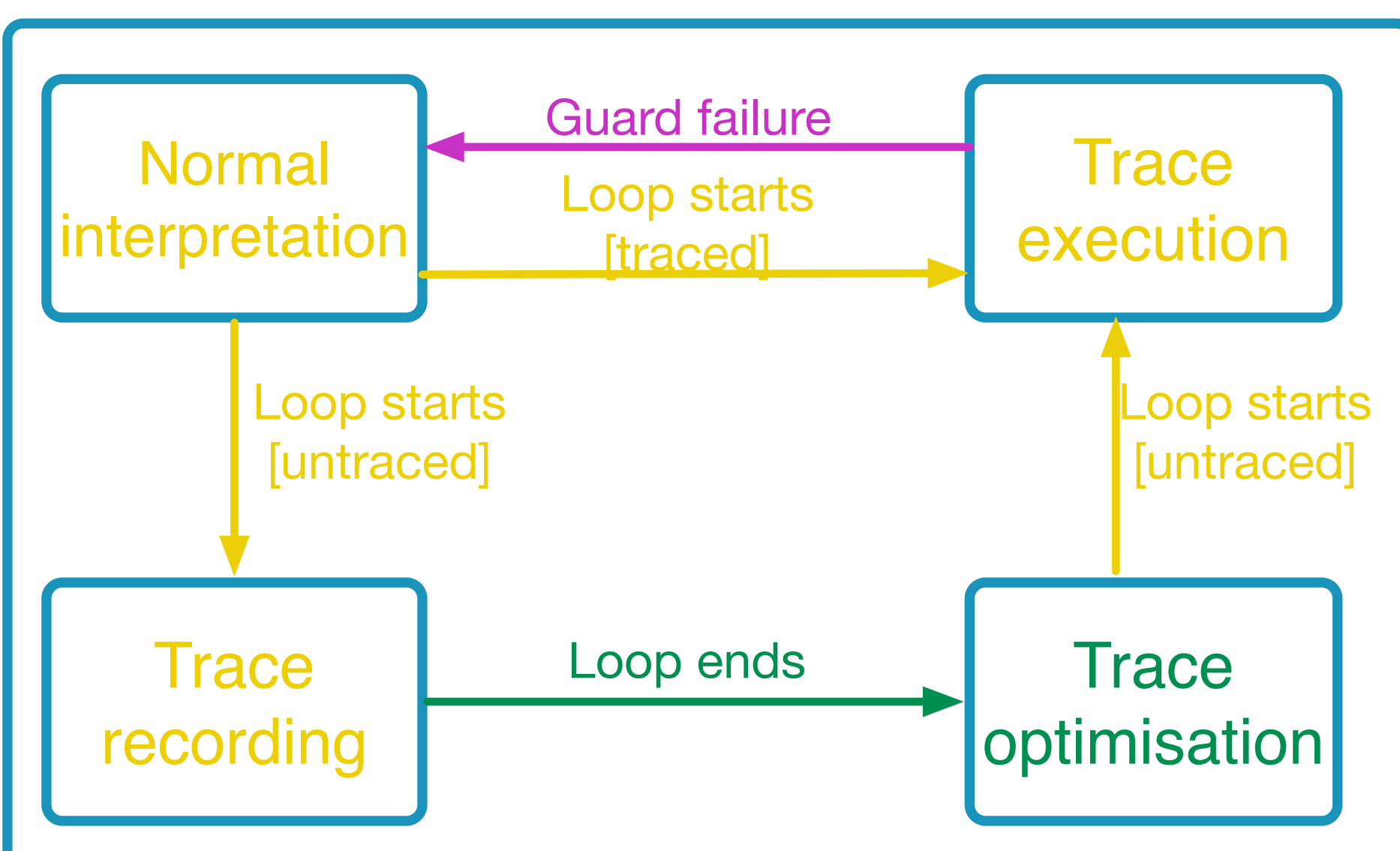
Language interpreter

Tracing interpreter

Underlying runtime

Implementation: formalism + implementation

State machine



Formal semantics

```
ts(TE, tc, ς, tn(lbl, ι : τ)) →
  ts(TE, tc, ς', tn(lbl, τ))
  if ι(ς) = traceStep(ς')
ts(TE, tc, ς, tn(lbl, ι : τ)) →
  ts(NI, tc, ς', False)
  if ι(ς) = guardFailed(rp)
  and where ς' = restart(rp, ς)
ts(TE, tc, ς, tn(lbl, '())) →
  ts(TE, tc, ς, tn(lbl, τ))
  and where τ is the trace that has already been recorded for lbl
```

Implementation

```
(define (do-trace-executing-step)
  (let* ((trace-node (evaluator-state-trace-executing
                     evaluator-state))
        (trace (trace-node-trace trace-node))
        (label (trace-key-label (trace-node-trace-key
                                trace-node))))
    (if (null? trace)
        (let* ((old-trace-node (evaluator-state-trace-executing
                                evaluator-state))
              (old-trace-key (trace-node-trace-key old-trace-node))
              (trace-key (make-label-trace-key
                          (trace-key-label old-trace-key)
                          (trace-key-debug-info old-trace-key)))
              (new-trace-node (get-trace tracer-context trace-key)))
          (evaluator-state-copy evaluator-state
                                (trace-executing new-trace-node)))
        (let* ((instruction (car trace))
              (program-state (evaluator-state-program-state evaluator-state)))
          (handle-response-executing (instruction program-state))))))
```