Trace Based Just-In-Time Type Specialization for Dynamic Languages (PLDI 2009)

Gal, Eich, Shaver, Anderson, Mandelin, Kaplan, Hoare, Zbarsky, Orendorff, Ruderman, Smith, Reitmaier, Haghighat, Bebenita, Chang, Franz

16th February 2015

Outline

Motivation

- Generate efficient machine code for dynamic type languages like Javascript.
- Challenges
  - Types decided at runtime
  - Even type inferencing does not help much

Motivation

- Generate efficient machine code for dynamic type languages like Javascript.
- Challenges
  - Types decided at runtime
  - Even type inferencing does not help much

Motivation

- Generate efficient machine code for dynamic type languages like Javascript.
- Challenges
  - Types decided at runtime
  - Even type inferencing does not help much

TraceMonkey: Proposed Compilation strategy

- Is implemented for Javascript Interpreter SpiderMonkey
- Dynamic compilation on loop level granularity, not method based. Why??

TraceMonkey: Proposed Compilation strategy

- Is implemented for Javascript Interpreter SpiderMonkey
- Dynamic compilation on loop level granularity, not method based. Why??

Trace

- Typed loop traces
  - Single entry, multiple exits loop paths
  - Type specialized compilation

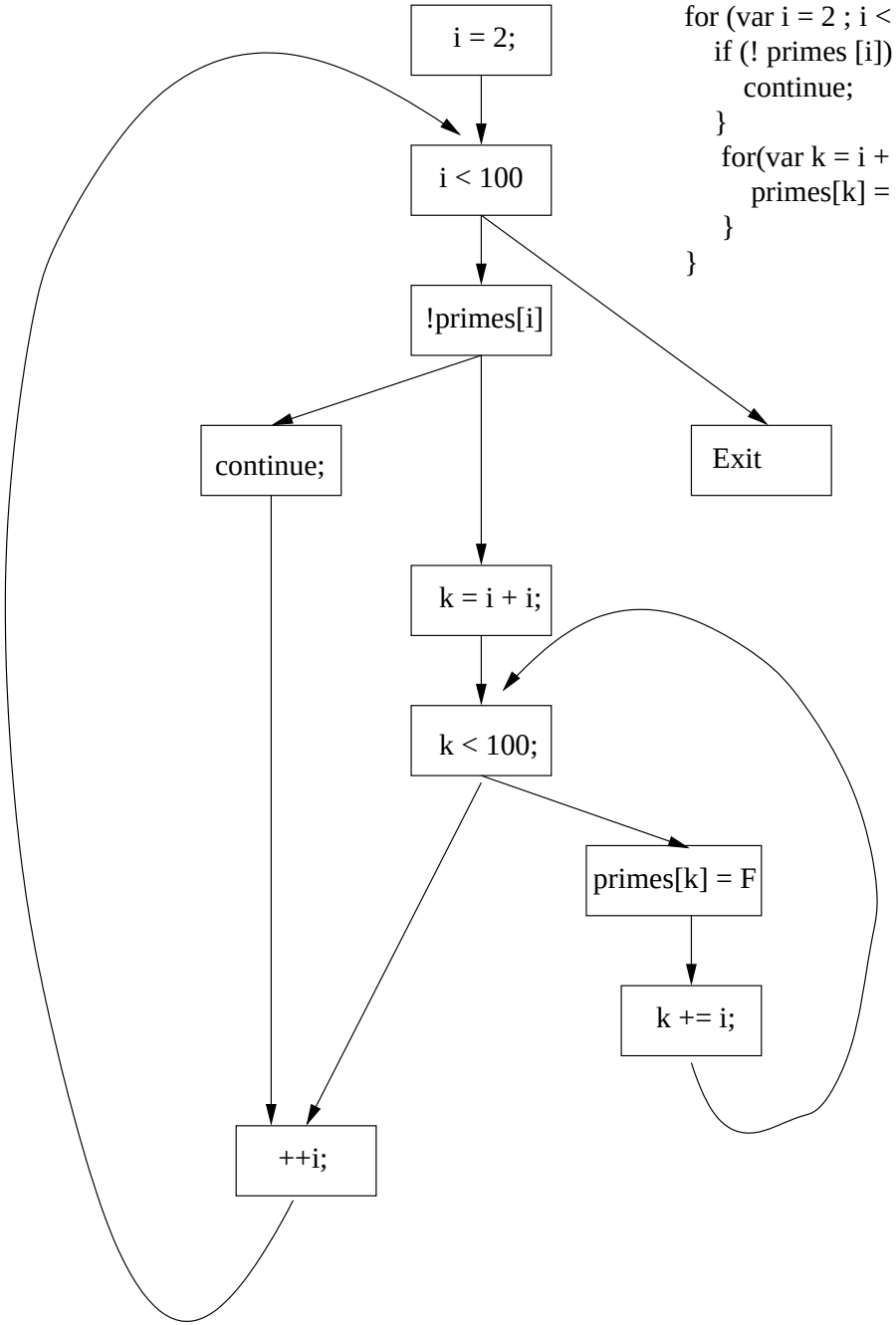
Trace

- Typed loop traces
  - Single entry, multiple exits loop paths
  - Type specialized compilation

Trace

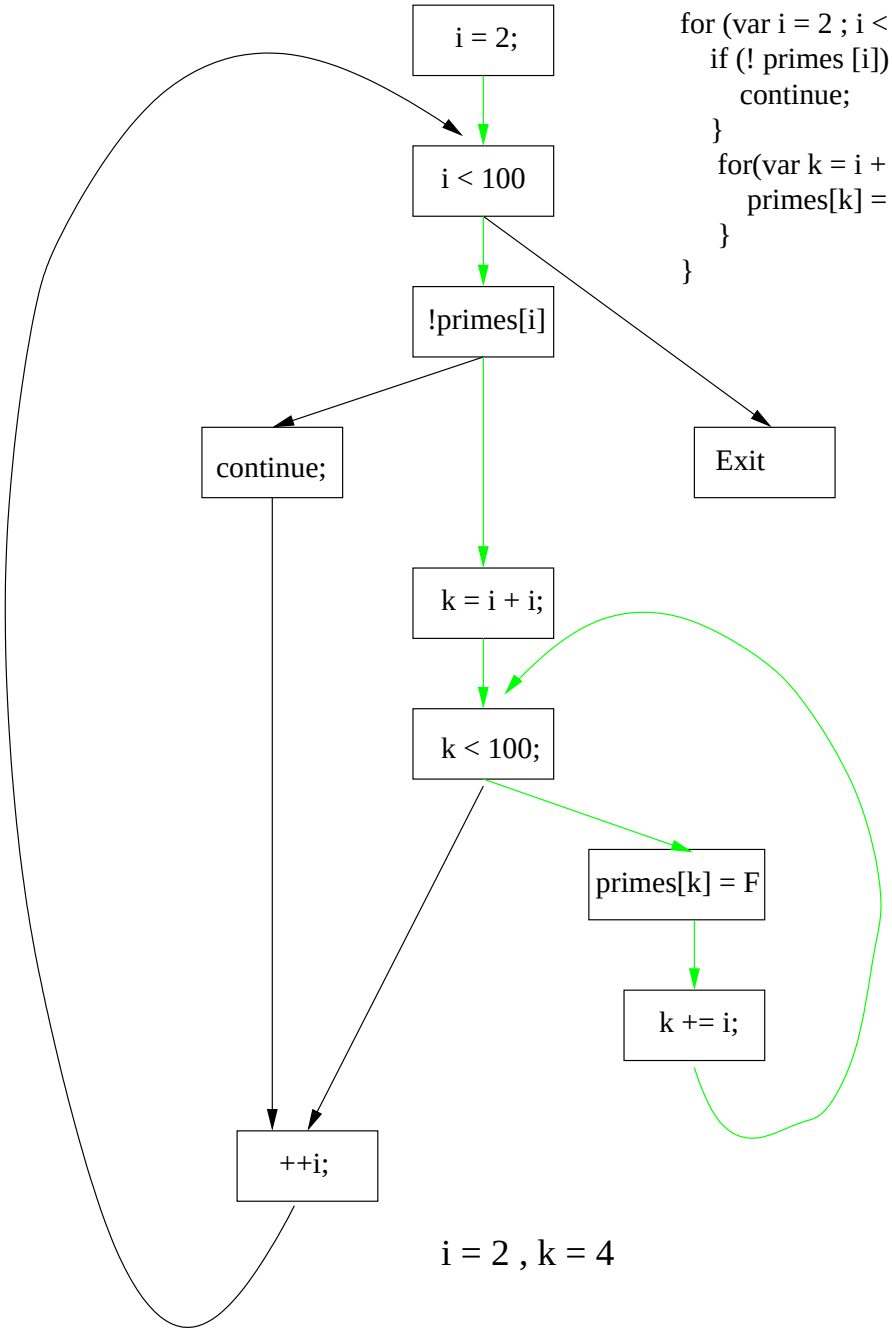
- Typed loop traces
  - Single entry, multiple exits loop paths
  - Type specialized compilation

Outline




```
for (var i = 2 ; i < 100 ; ++i) {
   if (! primes [i]) {
      continue;
   }
    for(var k = i + i ; k < 100 ; k += i) {
       primes[k] = false;
    }
}
```

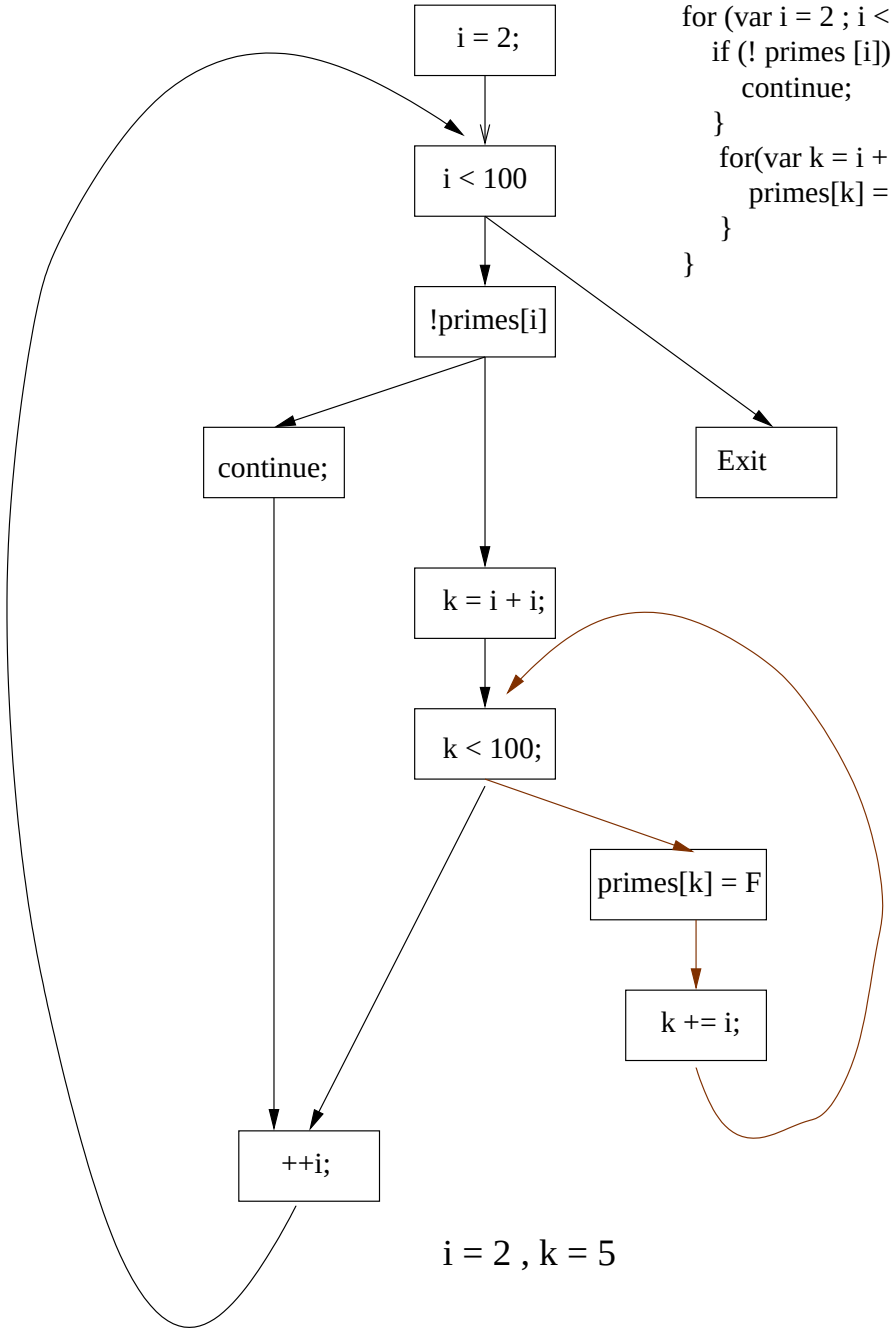
interpreting
recording
compiled code
side exit check (guard)




```
for (var i = 2 ; i < 100 ; ++i) {
   if (! primes [i]) {
      continue;
   }
    for(var k = i + i ; k < 100 ; k += i) {
       primes[k] = false;
    }
}
```

interpreting
recording
compiled code

i = 2 , k = 4

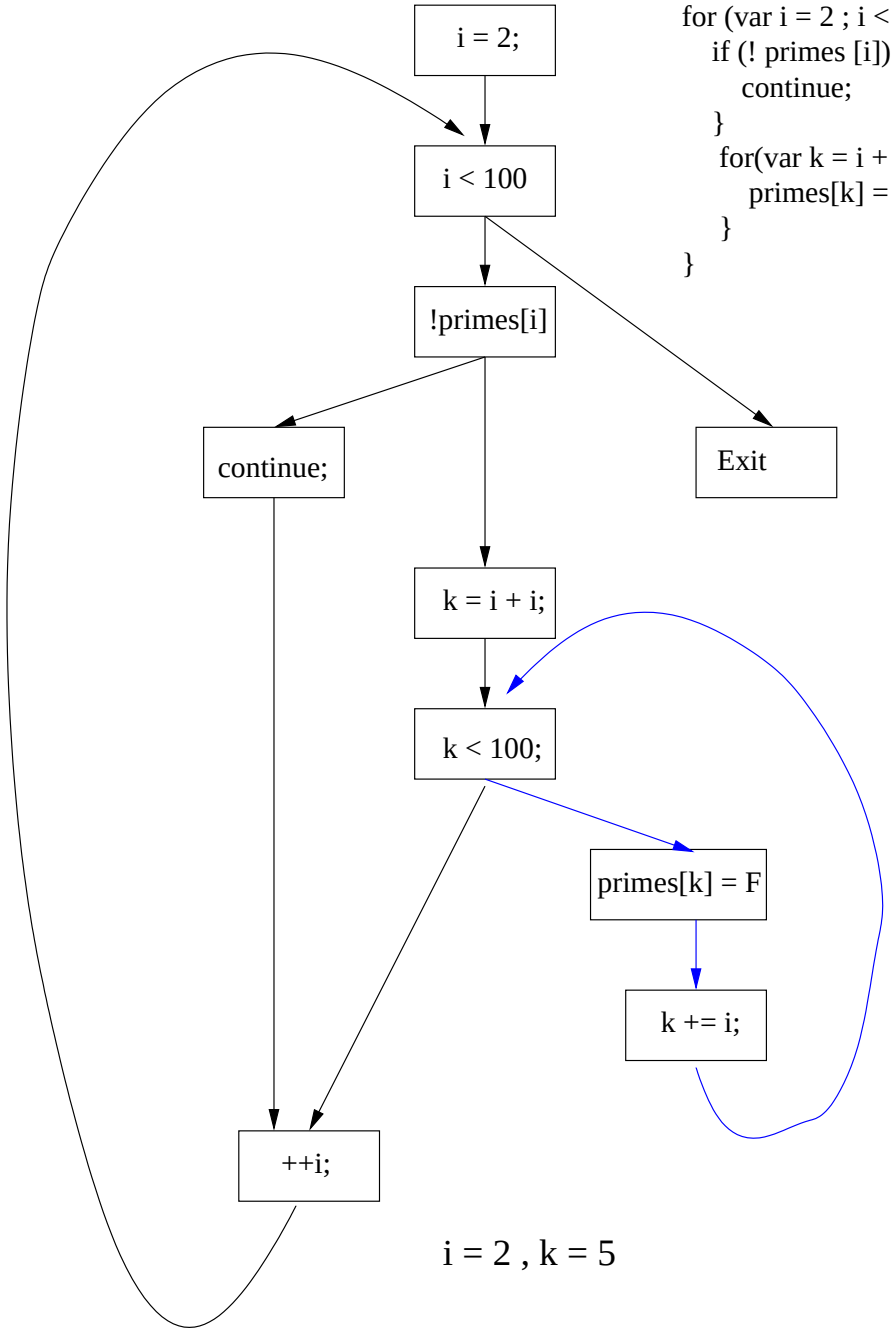



```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

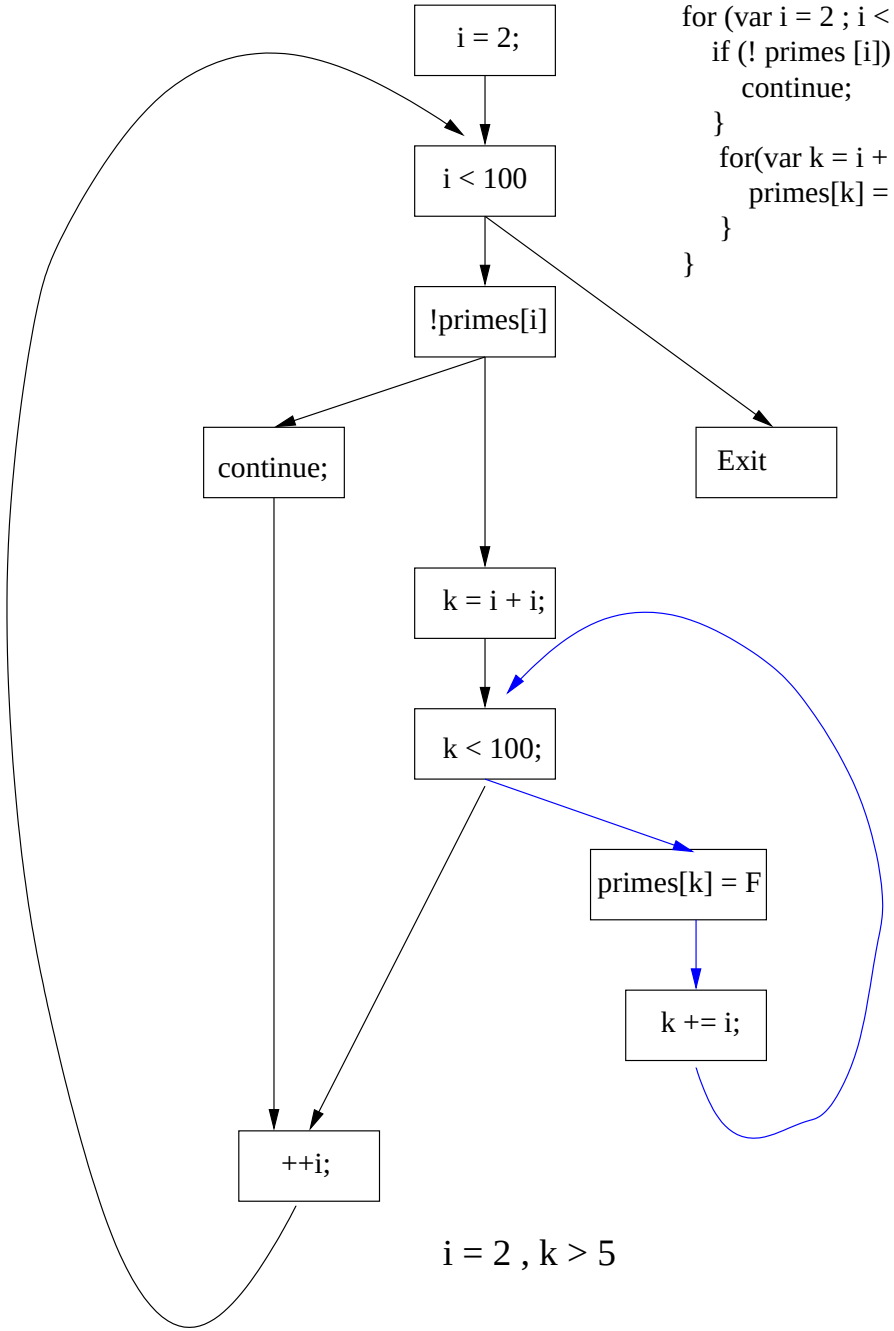
interpreting
recording
compiled code
side exit check (guard)

i = 2 , k = 5

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```



interpreting
recording
compiled code
side exit check (guard)



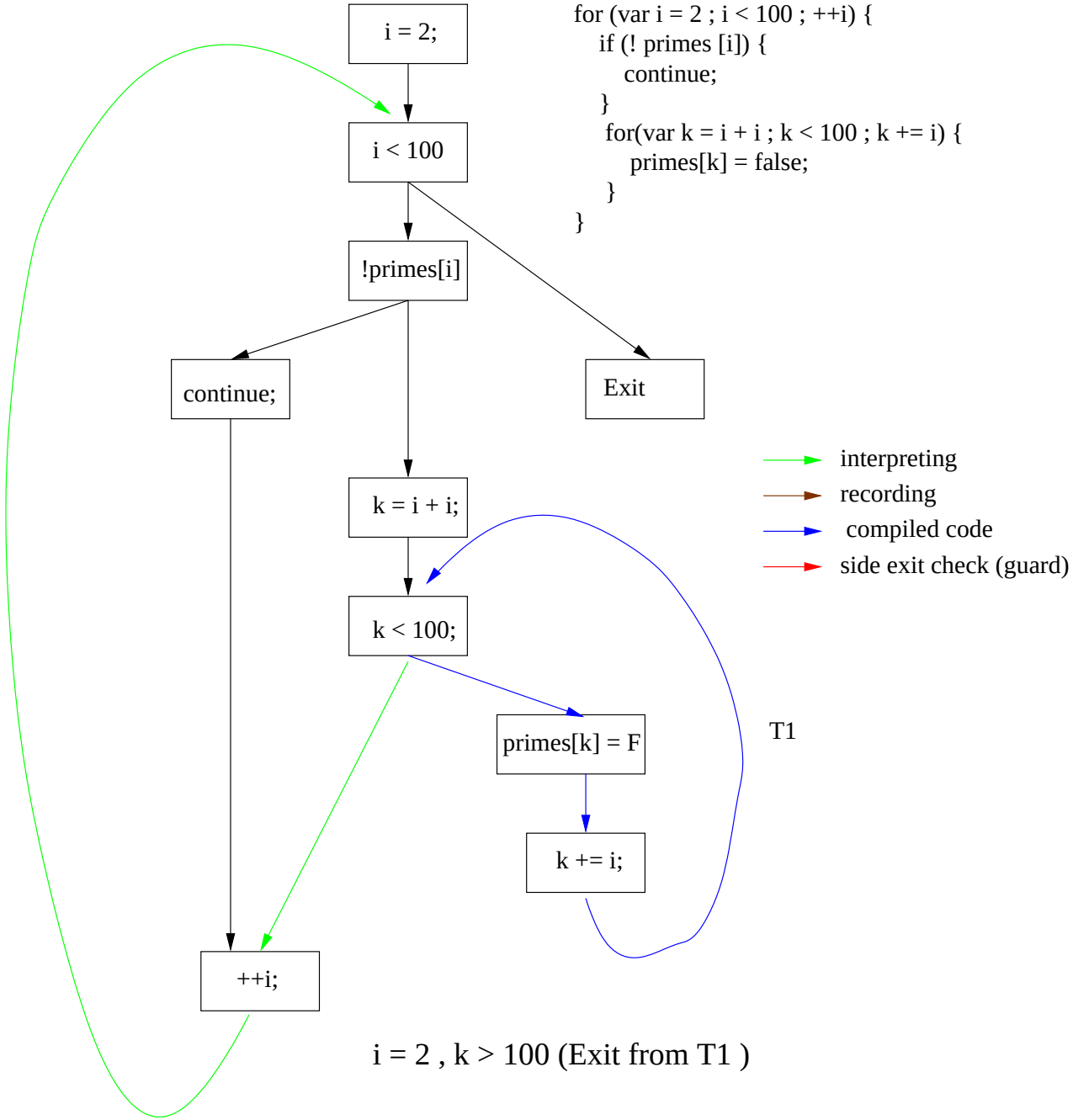

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

interpreting
recording
compiled code
side exit check (guard)

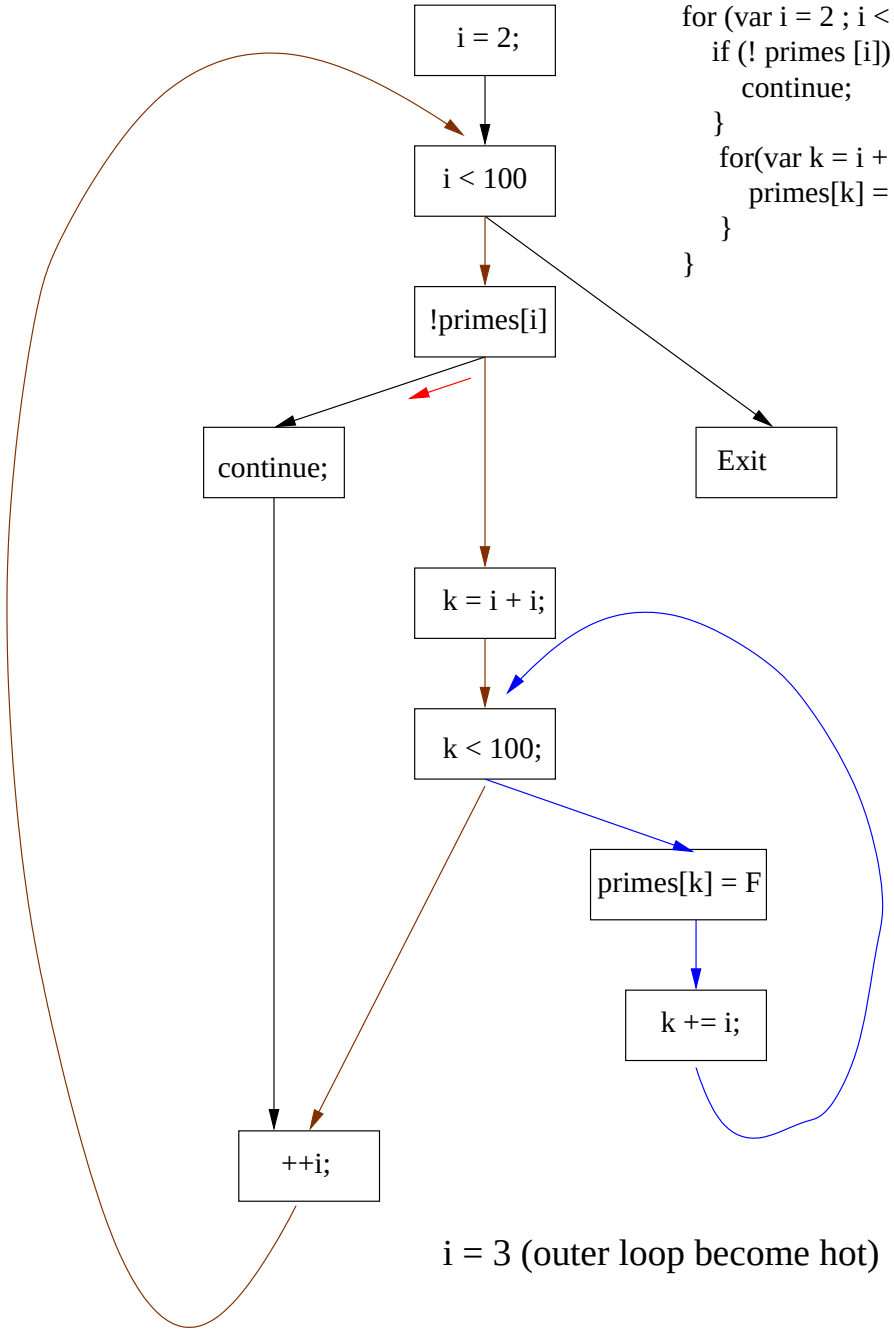
T1

i = 2 , k > 5

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```



i = 2 , k > 100 (Exit from T1)

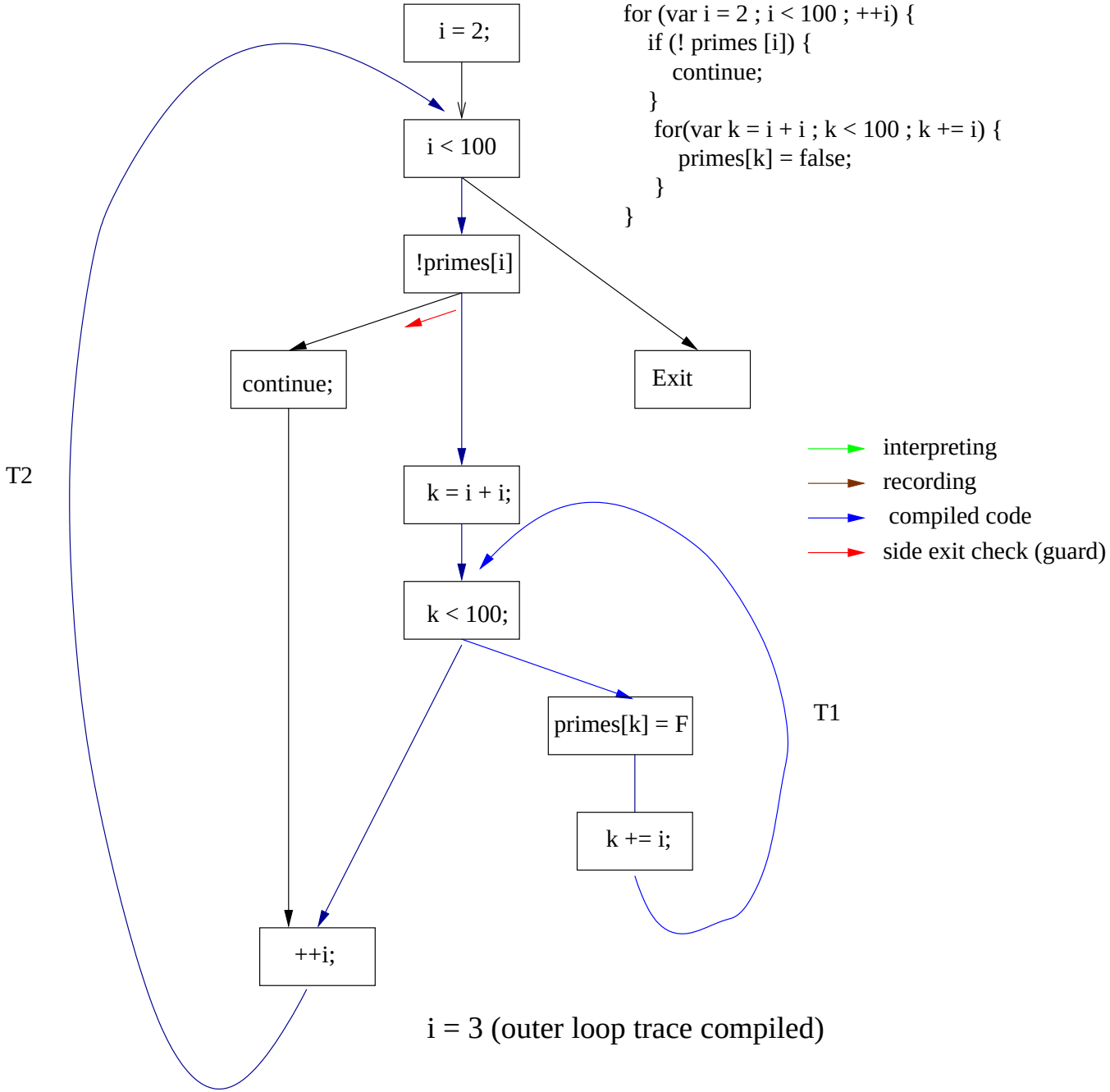



```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

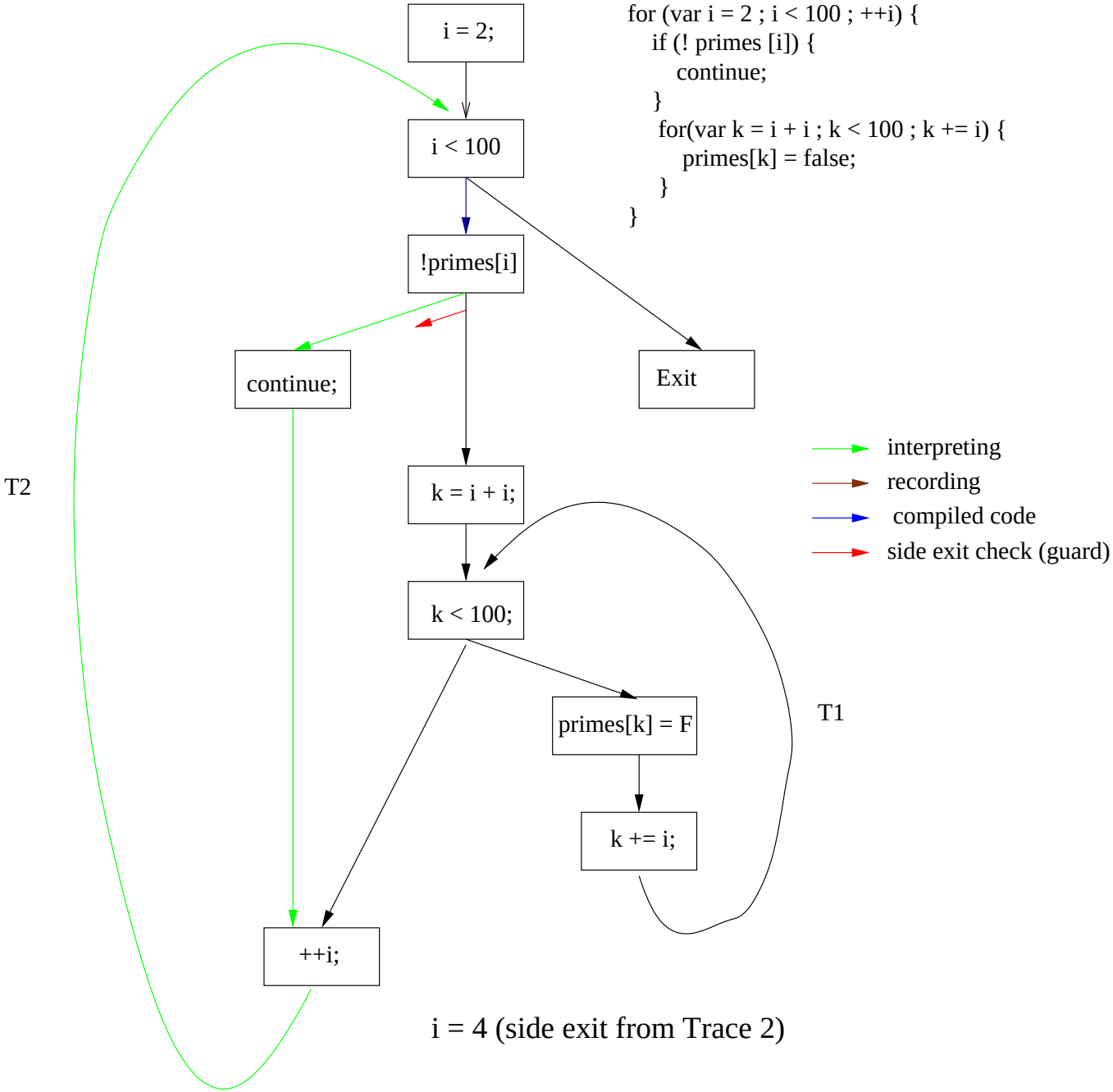
interpreting
recording
compiled code
side exit check (guard)

T1

i = 3 (outer loop become hot)

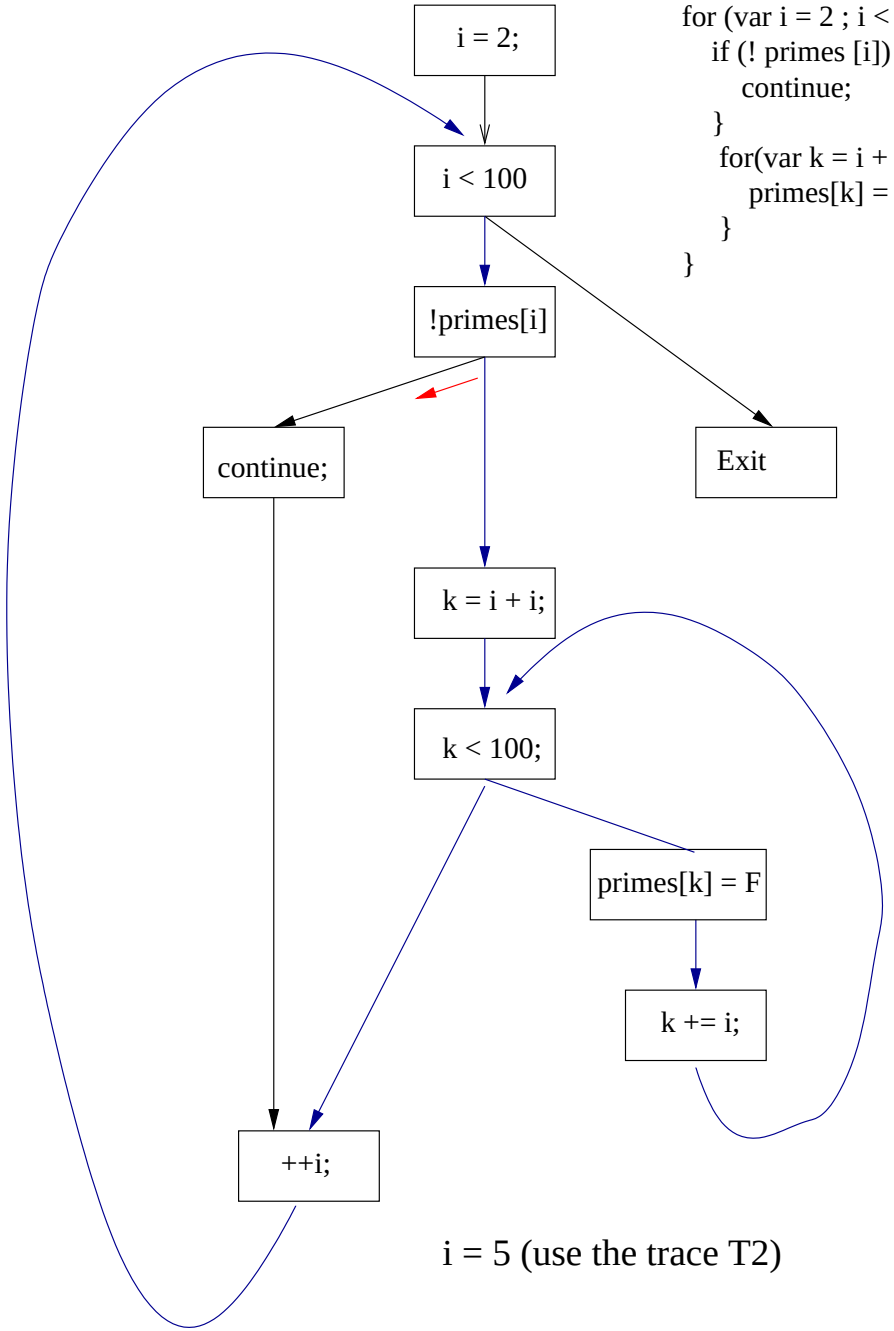


i = 3 (outer loop trace compiled)


for (var i = 2 ; i < 100 ; ++i) {
if (! primes [i]) {
continue;
}
for(var k = i + i ; k < 100 ; k += i) {
primes[k] = false;
}
}
i = 2;
i < 100
!primes[i]
continue;
Exit
k = i + i;
k < 100;
primes[k] = F
k += i;
++i;
T2
T1
interpreting
recording
compiled code
side exit check (guard)
i = 4 (side exit from Trace 2)

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

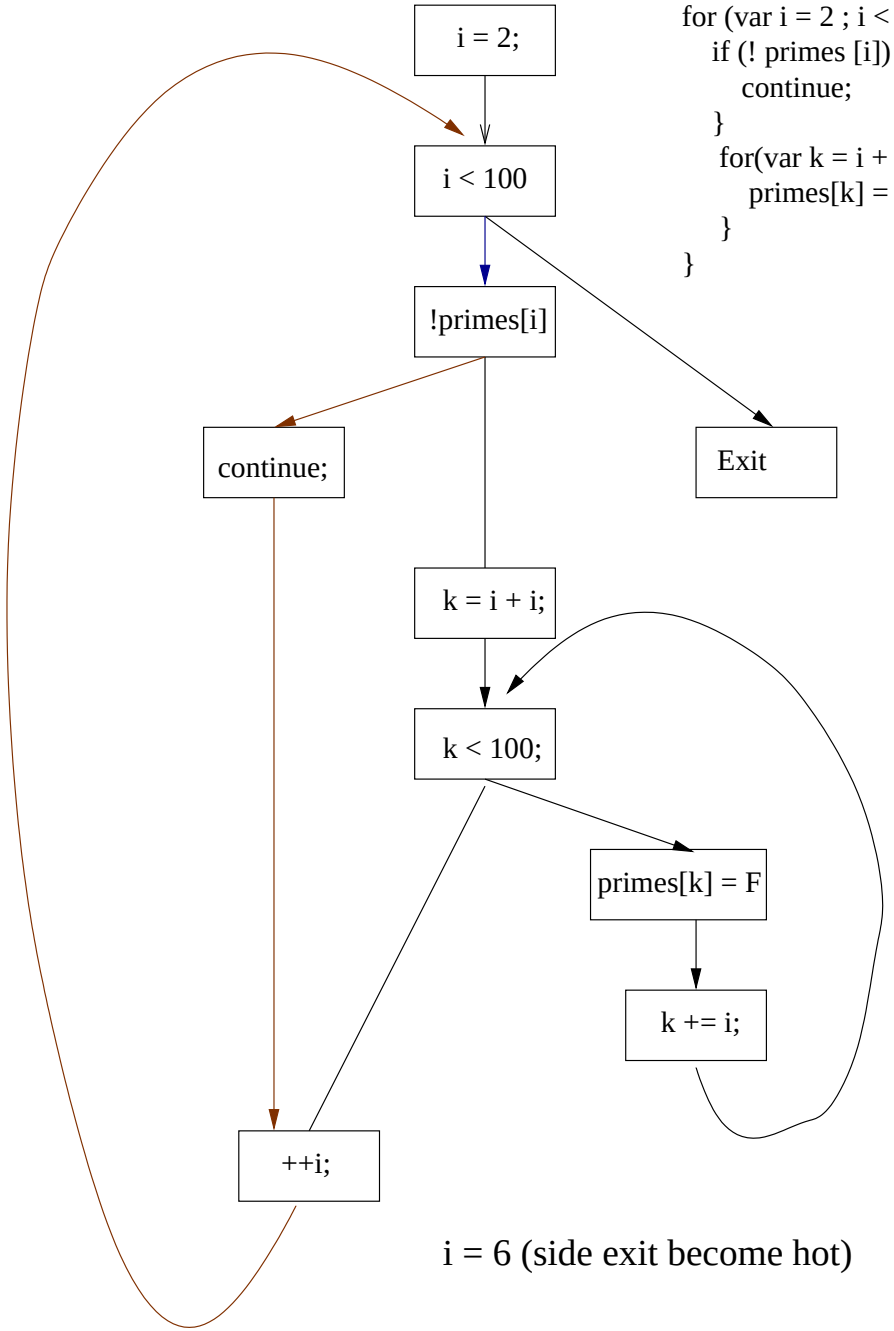
T2

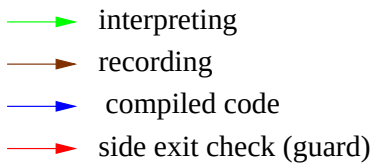


interpreting
recording
compiled code
side exit check (guard)

i = 5 (use the trace T2)

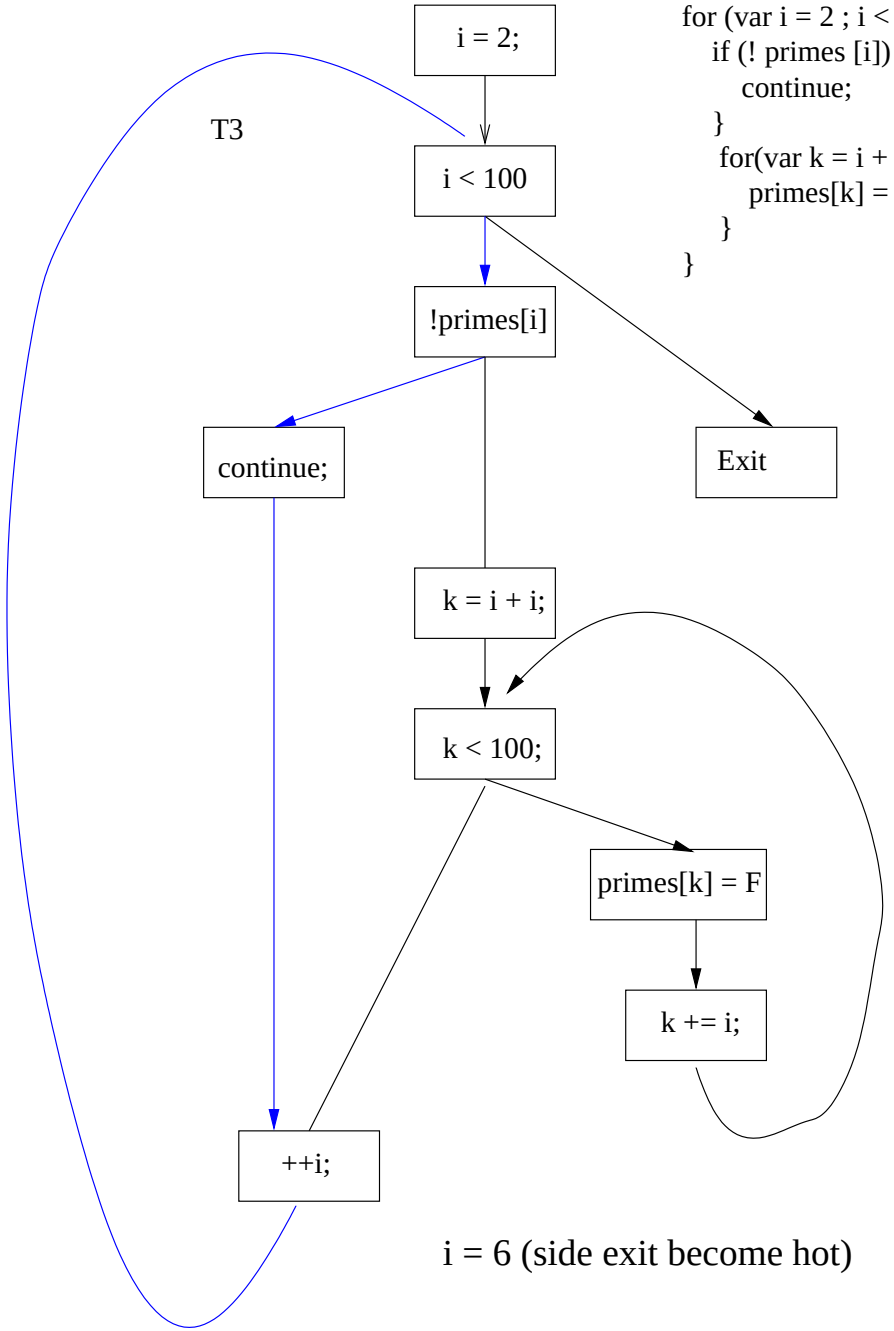
```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

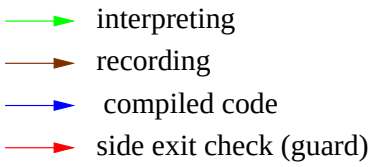




i = 6 (side exit become hot)

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

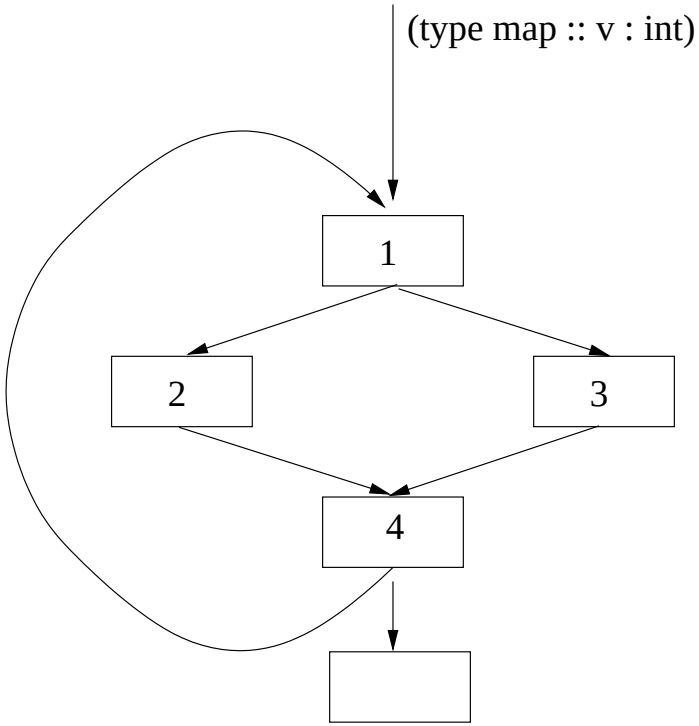




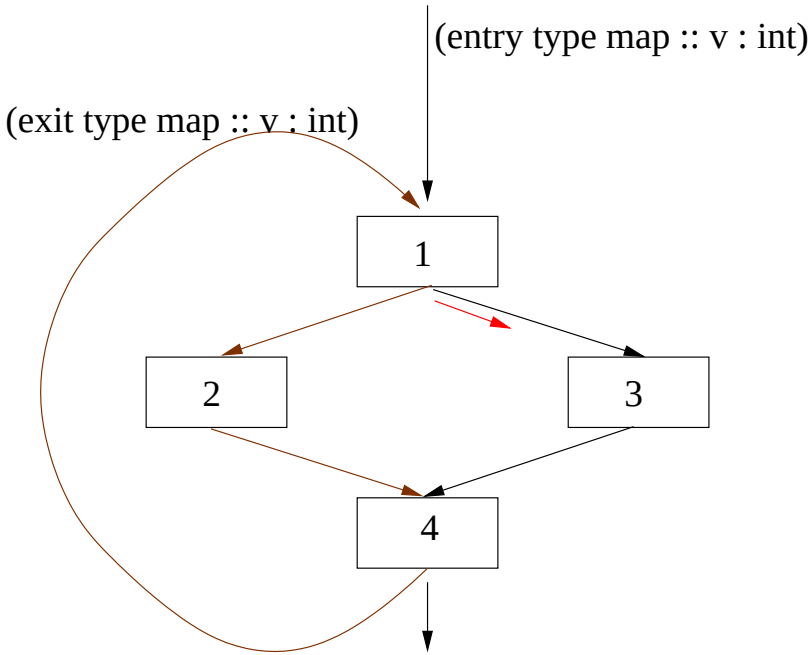
i = 6 (side exit become hot)

Outline

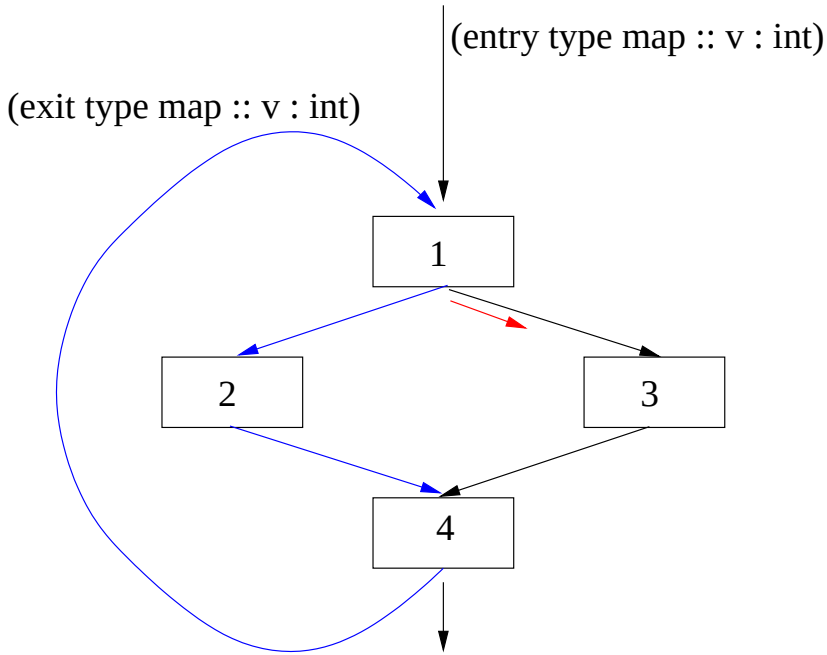
Type Stable Trace



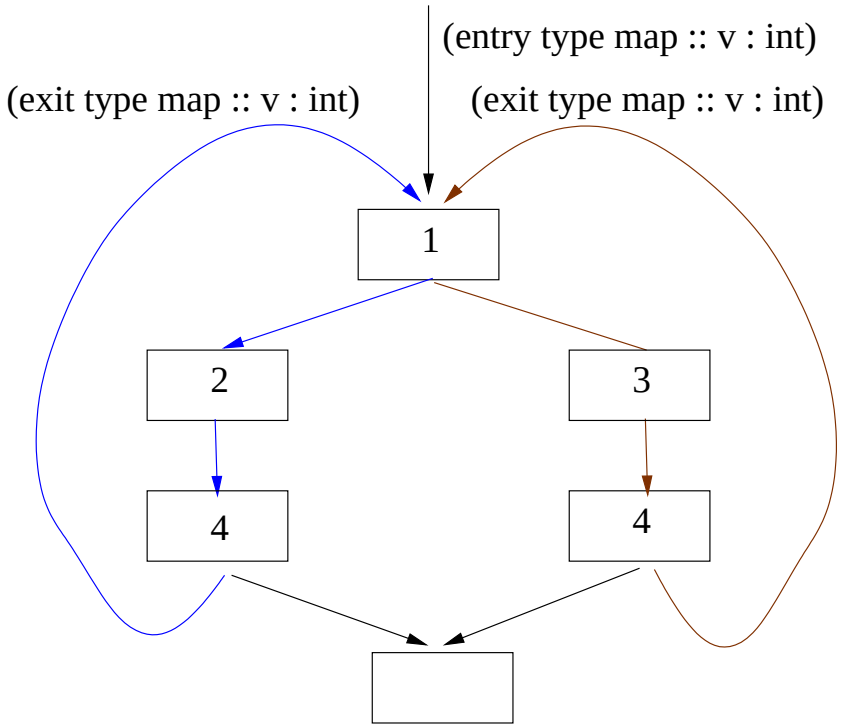
Type Stable Trace



Type Stable Trace



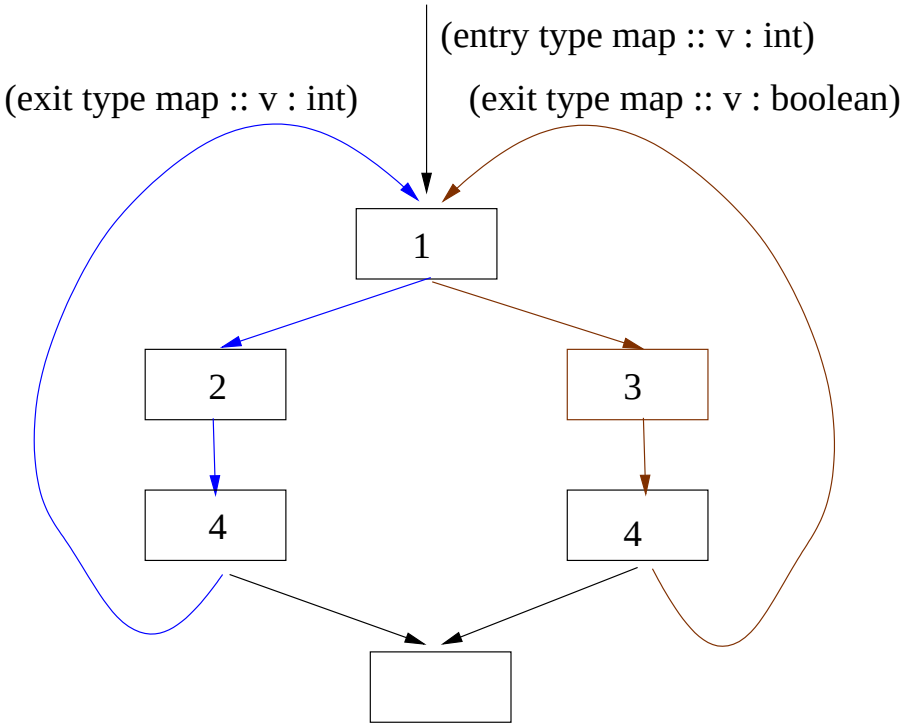
Trace Tree Extension



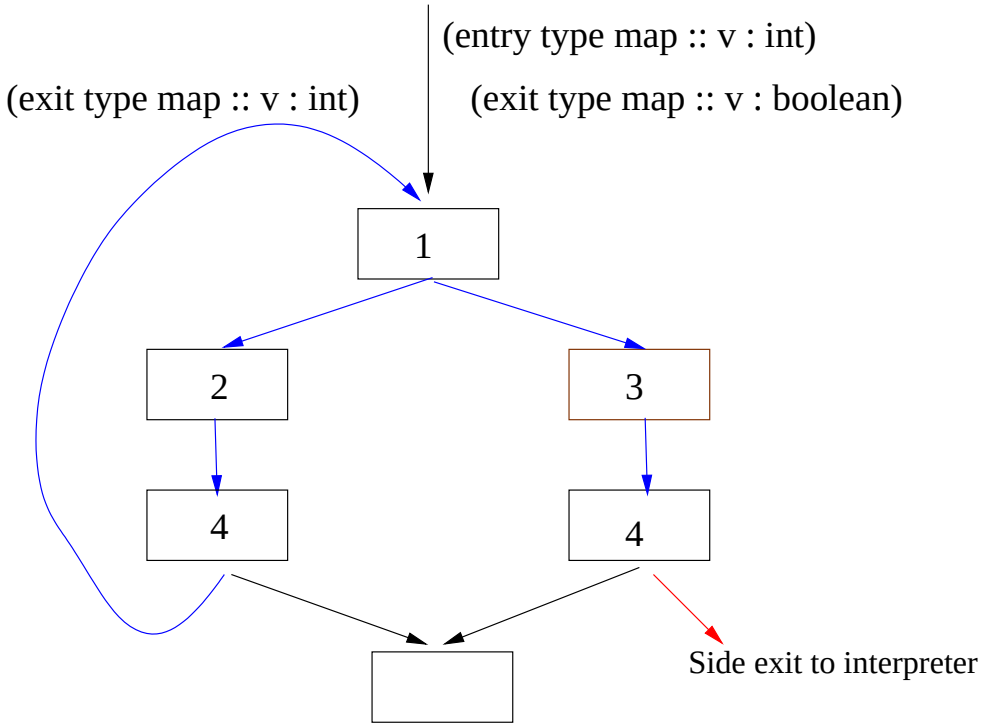
Note: Current Implepentation extends only if

- side exit is for control flow branch
- does not leave the loop

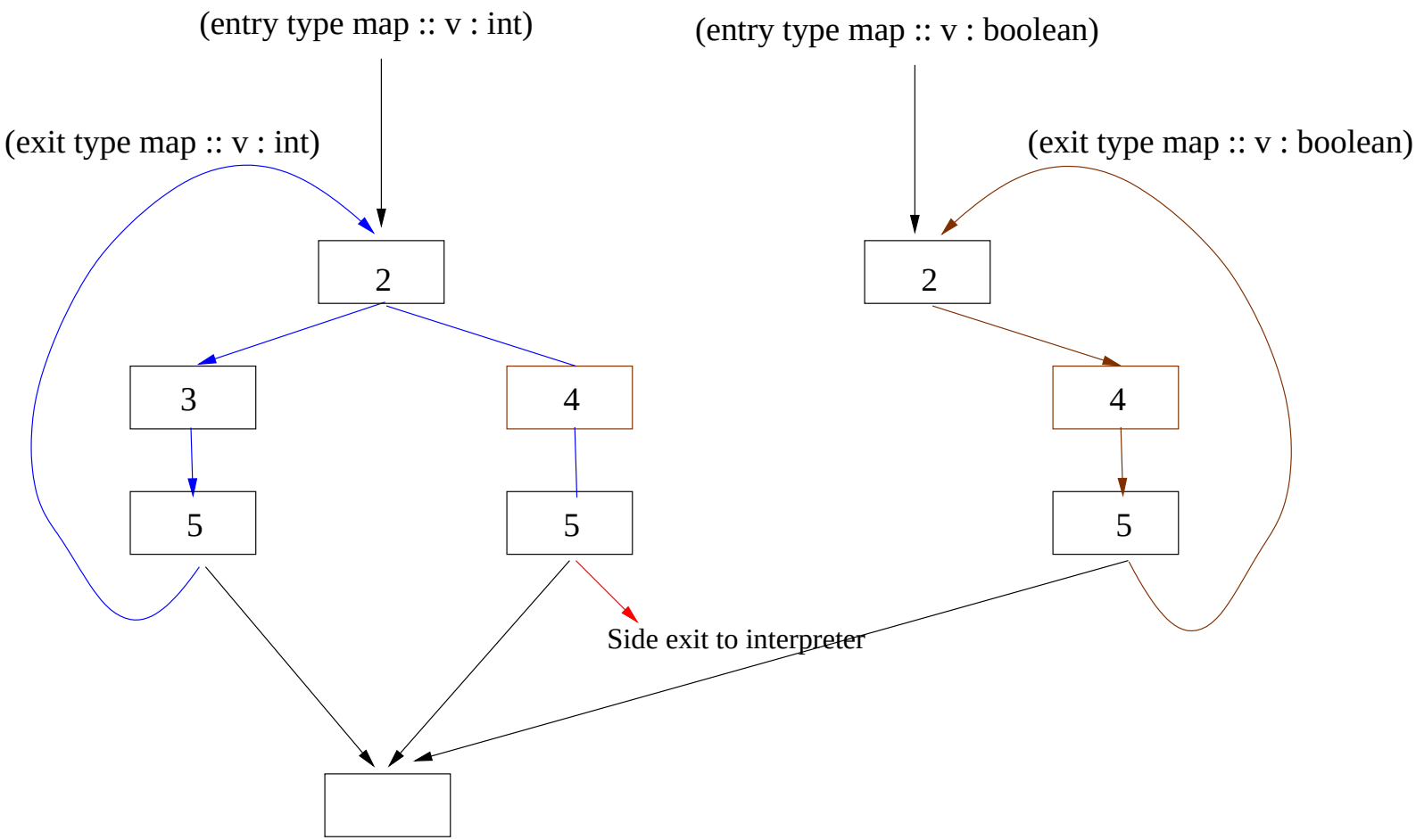
Type Unstable Trace Handling



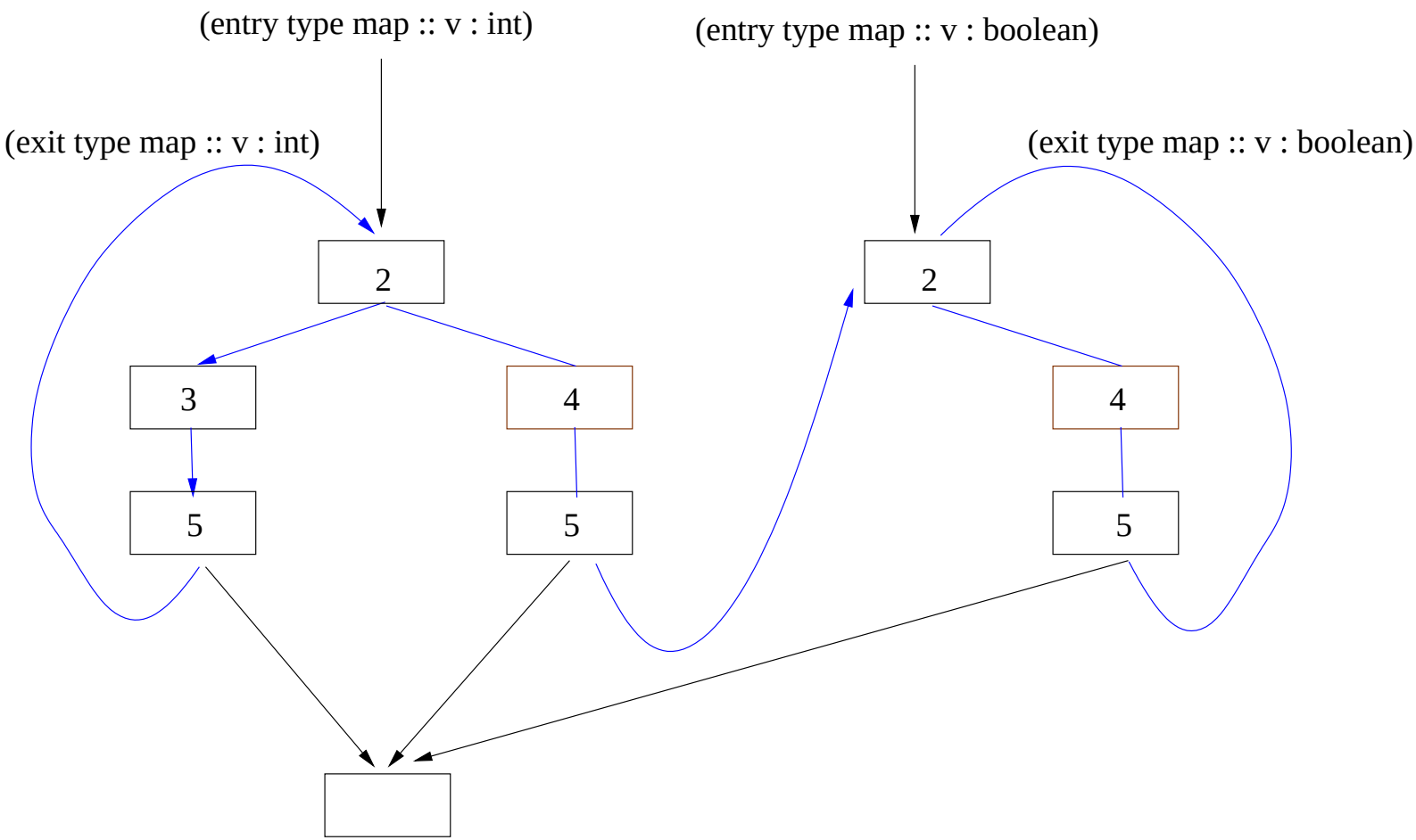
Type Unstable Trace Handling



Type Unstable Trace Handling



Type Unstable Trace Handling



Blacklisting

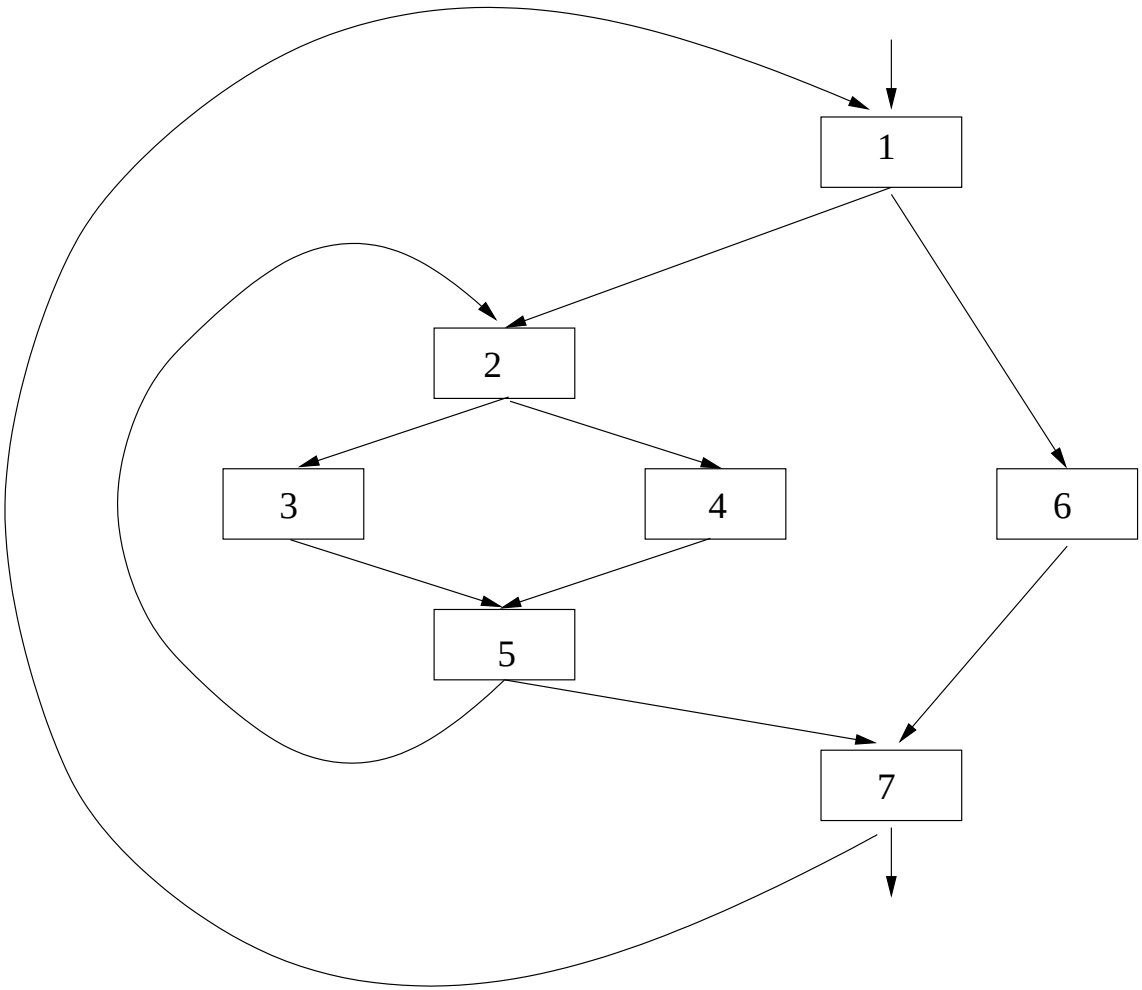
- What if a hot loop contain traces that always fail recording ?
- Solution : Blacklist with backoff on recording

Blacklisting

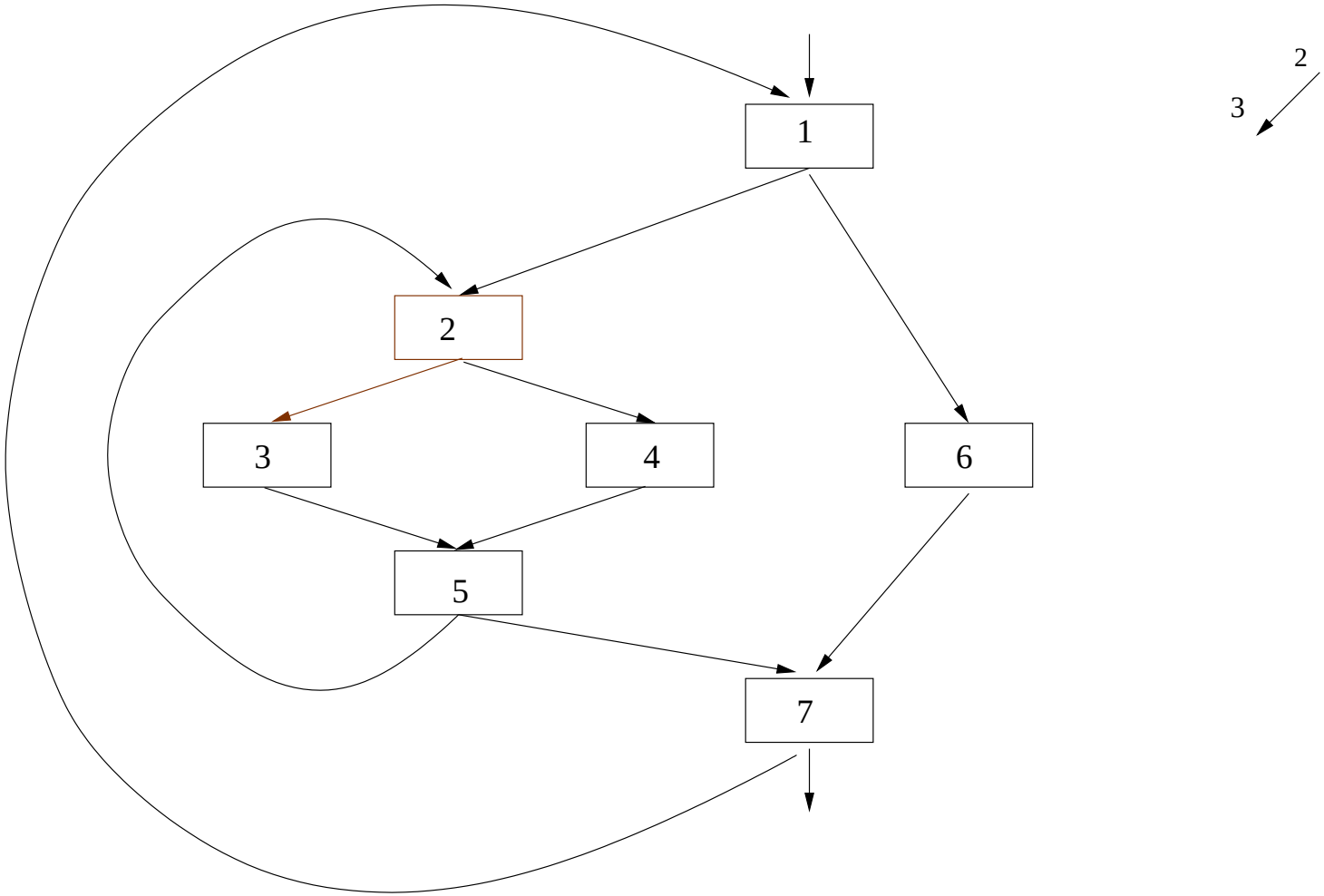
- What if a hot loop contain traces that always fail recording ?
- Solution : Blacklist with backoff on recording

Outline

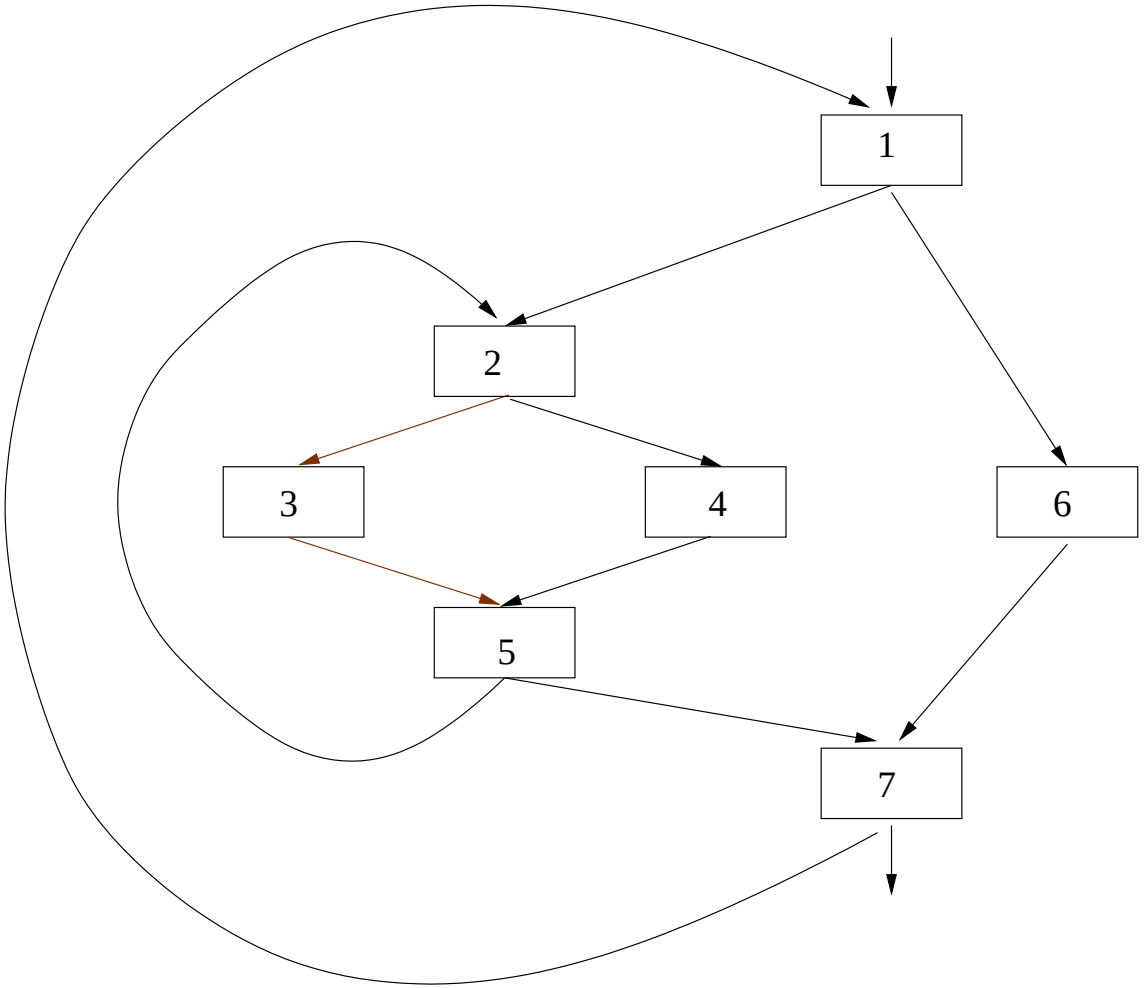
Nested Trace Tree Formation

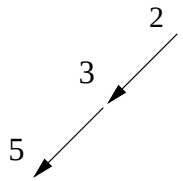


Nested Trace Tree Formation

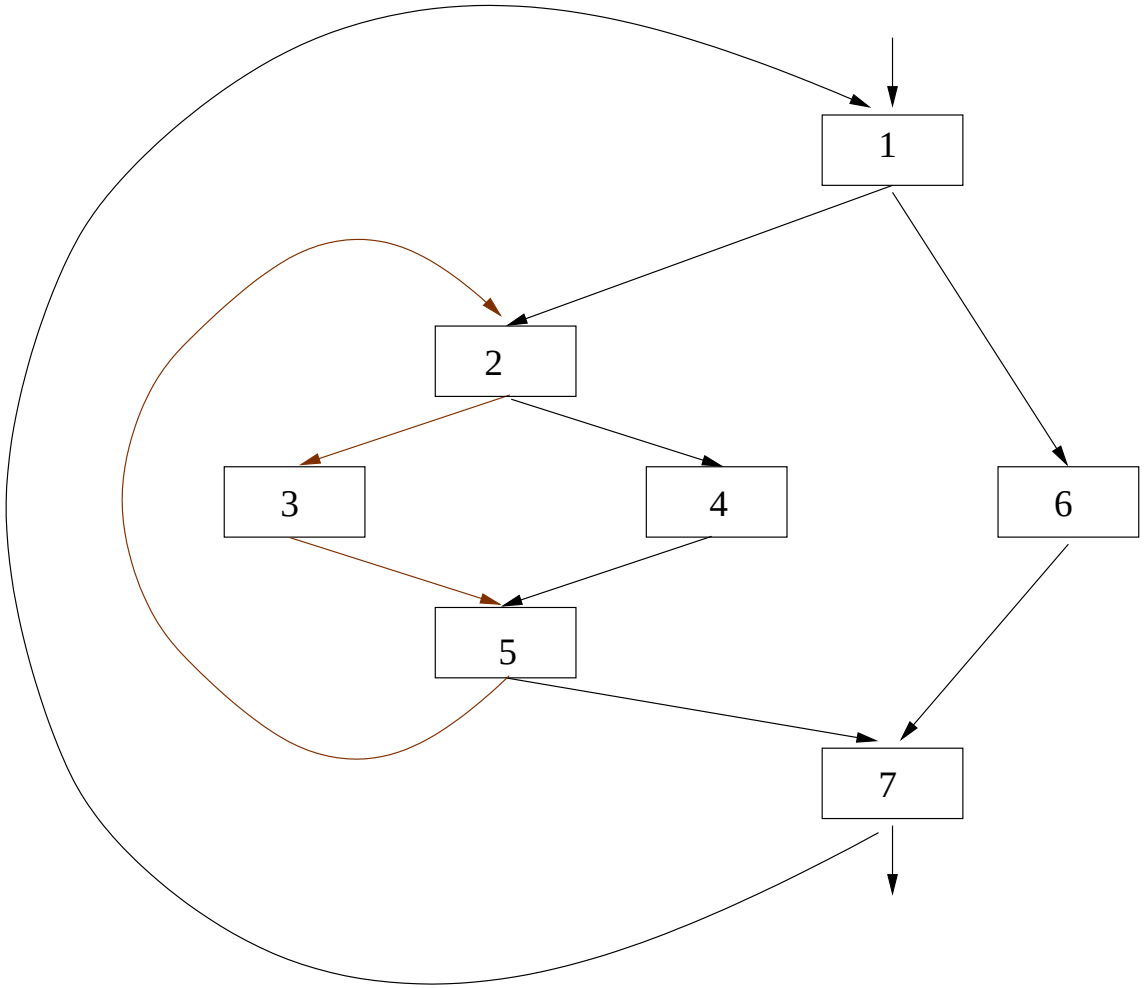


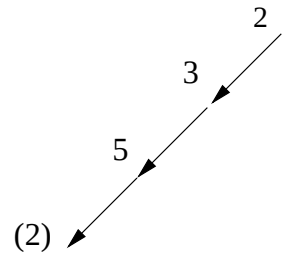
Nested Trace Tree Formation



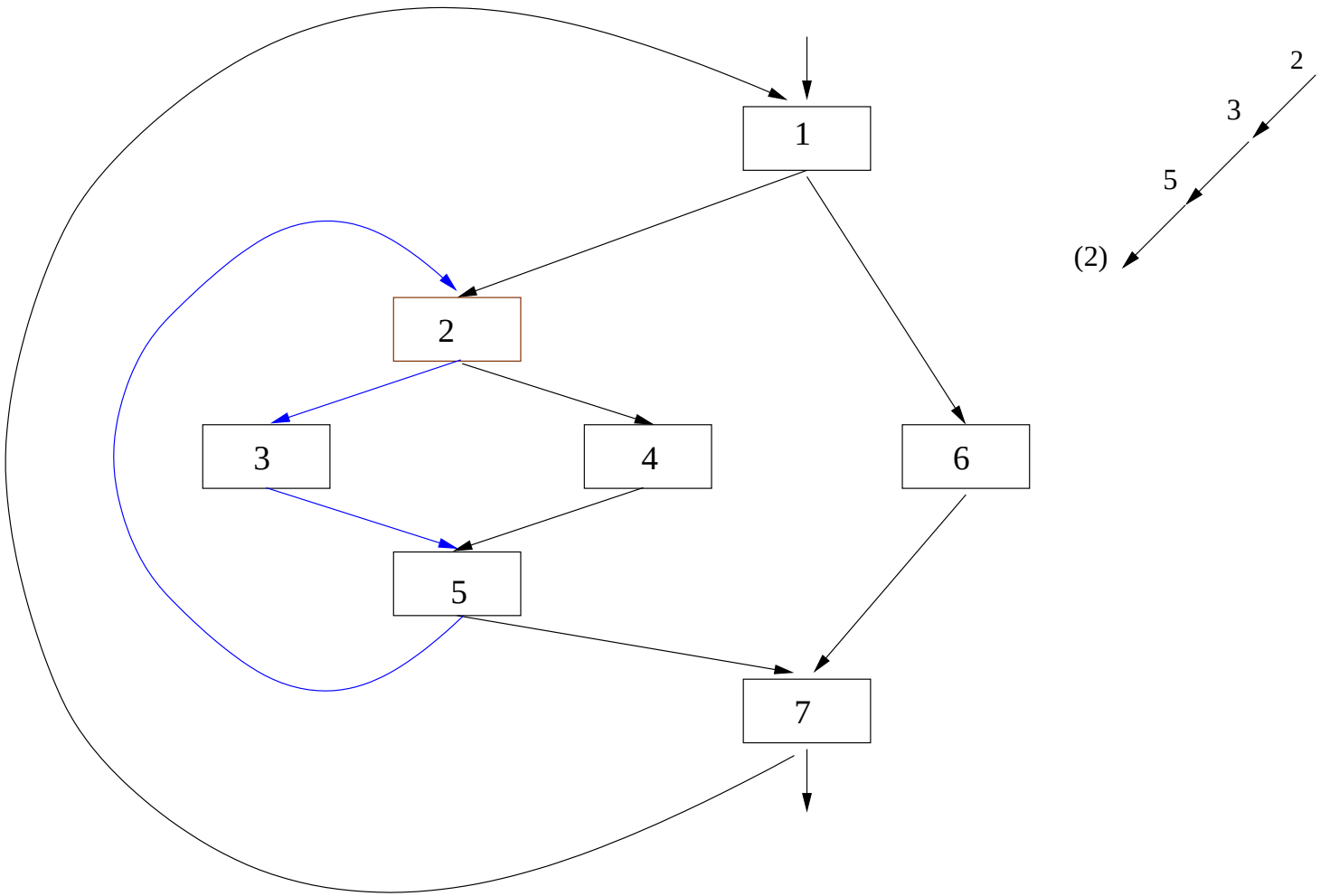


Nested Trace Tree Formation

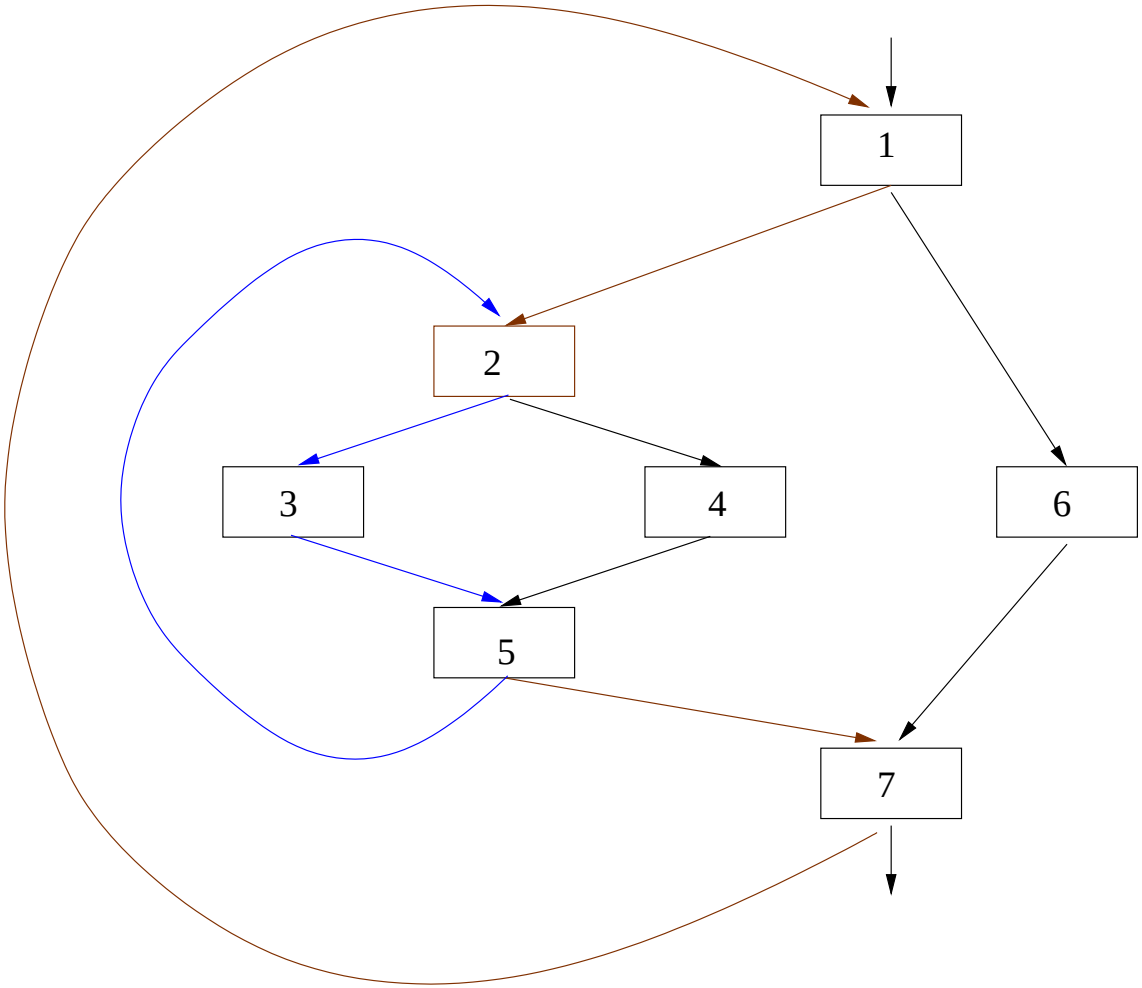


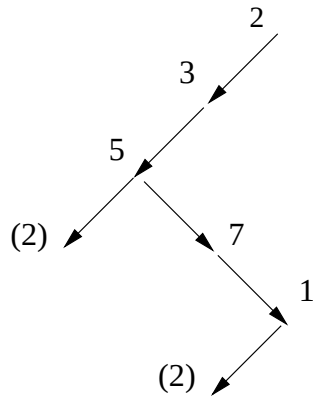


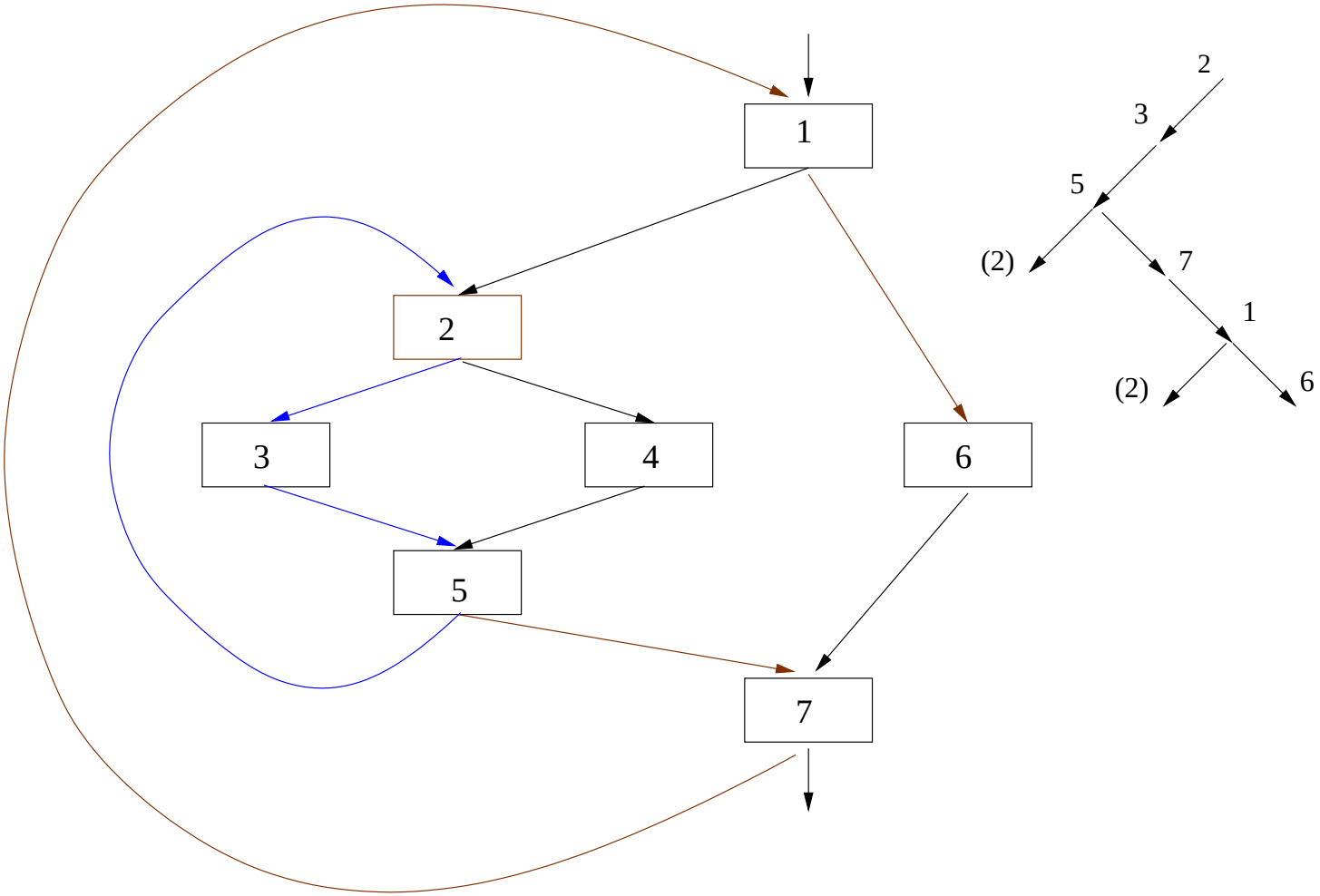
Nested Trace Tree Formation



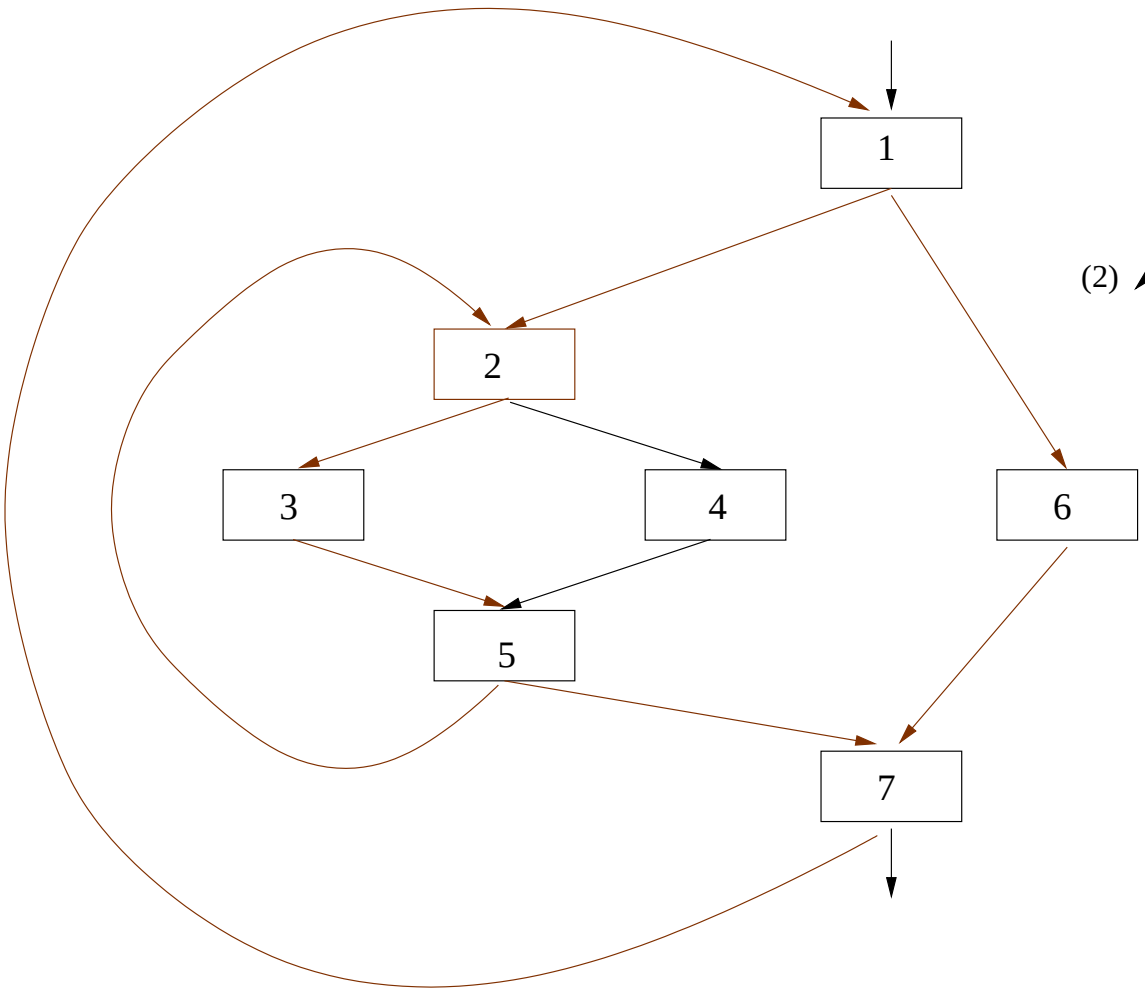
Continue tracing for outer loop

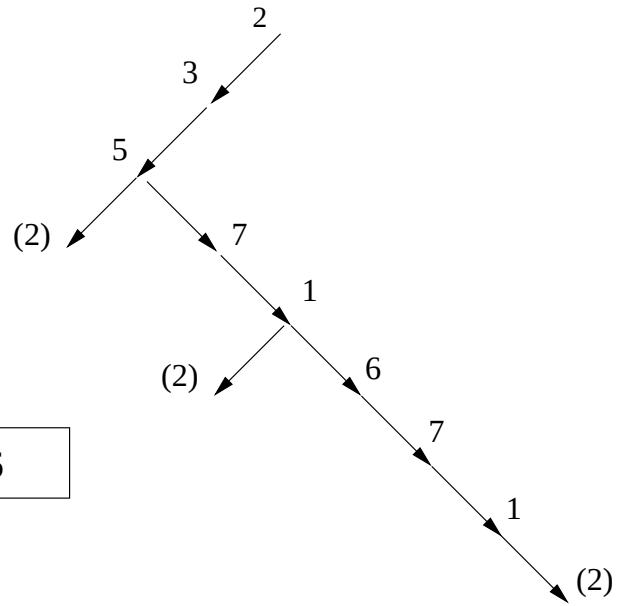




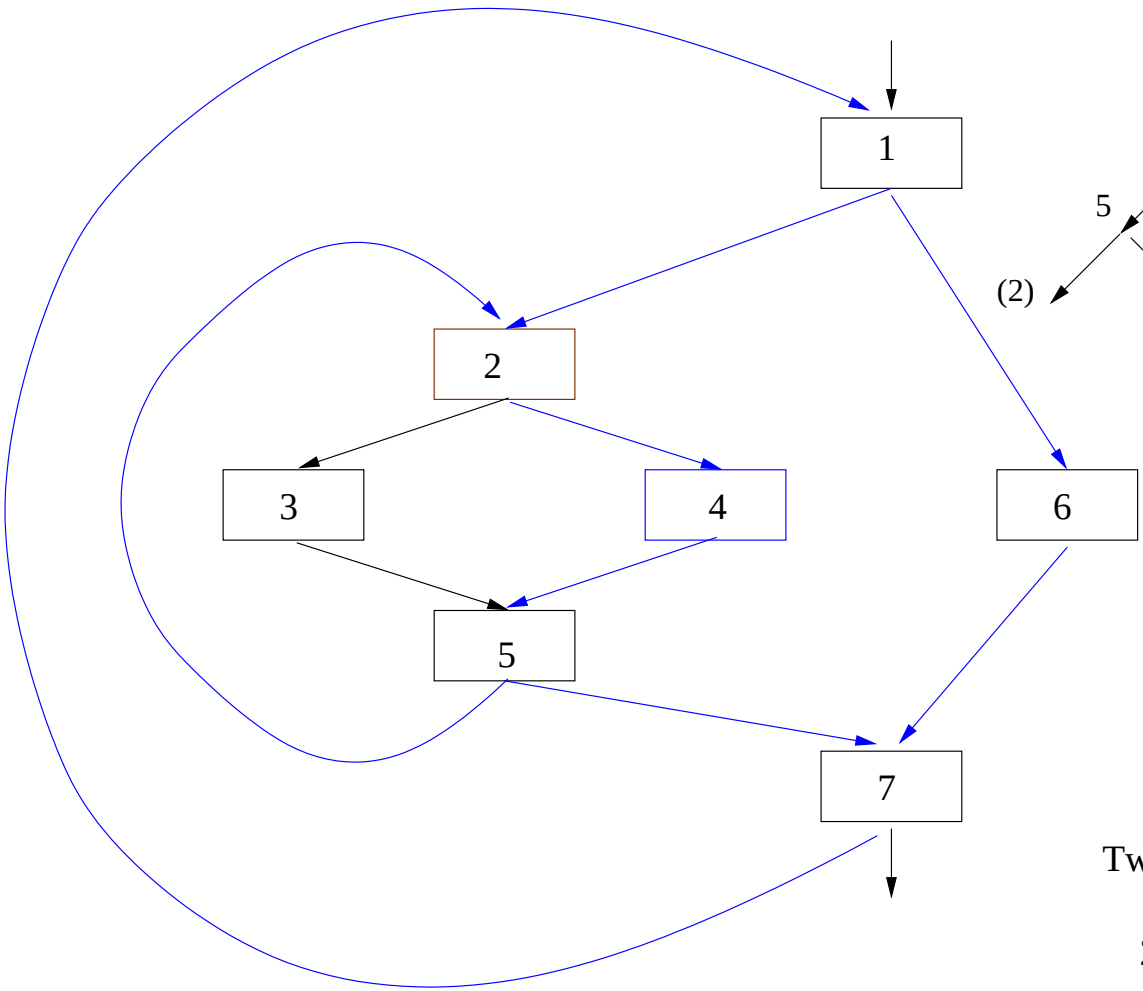

1
2
3
4
5
6
7
2
3
5
(2)
7
1
(2)
6

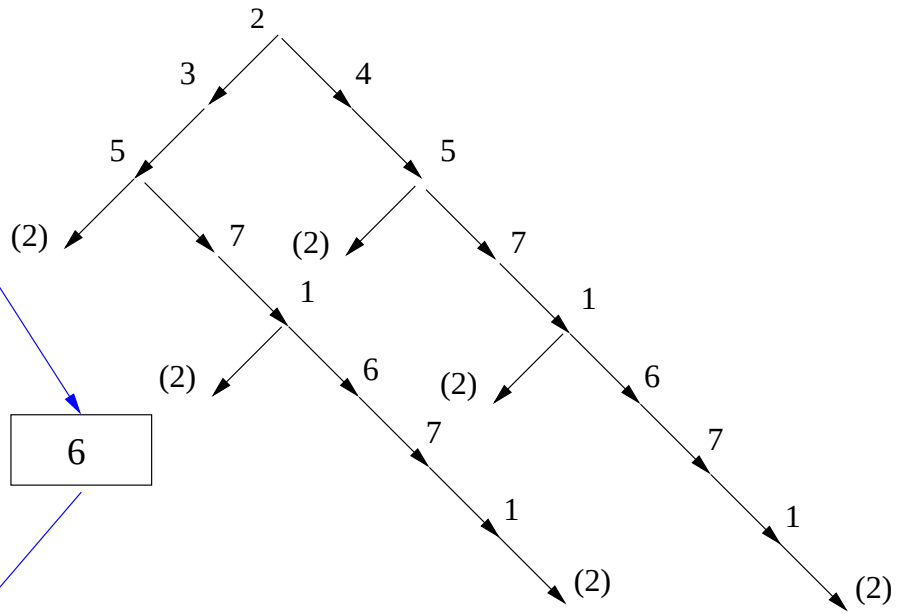
Continue tracing for outer loop





Continue tracing for outer loop

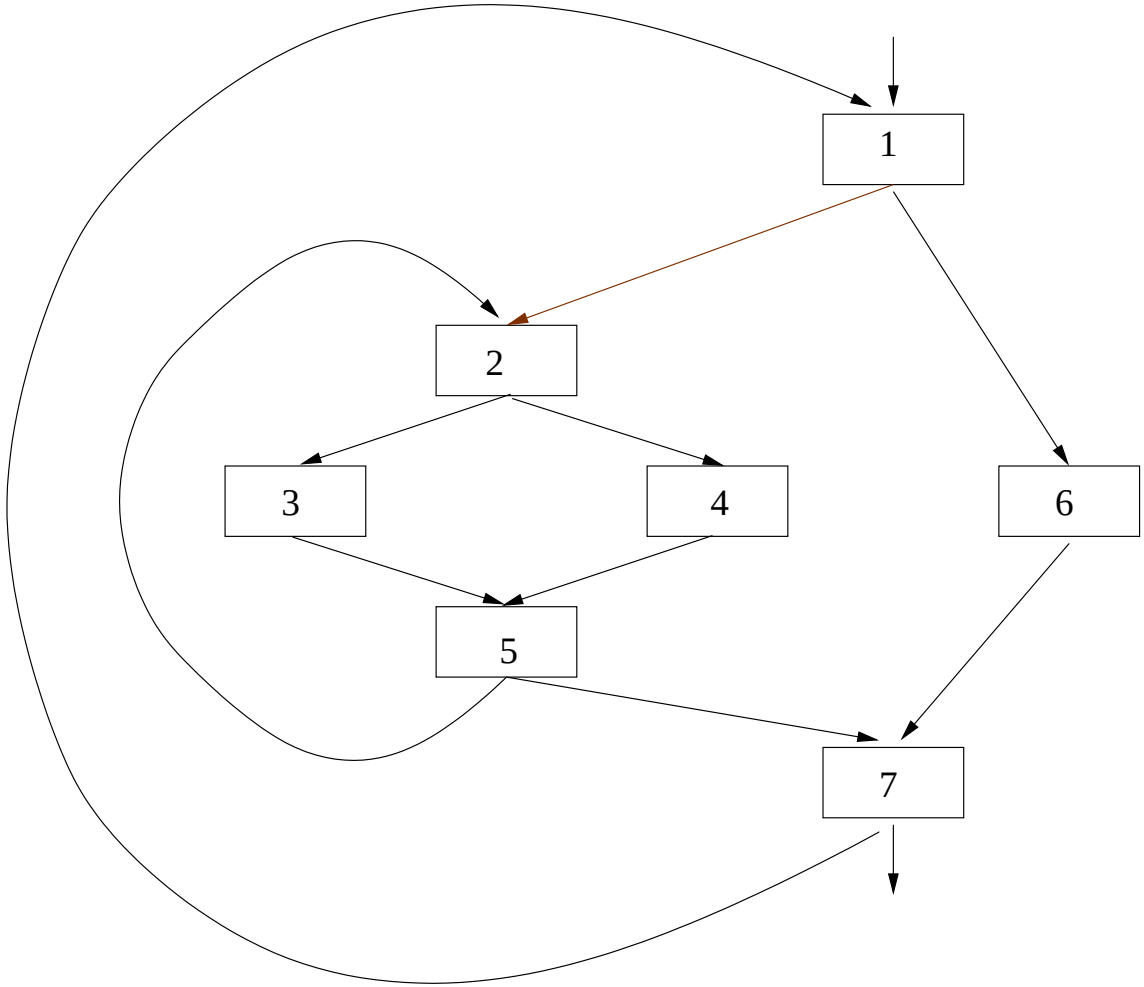




Two Problems:

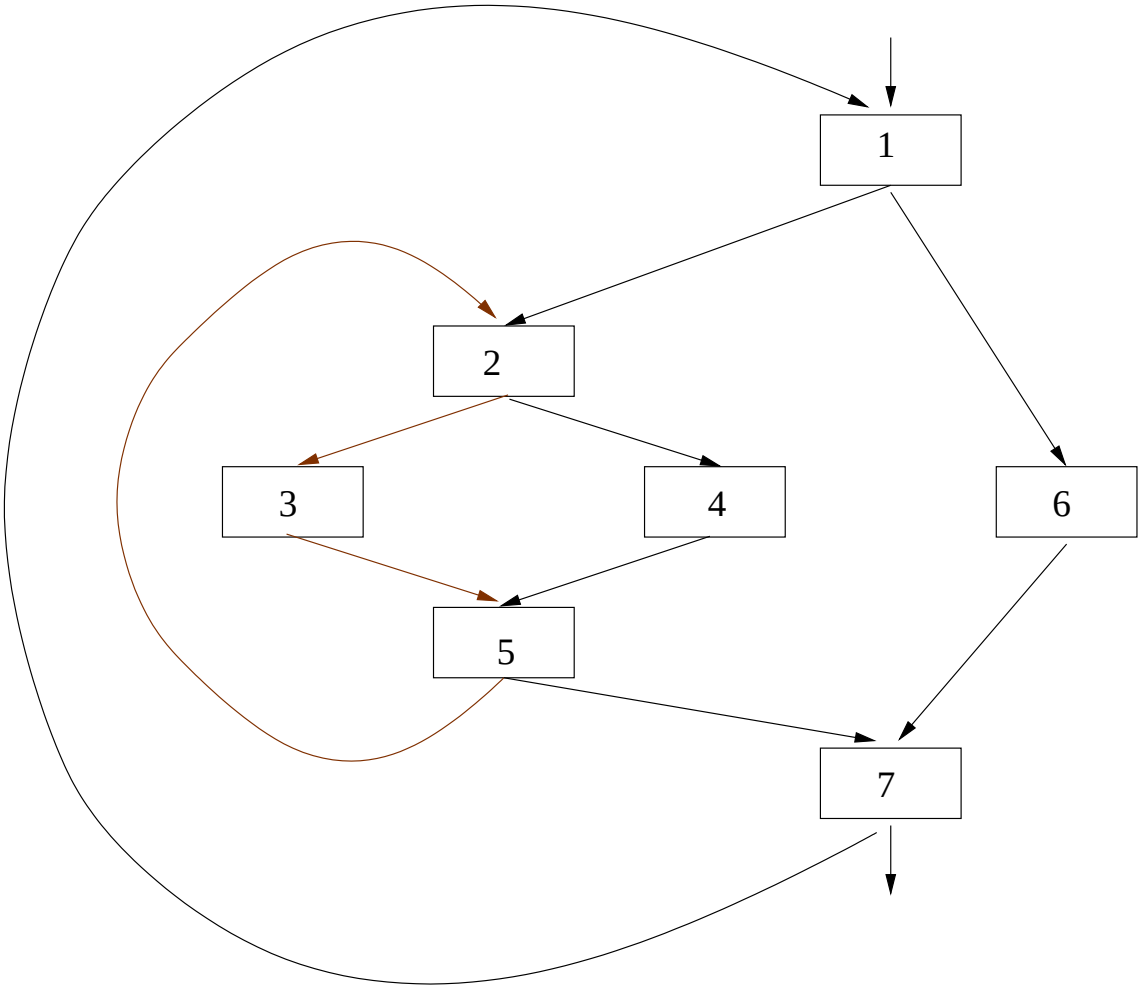
1. Trace Duplication
2. Traces may grow indefinitely

Separate traces



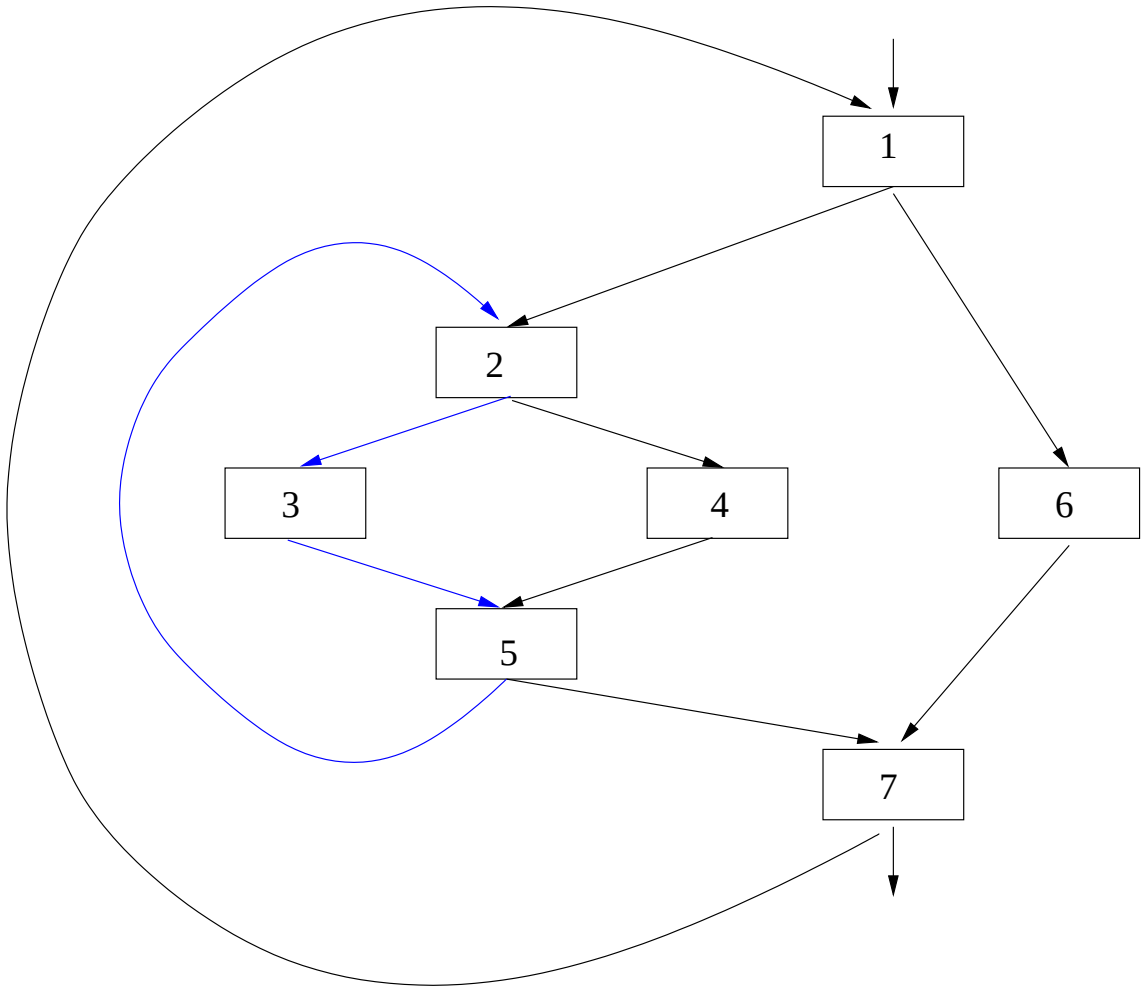
Outer loop starts recording

Separate traces



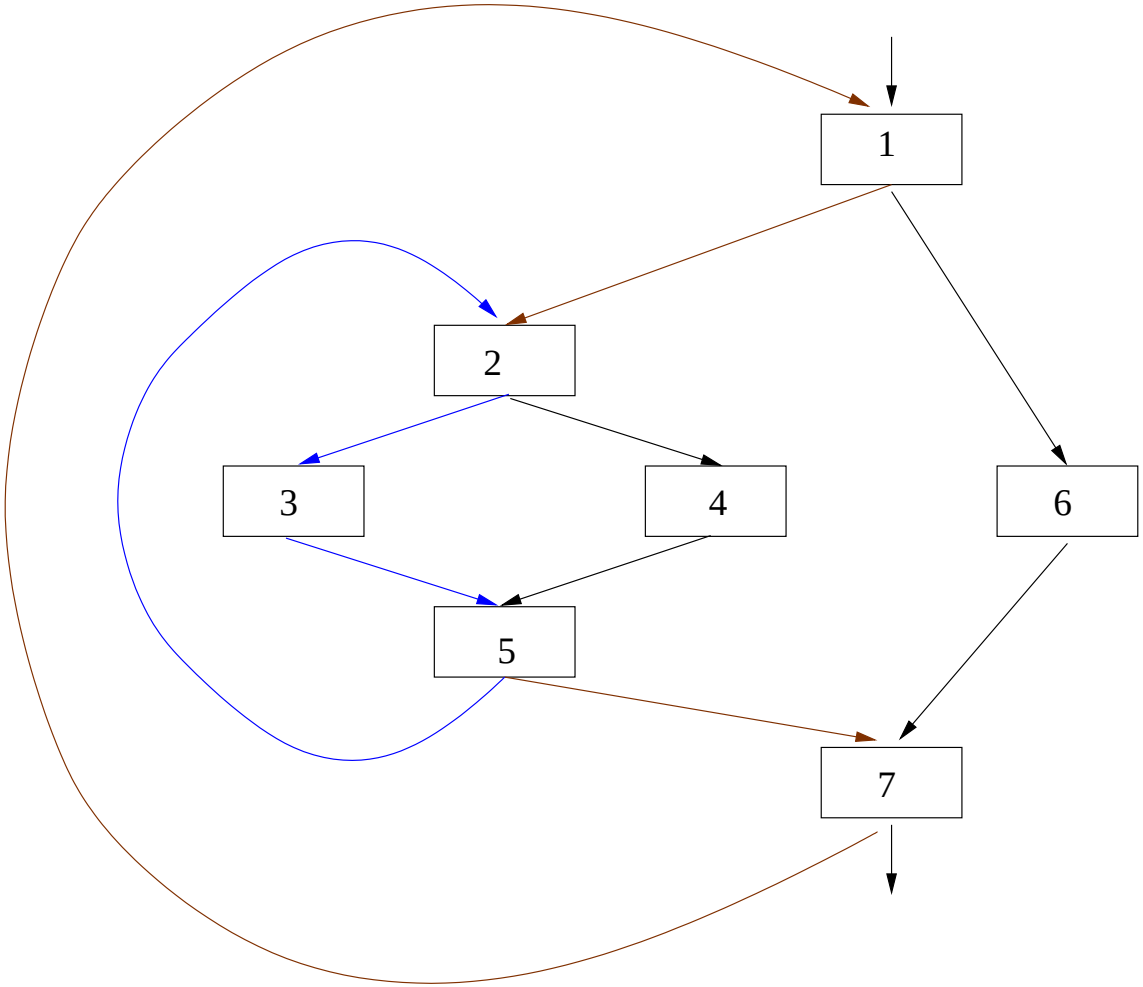
Outer Loop recording stops as "type matched" inner compiled trace not available.

Separate traces



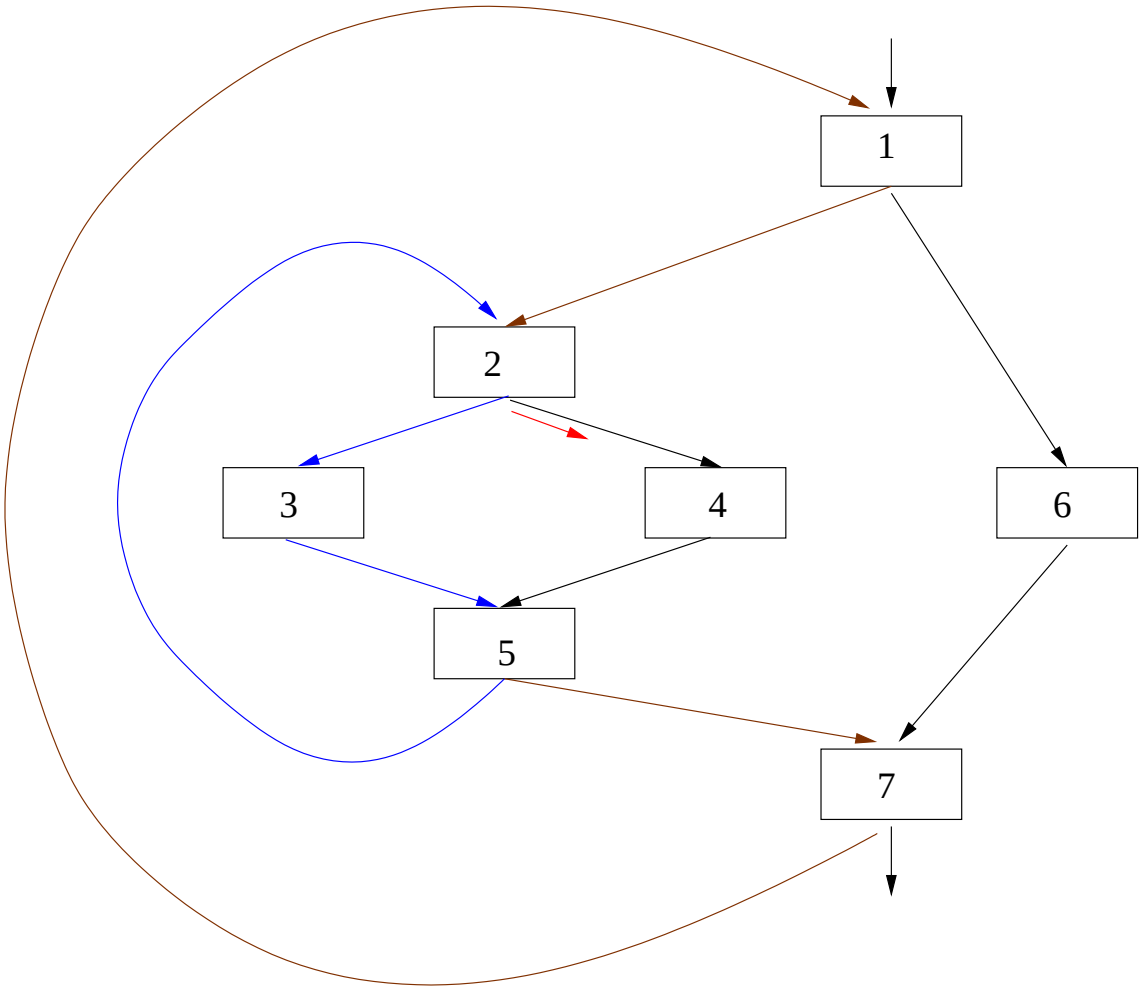
Inner compiled trace ready

Separate traces



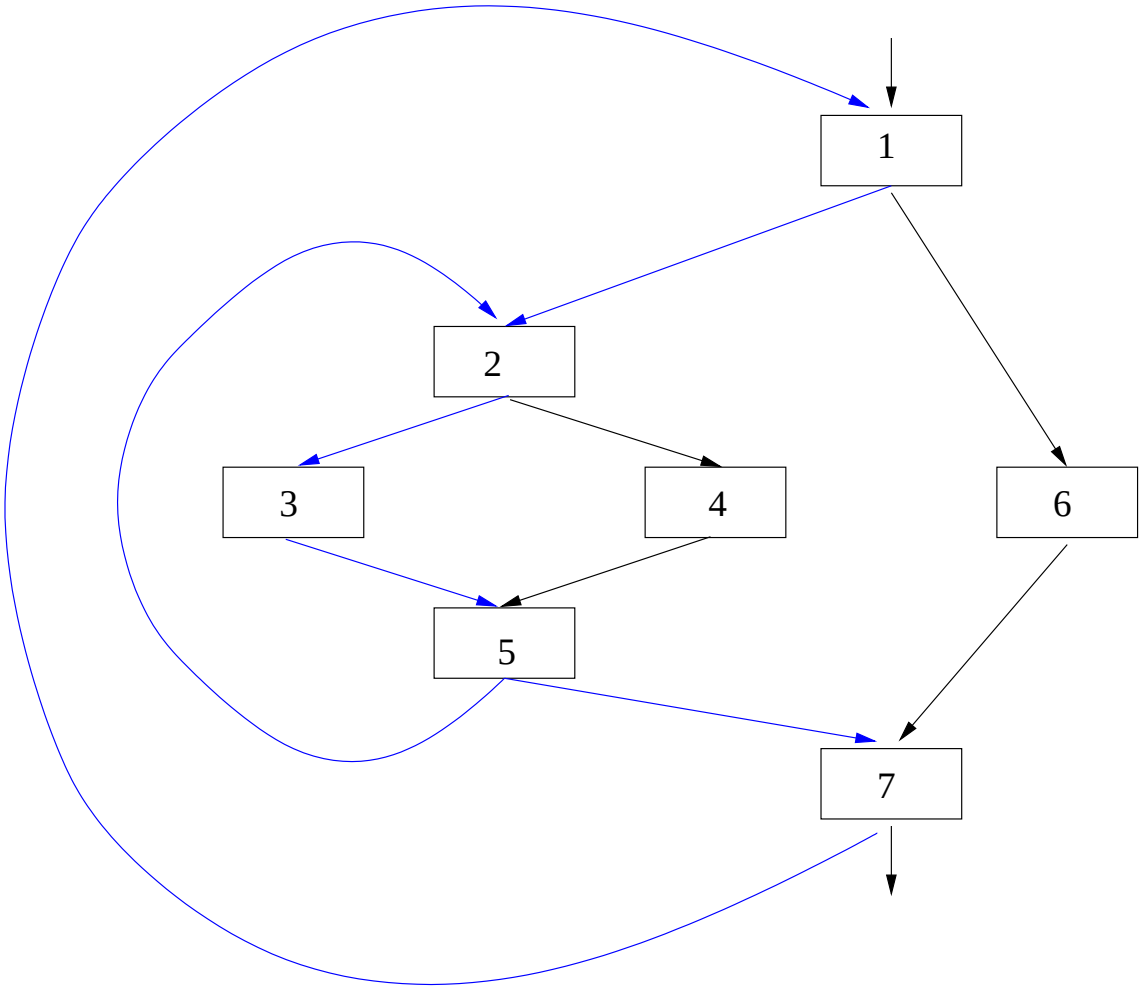
Outer loop records the inner trace call in its recording

Separate traces



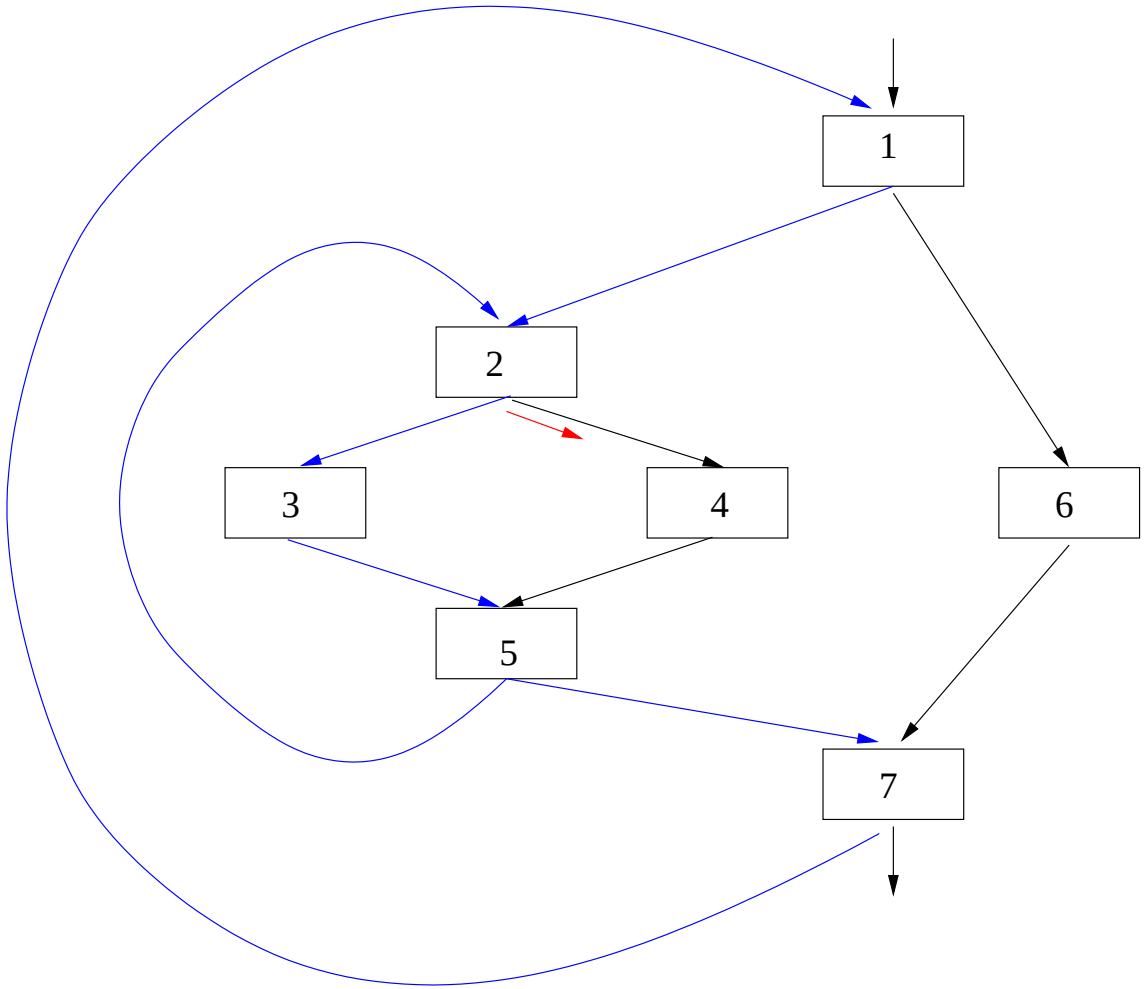
What if during recording of outer loop, the inner trace got a side exit !!

Separate traces



A compiled trace of outer loop

Separate traces



What if during execution of outer loop, the inner trace got a side exit !!

Modified Blacklisting with Nesting

- The outer loop blacklisted very quickly.
- Solution : Blacklist with an option of “forgiving”

Modified Blacklisting with Nesting

- The outer loop blacklisted very quickly.
- Solution : Blacklist with an option of “forgiving”

Outline

Recording

- While interpreting traces are recorded and transformed into LIR. WHY ??
  - LIR traces are type specialized
  - LIR traces are representation specialized w.r.t objects
  - LIR traces are representation specialized w.r.t numbers
  - LIR traces inline functions
- All the above LIR features need guards

Recording

- While interpreting traces are recorded and transformed into LIR. WHY ??
  - LIR traces are type specialized
  - LIR traces are representation specialized w.r.t objects
  - LIR traces are representation specialized w.r.t numbers
  - LIR traces inline functions
- All the above LIR features need guards

Recording

- While interpreting traces are recorded and transformed into LIR. WHY ??
  - LIR traces are type specialized
  - LIR traces are representation specialized w.r.t objects
  - LIR traces are representation specialized w.r.t numbers
  - LIR traces inline functions
- All the above LIR features need guards

Recording

- While interpreting traces are recorded and transformed into LIR. WHY ??
  - LIR traces are type specialized
  - LIR traces are representation specialized w.r.t objects
  - LIR traces are representation specialized w.r.t numbers
  - LIR traces inline functions
- All the above LIR features need guards

Recording

- While interpreting traces are recorded and transformed into LIR. WHY ??
  - LIR traces are type specialized
  - LIR traces are representation specialized w.r.t objects
  - LIR traces are representation specialized w.r.t numbers
  - LIR traces inline functions
- All the above LIR features need guards

Recording

- While interpreting traces are recorded and transformed into LIR. WHY ??
  - LIR traces are type specialized
  - LIR traces are representation specialized w.r.t objects
  - LIR traces are representation specialized w.r.t numbers
  - LIR traces inline functions
- All the above LIR features need guards

Trace Tree Optimization

- Optimization Goal: Make compilation fast. (Done by NanoJIT)
  - Retrict to small set of optimization
  - Avoid any reordering optimizations
  - leverage the fact that the LIR is in SSA form
- Optimizations are performed in 2 phases: forward and backward

Trace Tree Optimization

- Optimization Goal: Make compilation fast. (Done by NanoJIT)
  - Retrict to small set of optimization
  - Avoid any reordering optimizations
  - leverage the fact that the LIR is in SSA form
- Optimizations are performed in 2 phases: forward and backward

Trace Tree Optimization

- Optimization Goal: Make compilation fast. (Done by NanoJIT)
  - Retrict to small set of optimization
  - Avoid any reordering optimizations
  - leverage the fact that the LIR is in SSA form
- Optimizations are performed in 2 phases: forward and backward

Trace Tree Optimization

- Optimization Goal: Make compilation fast. (Done by NanoJIT)
  - Retrict to small set of optimization
  - Avoid any reordering optimizations
  - leverage the fact that the LIR is in SSA form
- Optimizations are performed in 2 phases: forward and backward

Trace Tree Optimization

- Optimization Goal: Make compilation fast. (Done by NanoJIT)
  - Retrict to small set of optimization
  - Avoid any reordering optimizations
  - leverage the fact that the LIR is in SSA form
- Optimizations are performed in 2 phases: forward and backward

Forward Optimizations

- While recording on each instruction basis.
  - Constant subexpression elimination
  - Expression simplification and algebraic identities
  - ISA specific simplification

Forward Optimizations

- While recording on each instruction basis.
  - Constant subexpression elimination
  - Expression simplification and algebraic identities
  - ISA specific simplification

Forward Optimizations

- While recording on each instruction basis.
  - Constant subexpression elimination
  - Expression simplification and algebraic identities
  - ISA specific simplification

Forward Optimizations

- While recording on each instruction basis.
  - Constant subexpression elimination
  - Expression simplification and algebraic identities
  - ISA specific simplification

Backward optimizations

- When recording is complete.
  - dead data-stack store elimination
  - dead call-stack store elimination
  - dead code elimination
  - Register Allocation

Backward optimizations

- When recording is complete.
  - dead data-stack store elimination
  - dead call-stack store elimination
  - dead code elimination
  - Register Allocation

Backward optimizations

- When recording is complete.
  - dead data-stack store elimination
  - dead call-stack store elimination
  - dead code elimination
  - Register Allocation

Backward optimizations

- When recording is complete.
  - dead data-stack store elimination
  - dead call-stack store elimination
  - dead code elimination
  - Register Allocation

Backward optimizations

- When recording is complete.
  - dead data-stack store elimination
  - dead call-stack store elimination
  - dead code elimination
  - Register Allocation

Outline

Trace stitching

- when a trace calls a branch trace, trace stitching can be applied
- identical type maps yield identical activation records

Outline

Setup

- SunSpider Benchmark suite is used
- SunSpider test driver starts a JS interpreter, load and run each program
  - one time to warm up
  - 10 times, which are measured the average time is taken
  - runs on 4 interpreter
    - SpiderMonkey: JS interpreter. **Baseline for comparision**
    - TraceMonkey: The proposed compilation strategy
    - SquirrelFish Extreme (SFX): Call threaded JS interpreter
    - V8: Method compiling JS VM

Results

- Speed up spreads between 0.9 and 25.
- Performance breakup on Sunspider benchmark
  - 9 / 26 : TraceMonkey is the fastest
  - 5 / 26 : The current implementation does not trace recursion.
  - 5 / 26 : nested loop with small bodies. May be more time in calling nested trace.
  - 2 / 26 : Does not trace eval and some other C implememnted functions.
  - 2 / 26 : Trace well, but long compilation.
  - 1 / 26 : Does not trace through regular expresion replace operations.
  - 1 / 26 : run time dominated by string processing buitlins.
  - 1 / 26 : dominated by regular expression matching (implemented in all 3 VM as special regular expression compiler)

Results Continued

- Overall performance speedup of native trace execution = $\frac{\text{Time per bytecode execution in Interpreter}}{\text{Time per bytecode execution in native code}} = 3.9$
- Other interpreter have an overall performance speedup of 3.0
- TraceMonkey recording & compilation 200 times slower than interpreter speed.
- the performance has catched up after 270 iterations of a trace.

My Critique

- Web as Desktop
- Incremental recompilation.

Outline

Outline

Correctness

- Mozillas JavaScript fuzz tester, JSFUNFUZZ, random test generator.
- Modified JSFUNFUZZ to generate
  - loops
  - type unstable loops
  - heavy branching code.

Implementation

- Implemented for SpiderMonkey JS virtual machine.
- SpiderMonkey is the interpreter for JS
- first compiled into bytecode, then interpreted.
- is garbage collected (non-generational, stop the world, mark and sweep)

Calling compiled trace

- traces are stored in a trace cache indexed by interpreter PC and type map.
- traces are compiled so that they can be called as functions using standard native calling conventions.