Trace Based Just-In-Time Type Specialization for Dynamic Languages

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

20th March 2015

Name: Sandeep Dasgupta
Advisor: Dr. Vikram Adve
Primary: Compilers
Secondary: Parallel Programming, Architecture

Outline

Motivation

- Generate efficient machine code for dynamic type languages like Javascript

Outline

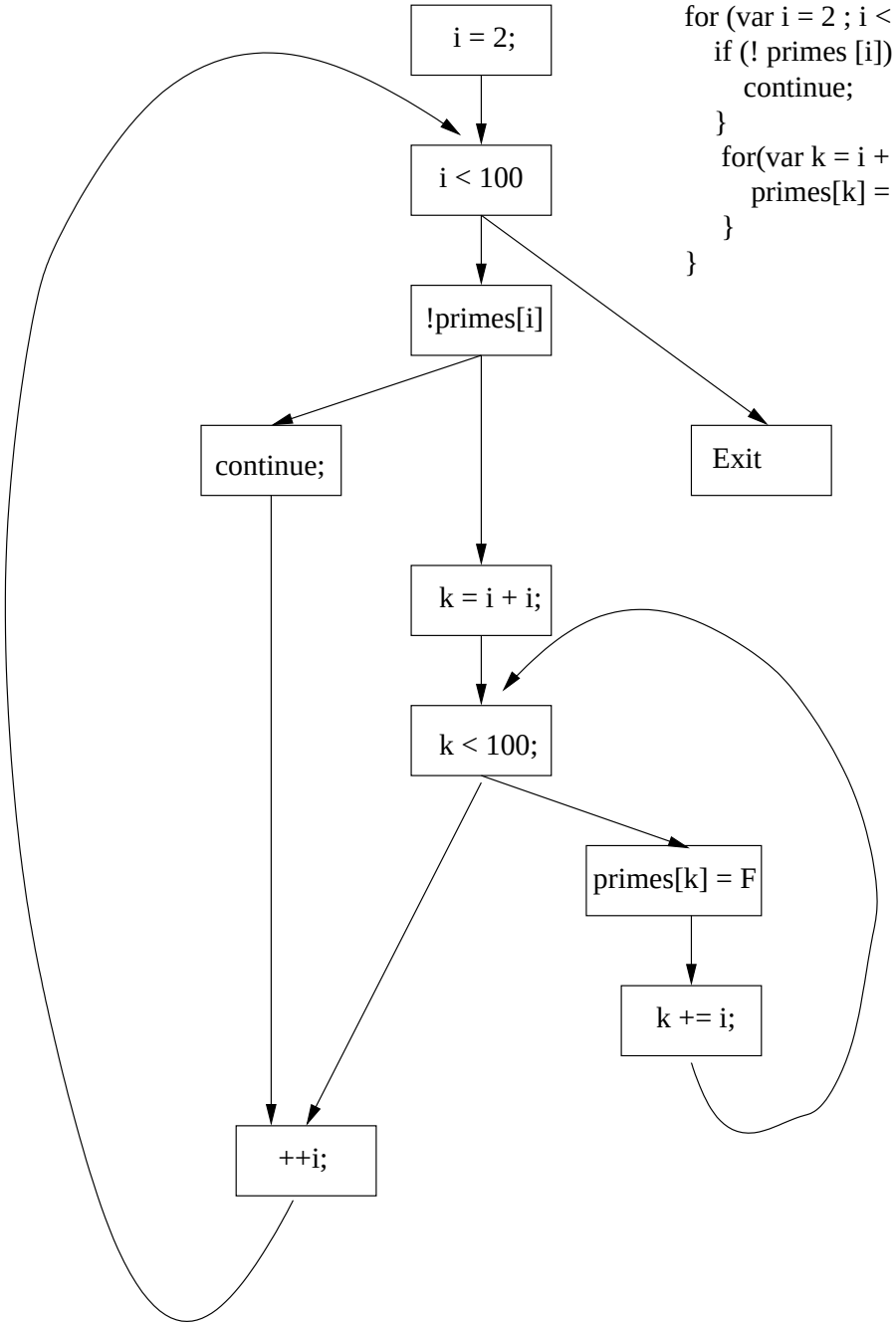
TraceMonkey: Proposed Compilation strategy

- Is implemented for Javascript Interpreter SpiderMonkey
- Design Decisions
  - Dynamic compilation on loop level granularity, not method based.
  - No static type inferencing.

Trace

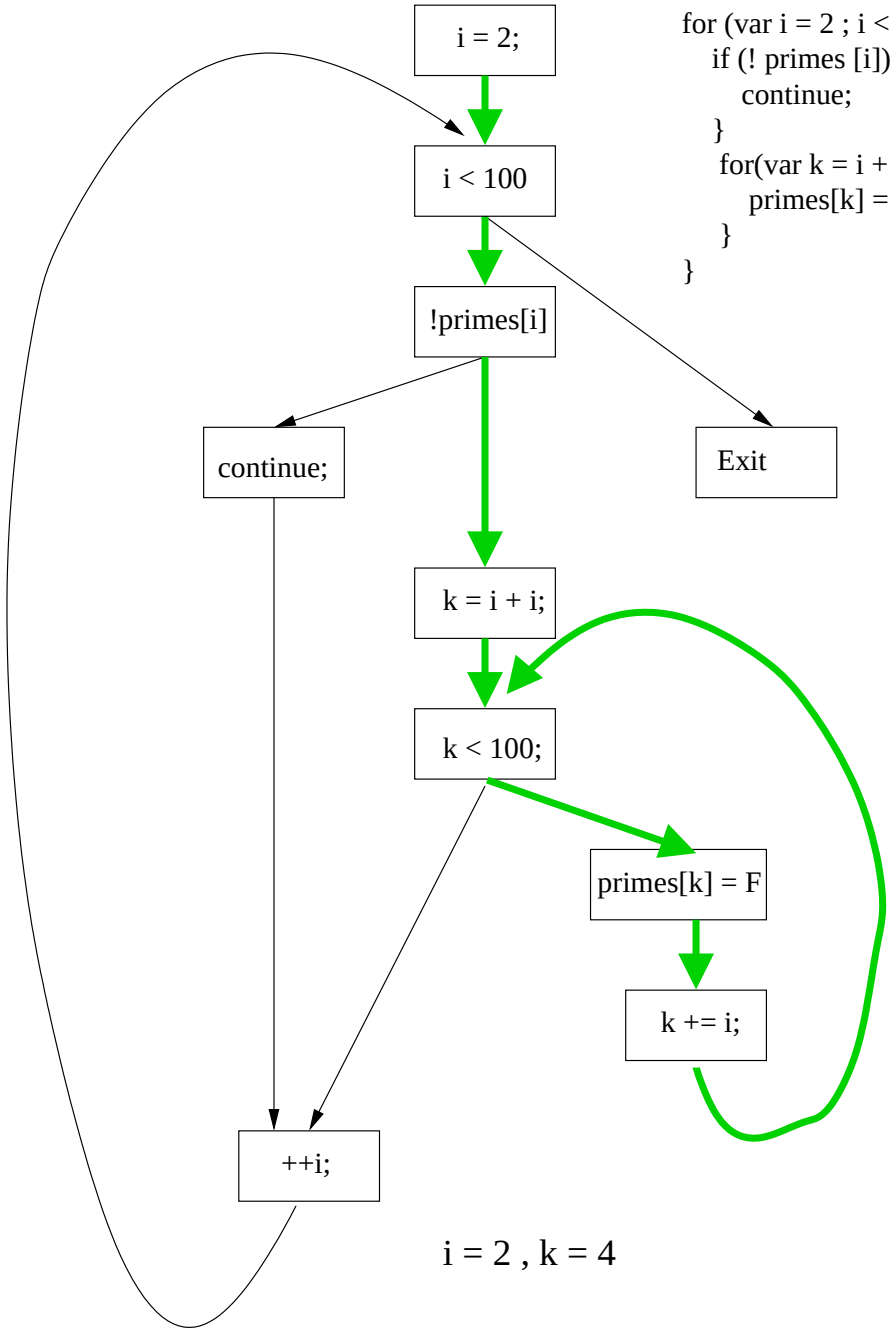
- Typed loop traces
  - Single entry, multiple exits loop paths
  - Type specialized

Outline




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

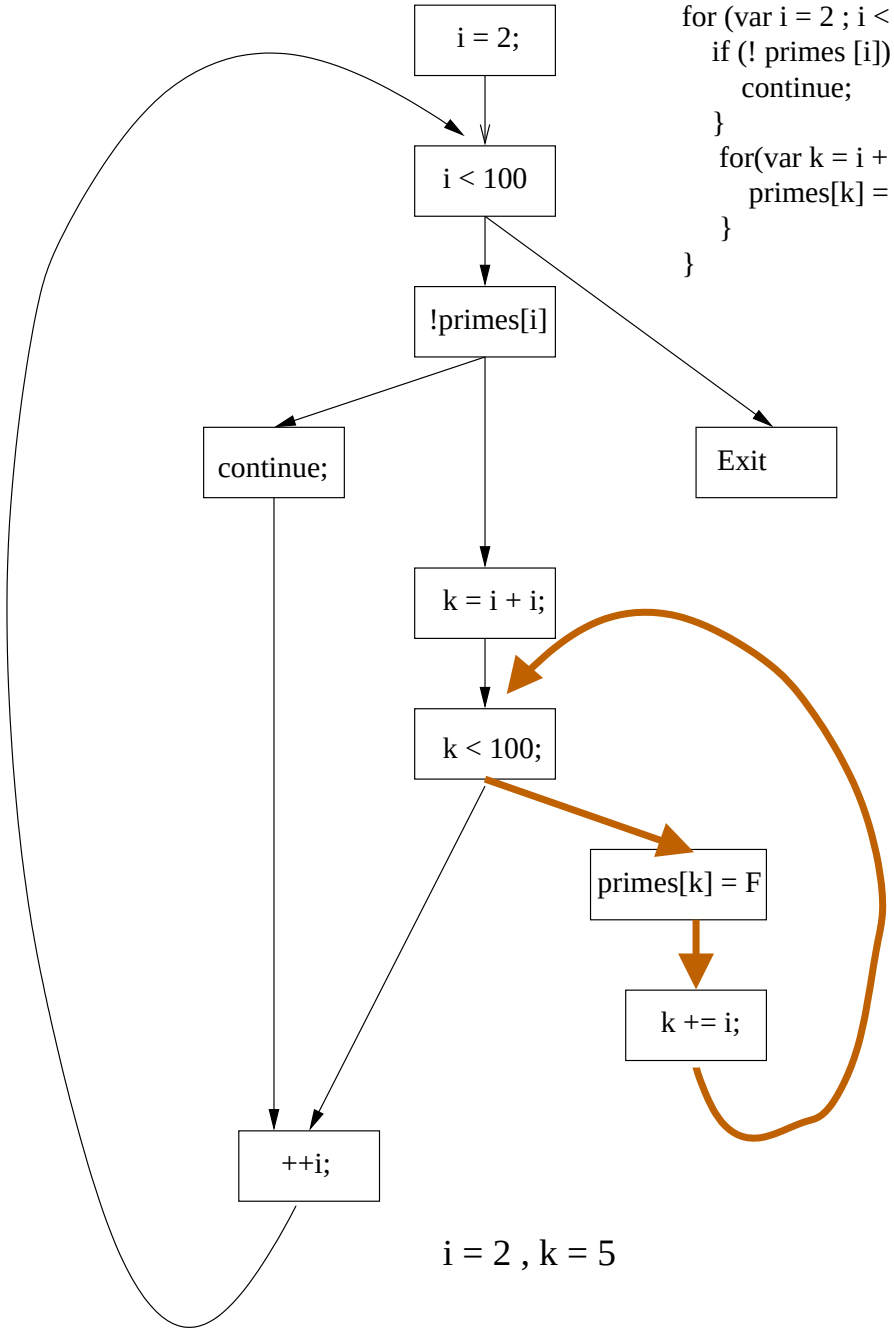
interpreting
recording
compiled code
side exit check (guard)




```
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
side exit check (guard)

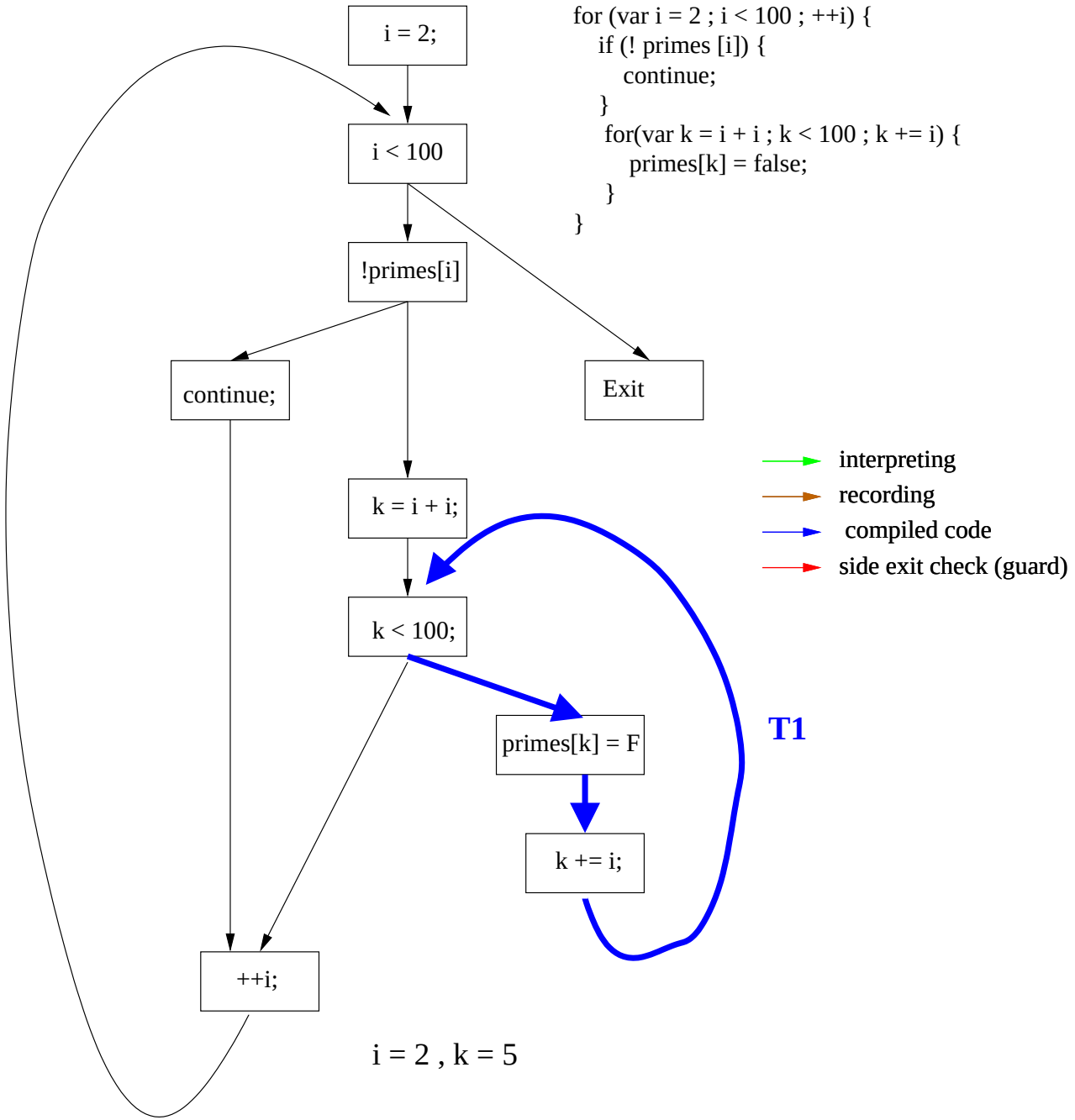
i = 2 , k = 4



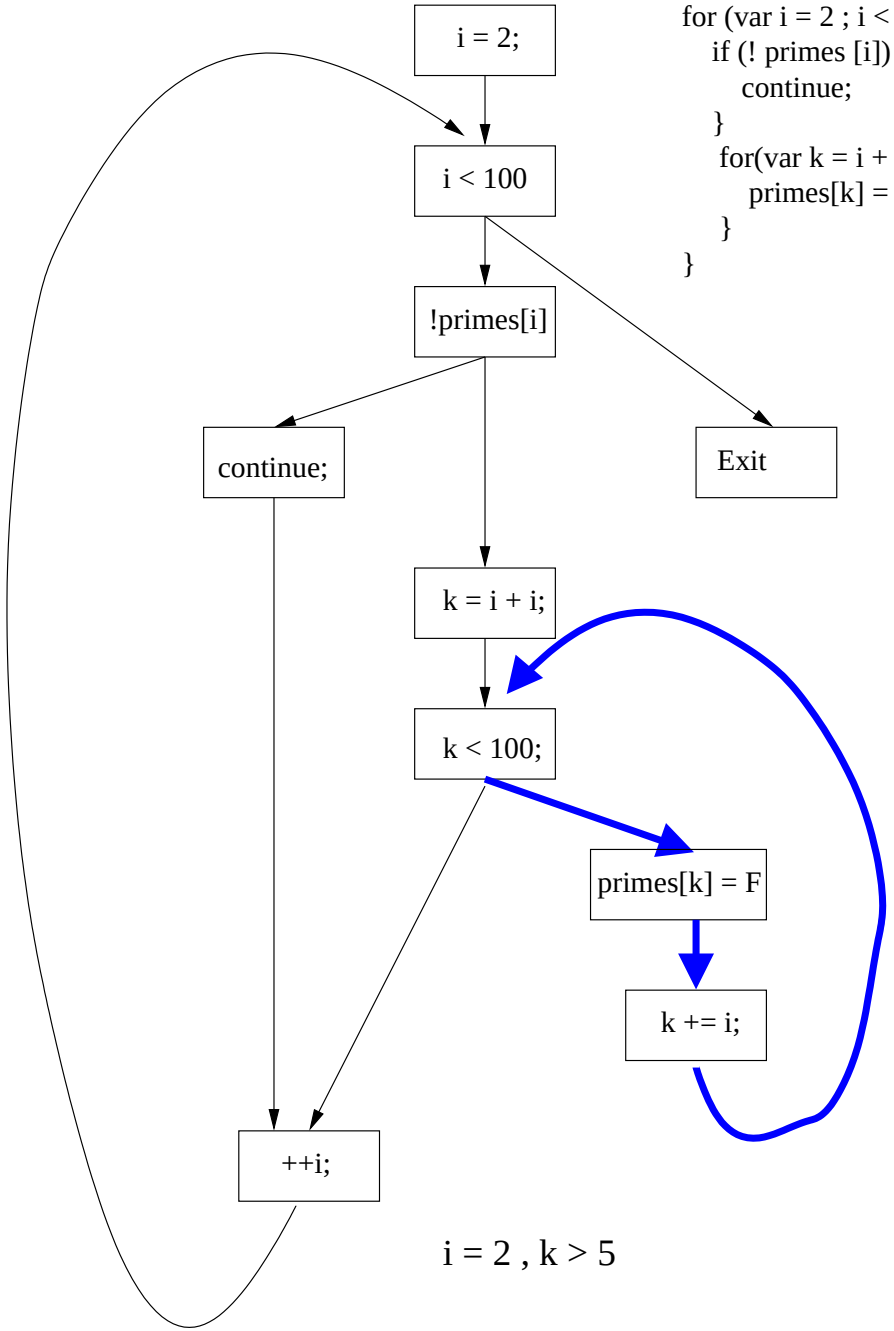

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

interpreting
recording
compiled code
side exit check (guard)

i = 2 , k = 5



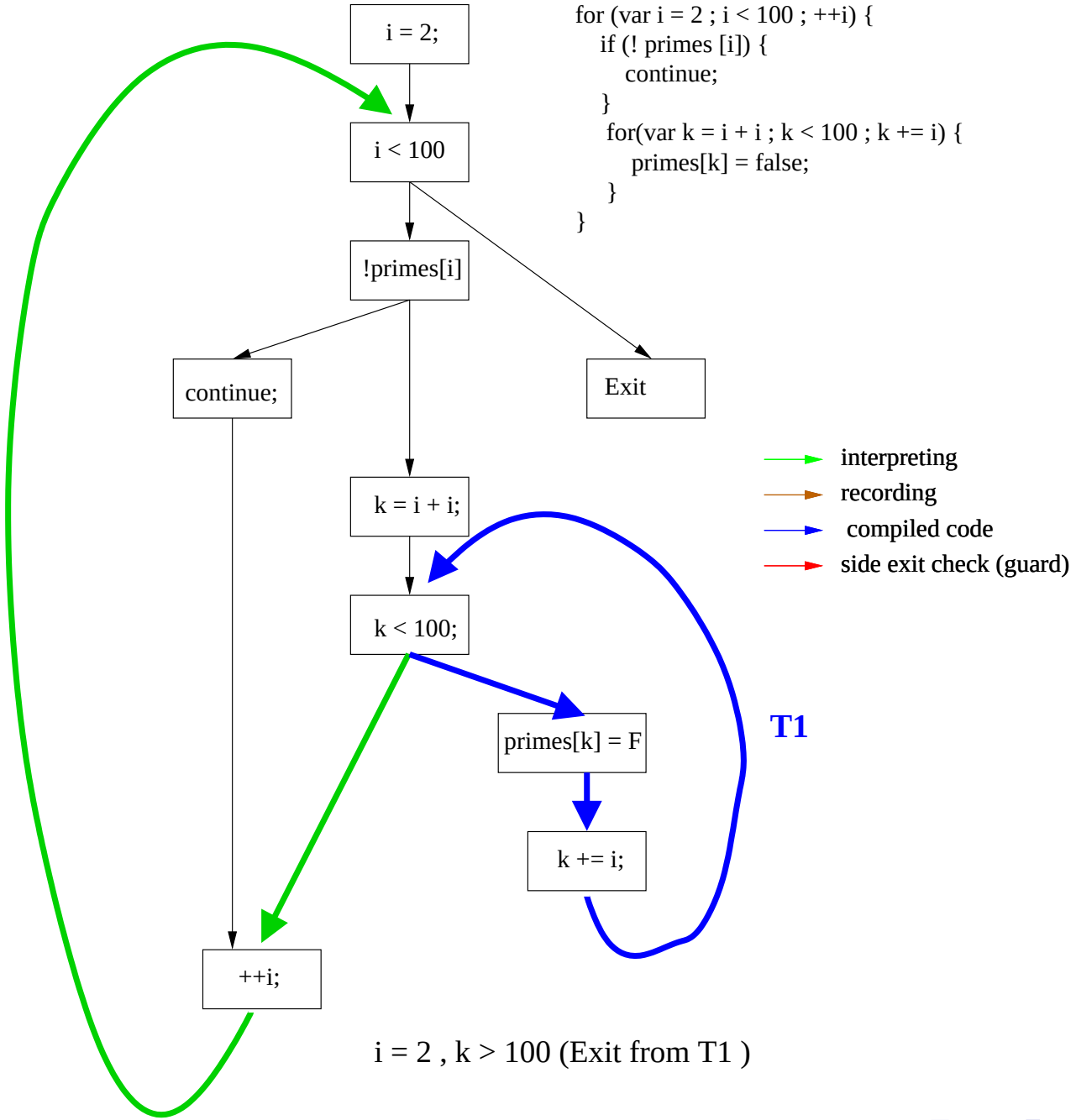

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```




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

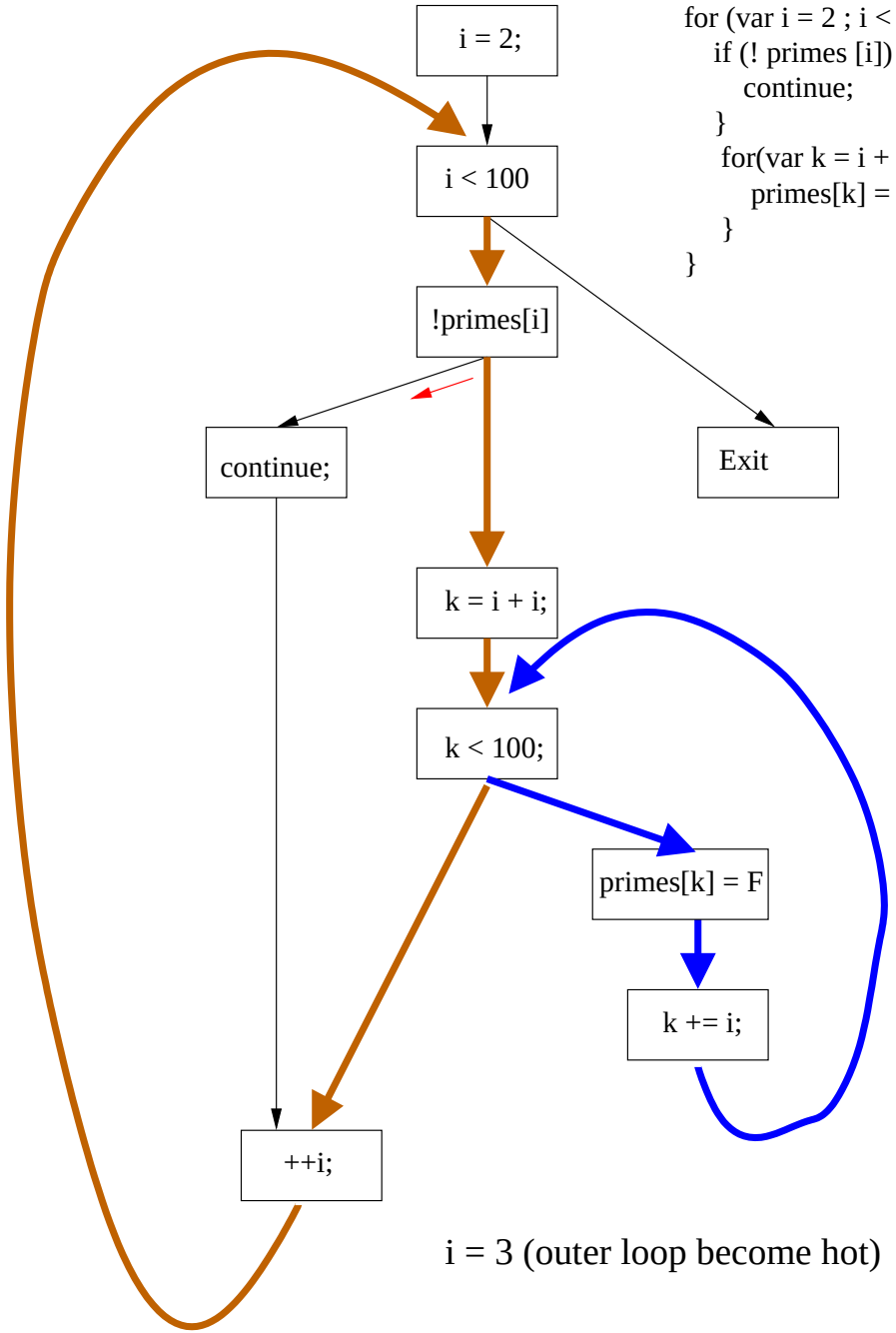
interpreting
recording
compiled code
side exit check (guard)

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;
     }
}
```

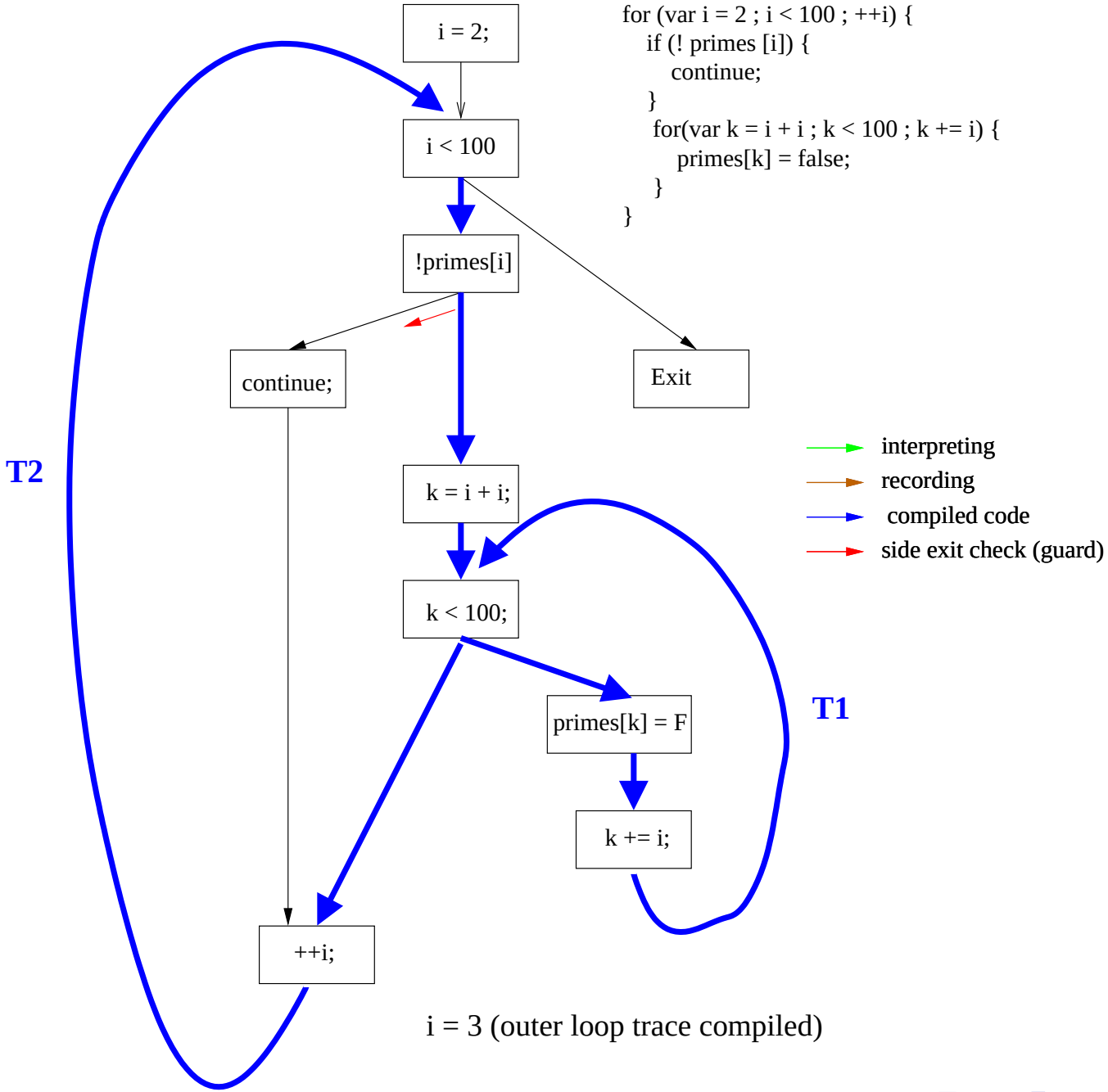
i = 2 , k > 100 (Exit from T1)



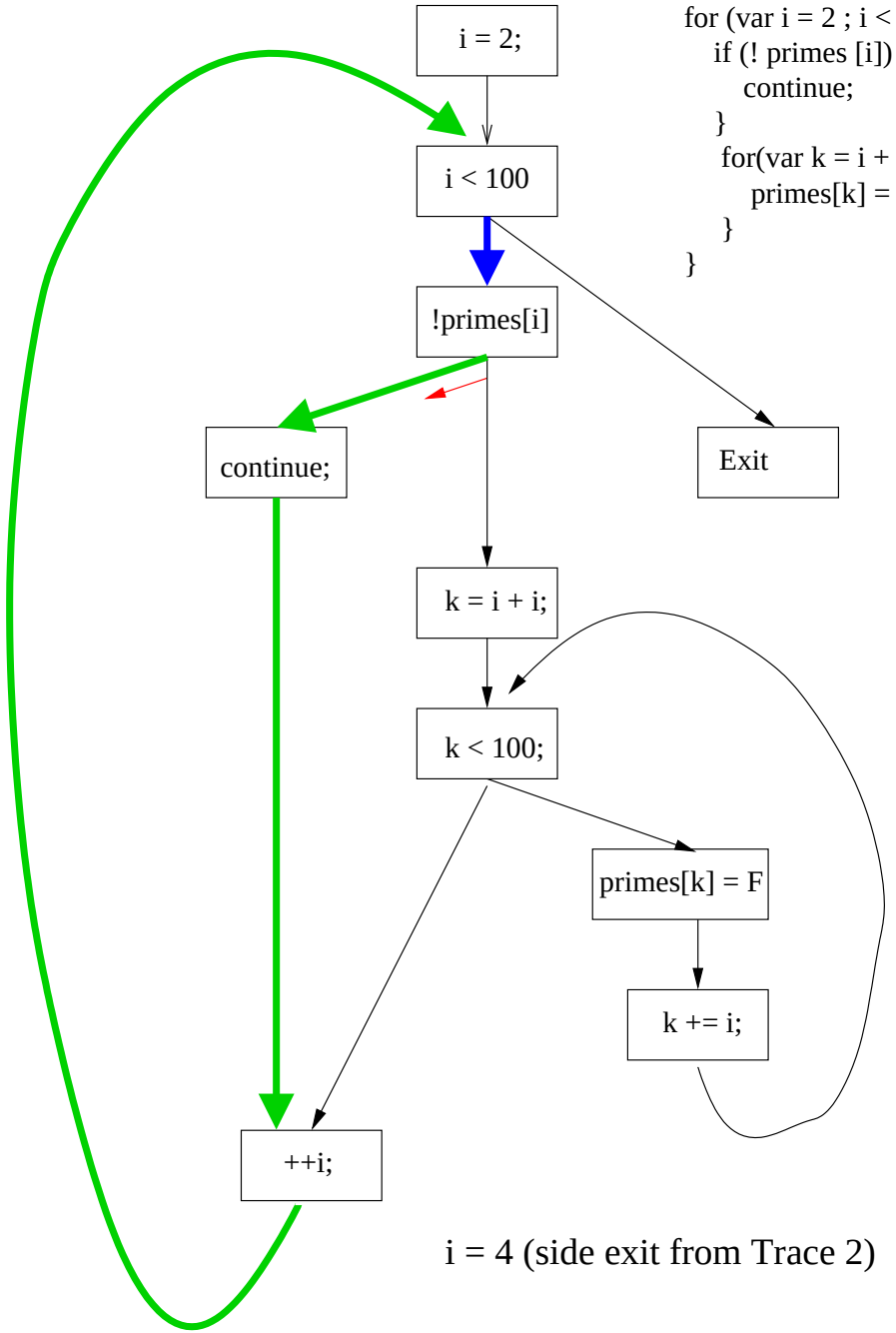

```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

interpreting
recording
compiled code
side exit check (guard)

i = 3 (outer loop become hot)



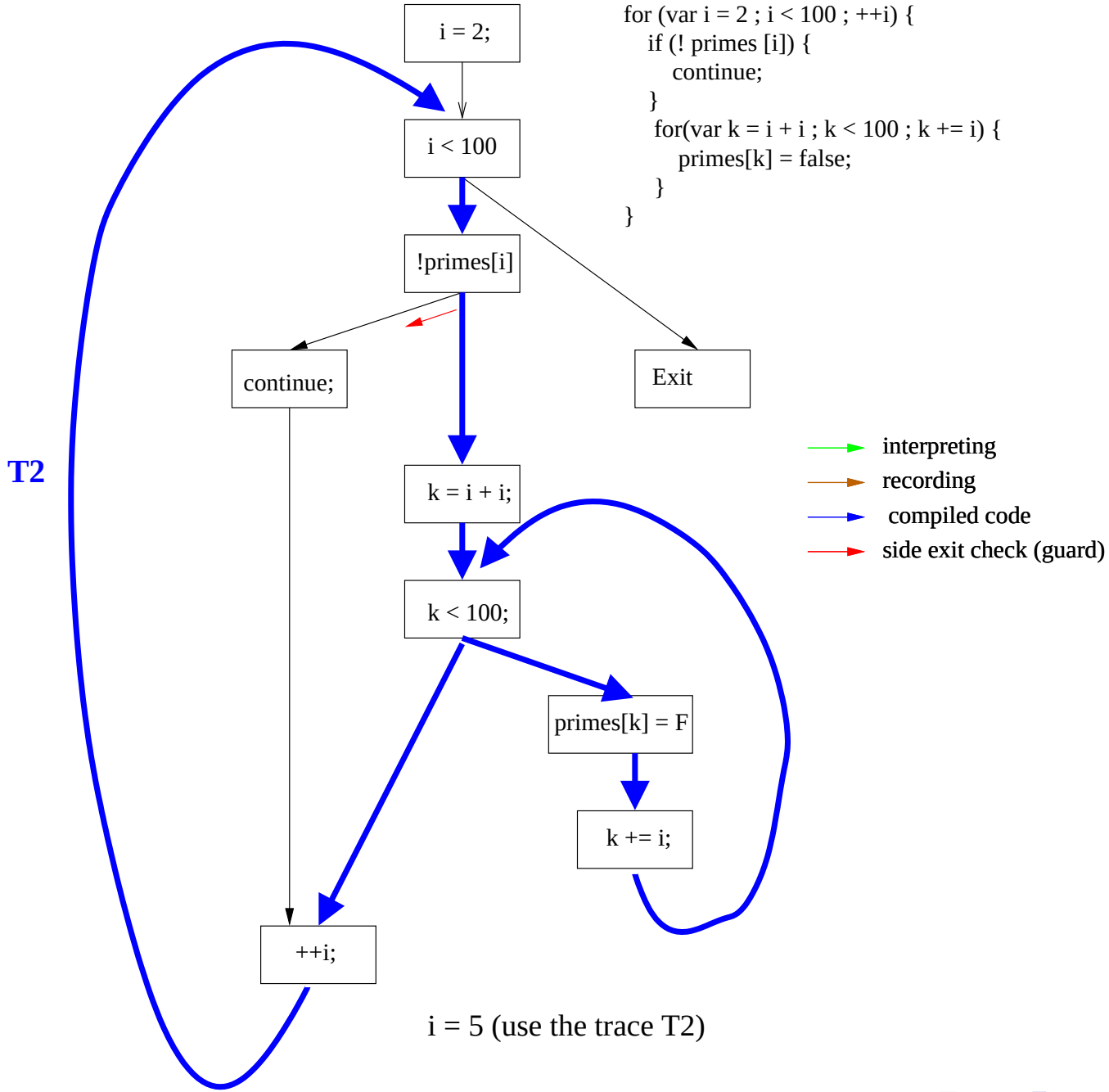
i = 3 (outer loop trace compiled)

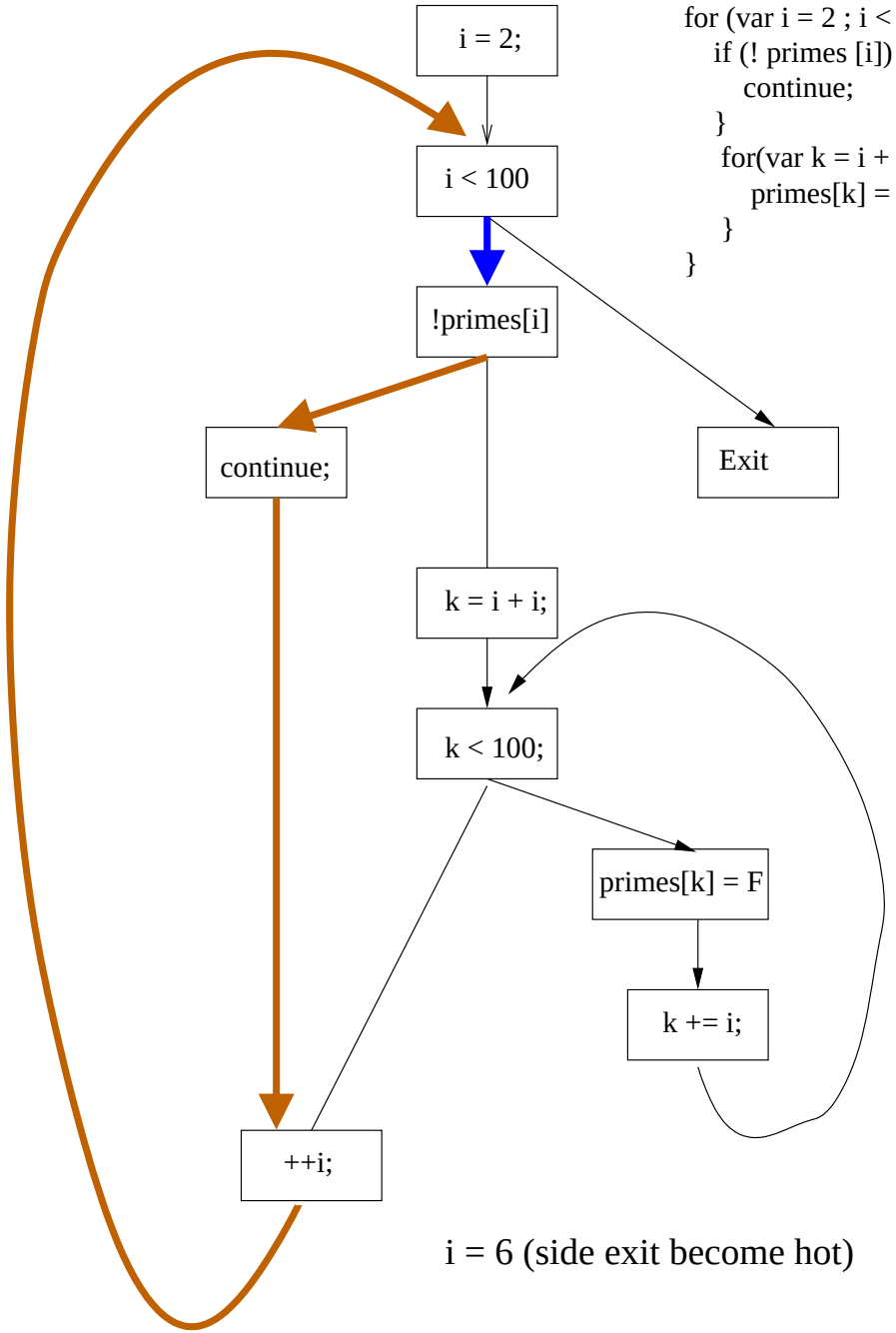



```
for (var i = 2 ; i < 100 ; ++i) {
    if (! primes [i]) {
        continue;
    }
     for(var k = i + i ; k < 100 ; k += i) {
         primes[k] = false;
     }
}
```

interpreting
recording
compiled code
side exit check (guard)

i = 4 (side exit from Trace 2)

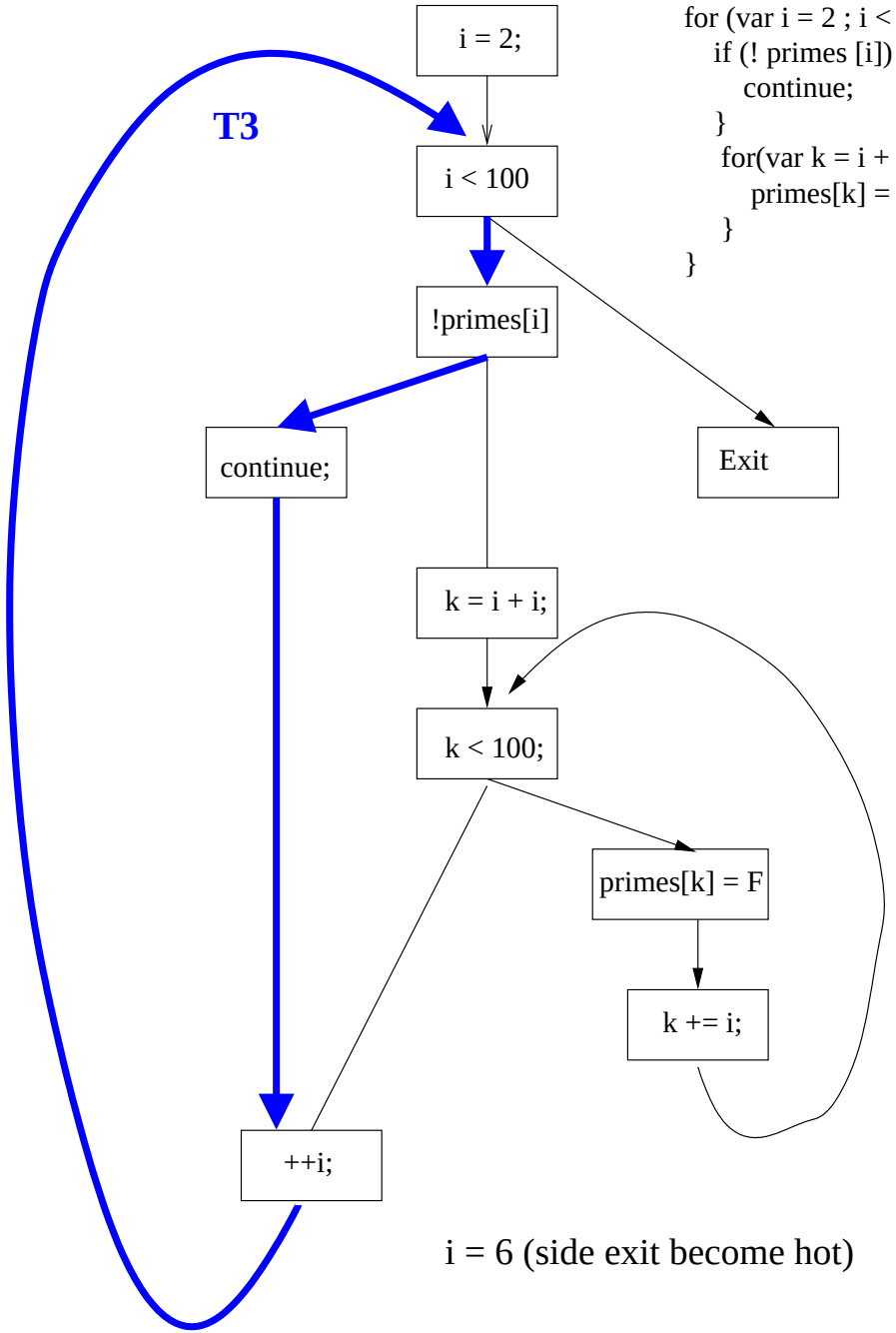

for (var i = 2 ; i < 100 ; ++i) {
if (! primes [i]) {
continue;
}
for(var k = i + i ; k < 100 ; k += i) {
primes[k] = false;
}
}
i = 2;
i < 100
!primes[i]
continue;
Exit
k = i + i;
k < 100;
primes[k] = F
k += i;
++i;
T2
interpreting
recording
compiled code
side exit check (guard)
i = 5 (use the trace T2)




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

interpreting
recording
compiled code
side exit check (guard)

i = 6 (side exit become hot)



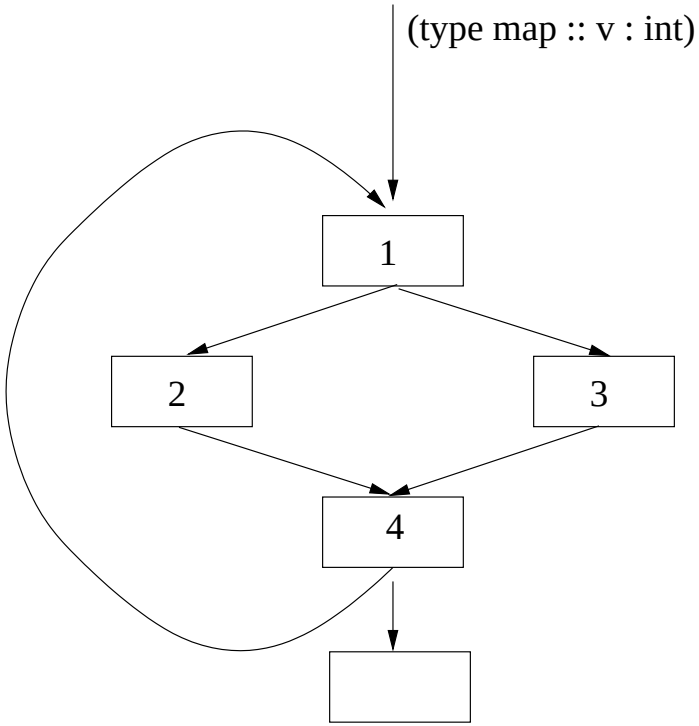

```
for (var i = 2 ; i < 100 ; ++i) {
   if (! primes [i]) {
      continue;
   }
    for(var k = i + i ; k < 100 ; k += i) {
       primes[k] = false;
    }
}
```

interpreting
recording
compiled code
side exit check (guard)

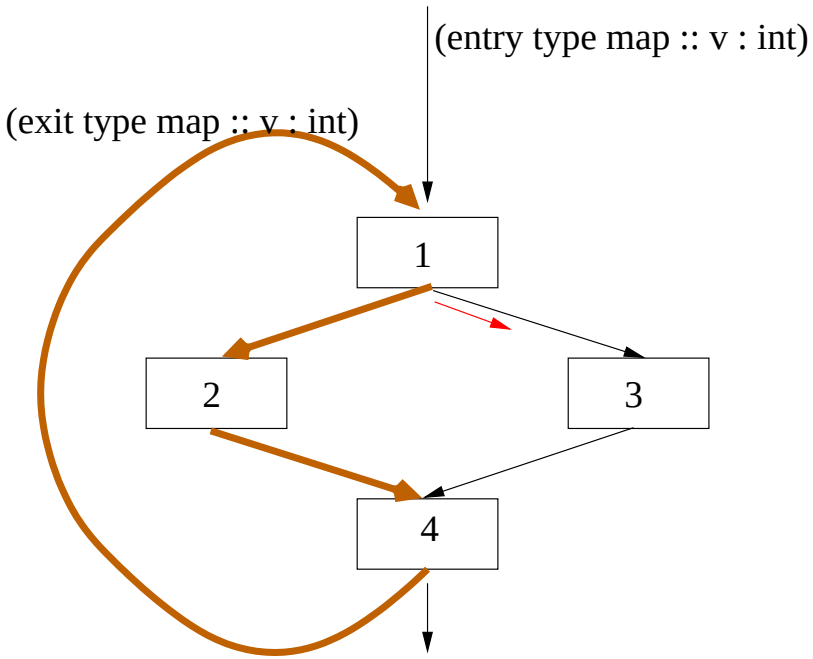
i = 6 (side exit become hot)

Outline

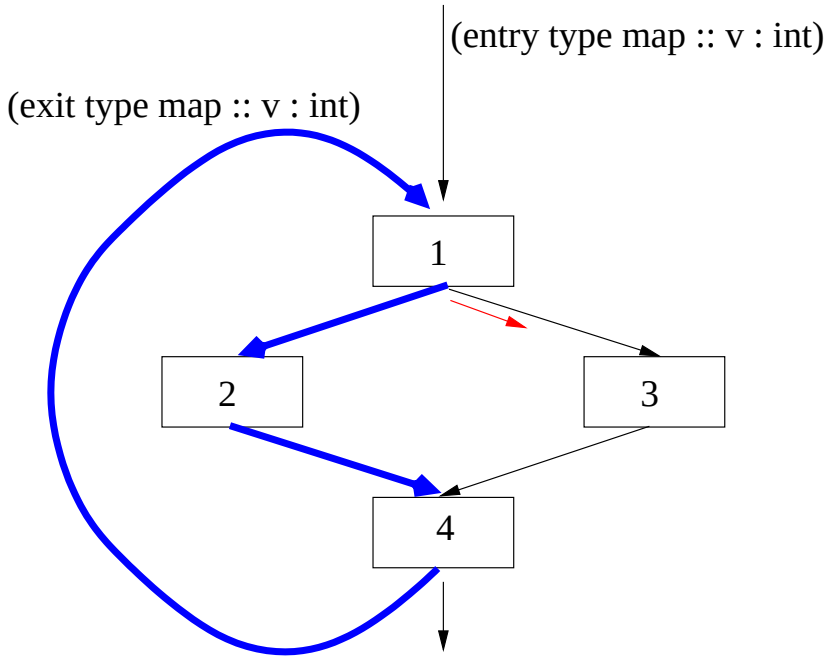
Type Stable Trace



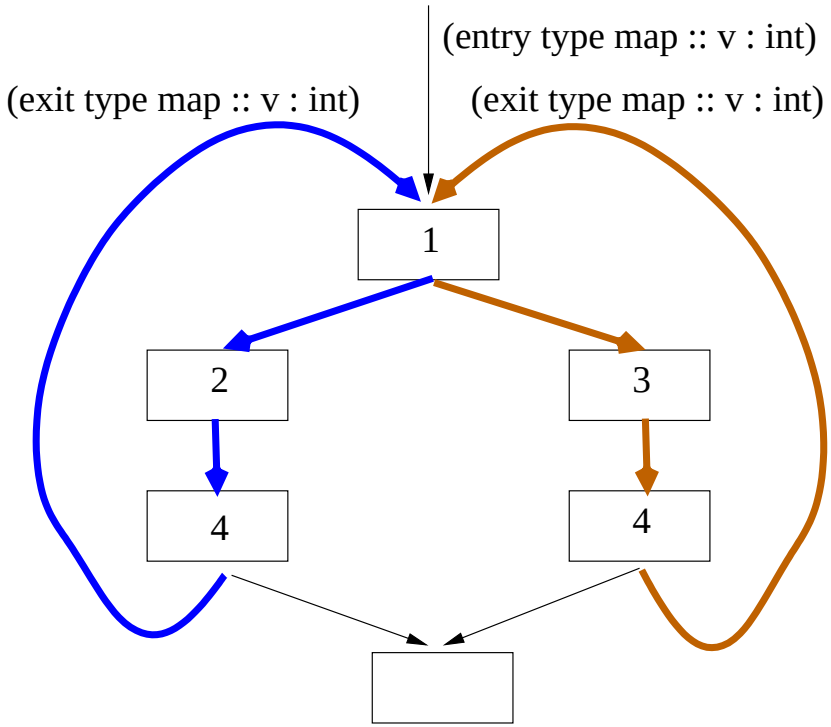
Type Stable Trace



Type Stable Trace



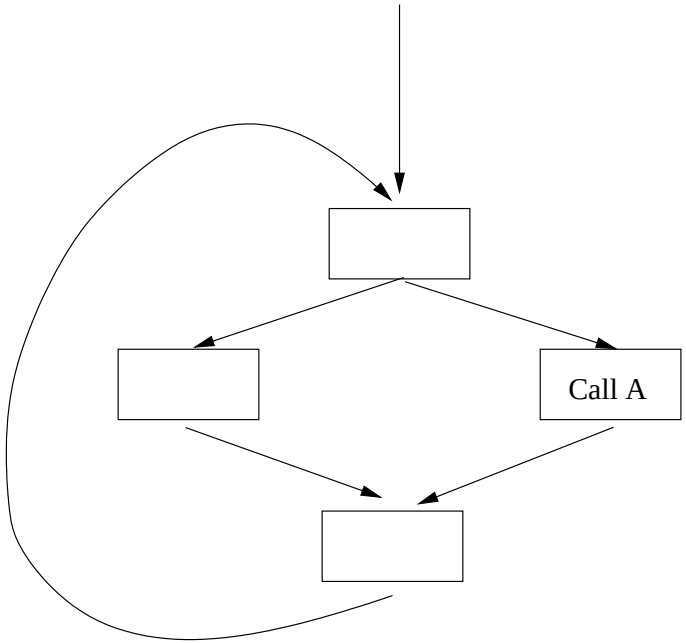
Trace Tree Extension



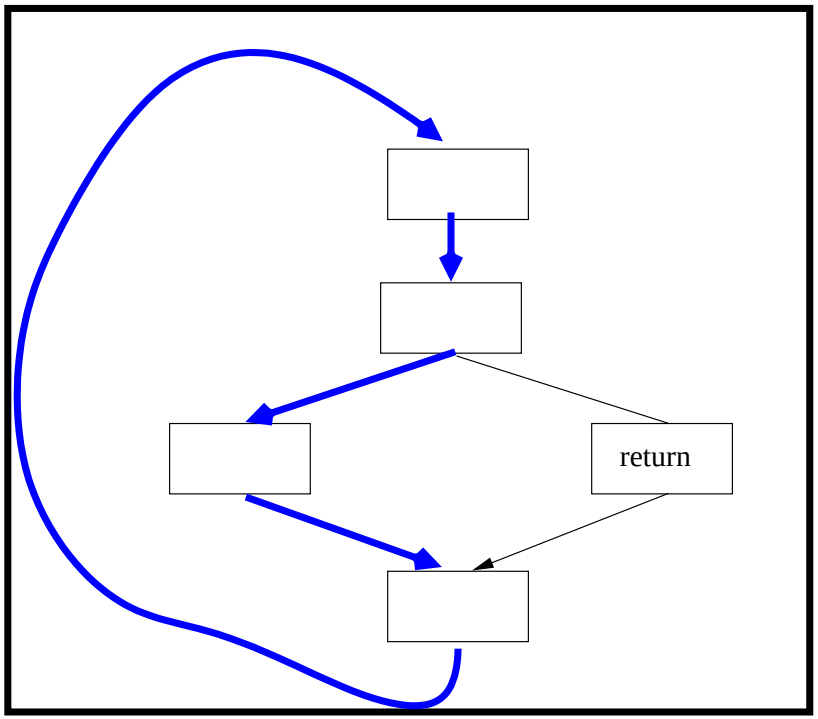
Note: Current Implementation extends only if

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

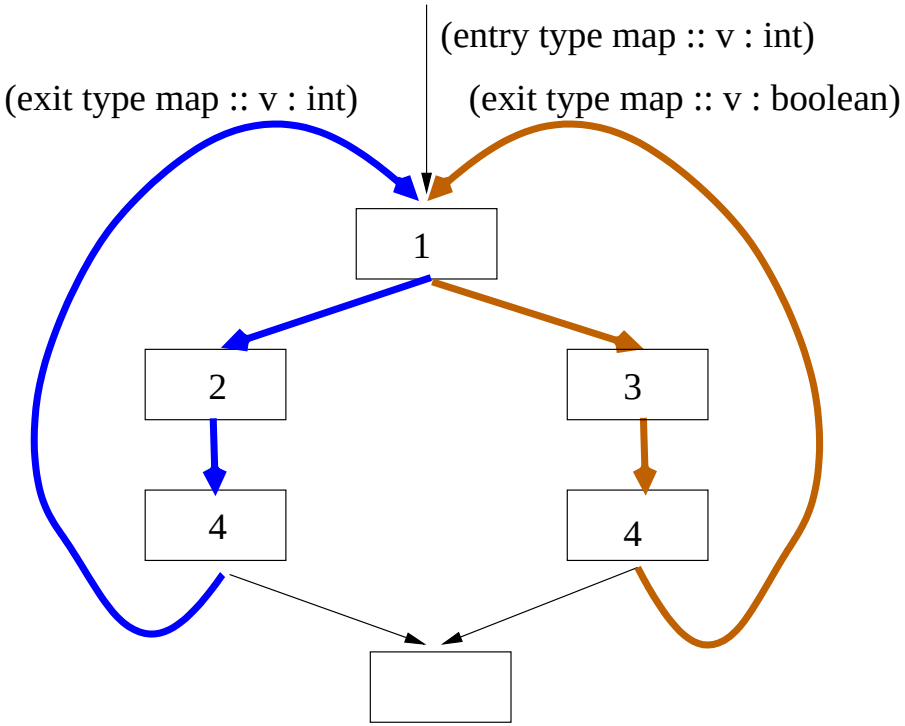
Problems with extension on return



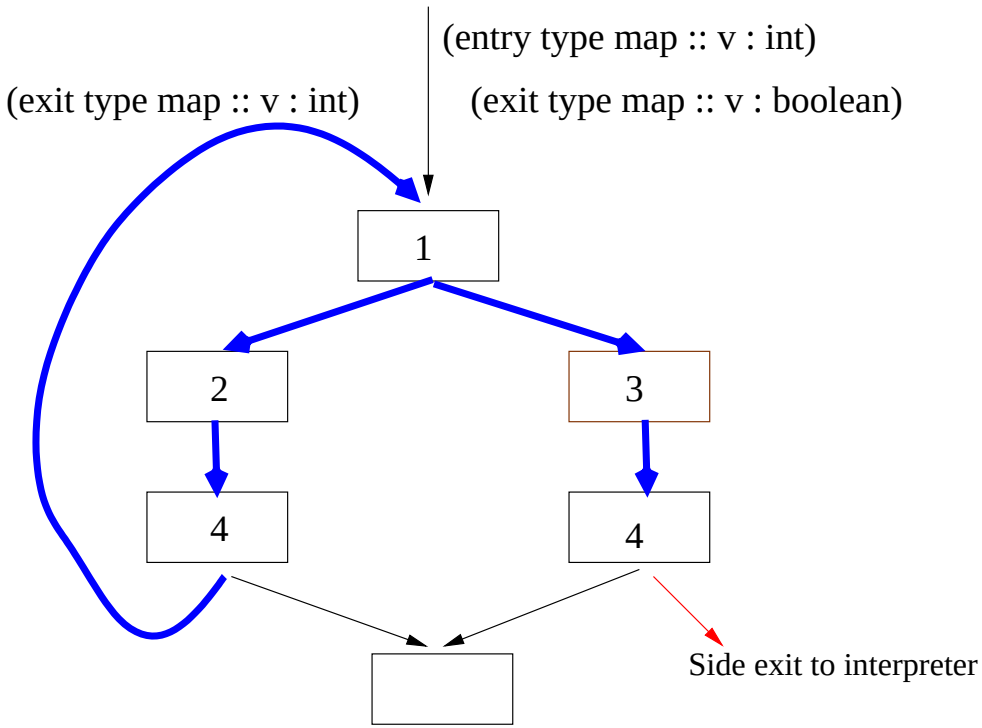
Procedure A



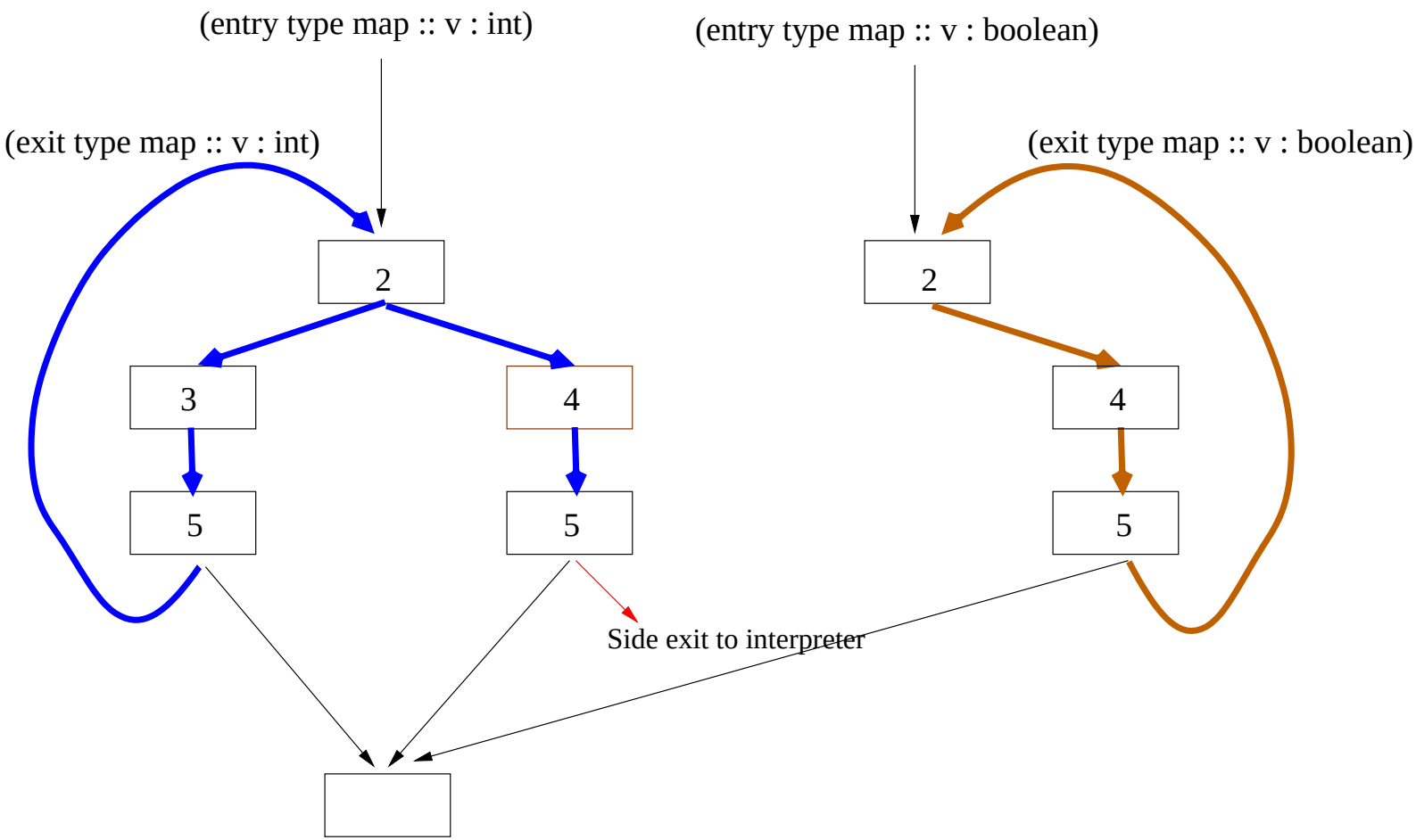
Type Unstable Trace Handling



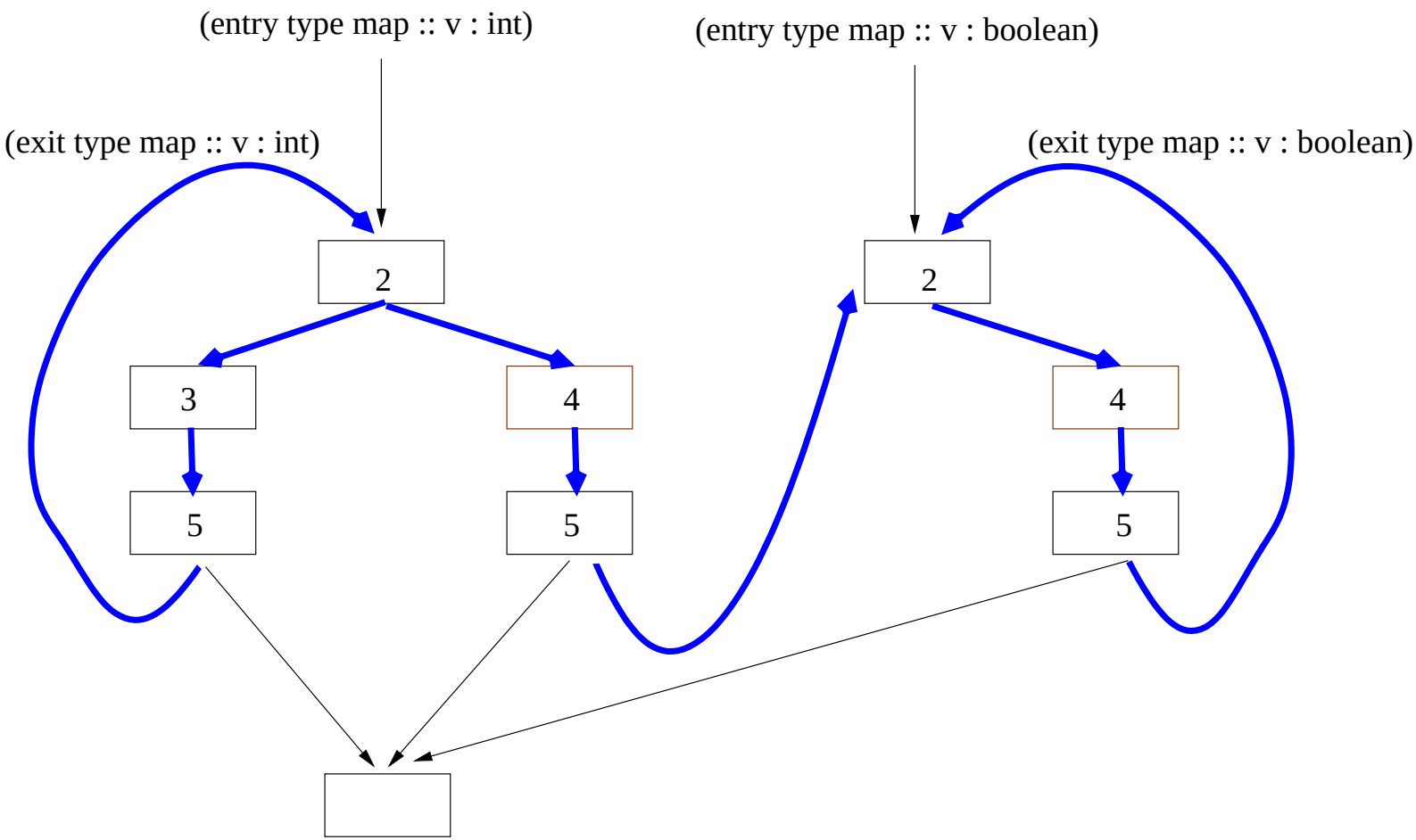
Type Unstable Trace Handling



Type Unstable Trace Handling

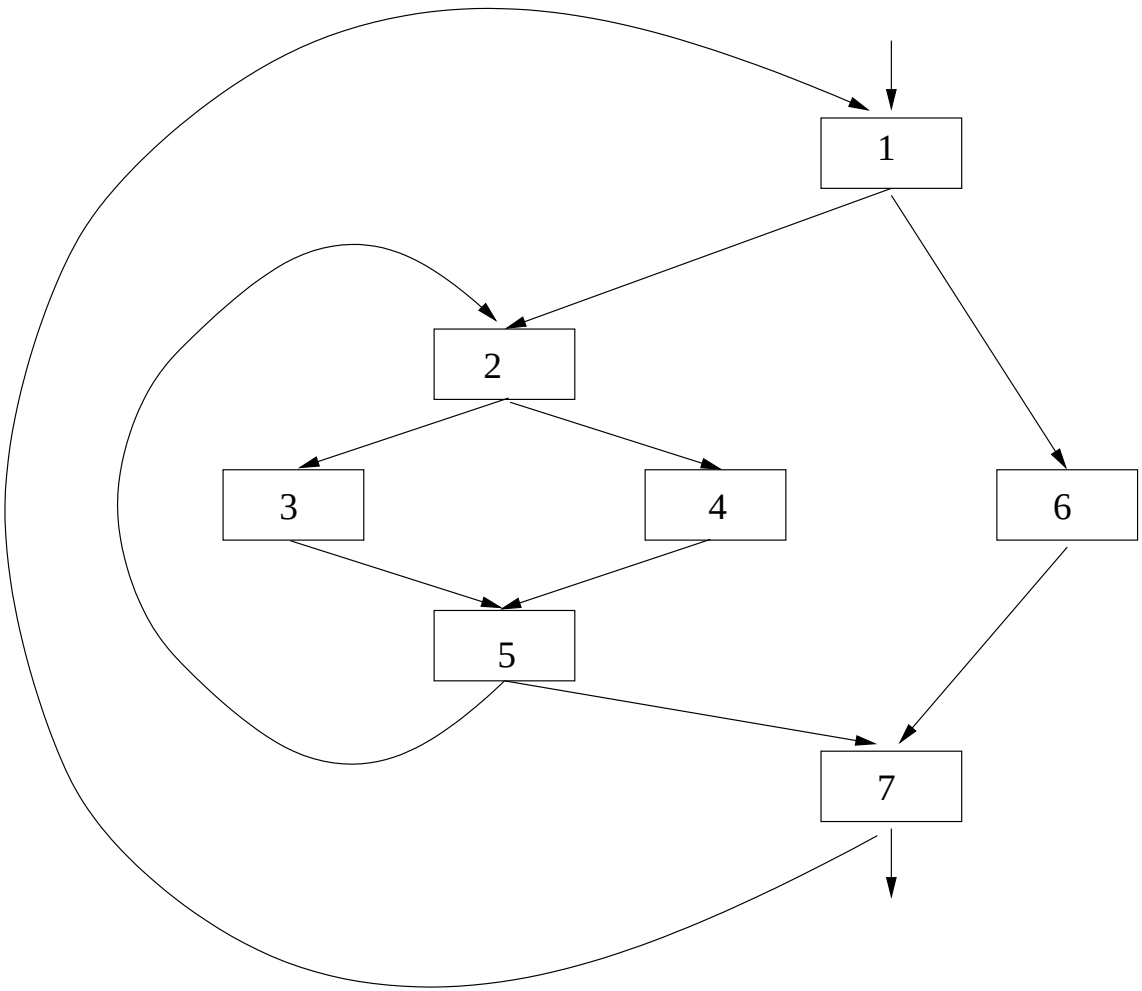


Type Unstable Trace Handling

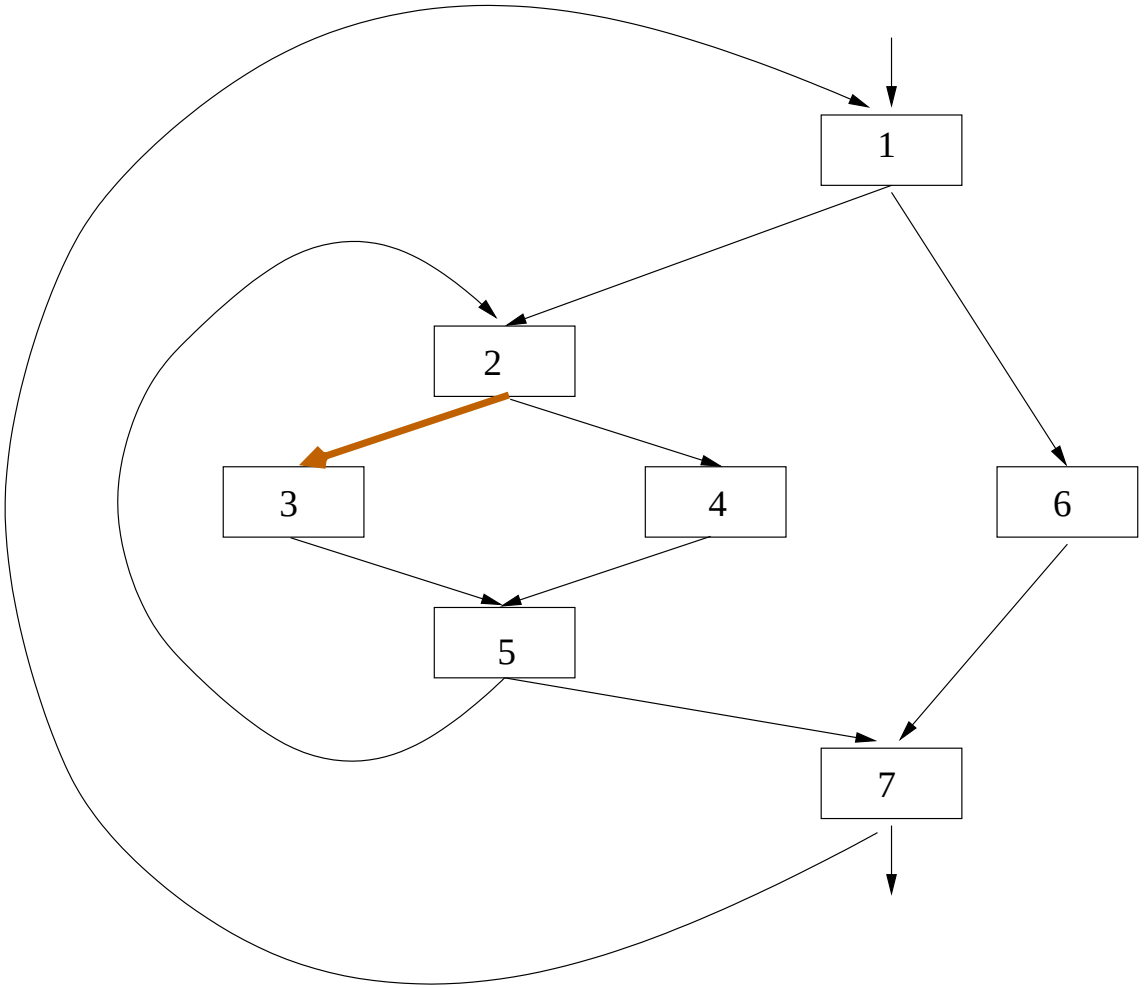


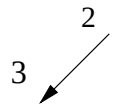
Outline

Nested Trace Tree Formation

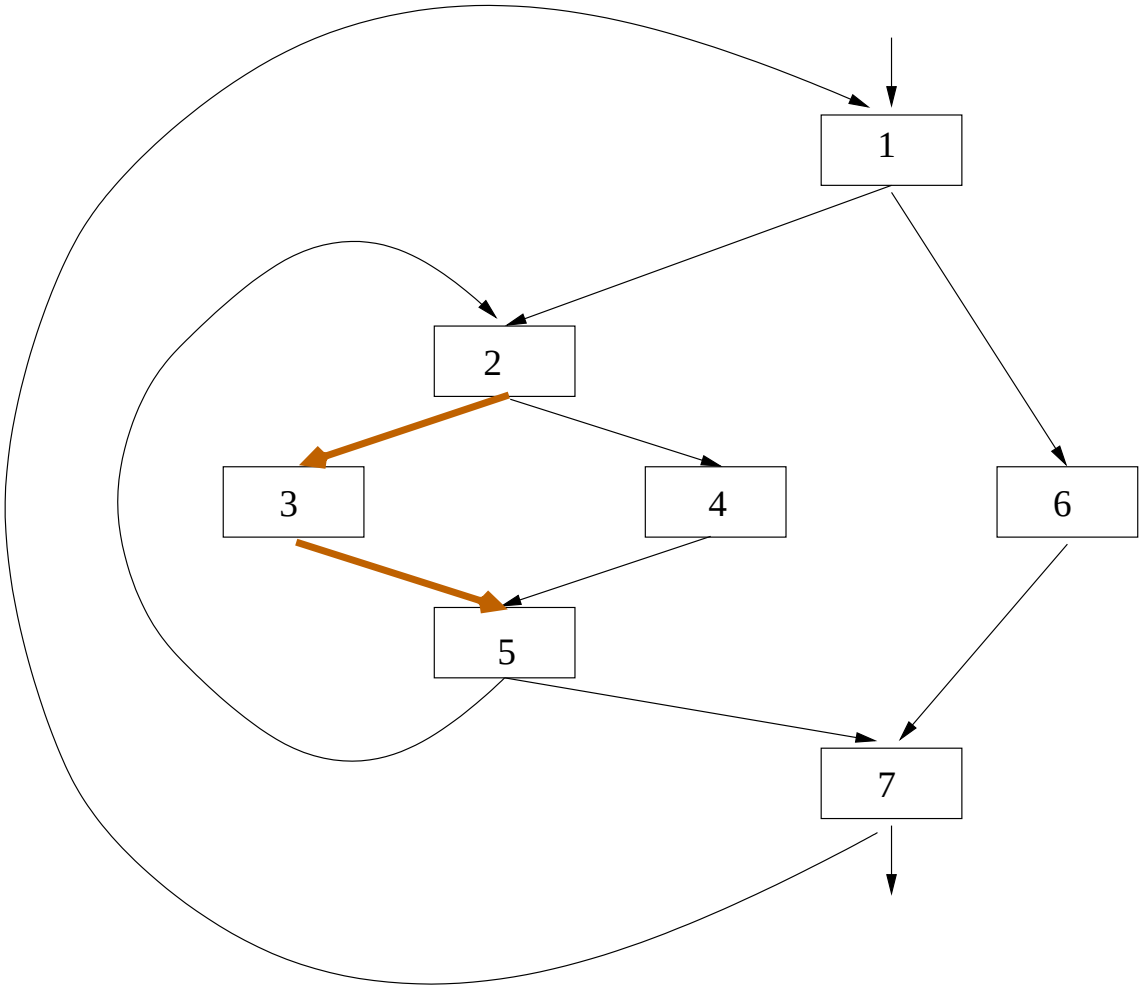


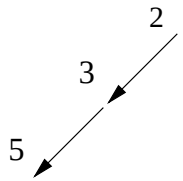
Nested Trace Tree Formation



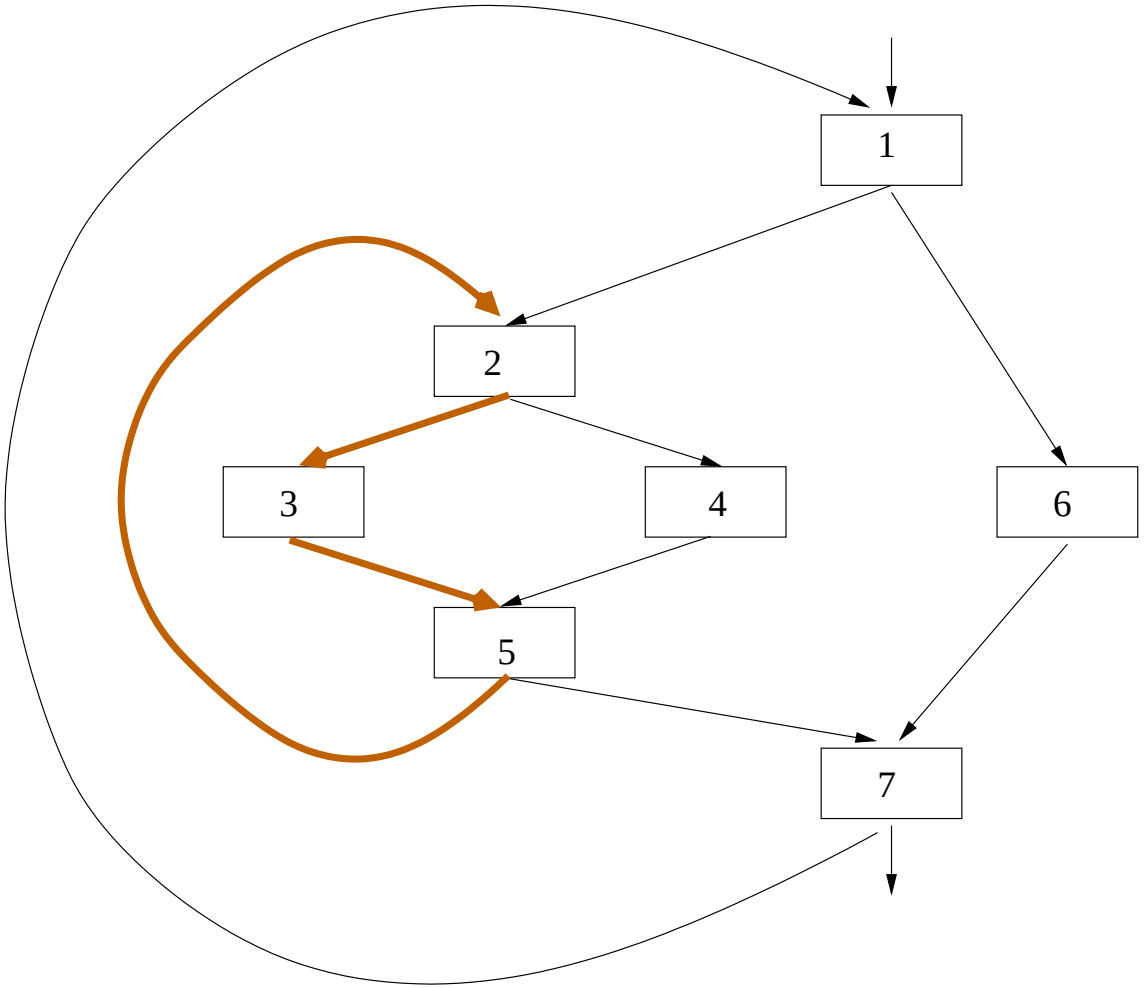


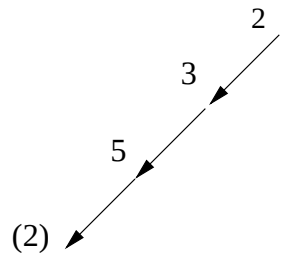
Nested Trace Tree Formation



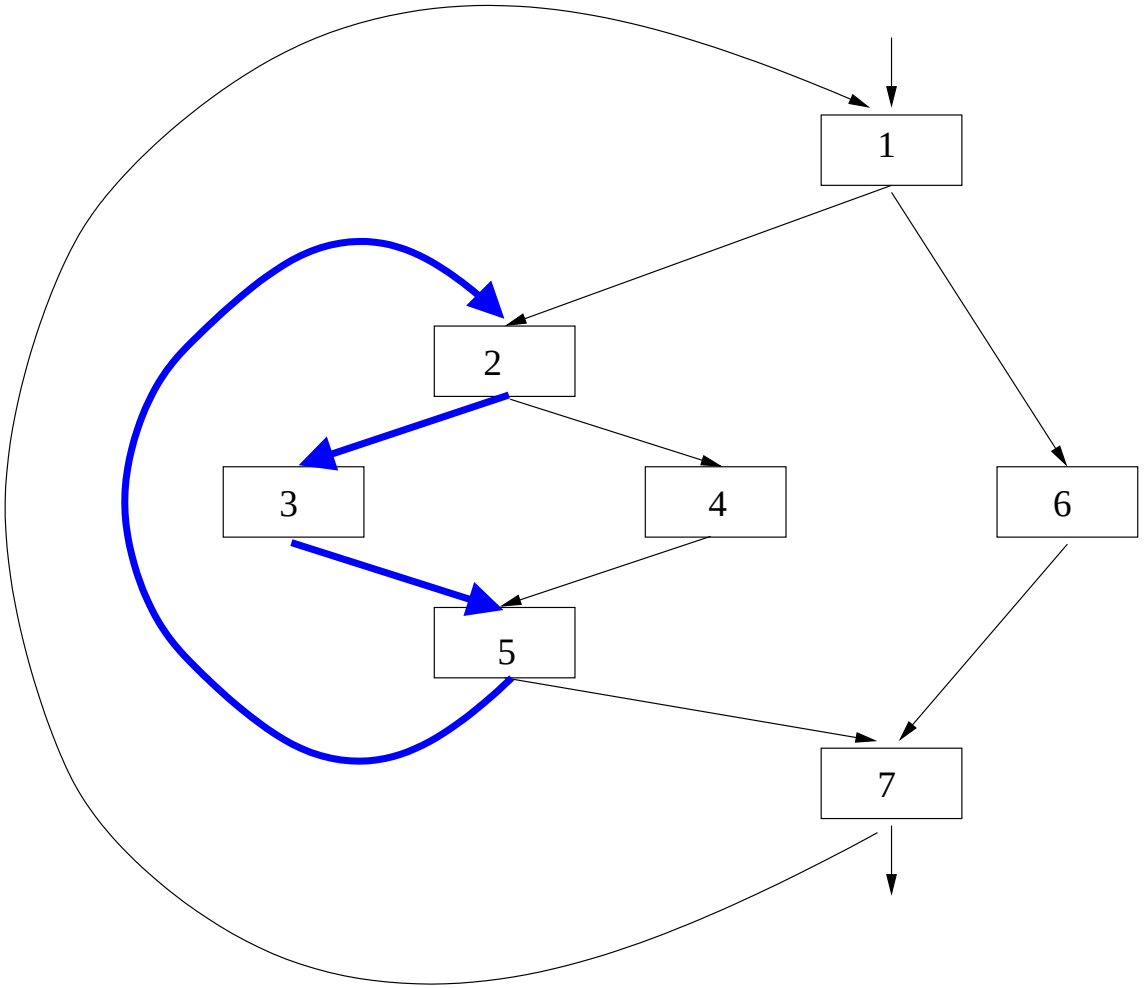


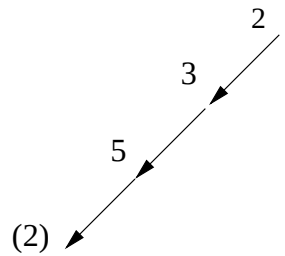
Nested Trace Tree Formation



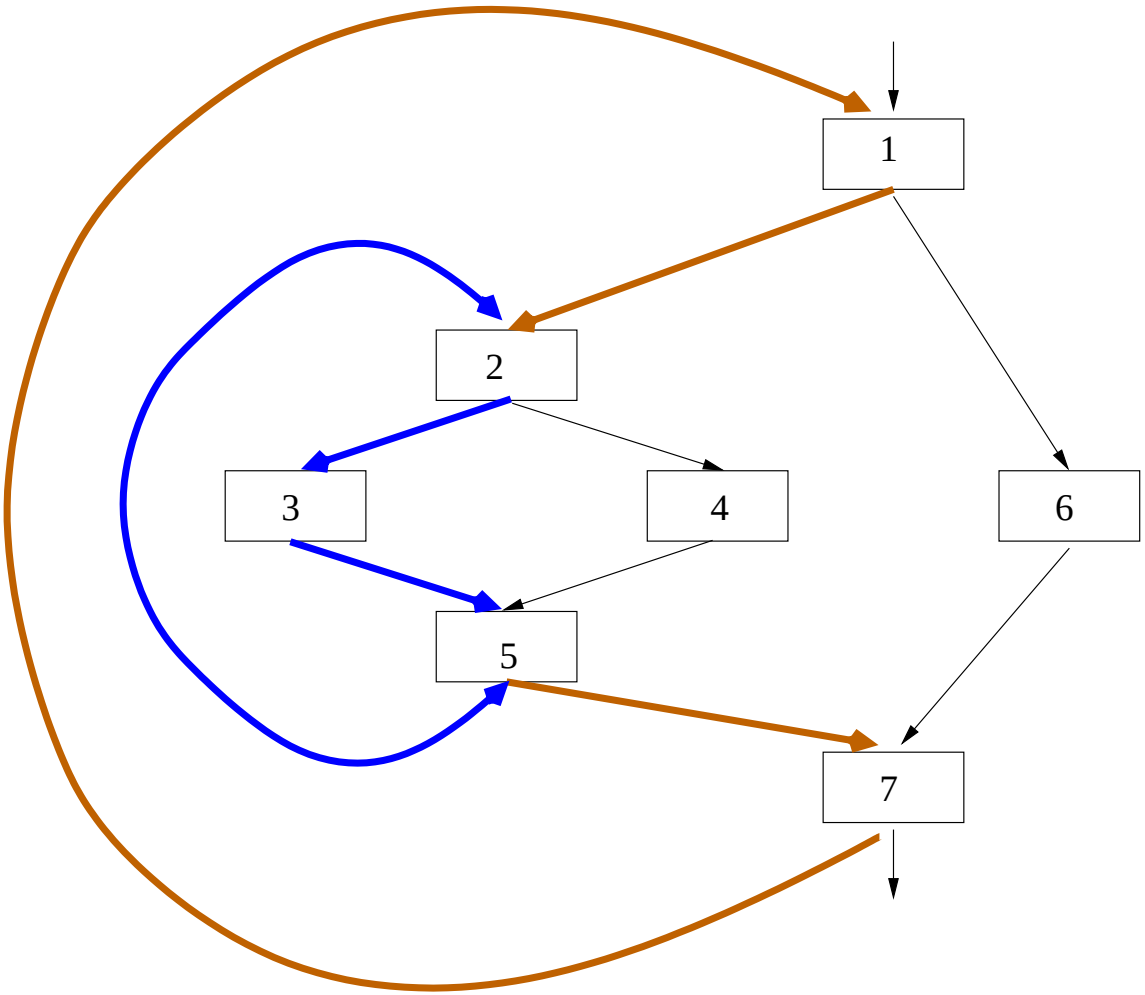


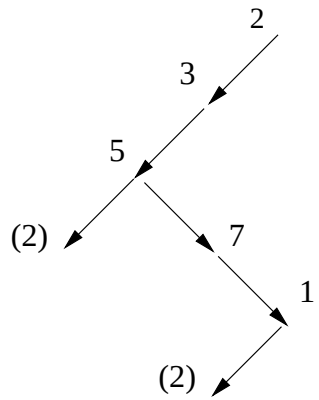
Nested Trace Tree Formation

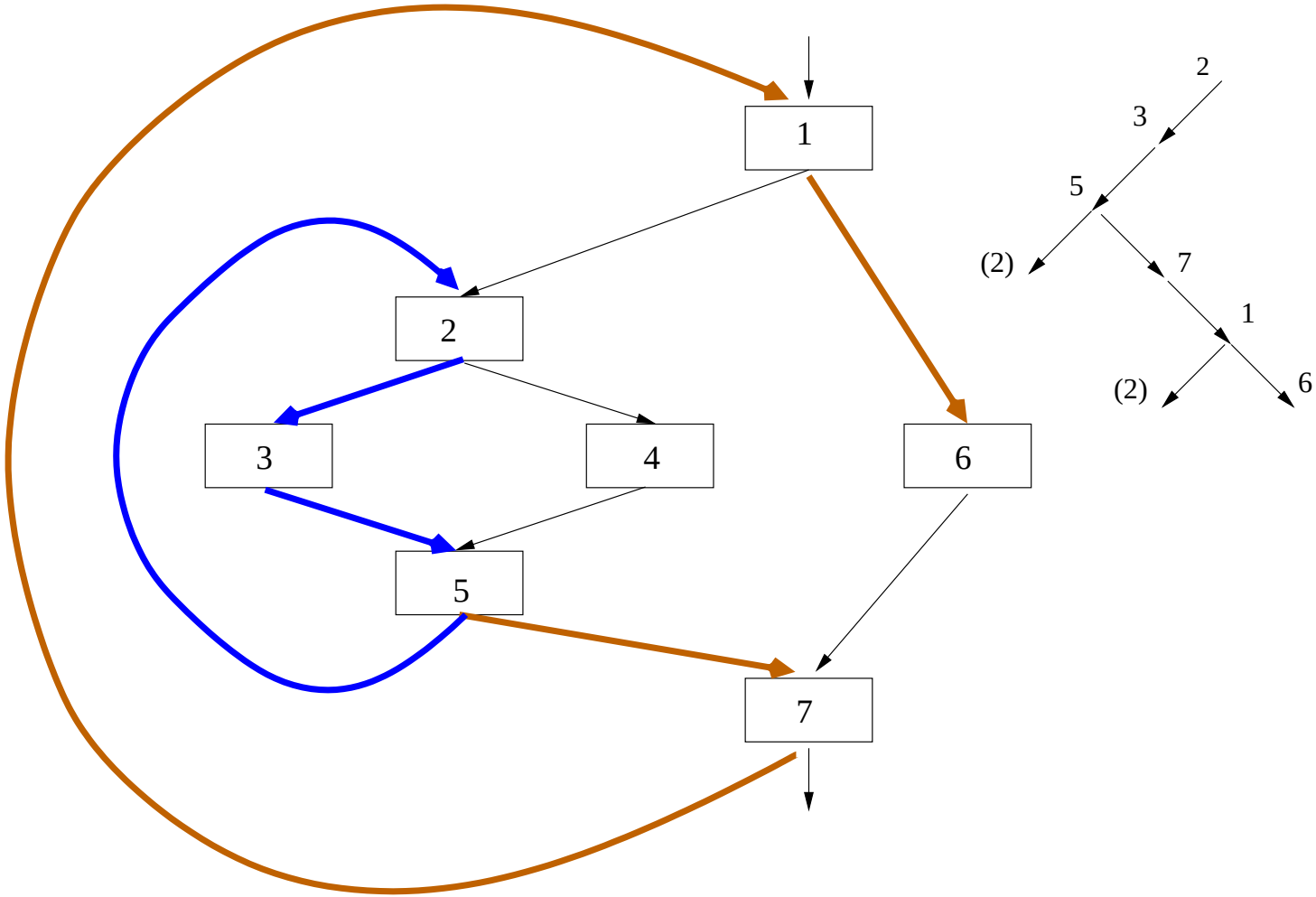




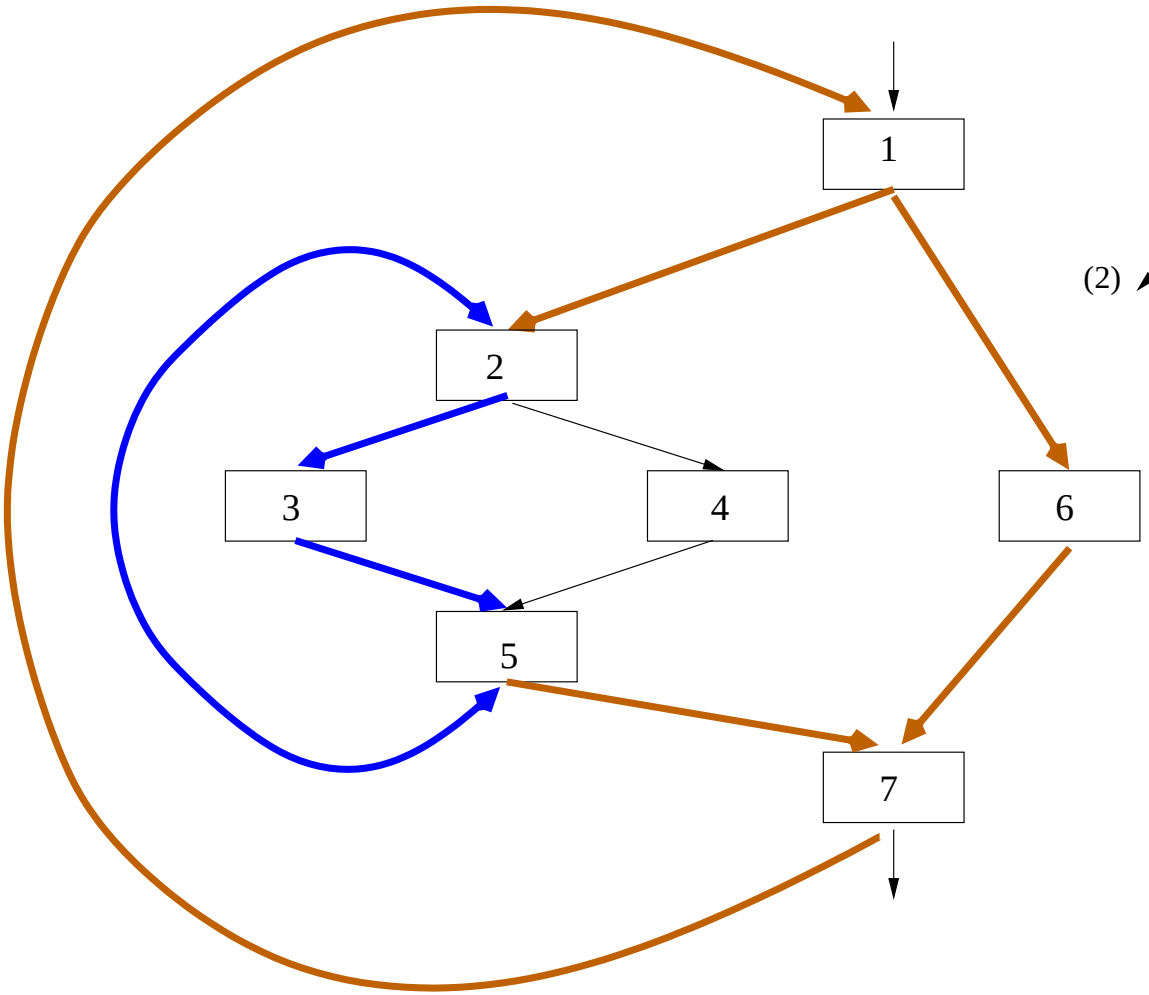
Continue tracing for outer loop

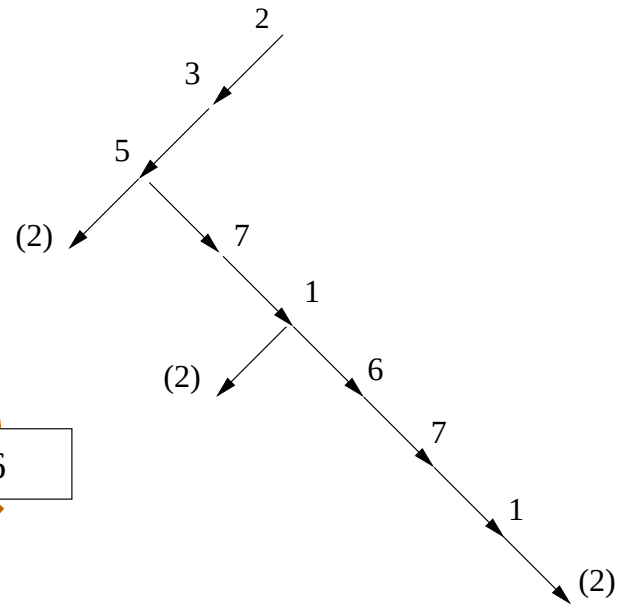




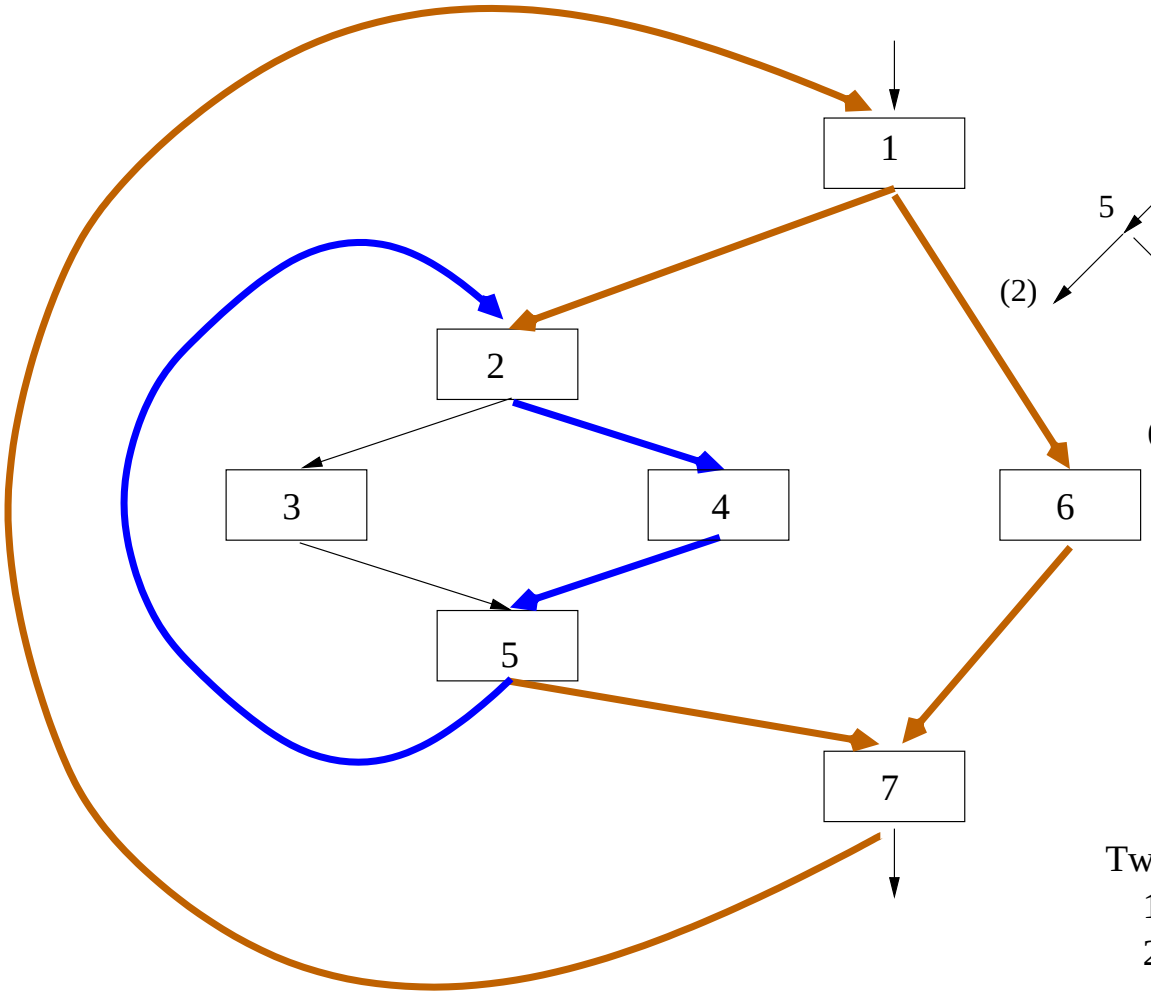

1
2
3
4
5
6
7
2
3
5
(2)
7
1
(2)
6

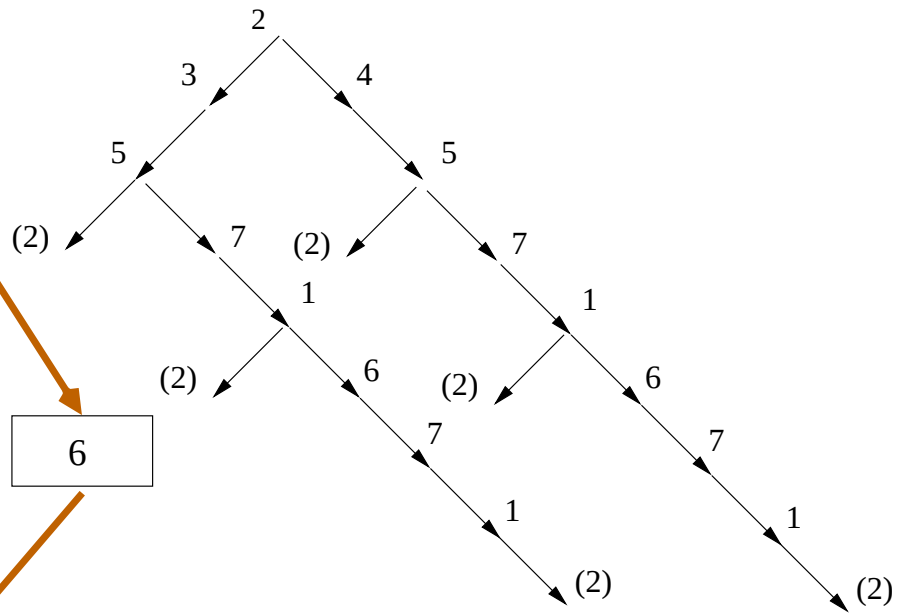
Continue tracing for outer loop





Continue tracing for outer loop

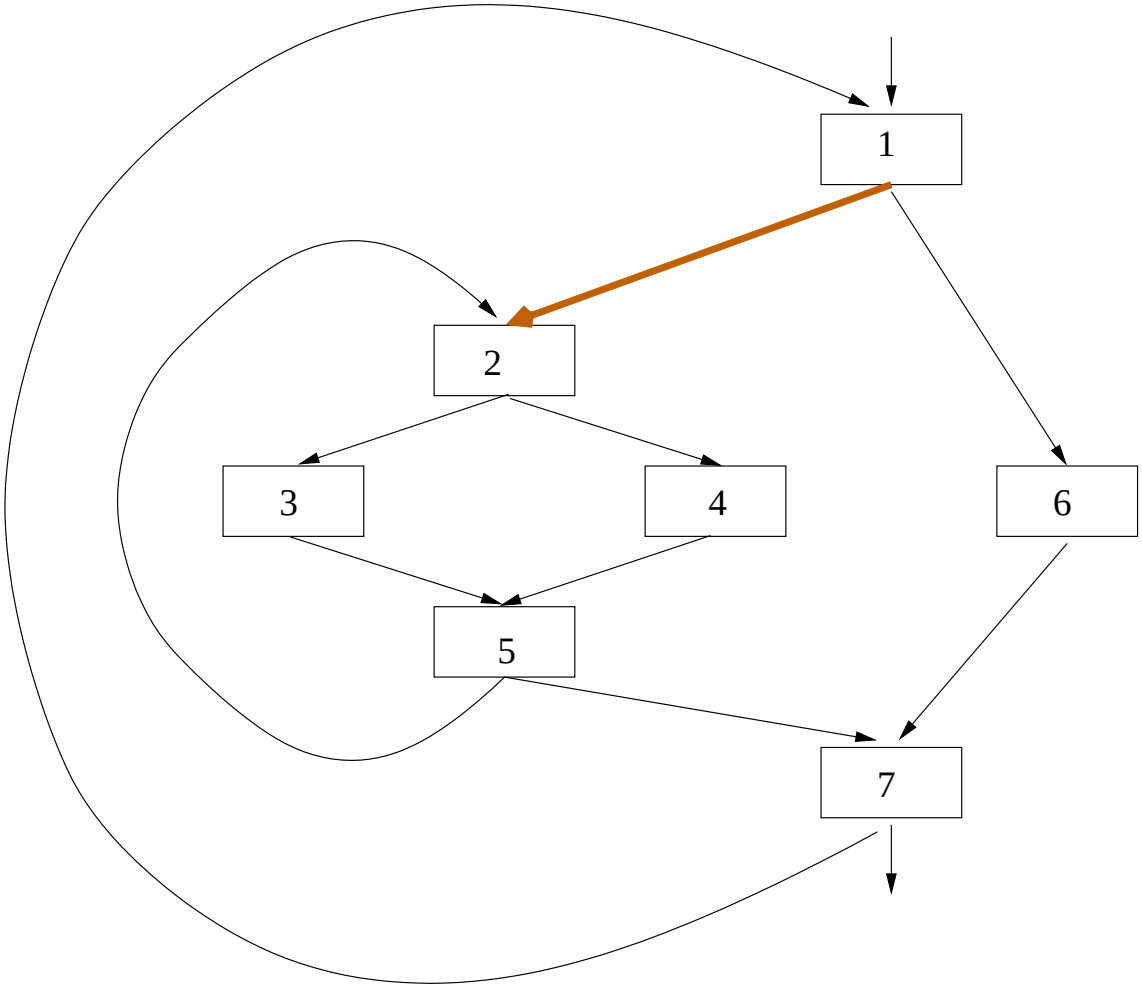




Two Problems:

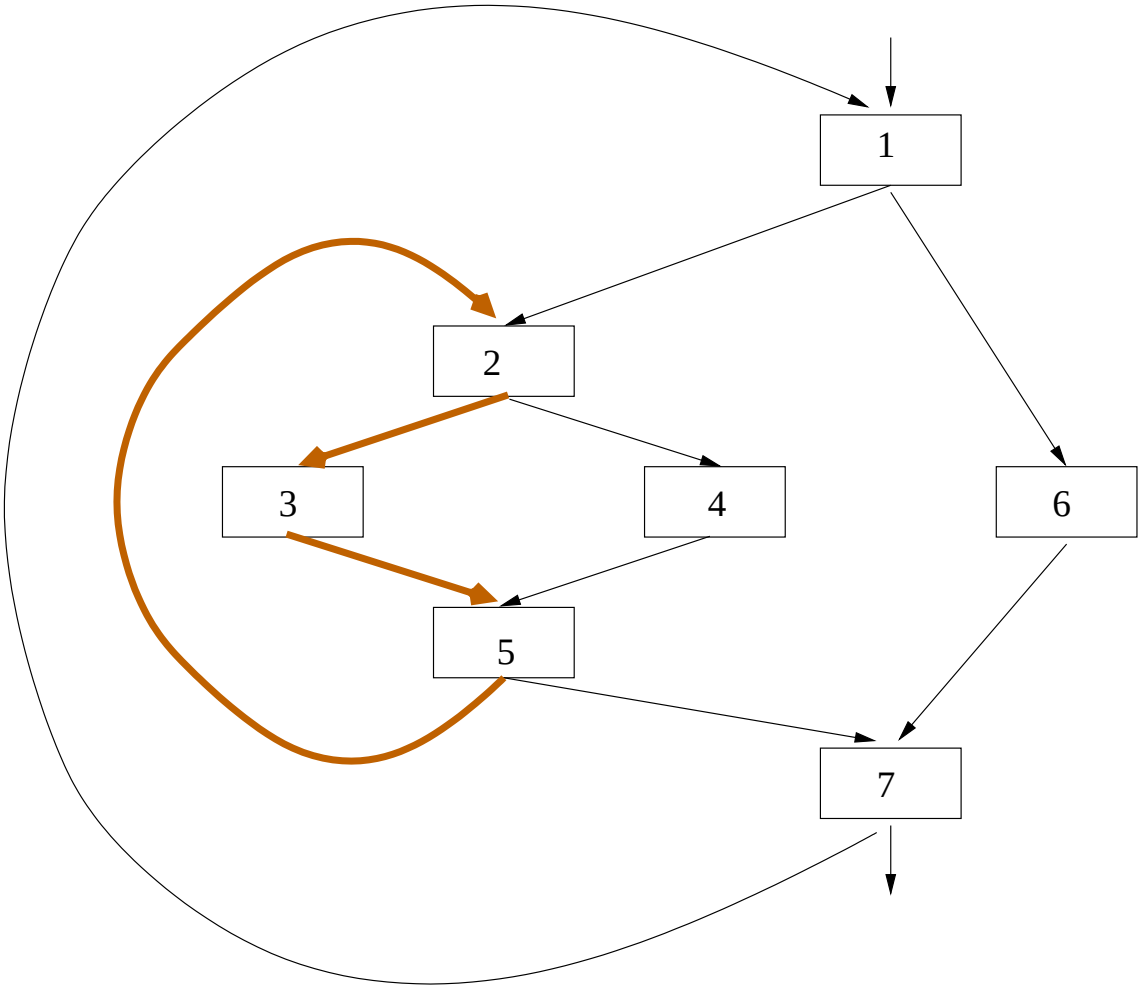
1. Trace Duplication
2. Traces may grow indefinitely

Separate traces



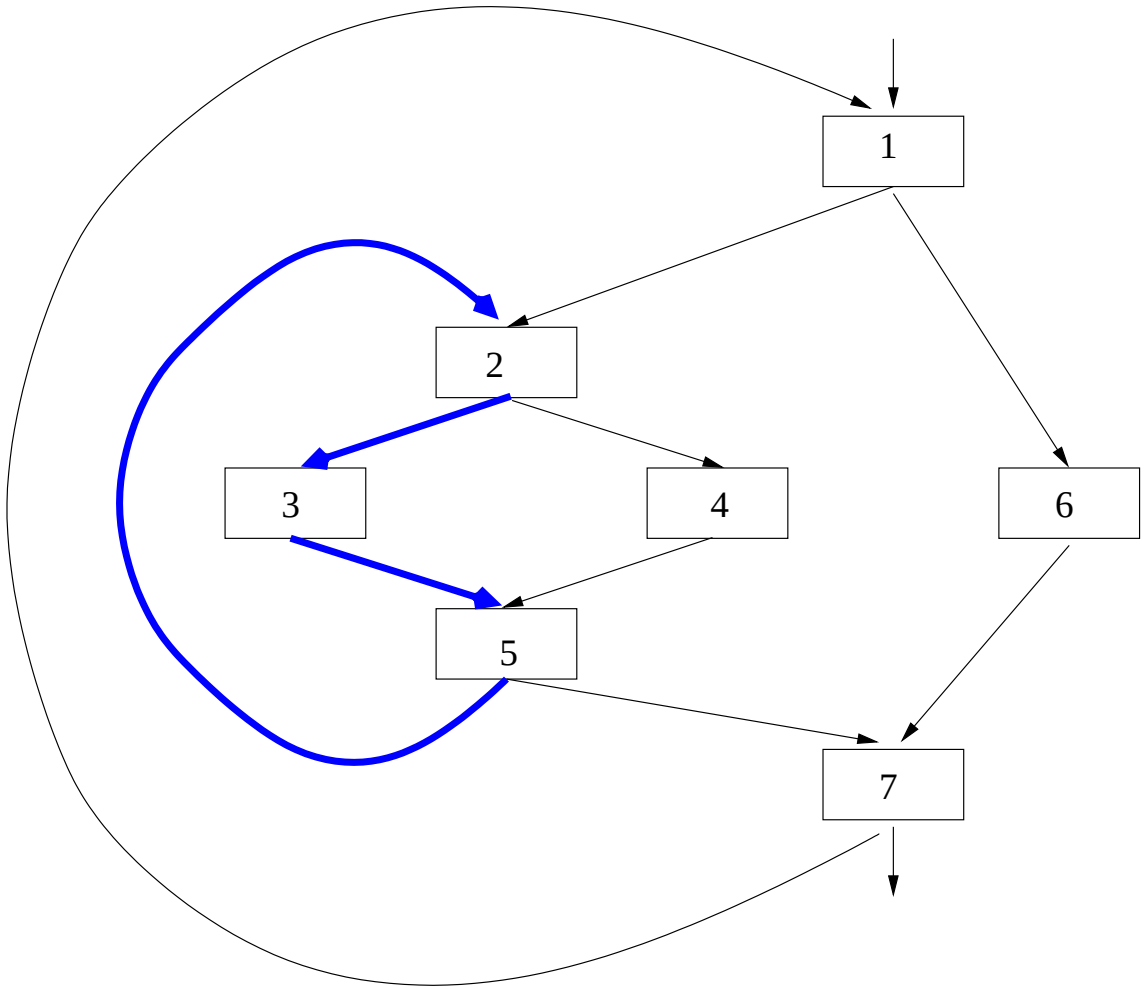
Outer loop starts recording

Separate traces



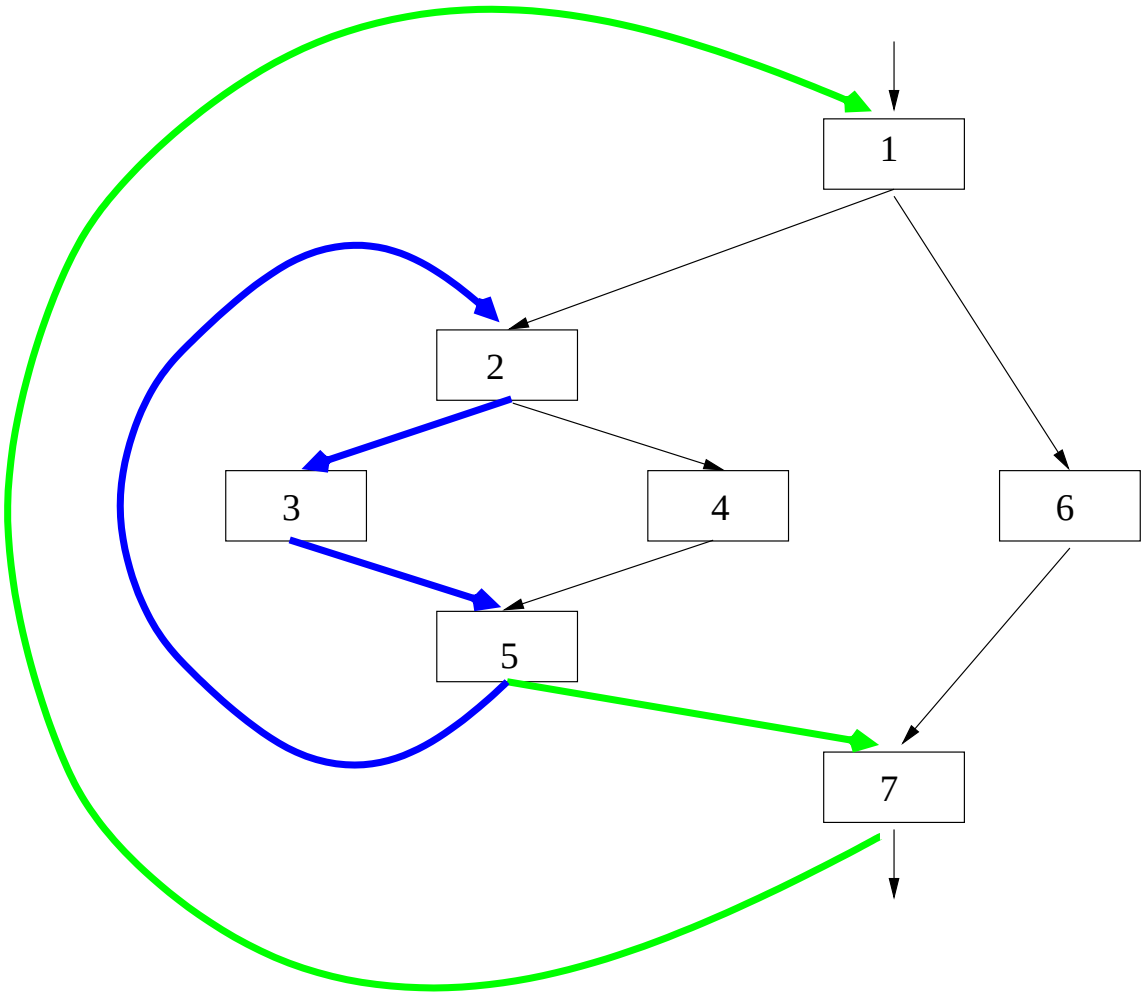
Outer Loop recording stops as "type matched" inner compiled trace not available.

Separate traces



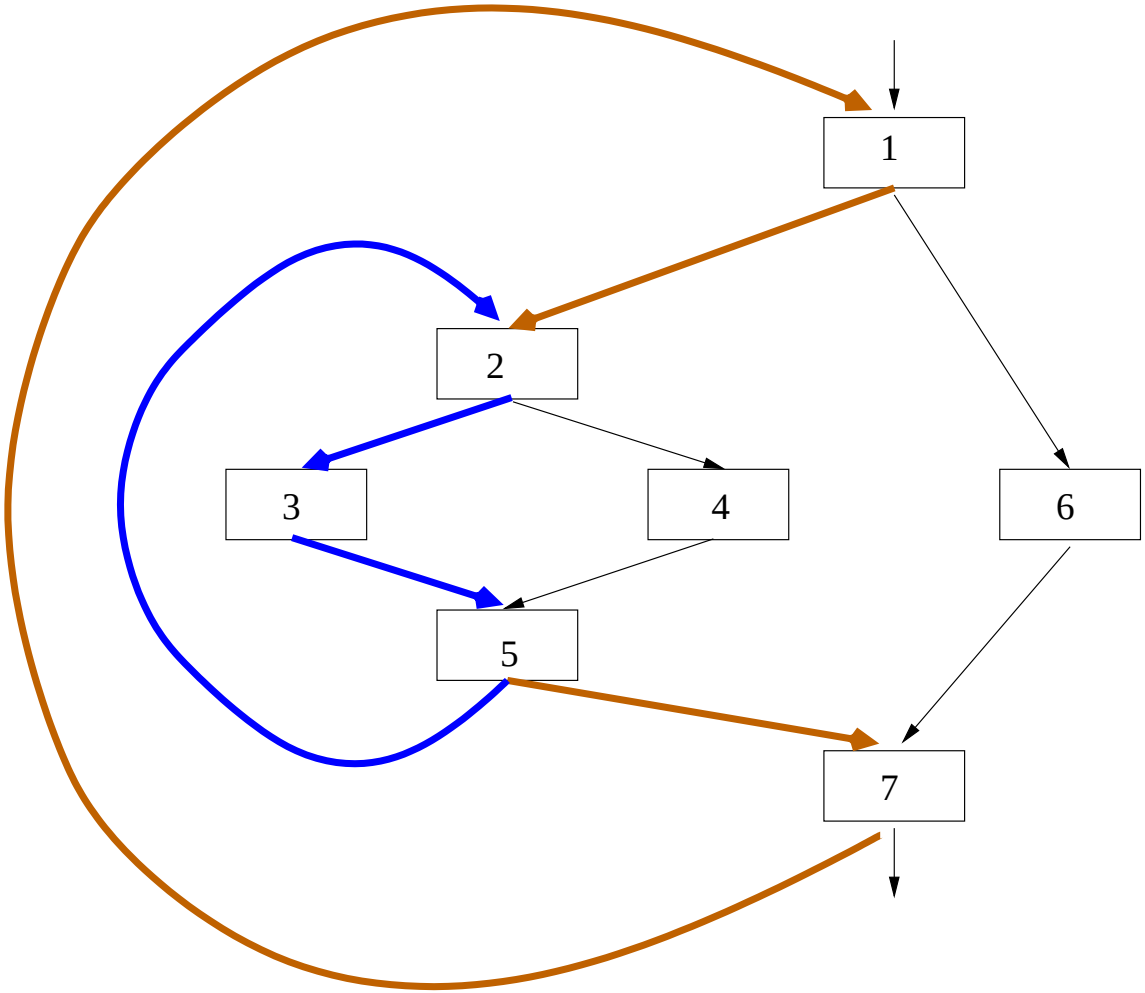
Inner compiled trace ready

Separate traces



Inner compiled finishes (No further recording)

Separate traces



Outer loop records the inner trace call in its recording

Outline

Trace Tree Optimization

- Optimization Goal: Make compilation fast. (Done by NanoJIT)
- 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

Backward optimizations

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

Register Allocation

V0 = ...

V1 = ...

V2 =

V3 [r3] = V1 + V2 // Freelist = {r3}

... = V3 [r3]

... = V0 [r0]

Register Allocation

V0 = ...

V1 = ...

V2 =

V3 [r3] = V1 [r3] + V2 // Freelist = {}

... = V3 [r3]

... = V0 [r0]

//spill Candidate = {V0, V1}

Register Allocation

V0 [r0] = ...

Mem0 = r0

V1 [r3] = ...

V2 [r0] = ...

V3 [r3] = V1 [r3] + V2 [r0]

r0 = Mem0

... = V3 [r3]

... = V0 [r0]

Outline

Setup

- SunSpider Benchmark suite is used
  - SpiderMonkey: JS interpreter. **Baseline for comparison**
  - TraceMonkey: The proposed compilation strategy
  - SquirrelFish Extreme (SFX)
  - V8, Javascript VM from Google

Results

- Speedup (TraceMonkey) spreads between 0.9 and 25.
- Performance depends on
  - Fraction of bytecode executed as native trace
  - Cost of Recording, Compilation, Calling Traces
- Outperform both the VMs on 9/26 benchmarks.

My Critique

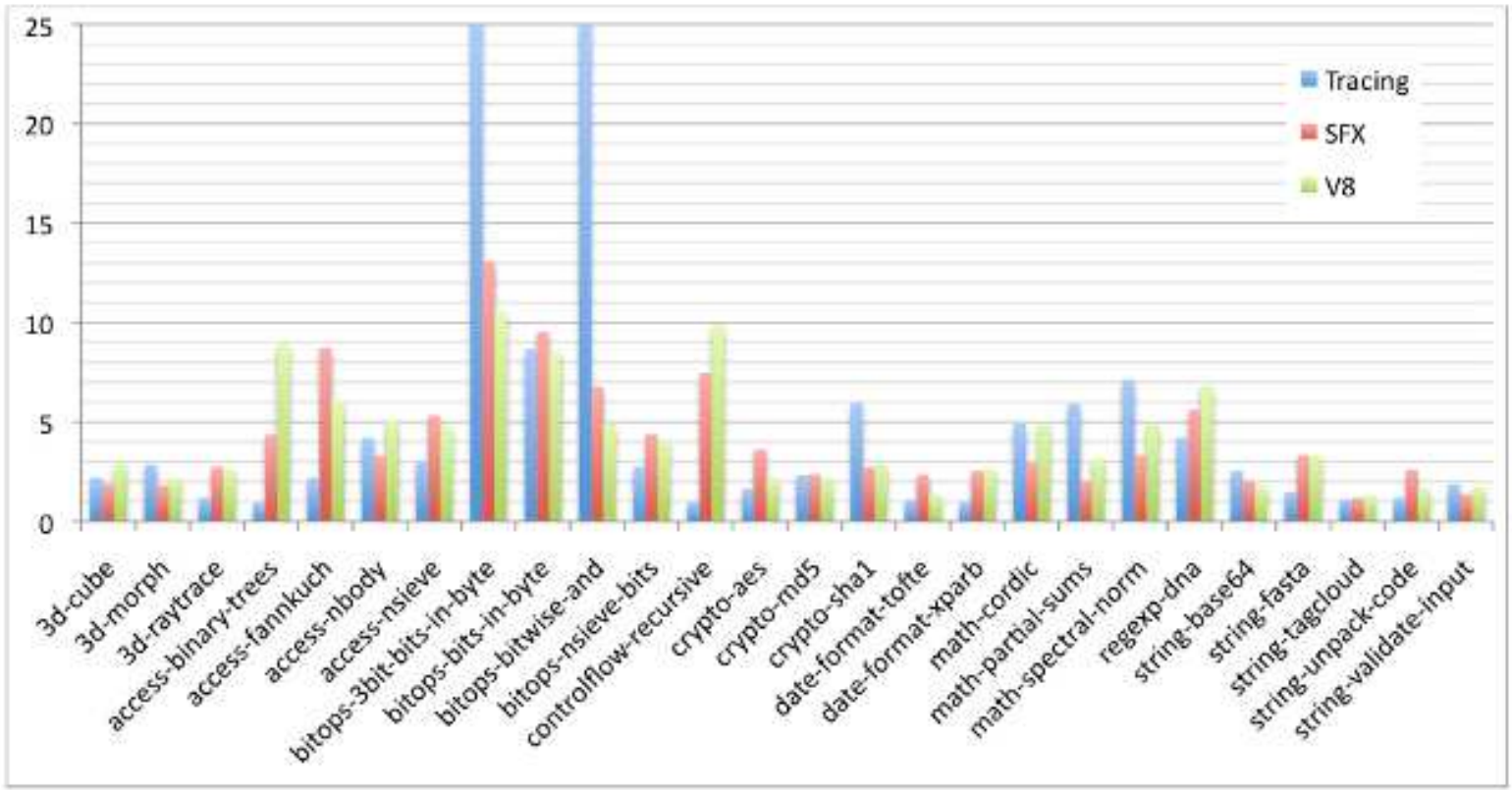
- Novel approach.
- Entirely loop based.
- Sunspider is too small a benchmark to show the effectiveness.

Outline

Static Type Inferencing

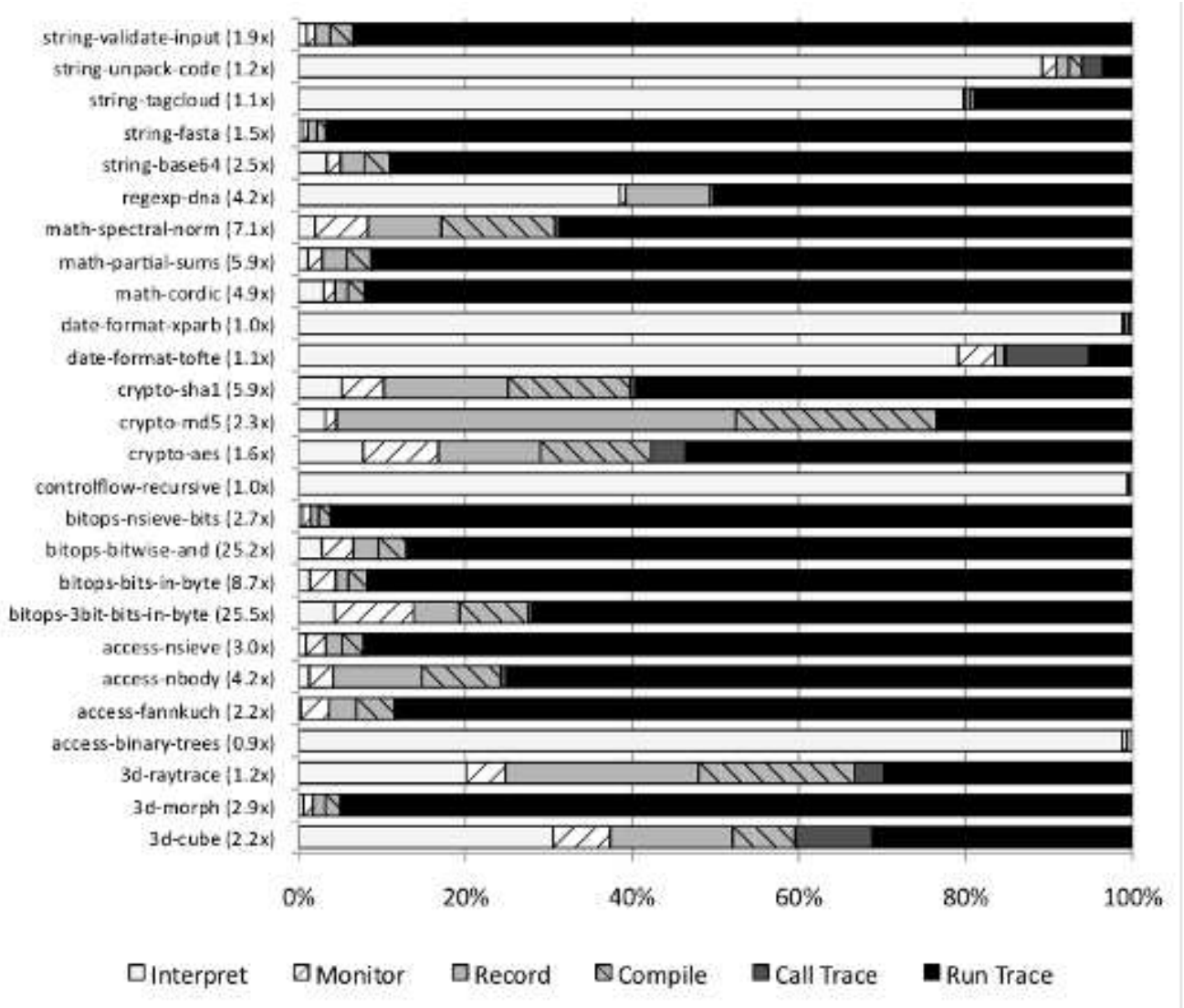
```
var x = 0 ;
for (var i = 0 ; i < a; i ++ ) {
   x = x + i;
   if(x == b)    {
        x = "number : " + x;
     }
 }
```

Speedup Comparison



[Figure copied from presentation paper]

Fraction of time spend in major VM activities



[Figure copied from presentation paper]

Trace recording statistics for Sunspider Benchmark

	Loops	Trees	Traces	Aborts	Flushes	Trees/Loop	Traces/Tree	Traces/Loop	Speedup
3d-cube	25	27	29	3	0	1.1	1.1	1.2	2.20x
3d-morph	5	8	8	2	0	1.6	1.0	1.6	2.86x
3d-raytrace	10	25	100	10	1	2.5	4.0	10.0	1.18x
access-binary-trees	0	0	0	5	0	-	-	-	0.93x
access-fannkuch	10	34	57	24	0	3.4	1.7	5.7	2.20x
access-nbody	8	16	18	5	0	2.0	1.1	2.3	4.19x
access-nsieve	3	6	8	3	0	2.0	1.3	2.7	3.05x
bitops-3bit-bits-in-byte	2	2	2	0	0	1.0	1.0	1.0	25.47x
bitops-bits-in-byte	3	3	4	1	0	1.0	1.3	1.3	8.67x
bitops-bitwise-and	1	1	1	0	0	1.0	1.0	1.0	25.20x
bitops-nsieve-bits	3	3	5	0	0	1.0	1.7	1.7	2.75x
controlflow-recursive	0	0	0	1	0	-	-	-	0.98x
crypto-aes	50	72	78	19	0	1.4	1.1	1.6	1.64x
crypto-md5	4	4	5	0	0	1.0	1.3	1.3	2.30x
crypto-sha1	5	5	10	0	0	1.0	2.0	2.0	5.95x
date-format-tofte	3	3	4	7	0	1.0	1.3	1.3	1.07x
date-format-xparb	3	3	11	3	0	1.0	3.7	3.7	0.98x
math-cordic	2	4	5	1	0	2.0	1.3	2.5	4.92x
math-partial-sums	2	4	4	1	0	2.0	1.0	2.0	5.90x
math-spectral-norm	15	20	20	0	0	1.3	1.0	1.3	7.12x
regexp-dna	2	2	2	0	0	1.0	1.0	1.0	4.21x
string-base64	3	5	7	0	0	1.7	1.4	2.3	2.53x
string-fasta	5	11	15	6	0	2.2	1.4	3.0	1.49x
string-tagcloud	3	6	6	5	0	2.0	1.0	2.0	1.09x
string-unpack-code	4	4	37	0	0	1.0	9.3	9.3	1.20x
string-validate-input	6	10	13	1	0	1.7	1.3	2.2	1.86x

Figure 13. Detailed trace recording statistics for the SunSpider benchmark set.

[Figure copied from presentation paper]

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 VMs have an overall performance speedup of 3.0
- TraceMonkey recording & compilation 200 times slower than interpreter speed.

When recording aborts

- Type unstable loop.
- Trace length exceeds a threashold.
- Outer loop recording finds an uncompiled inner trace.

Types in C Vs JS

C Code

```
struct A {
    int x;
    int y;
};

A o;
o.x = 1;
o.y =  2;
```

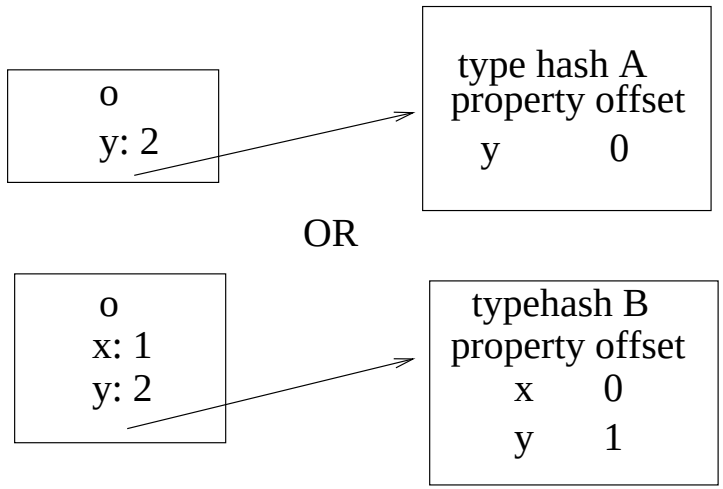
Asembly Code

```
store &o[0], 1

    store &o[1], 2
```

JS Code

```
var o;
if(...) {
    o.x = 1;
}
o.y = 2;
```



NanoJIT

- Responsible for compiling a recorded trace into machine code.
- Input: The recorded LIR SSA instructions.

Modes of TraceMonkey: Monitoring

- Interpreting bytecode.
- Every time SpiderMonkey interprets a backward-jump bytecode, the monitor makes note of the number of times the jump-target program-counter (PC) value has been jumped-to.
- If particular PC reaches a threshold value, the target is considered hot.
- When the monitor decides a target PC is hot, it looks compiled trace for that PC. If it finds, it transitions to executing mode. Otherwise it transitions to recording mode.

Modes of TraceMonkey: Recording

- Interpreting bytecode but with a difference (Recording LIR)
- All transitions out of recording mode eventually involve returning to monitoring mode.
- The recording may abort because:
  - If the recorder is asked to record a bytecode that it cannot, for various low-level reasons. The monitor also keeps track of how many times it has attempted to record a trace starting at each PC value. Helps in blacking recording of a loop if a particular PC causes too many aborted recordings.
  - If the recorder completes recording at a backward branch back to the initial PC value that triggered recording mode, it is said to have successfully closed the loop. A closed loop is passed through the NanoJIT and thereby compiled to native machine code.

Modes of TraceMonkey: Executing

- When the interpreter hits a loop back edge, it checks for a compiled trace in trace cache with computed type map and current PC.
- First the monitor allocates a native stack for the trace to use.
- It then imports the set of jsval values (local and global variables) from the SpiderMonkey interpreter that the trace is known to read or write during its execution. This set was determined during recording, and the imported values are stored locally within the native stack during execution.
- The monitor then calls into the native code of the trace as a normal C function.
- The native code returns a pointer to a structure called a side exit.
- The monitor inspects the side exit and, depending on the exit condition, reconstructs a certain number of interpreter frames and flushes a certain set of writes back. Frame reconstruction is necessary because a trace may inline a number of function calls, and may exit before those function calls return. Write flushing is necessary because all writes to imported values in the native stack are deferred until trace-exit, as an optimization.
- If the side exit condition indicates that the trace exited successfully – simply running out of native code or hitting a branch condition that no native code yet exists to handle – the monitor will inspect the hit count for the trace-exiting PC. If that PC has itself grown hot, the monitor will immediately transition to

Calling External Functions

- A trace will update the interpreter state while it exits. Now if the called function reenters the interpreter and try to use some interpreter data values, they will all be stale.

Modified Blacklisting with Nesting

- The outer loop blacklisted very quickly.
- Solution : Increment blacklist counter on such aborts, and back-off on compiling it, but decrement it when the inner loop successfully finishes a trace and undo the back-off.

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)