Deterministic performance tests

quasilyte @ GDG Nizhny Novgorod
Go meetup, 2021

Measurable performance attributes

Measurable performance attributes

- Execution time

Measurable performance attributes

- Execution time
- Memory consumption

Measurable performance attributes

- Execution time
- Memory consumption
- Heap allocations / GC pressure

Non-deterministic ways

Non-deterministic ways

- Benchmark tests

Non-deterministic ways

- Benchmark tests
- Profiling (of any kind)

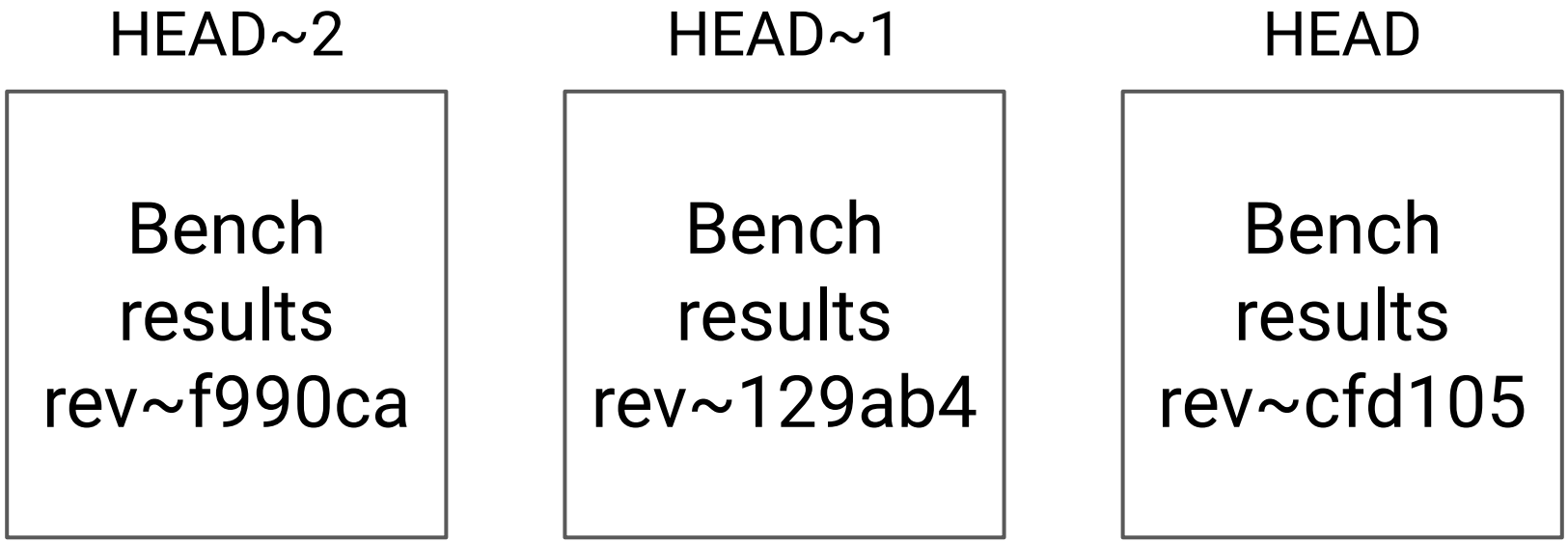
Non-deterministic ways

- Benchmark tests
- Profiling (of any kind)
- Metrics, logs, traces (allocs, rps, etc.)

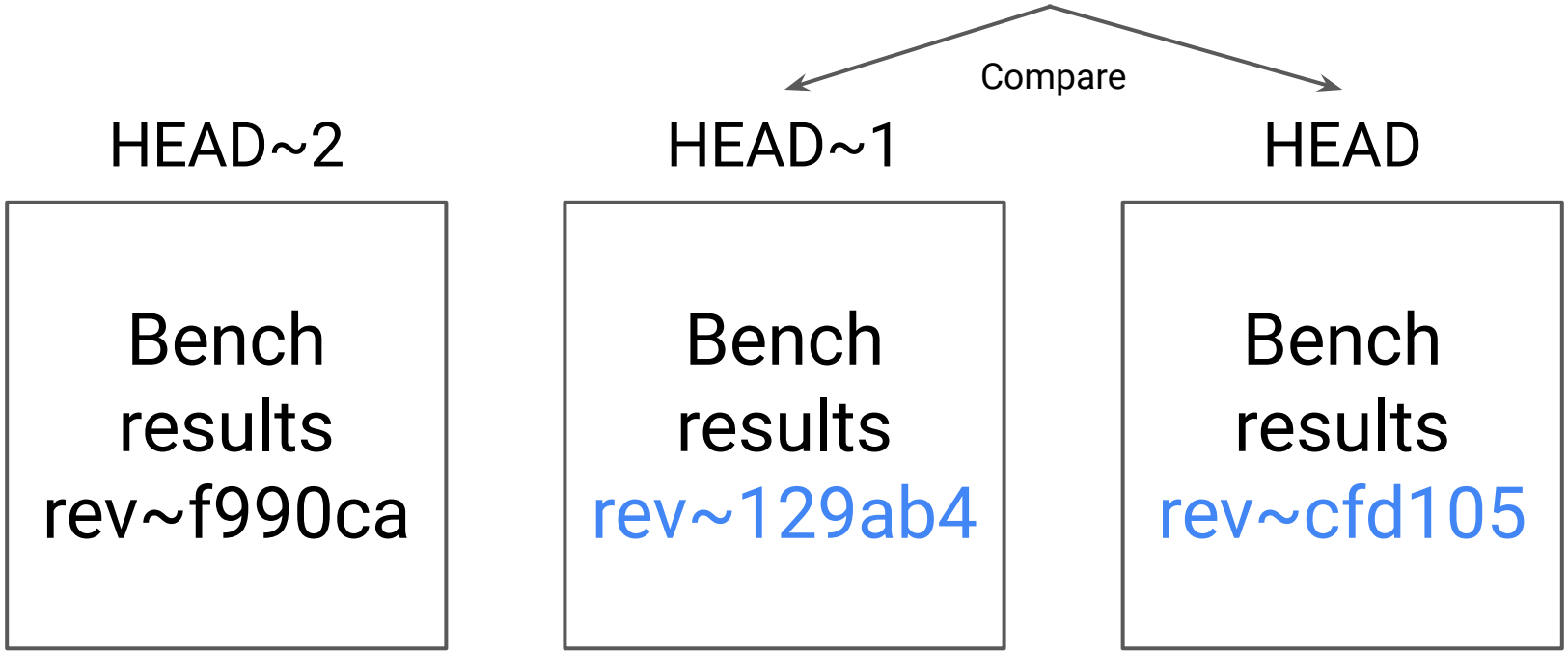
A conventional work-around

1. Run benchmarks on a dedicated machine
2. Save benchmark results from various revisions
3. Compare new rev result with older results

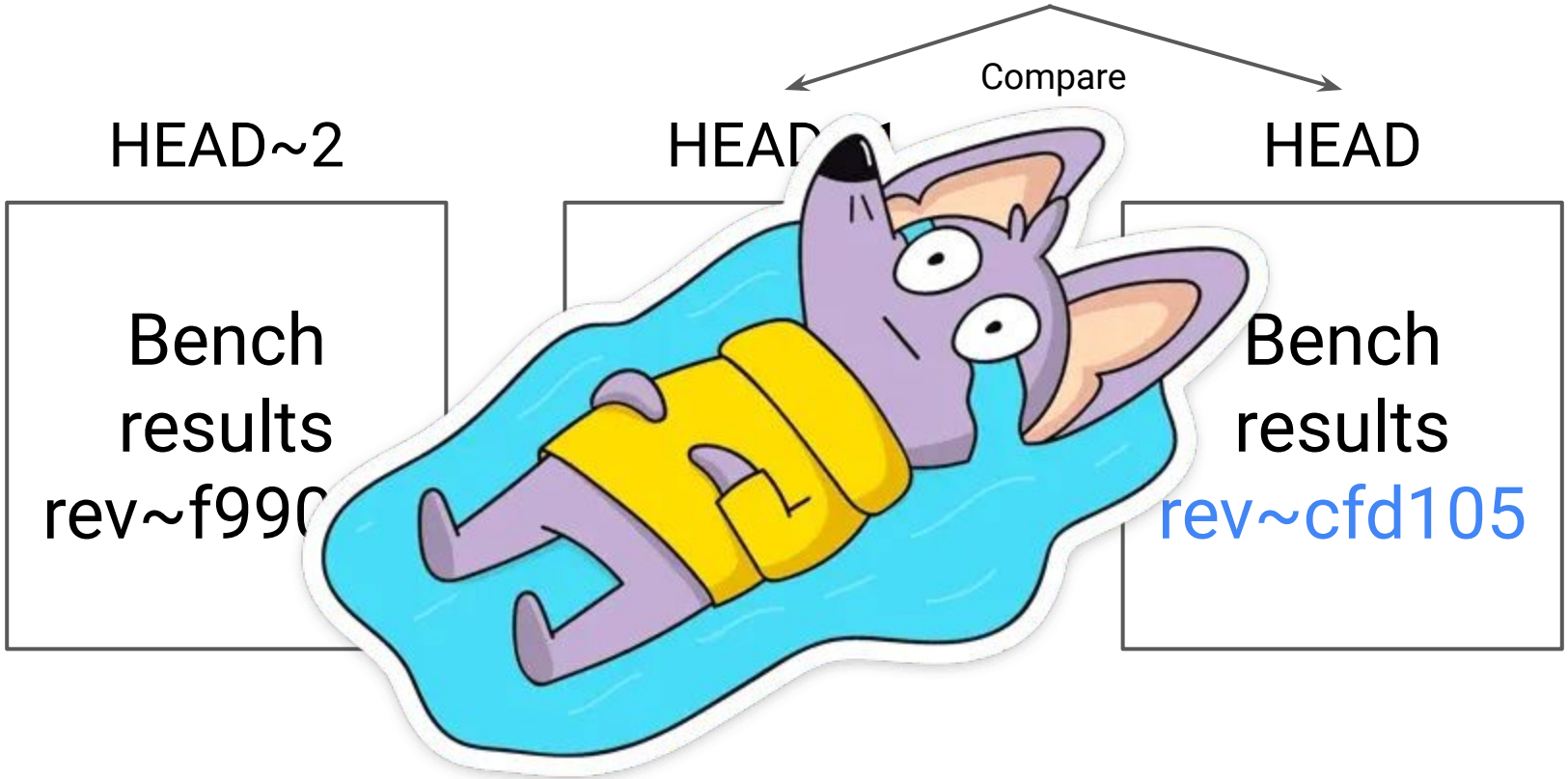
The “test” is failed if the new code gave worse results than we recorded before.



Comparing benchmark results



Comparing benchmark results



Comparing benchmark results

What is a “deterministic perf test”?

Defining a deterministic performance test

Defining a deterministic performance test

- Results are stable

Defining a deterministic performance test

- Results are stable
- Results are reproducible on different machines

Defining a deterministic performance test

- Results are stable
- Results are reproducible on different machines

But, more importantly…

Defining a deterministic performance test

- Results are stable
- Results are reproducible on different machines

But, more importantly…

- They **fail** if their invariant is broken

What we can measure reliably

- Number of the computational steps
- Heap allocations
- Whether Go compiler applied some optimization

What we can measure reliably

- Number of the computational steps
- Heap allocations
- Whether Go compiler applied some optimization

Safe zone

What we can measure reliably

- Number of the computational steps
- Heap allocations
- Whether Go compiler applied some optimization

Gray zone!

Test may be more fragile than you want it to be

What we can measure reliably

- Number of the computational steps
- Heap allocations
- Whether Go compiler applied some optimization

Danger zone!
Hard Go toolchain version dependency

The universal solution

The universal solution

Doesn’t exist

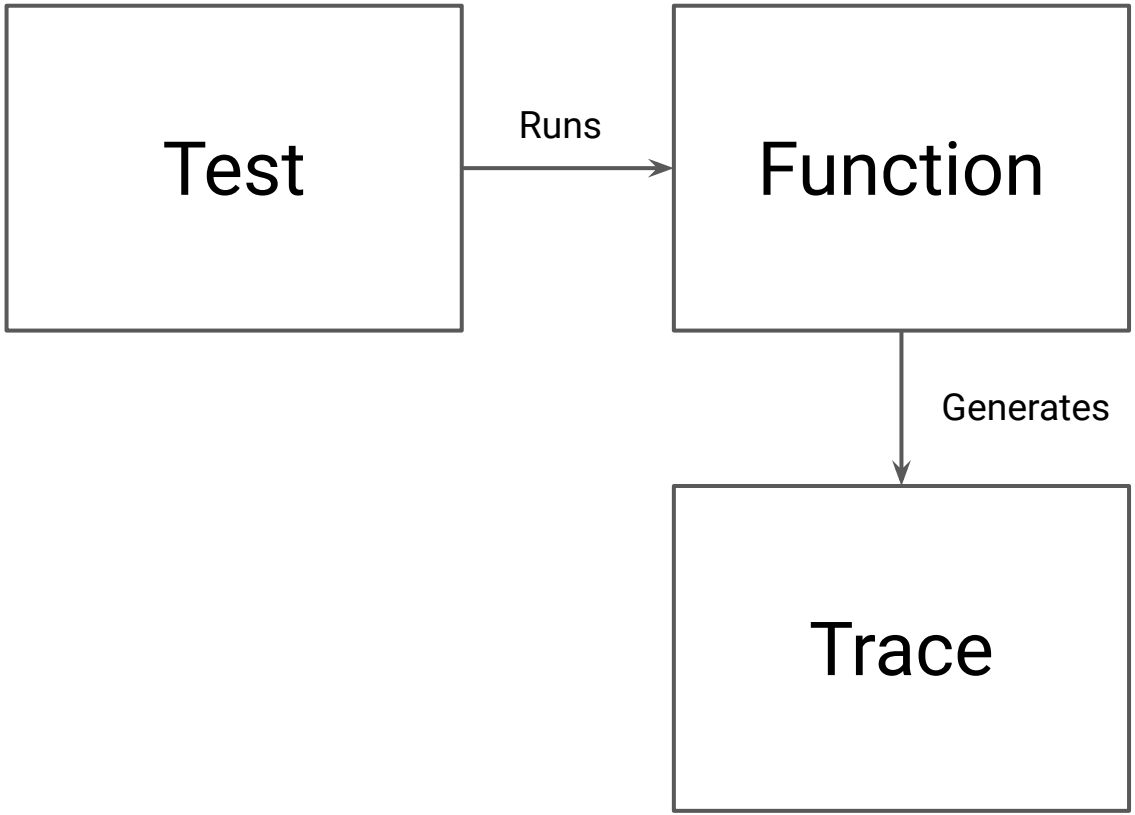
The universal solution

Doesn’t exist

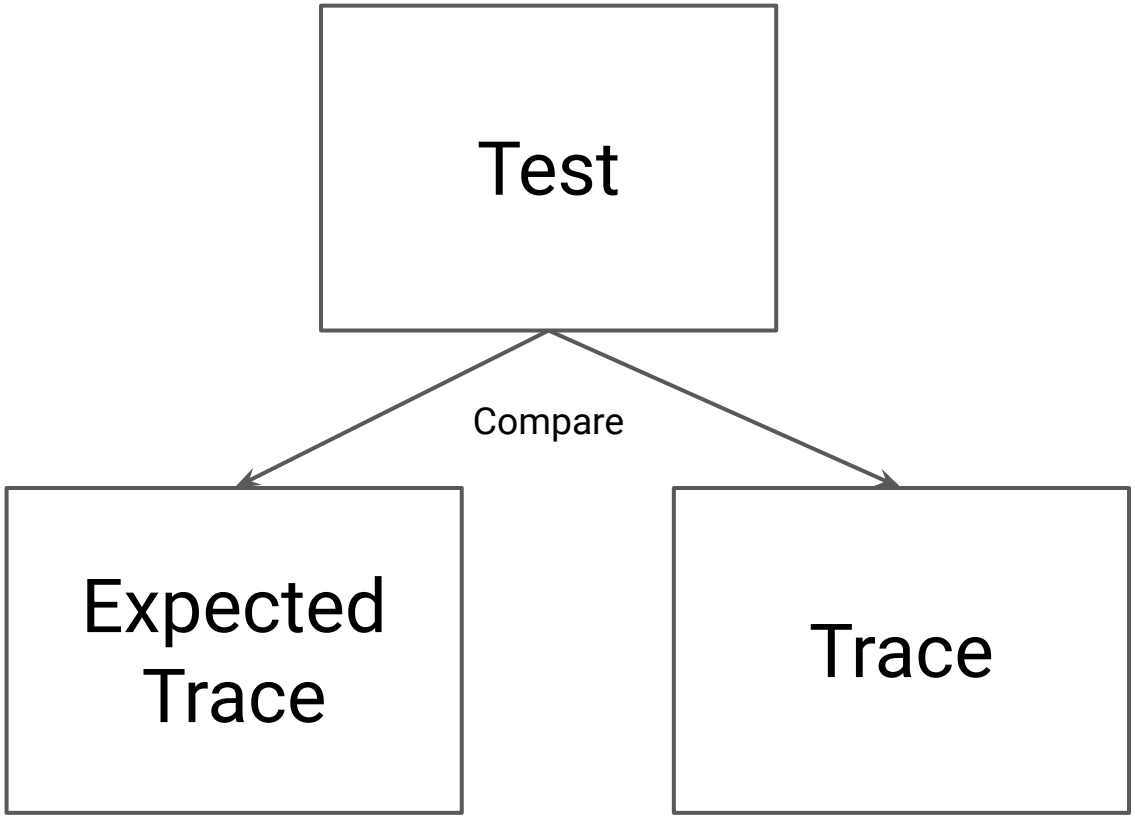
But I know some magic tricks

Magic trick #1

Execution trace tests



Execution trace tests



Execution trace tests

Execution trace tests

```
// When outer matching fails,
// no other matching should be performed.
{
    pattern: `if (0) $_`,
    input:   `if (1) { if (2) { if (3) {}}}`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.Lnumber y=*ir.Lnumber",
    },
}
```

Execution trace tests

```
// When outer matching fails,
// no other matching should be performed.
{
    pattern: `if (0) $_`,
    input:   `if (1) { if (2) { if (3) {}}}`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.Lnumber y=*ir.Lnumber",
    },
}
```

Execution trace tests

```
// When outer matching fails,
// no other matching should be performed.
{
    pattern: `if (0) $_`,
    input:   `if (1) { if (2) { if (3) {}}}`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.Lnumber y=*ir.Lnumber",
    },
}
```

Execution trace tests

```
// When $_ is used for statement body it should quickly
// match the body without checking the actual contents.
{
    pattern: `if ($cond) $_`,
    input:   `if ($x && g()) { echo 1, 2, 3; }`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.SimpleVar y=*ir.BooleanAndExpr",
        " • eqNode x=*ir.ExpressionStmt y=*ir.StmtList",
    },
},
```

Execution trace tests

```
// When $_ is used for statement body it should quickly
// match the body without checking the actual contents.
{
    pattern: `if ($cond) $_`,
    input:   `if ($x && g()) { echo 1, 2, 3; }`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.SimpleVar y=*ir.BooleanAndExpr",
        " • eqNode x=*ir.ExpressionStmt y=*ir.StmtList",
    },
},
```

Execution trace tests

```
// When $_ is used for statement body it should quickly
// match the body without checking the actual contents.
{
    pattern: `if ($cond) $_`,
    input:   `if ($x && g()) { echo 1, 2, 3; }`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.SimpleVar y=*ir.BooleanAndExpr",
        " • eqNode x=*ir.ExpressionStmt y=*ir.StmtList",
    },
},
```

Execution trace tests

```
// When $_ is used for statement body it should quickly
// match the body without checking the actual contents.
{
    pattern: `if ($cond) $_`,
    input:   `if ($x && g()) { echo 1, 2, 3; }`,
    trace: []string{
        "eqNode x=*ir.IfStmt y=*ir.IfStmt",
        " • eqNode x=*ir.SimpleVar y=*ir.BooleanAndExpr",
        " • eqNode x=*ir.ExpressionStmt y=*ir.StmtList",
    },
},
```

Execution trace tests

```
// Match $_ once, then accept match as ${"*"} is the last node.
{
    pattern: `f($_, ${"*"})`,
    input:   `f(1, 2, 3, 4, g())`,
    trace: []string{
        "eqNode x=*ir.FunctionCallExpr y=*ir.FunctionCallExpr",
        " • eqNode x=*ir.Argument y=*ir.Argument",
        " •  • eqNode x=*ir.SimpleVar y=*ir.Lnumber",
    },
},
```

Execution trace tests

```
// Match $_ once, then accept match as ${"*"} is the last node.
{
    pattern: `f($_, ${"*"})`,
    input:   `f(1, 2, 3, 4, g())`,
    trace: []string{
        "eqNode x=*ir.FunctionCallExpr y=*ir.FunctionCallExpr",
        " • eqNode x=*ir.Argument y=*ir.Argument",
        " •  • eqNode x=*ir.SimpleVar y=*ir.Lnumber",
    },
},
```

Execution trace tests

```
// Match $_ once, then accept match as ${"*"} is the last node.
{
    pattern: `f($_, ${"*"})`,
    input:   `f(1, 2, 3, 4, g())`,
    trace: []string{
        "eqNode x=*ir.FunctionCallExpr y=*ir.FunctionCallExpr",
        " • eqNode x=*ir.Argument y=*ir.Argument",
        " •  • eqNode x=*ir.SimpleVar y=*ir.Lnumber",
    },
},
```

Execution trace tests

```
// Match $_ once, then accept match as ${"*"} is the last node.
{
    pattern: `f($_, ${"*"})`,
    input:   `f(1, 2, 3, 4, g())`,
    trace: []string{
        "eqNode x=*ir.FunctionCallExpr y=*ir.FunctionCallExpr",
        " • eqNode x=*ir.Argument y=*ir.Argument",
        " •  • eqNode x=*ir.SimpleVar y=*ir.Lnumber",
    },
},
```

Making the traces “free”

1. Collect traces under if with const condition
2. Set const value via the compilation tag

Trace collector struct can be empty for release builds and a slice of records for test runs.

tracing_enabled.go

```
// +build tracing

package phpgrep

const tracingEnabled = true
```

tracing_enabled.go

```
// +build tracing

package phpgrep

const tracingEnabled = true
```

Or with a newer “go:build” syntax

tracing_disabled.go

```
// +build !tracing

package phpgrep

const tracingEnabled = false
```

Matcher struct

```
type matcher struct {
    // . . . other fields

    // Used only when -tracing build tag is specified.
    tracingBuf   *bytes.Buffer
    tracingDepth int
}
```

Writing the trace

```
if tracingEnabled {
    pad := strings.Repeat(" · ", m.tracingDepth)
    fmt.Fprintf(m.tracingBuf, "%seqNode x=%T y=%T\n", pad, x, y)
    m.tracingDepth++
    defer func() {
        m.tracingDepth--
    }()
}
```

Writing the trace

```
if tracingEnabled {
    pad := strings.Repeat(" · ", m.tracingDepth)
    fmt.Fprintf(m.tracingBuf, "%seqNode x=%T y=%T\n", pad, x, y)
    m.tracingDepth++
    defer func() {
        m.tracingDepth--
    }()
}
```

Const condition:
If tracingEnabled=false, this branch is removed

Where applicable

Where applicable

- Any kinds of state machines

Where applicable

- Any kinds of state machines
- Data transformers

Magic trick #2

Codegen tests

Codegen kinds (simplified!)

Codegen kinds (simplified!)

- Ad-hoc (go:generate, codegen scripts)

Codegen kinds (simplified!)

- Ad-hoc (go:generate, codegen scripts)
- Compilers-like software

Why testing the codegen?

Why testing the codegen?

- Avoid unexpected perf regressions

Why testing the codegen?

- Avoid unexpected perf regressions
- Keep codegen optimization promises

Why testing the codegen?

- Avoid unexpected perf regressions
- Keep codegen optimization promises
- You’ll probably need these tests anyway

Codegen test: compilers

```
{
    `if b { return 1 }; return 0`: {
        `  PushParam 2 # b`,
        `  JumpFalse 6 # L0`,
        `  PushIntConst 0 # value=1`,
        `  ReturnIntTop`,
        `L0:`,
        `  PushIntConst 1 # value=0`,
        `  ReturnIntTop`,
    },
    // ...
}
```

Codegen test: compilers

```
{
    `if b { return 1 }; return 0`: {
        `  PushParam 2 # b`,
        `  JumpFalse 6 # L0`,
        `  PushIntConst 0 # value=1`,
        `  ReturnIntTop`,
        `L0:`,
        `  PushIntConst 1 # value=0`,
        `  ReturnIntTop`,
    },
    // ...
}
```

Input (source)

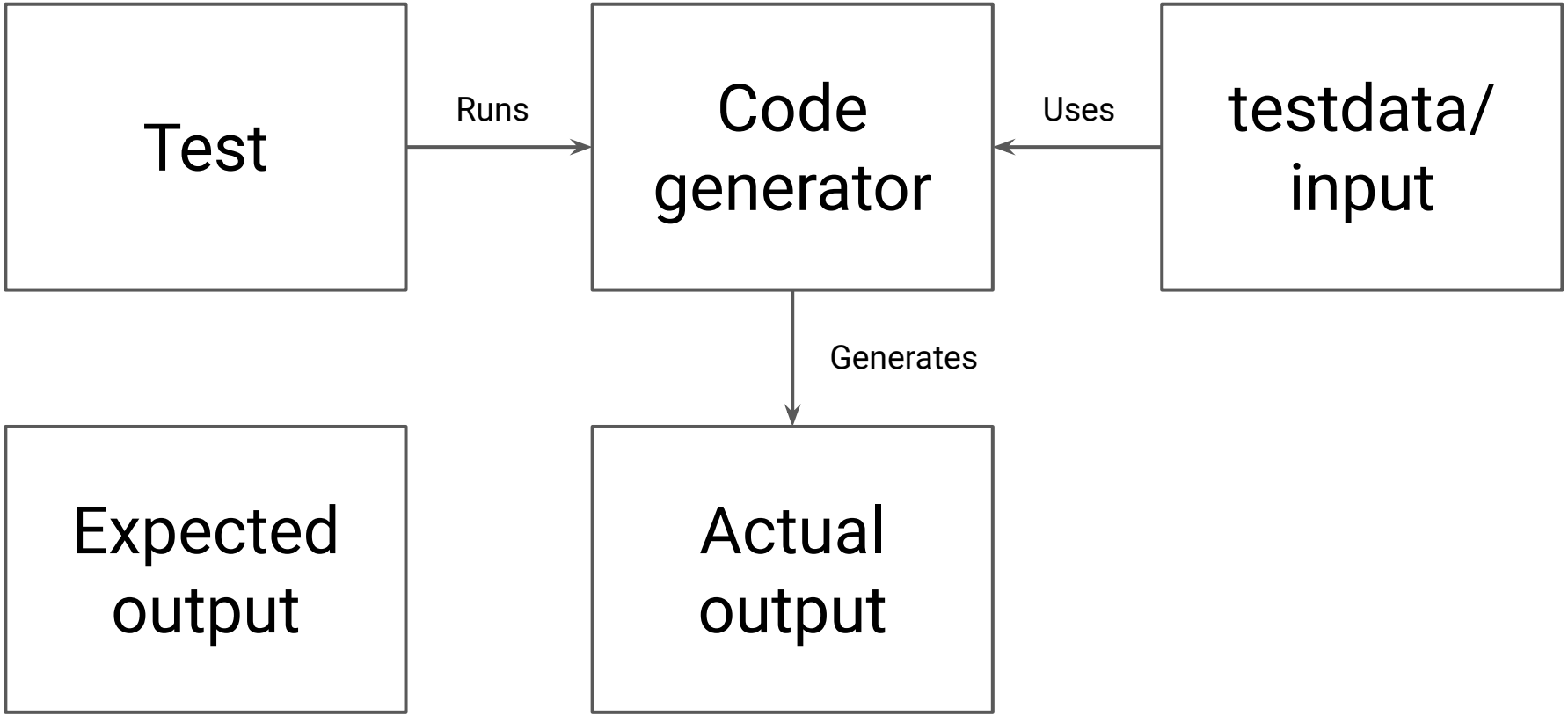
Codegen test: compilers

```
{
    `if b { return 1 }; return 0`: {
        `  PushParam 2 # b`,
        `  JumpFalse 6 # L0`,
        `  PushIntConst 0 # value=1`,
        `  ReturnIntTop`,
        `L0:`,
        `  PushIntConst 1 # value=0`,
        `  ReturnIntTop`,
    },
    // ...
}
```

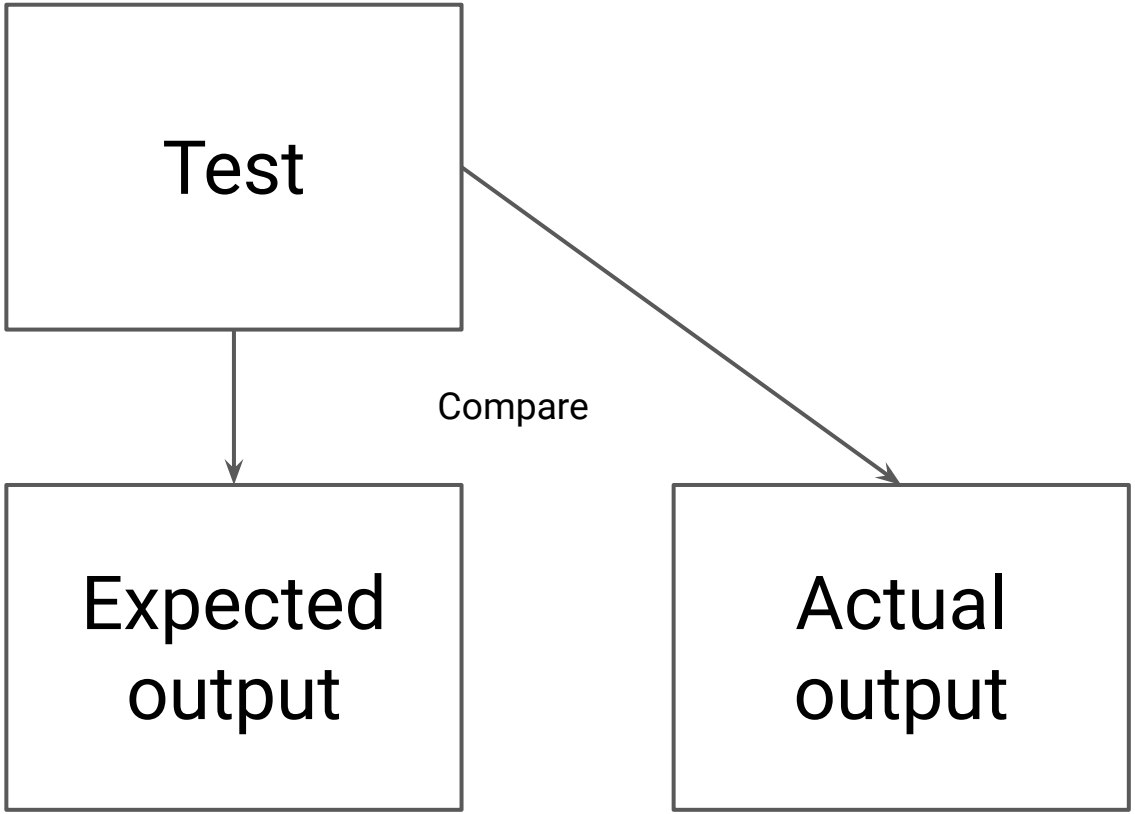
Output (generated code)

Codeget test: ad-hoc

1. Put input files to “testdata/”
2. Put “expected result” files somewhere close
3. Run codegen over a testdata folder
4. Compare actual and expected results



Ad-hoc codegen test



Ad-hoc codegen test

Where applicable

Where applicable

- ORM-like code generators

Where applicable

- ORM-like code generators
- Any auto-generated perf sensitive code

Where applicable

- ORM-like code generators
- Any auto-generated perf sensitive code
- Compilers, interpreters, optimizers

Magic trick #3

Zero alloc tests

Zero alloc tests

1. Collect mem stats (“before”)
2. Run the code to be tested
3. Collect mem stats again (“after”)
4. Compare “before” and “after” allocs

Zero alloc test

```
var before, after runtime.MemStats
runtime.GC()

runtime.ReadMemStats(&before)
funcToTest() // <- HERE
runtime.ReadMemStats(&after)

allocated := after.Alloc - before.Alloc
if allocated != 0 {
   t.Error(...)
}
```

Problem

This method can be flaky*

(*) fails from time to time

Solution

Run the test several times

If it f() allocates, it always allocates

Zero alloc test (improved)

```
const numTests = 5 // Five is good enough, 10 is even better
failures := 0

for i := 0; i < numTests; i++ {
    // ... test body.

    if allocated != 0 {
       failures++
    }
}

if failures == numTests {
    t.Error(...)
}
```

Where applicable

Where applicable

- Low-level functions

Where applicable

- Low-level functions
- High-performance primitives

Where applicable

- Low-level functions
- High-performance primitives
- Latency-sensitive code paths

Magic trick #4

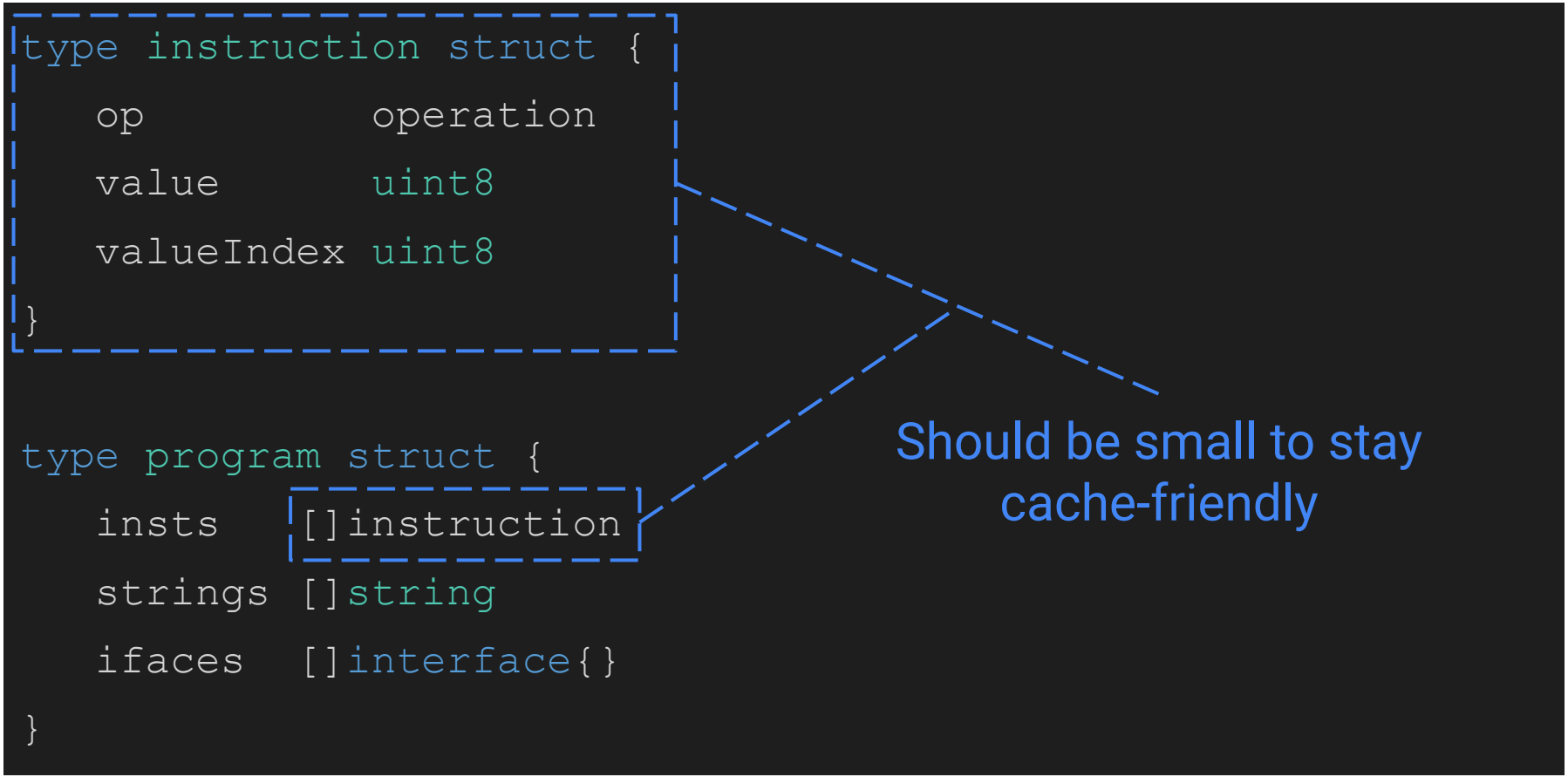
Sizeof tests

ruleguard AST matching engine

```
type instruction struct {
    op         operation
    value      uint8
    valueIndex uint8
}

type program struct {
    insts   []instruction
    strings []string
    ifaces  []interface{}
}
```

ruleguard AST matching engine



Sizeof test

```
wantSize := 3
haveSize := int(unsafe.Sizeof(instruction{}))
if wantSize != haveSize {
    t.Errorf("sizeof(instruction): have %d, want %d",
        haveSize, wantSize)
}
```

Problem

Sizeof is platform-dependent

Solution 1

Check for the platform and/or int size

Sizeof test: adjusting expected sizes

```
is64bit := bits.UintSize == 64
wantSize := 12
if is64bit {
    wantSize = 24
}
```

Solution 2

Skip platforms that you don’t care about

Sizeof test: skipping platforms

```
if bits.UintSize != 64 {
    t.Skip("not 64-bit platform")
}
```

Where applicable

Where applicable

- Core data points

Where applicable

- Core data points
- Layout-sensitive data

Magic trick #5

Go compiler optimizations test

Disclaimer

You probably don’t need to go this far

What we may want to test

- Is function f() inlineable?
- Does function f() contain a bound check?
- Do f() params/results escape to heap?

What we may want to test

- Is function f() inlineable?
- Does function f() contain a bound check?
- Do f() params/results escape to heap?

Go compiler can provide answers to all these questions if we'll just ask.

Algorithm

- Run Go compiler with “-m -m” flags
- Collect the output
- Parse output for the relevant info

Algorithm

- Run Go compiler with “-m -m” flags
- Collect the output
- Parse output for the relevant info

Note that the output format is not stable between the Go versions.

inltest library

A legacy library that probably doesn’t work anymore, but it can be used as a reference:

https://github.com/quasilyte/inltest

Where applicable

Where applicable

- Anywhere if you’re a crazy person

Don’t just run benchmarks

Don’t just run benchmarks

Write performance tests

Deterministic performance tests

quasilyte @ GDG Nizhny Novgorod
Go meetup, 2021