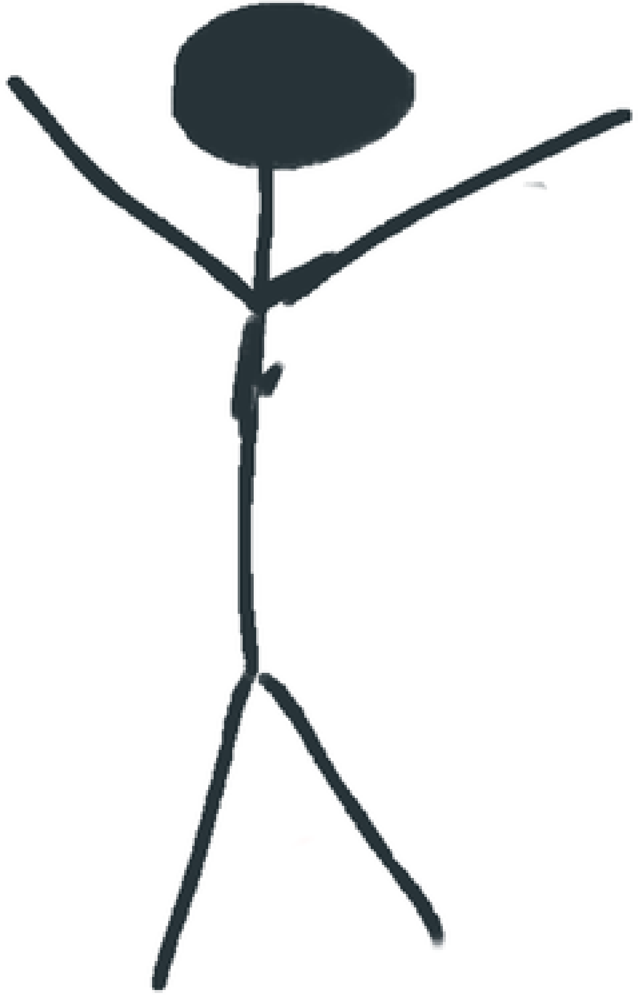
HOW WE DON'T DO THINGS

HOW WE DO THINGS

[BUT REALLY SHOULD NOT]

people WRiting VMs
in C++



people WRiting VMs
in RPython, Java

ME

Excelsior JET

V8

Dart VM

LuaJIT

Excelsior JET

[Java VM written in Oberon-2/Modula-2]

Excelsior JET

[Java VM written in Oberon-2/Modula-2]

[This days moved to Scala]

V8

[JavaScript VM written in C++]

V8

[JavaScript VM written in C++]

[Some of that C++ is assembly is disguise]

Dart VM

[Dart VM written in C++]

Dart VM

[Dart VM written in C++]

[Some of that C++ is assembly is disguise]

LuaJIT

[Lua VM written in C, Lua & assembly]

LuaJIT

[Lua VM written in C, Lua & assembly]

[has a tracing JIT]

LuaJIT

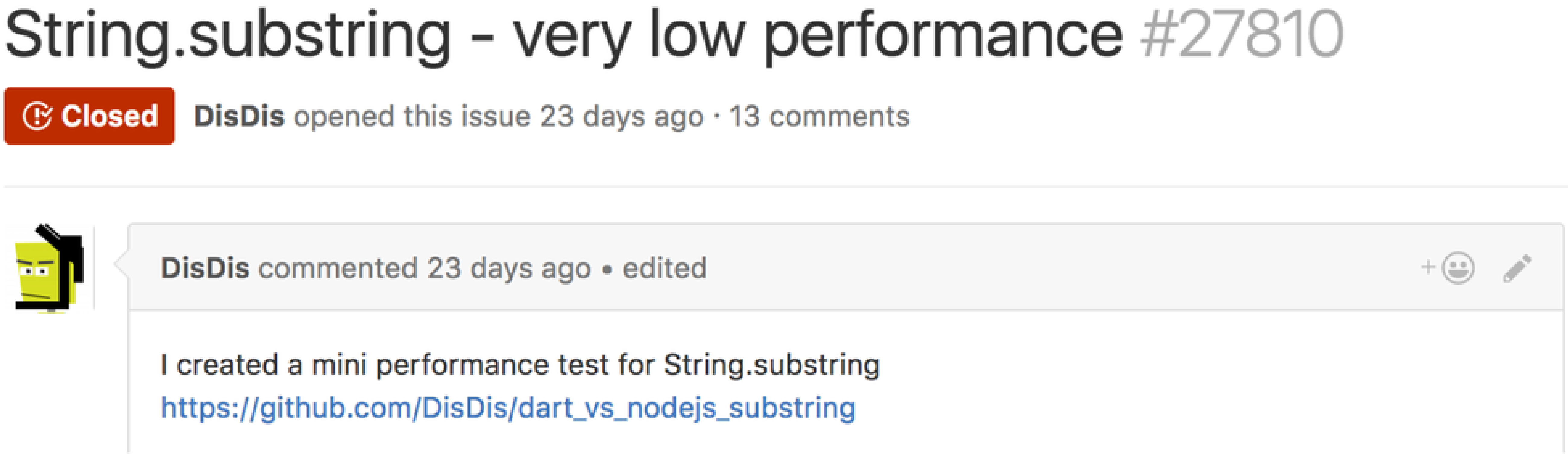
[Lua VM written in C, Lua & assembly]

[has a *tracing* JIT]

USERS

«Focus on the user and all else will follow.»

benchmarks are not our users



String.substring - very low performance #27810

Closed DisDis opened this issue 23 days ago · 13 comments

DisDis commented 23 days ago • edited

I created a mini performance test for String.substring
https://github.com/DisDis/dart_vs_nodejs_substring

```
benchmark(String s) {
  while (s.length > 1) {
    s = s.substring(1);
  }
}
```

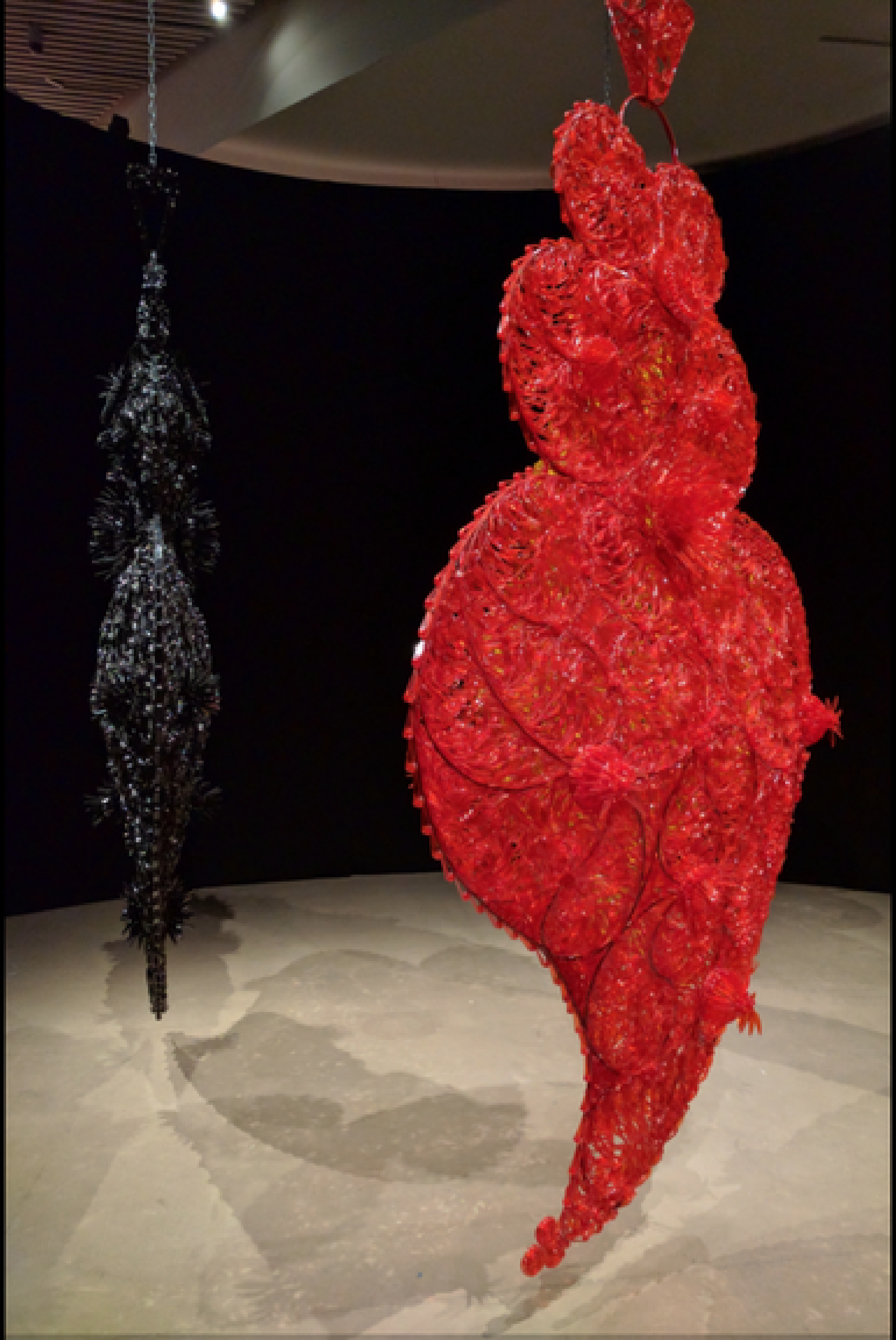
```
matchAt(String s, RegExp re, int index) =>
  re.firstMatch(s.substring(index));
```

```
matchAt(String s, RegExp re, int index) =>
  re.matchAsPrefix(s, index);
```

holistic

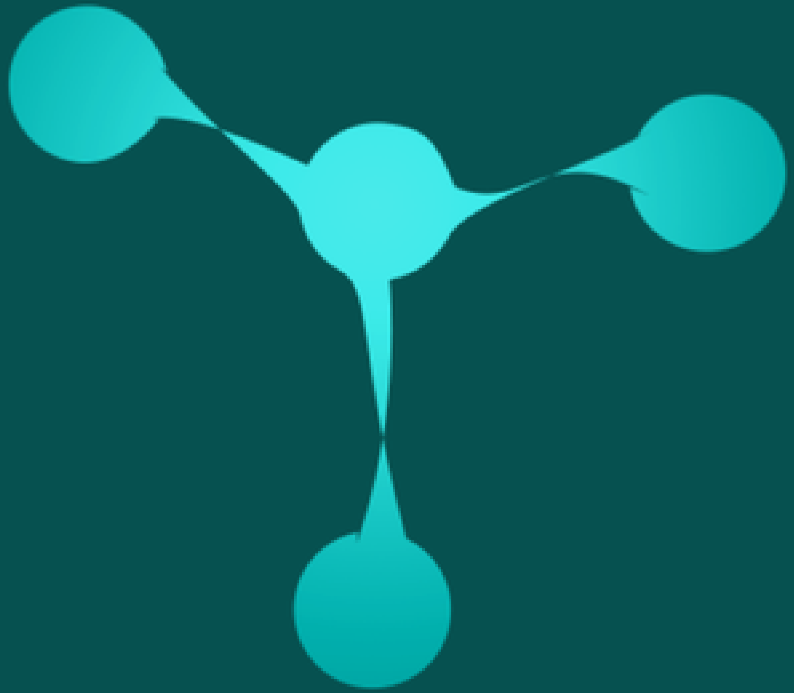
/hō'listik/

users are artists



Coração Independente Vermelho by Joana Vasconcelos

Coração Independente Vermelho by Joana Vasconcelos



torch

A SCIENTIFIC COMPUTING FRAMEWORK FOR LUAJIT

```
for j = 1, N do
  for i = 1, M do
    t[{i, j}] = 2 * i + j
  end
end
```

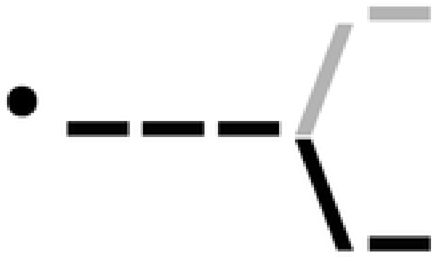
```
for j = 1, N do
  for i = 1, M do
    t[i][j] = 2 * i + j
  end
end
```

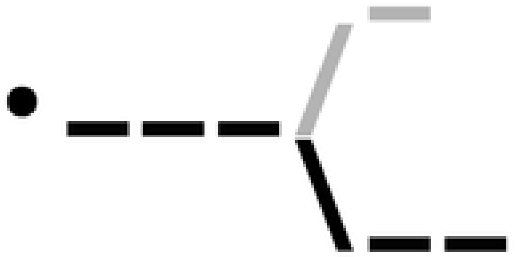
```lua
-- counterintuitively it is actually slower
-- than the t[{i, j}] code.
for j = 1, N do
  for i = 1, M do
    t[i][j] = 2 * i + j
  end
end
```

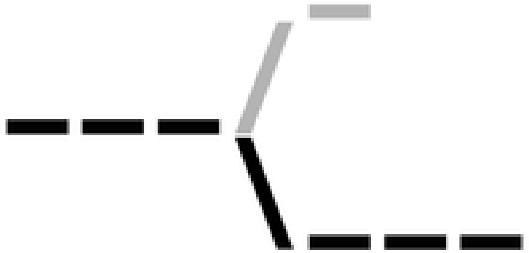
- _

- __

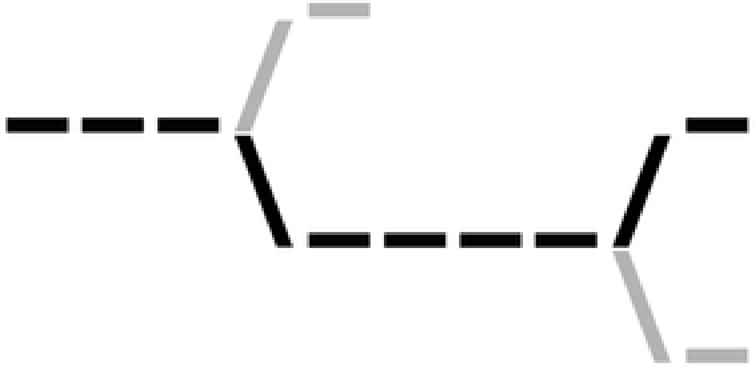
- _ _ _

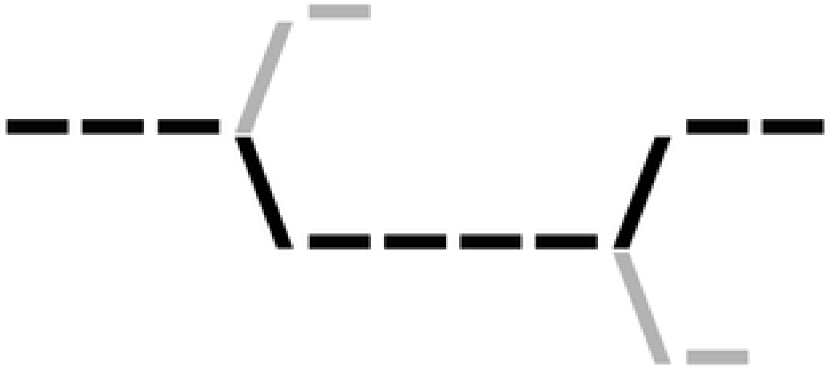


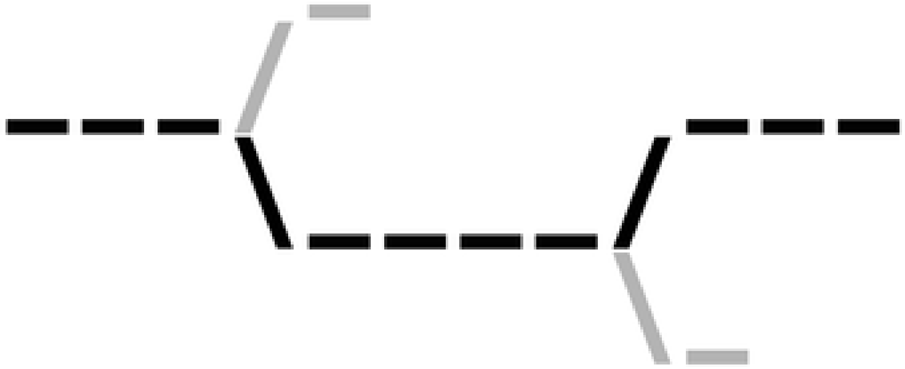


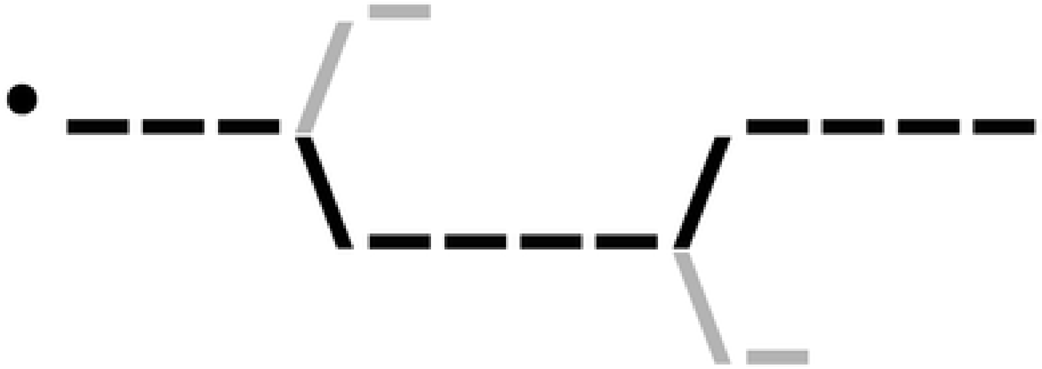
- 

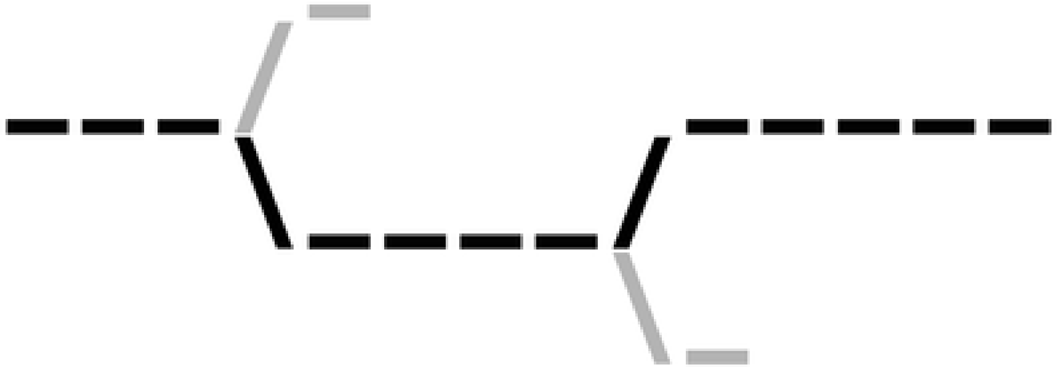
-

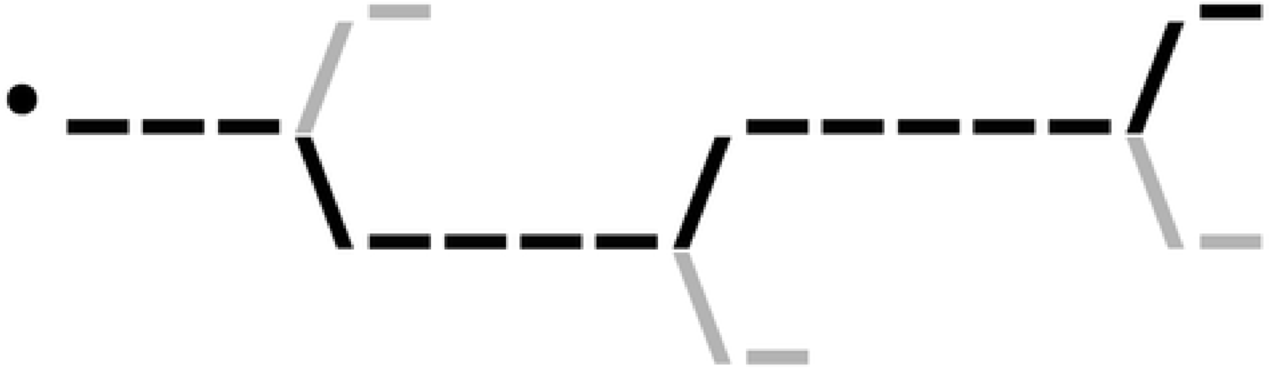
- 

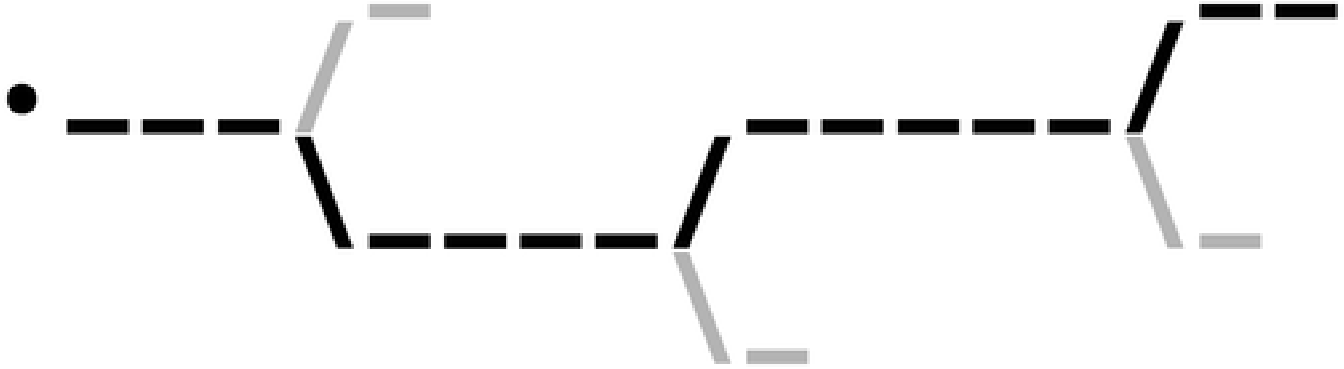
- 

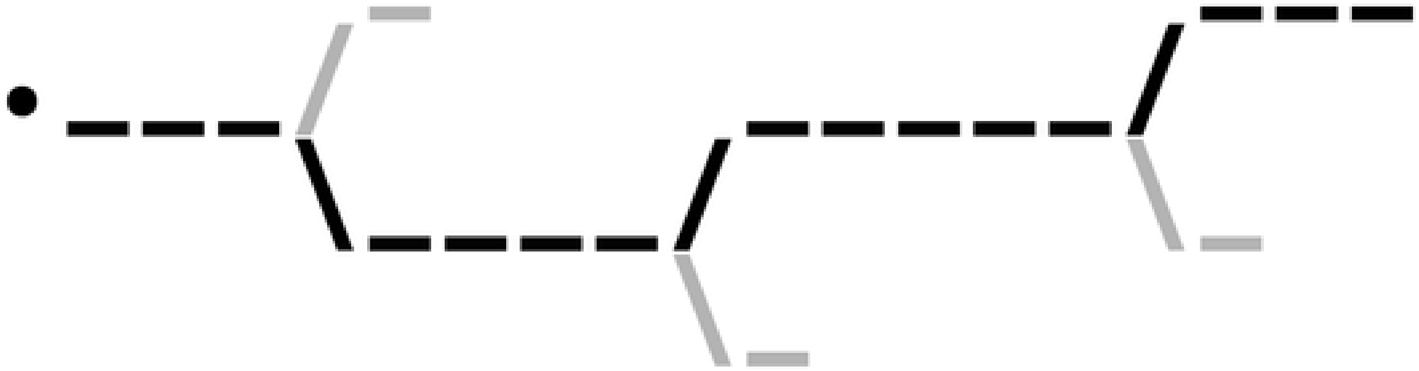
- 

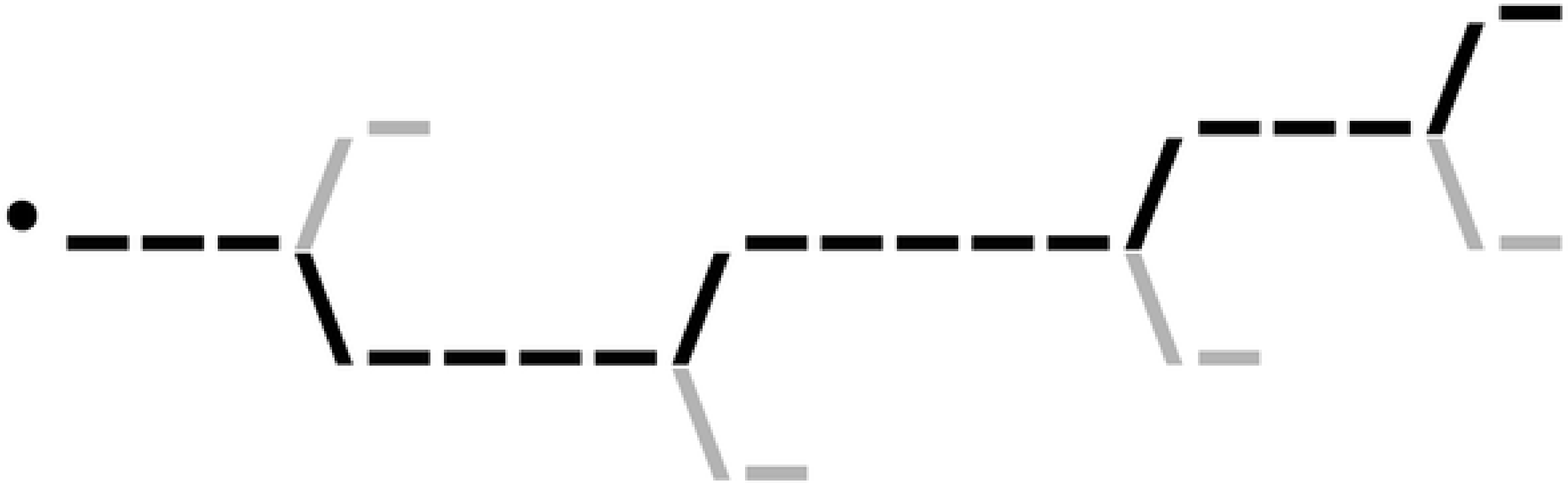


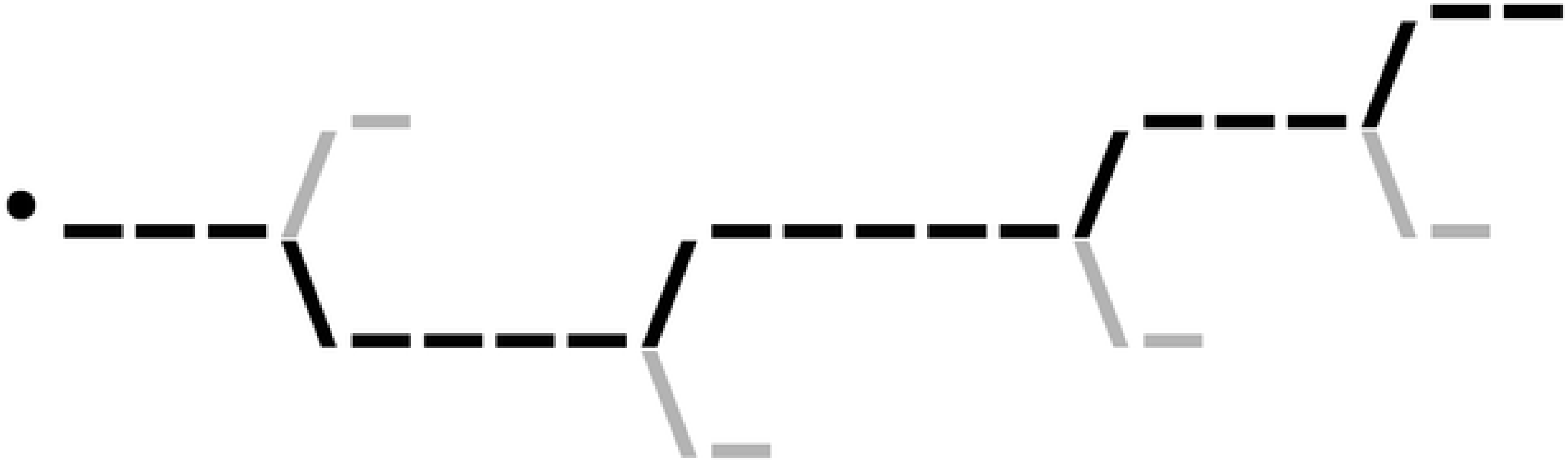
- 

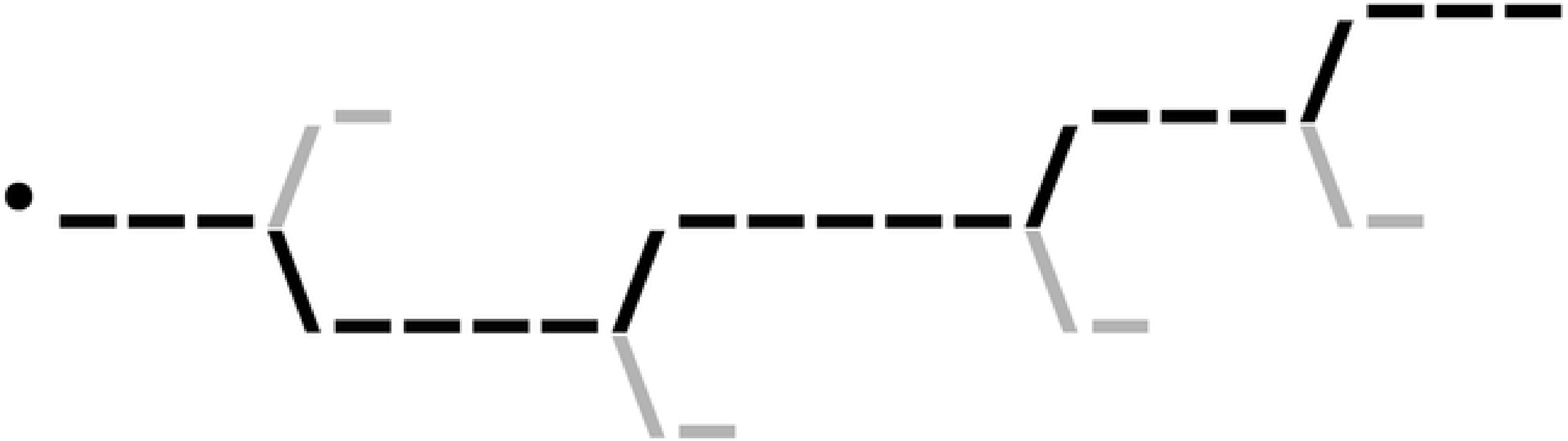


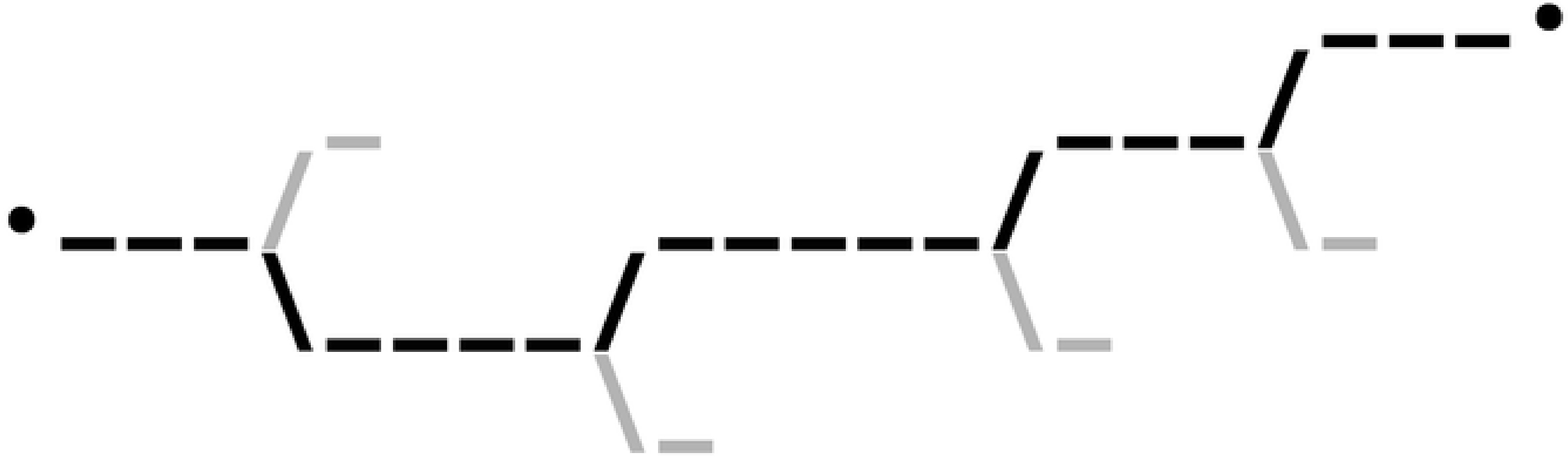












-

- _

- __

● ___ ___ ___ ☠

.___☠️ (trace abort)

build a trace visualizer

~~build a trace visualizer~~

~~build a trace visualizer~~

build a profiler!

deopt cargo cult

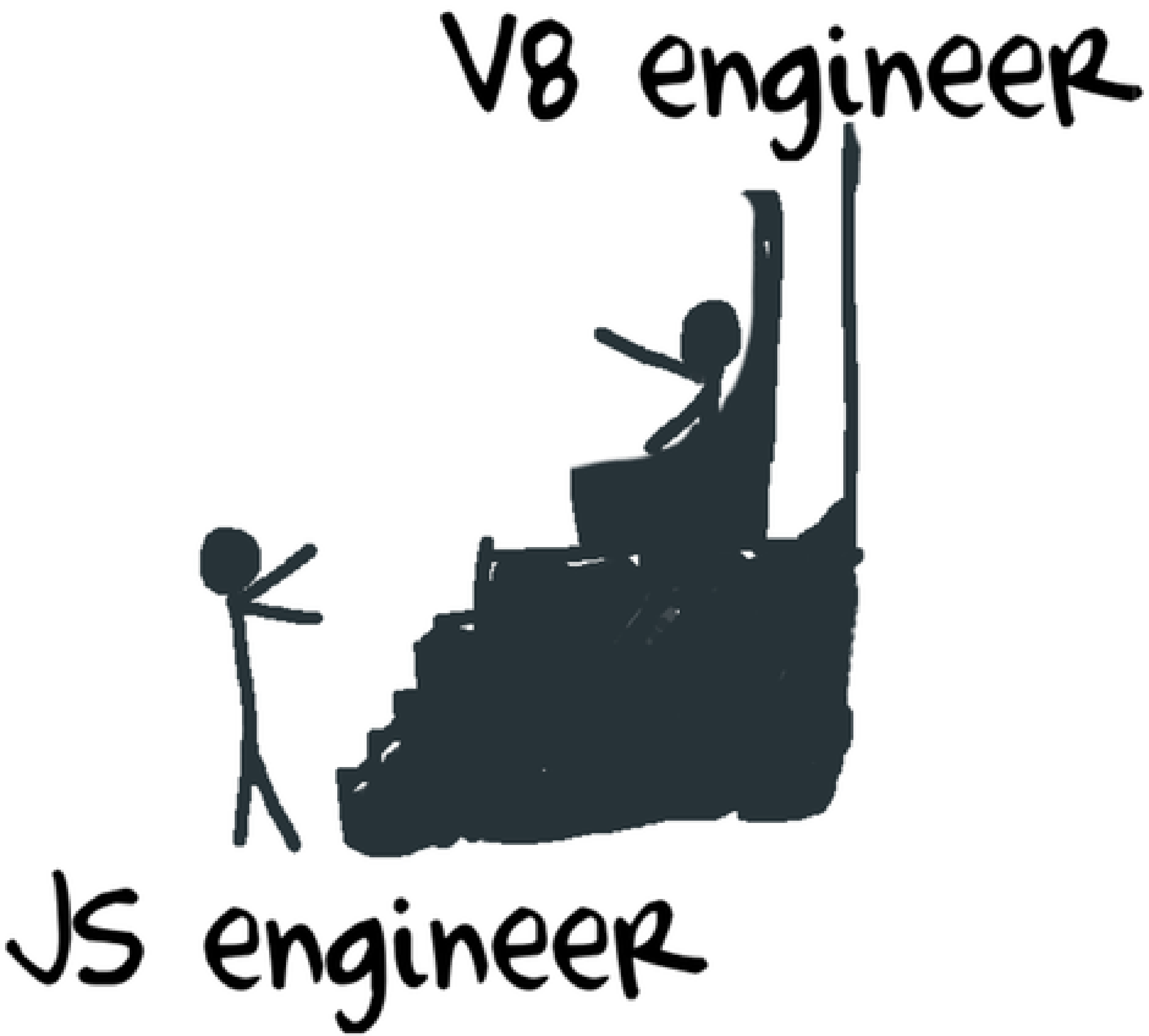
[deoptimizations are bad]

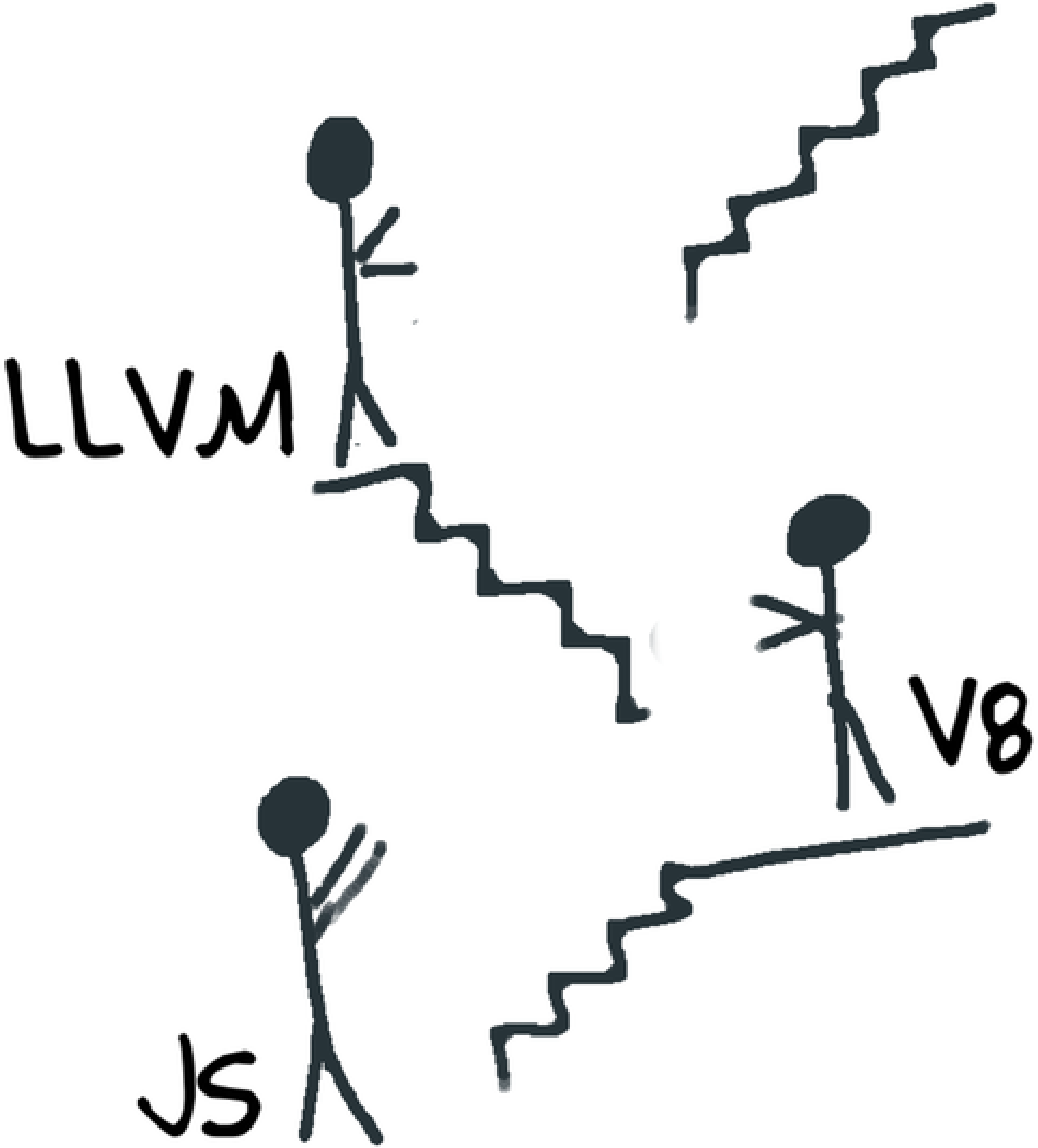
users don't know why fast is fast

users don't know what matters

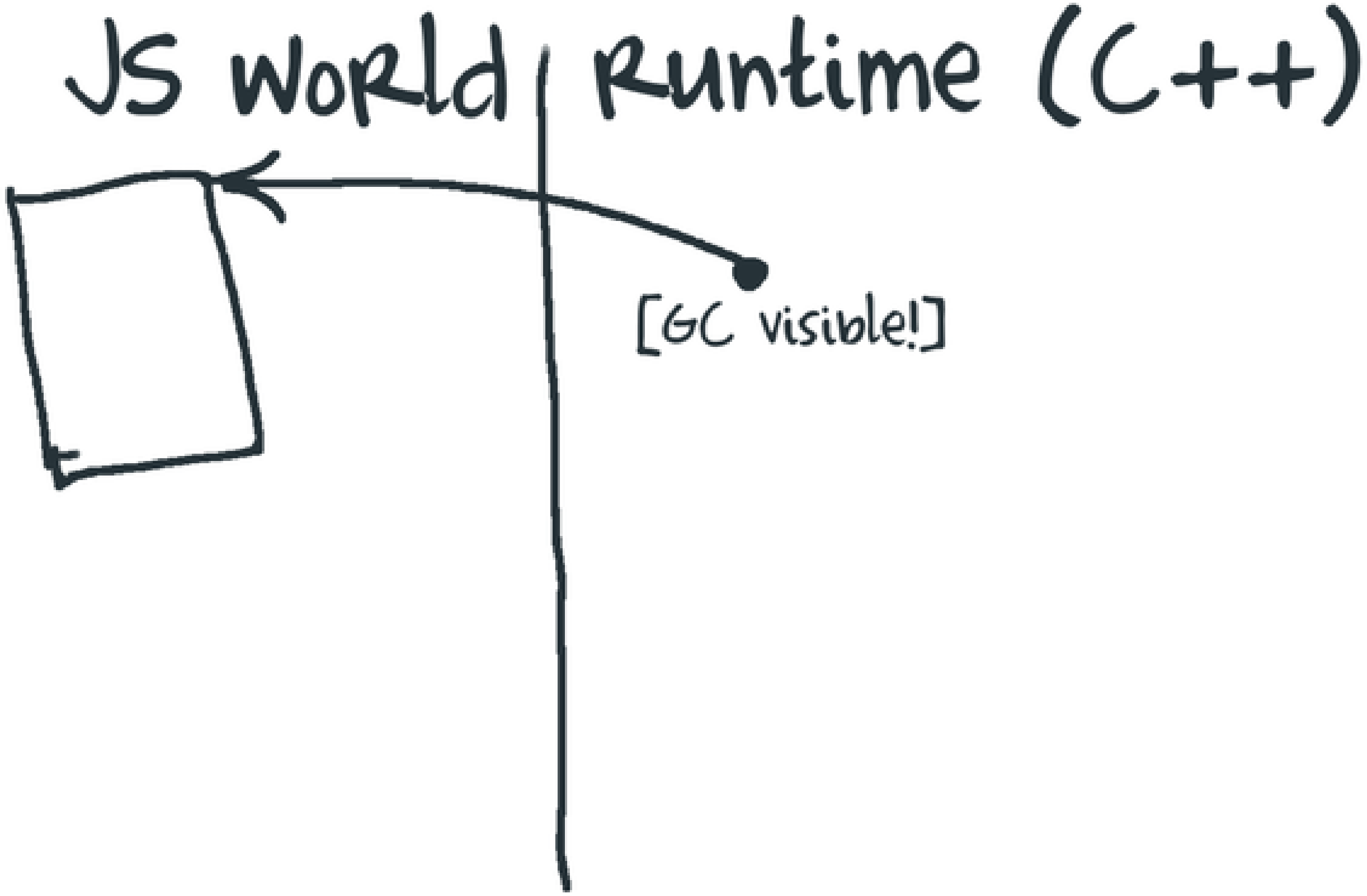
users are not our benchmarks

users need tools


V8 engineer
JS engineer


LLVM
V8
JS

TRAPPED BY C++


JS World
Runtime (C++)
[GC visible!]

```
class HeapNumber: public HeapObject {
 public:
  // [value]: number value.
  inline double value() const;
}
```

```
double HeapNumber::value() const {
  return READ_DOUBLE_FIELD(this, kValueOffset);
}
```

```
double HeapNumber::value() const {
  return ReadDoubleValue(
      FIELD_ADDR_CONST(this, kValueOffset));
}
```

```
double HeapNumber::value() const {
  return ReadDoubleValue(
      (reinterpret_cast(this)
       + kValueOffset
       - kHeapObjectTag));
}
```

```
double HeapNumber::value() const {
  return ReadUnalignedValue<double>(
      (reinterpret_cast<const byte*>](this)
       + kValueOffset
       - kHeapObjectTag));
}
```

```
double HeapNumber::value() const {
  const void* p =
    reinterpret_cast<const byte*>(this)
       + kValueOffset
       - kHeapObjectTag);
#if !(V8_TARGET_ARCH_MIPS ||
      V8_TARGET_ARCH_MIPS64 ||
      V8_TARGET_ARCH_ARM)
  return *reinterpret_cast<const double*>(p);
#else
  V r;
  memmove(&r, p, sizeof(V));
  return r;
#endif
}
```

(mis)represent objects as C++ objects

use `reinterpret_cast` for profit

What you think you've built

What you have actually built according to C++ UB Rules

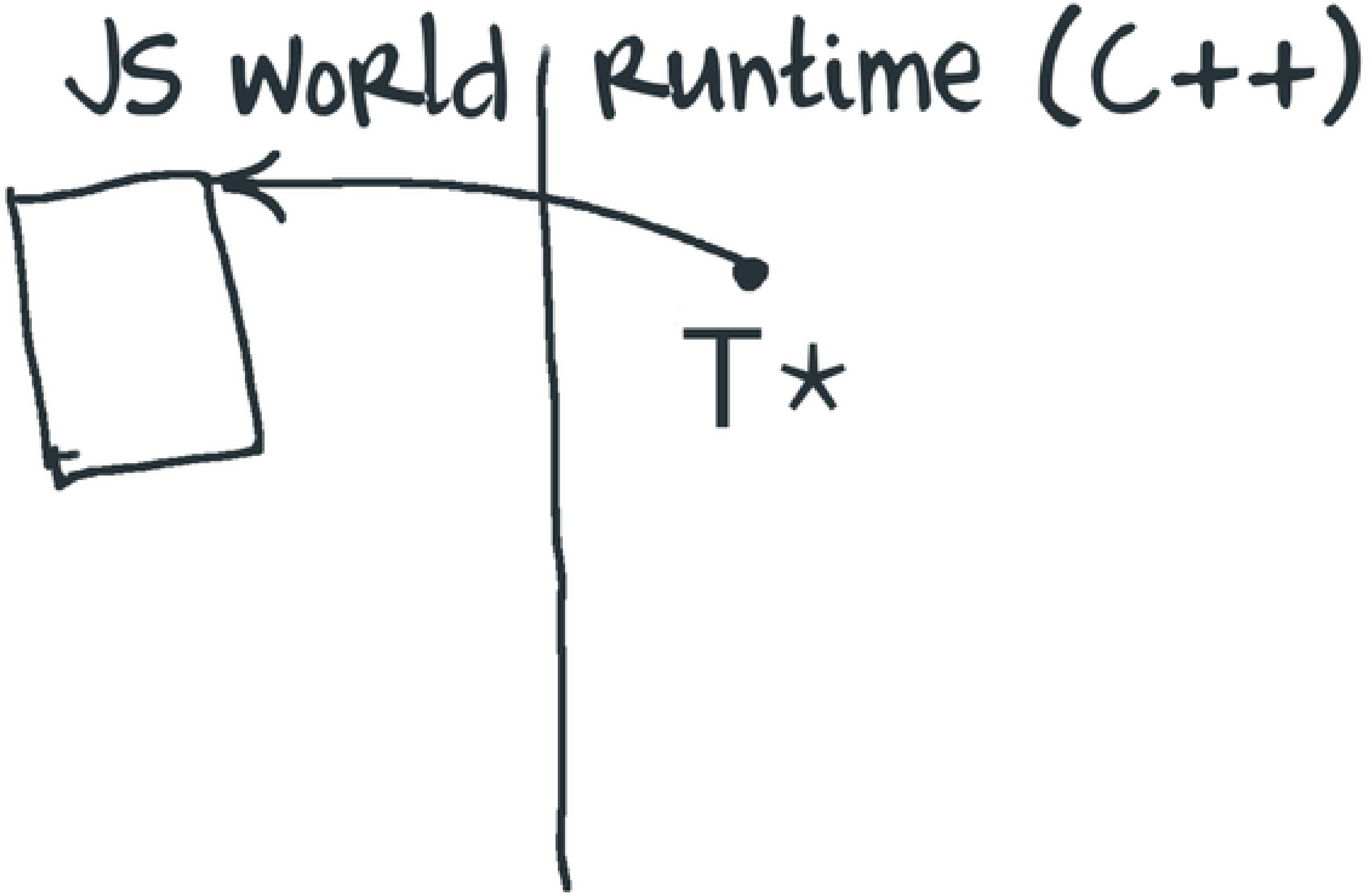
```
((Object*)0)->IsSmi()
```

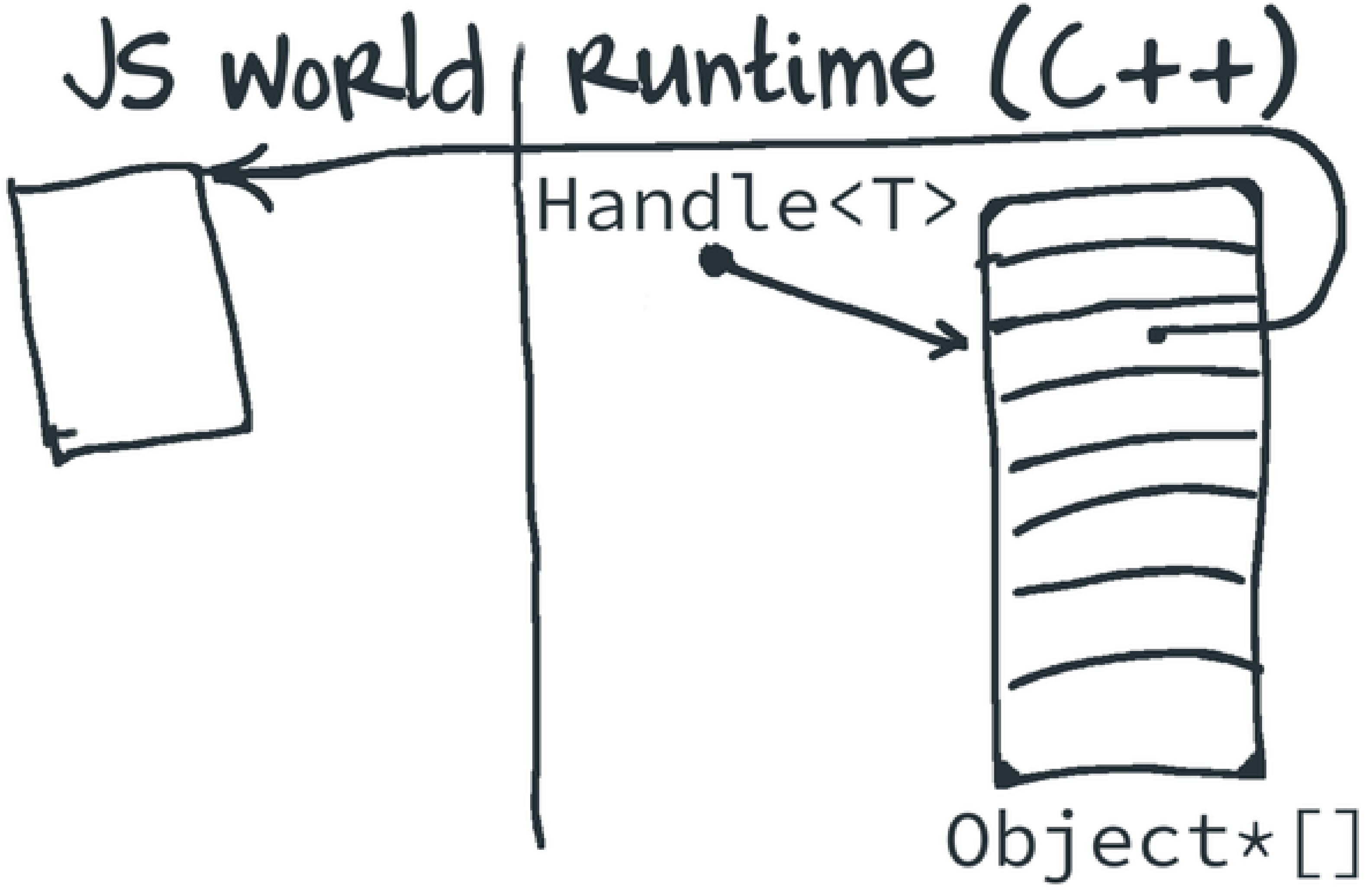
what about GC?

`T*` ⇒ `Handle<T>`

```
class HandleBase {
 protected:
  Object** location_;
};

template <typename T>
class Handle : public HandleBase {
 public:
  V8_INLINE T* operator->() const {
    return *reinterpret_cast<T**>(location_);
  }
};
```


JS world
Runtime (C++)
T*


JS World
Runtime (C++)
Handle<T>
Object*[]

```
Handle<Foo> foo;
foo->doSomethingElse(doSomething())
```

```
Handle<Foo> foo;
Foo* foo_ = foo.location_;
foo_->doSomethingElse(doSomething());
```

```
Handle<Foo> foo;
Foo* foo_ = foo.location_;
foo_->doSomethingElse(doSomething());
```

subtle bugs

unprotected this

no virtual behavior

`RawT⋆ ⇒ T`

```
class RawDouble : public RawNumber {
  ALIGN8 double value_;
};

class Double : public Number {
 public:
  double value() const { return raw_->ptr()->value_; }

  virtual const char* ToCString() const;

  static Double& Handle(RawDouble* raw);
 private:
  RawDouble* raw_;
};
```

```
Object& obj = Object::Handle();
obj = Something();
printf("obj = %s\n", obj.ToCString());
```

```
Object& obj = Object::Handle();
obj = Something();  // RawDouble?
printf("obj = %s\n", obj.ToCString());
```

```
DART_FORCE_INLINE void Object::SetRaw(RawObject* value) {
  raw_ = value;
  if ((reinterpret_cast(value) & kSmiTagMask) == kSmiTag) {
    set_vtable(Smi::handle_vtable_);
    return;
  }
  intptr_t cid = value->GetClassId();
  if (cid >= kNumPredefinedCids) {
    cid = kInstanceCid;
  }
  set_vtable(builtin_vtables_[cid]);
}
```

```
DART_FORCE_INLINE void Object::SetRaw(RawObject* value) {
  raw_ = value;
  if ((reinterpret_cast(value) & kSmiTagMask) == kSmiTag) {
    set_vtable(Smi::handle_vtable_);
    return;
  }
  intptr_t cid = value->GetClassId();
  if (cid >= kNumPredefinedCids) {
    cid = kInstanceCid;
  }
  set_vtable(builtin_vtables_[cid]);
}
```

solves *some* issues

entering RUNTIME is expensive

using RUNTIME is expensive

so people write *stubs* and *intrinsics*

```
// Access growable object array at specified index.
// On stack: growable array (+2), index (+1), return-address (+0).
void Intrinsifier::GrowableArrayGetIndexed(Assembler* assembler) {
  Label fall_through;
  __ movl(EBX, Address(ESP, + 1 * kWordSize));  // Index.
  __ movl(EAX, Address(ESP, + 2 * kWordSize));  // GrowableArray.
  __ testl(EBX, Immediate(kSmiTagMask));
  __ j(NOT_ZERO, &fall_through, Assembler::kNearJump);  // Non-smi index.
  // Range check using _length field.
  __ cmpl(EBX, FieldAddress(EAX, GrowableObjectArray::length_offset()));
  // Runtime throws exception.
  __ j(ABOVE_EQUAL, &fall_through, Assembler::kNearJump);
  __ movl(EAX, FieldAddress(EAX, GrowableObjectArray::data_offset()));  // data.

  // Note that EBX is Smi, i.e, times 2.
  ASSERT(kSmiTagShift == 1);
  __ movl(EAX, FieldAddress(EAX, EBX, TIMES_2, Array::data_offset()));
  __ ret();
  __ Bind(&fall_through);
}
```

```
bool Intrinsifier::Build_GrowableArrayGetIndexed(FlowGraph* flow_graph) {
  GraphEntryInstr* graph_entry = flow_graph->graph_entry();
  TargetEntryInstr* normal_entry = graph_entry->normal_entry();
  BlockBuilder builder(flow_graph, normal_entry);

  Definition* index = builder.AddParameter(1);
  Definition* growable_array = builder.AddParameter(2);

  PrepareIndexedOp(
      &builder, growable_array, index, GrowableObjectArray::length_offset());

  Definition* backing_store = builder.AddDefinition(
      new LoadFieldInstr(new Value(growable_array),
                         GrowableObjectArray::data_offset(),
                         Type::ZoneHandle(),
                         builder.TokenPos()));
  Definition* result = builder.AddDefinition(
      new LoadIndexedInstr(new Value(backing_store),
                           new Value(index),
                           Instance::ElementSizeFor(kArrayCid),  // index scale
                           kArrayCid,
                           Isolate::kNoDeoptId,
                           builder.TokenPos()));
  builder.AddIntrinsicReturn(new Value(result));
  return true;
}
```

```
DEFINE_NATIVE_ENTRY(GrowableList_getIndexed, 2) {
  const GrowableObjectArray& array =
      GrowableObjectArray::CheckedHandle(
          arguments->NativeArgAt(0));
  GET_NON_NULL_NATIVE_ARGUMENT(Smi, index,
                               arguments->NativeArgAt(1));
  if ((index.Value() < 0) ||
      (index.Value() >= array.Length())) {
    Exceptions::ThrowRangeError("index", index, 0,
                                array.Length() - 1);
  }
  const Instance& obj = Instance::CheckedHandle(
      array.At(index.Value()));
  return obj.raw();
}
```

```
Object* GrowableList_getIndexed(GrowableObjectArray* array,
                                intptr_t index) {
  if ((index < 0) || (index >= array->length())) {
    Exceptions::ThrowRangeError(
        "index", index, 0, array->length() - 1);
  }
  return array->At(index);
}
```

```
foo({a: 1, b: 2, c: 3, d: 4, e: 5, f: 6}) {
  return a + f;
}
```

```
// Generate code handling each optional parameter in alphabetical order.
__ movq(RBX, FieldAddress(R10, ArgumentsDescriptor::count_offset()));
__ movq(RCX,
        FieldAddress(R10, ArgumentsDescriptor::positional_count_offset()));
__ SmiUntag(RCX);
// Let RBX point to the first passed argument, i.e. to
// fp[kParamEndSlotFromFp + num_args]; num_args (RBX) is Smi.
__ leaq(RBX, Address(RBP, RBX, TIMES_4, kParamEndSlotFromFp * kWordSize));
// Let RDI point to the entry of the first named argument.
__ leaq(RDI,
        FieldAddress(R10, ArgumentsDescriptor::first_named_entry_offset()));
for (int i = 0; i < num_opt_named_params; i++) {
  Label load_default_value, assign_optional_parameter;
  const int param_pos = opt_param_position[i];
  // Check if this named parameter was passed in.
  // Load RAX with the name of the argument.
  __ movq(RAX, Address(RDI, ArgumentsDescriptor::name_offset()));
  ASSERT(opt_param[i]->name().IsSymbol());
  __ CompareObject(RAX, opt_param[i]->name());
  __ j(NOT_EQUAL, &load_default_value, Assembler::kNearJump);
  // Load RAX with passed-in argument at provided arg_pos, i.e. at
  // fp[kParamEndSlotFromFp + num_args - arg_pos].
  __ movq(RAX, Address(RDI, ArgumentsDescriptor::position_offset()));
  // RAX is arg_pos as Smi.
  // Point to next named entry.
  __ AddImmediate(RDI, Immediate(ArgumentsDescriptor::named_entry_size()));
  __ negq(RAX);
```

```
// Generate code handling each optional parameter in alphabetical order.
__ movq(RBX, FieldAddress(R10, ArgumentsDescriptor::count_offset()));
__ movq(RCX,
        FieldAddress(R10, ArgumentsDescriptor::positional_count_offset()));
__ SmiUntag(RCX);
// Let RBX point to the first passed argument, i.e. to
// fp[kParamEndSlotFromFp + num_args]; num_args (RBX) is Smi.
__ leaq(RBX, Address(RBP, RBX, TIMES_4, kParamEndSlotFromFp * kWordSize));
// Let RDI point to the entry of the first named argument.
__ leaq(RDI,
        FieldAddress(R10, ArgumentsDescriptor::first_named_entry_offset()));
for (int i = 0; i < num_opt_named_params; i++) {
  Label load_default_value, assign_optional_parameter;
  const int param_pos = opt_param_position[i];
  // Check if this named parameter was passed in.
  // Load RAX with the name of the argument.
  __ movq(RAX, Address(RDI, ArgumentsDescriptor::name_offset()));
  ASSERT(opt_param[i]->name().IsSymbol());
  __ CompareObject(RAX, opt_param[i]->name());
  __ j(NOT_EQUAL, &load_default_value, Assembler::kNearJump);
  // Load RAX with passed-in argument at provided arg_pos, i.e. at
  // fp[kParamEndSlotFromFp + num_args - arg_pos].
  __ movq(RAX, Address(RDI, ArgumentsDescriptor::position_offset()));
  // RAX is arg_pos as Smi.
  // Point to next named entry.
  __ AddImmediate(RDI, Immediate(ArgumentsDescriptor::named_entry_size()));
  __ negq(RAX);
```

WHYYYY?

```
intptr_t bar(const intptr_t* names, intptr_t argc, ...) {
  static const intptr_t kArgC = 6;
  static const intptr_t EXPECTED[kArgC] = {1, 2, 3, 4, 5, 6};
  static const intptr_t DEFAULT[kArgC] = {1, 2, 3, 4, 5, 6};

  intptr_t args[kArgC];

  const intptr_t *arg = names;
  const intptr_t *last = names + argc;

  va_list vl;
  va_start(vl, argc);
  for (intptr_t i = 0; i < kArgC && arg != last; i++) {
    if (*arg == EXPECTED[i]) {
      args[i] = va_arg(vl, intptr_t);
      arg++;
    } else {
      args[i] = DEFAULT[i];
    }
  }
  va_end(vl);

  return args[0] + args[5];
}
```

different kind of stub

e.g. fast path of property lookups

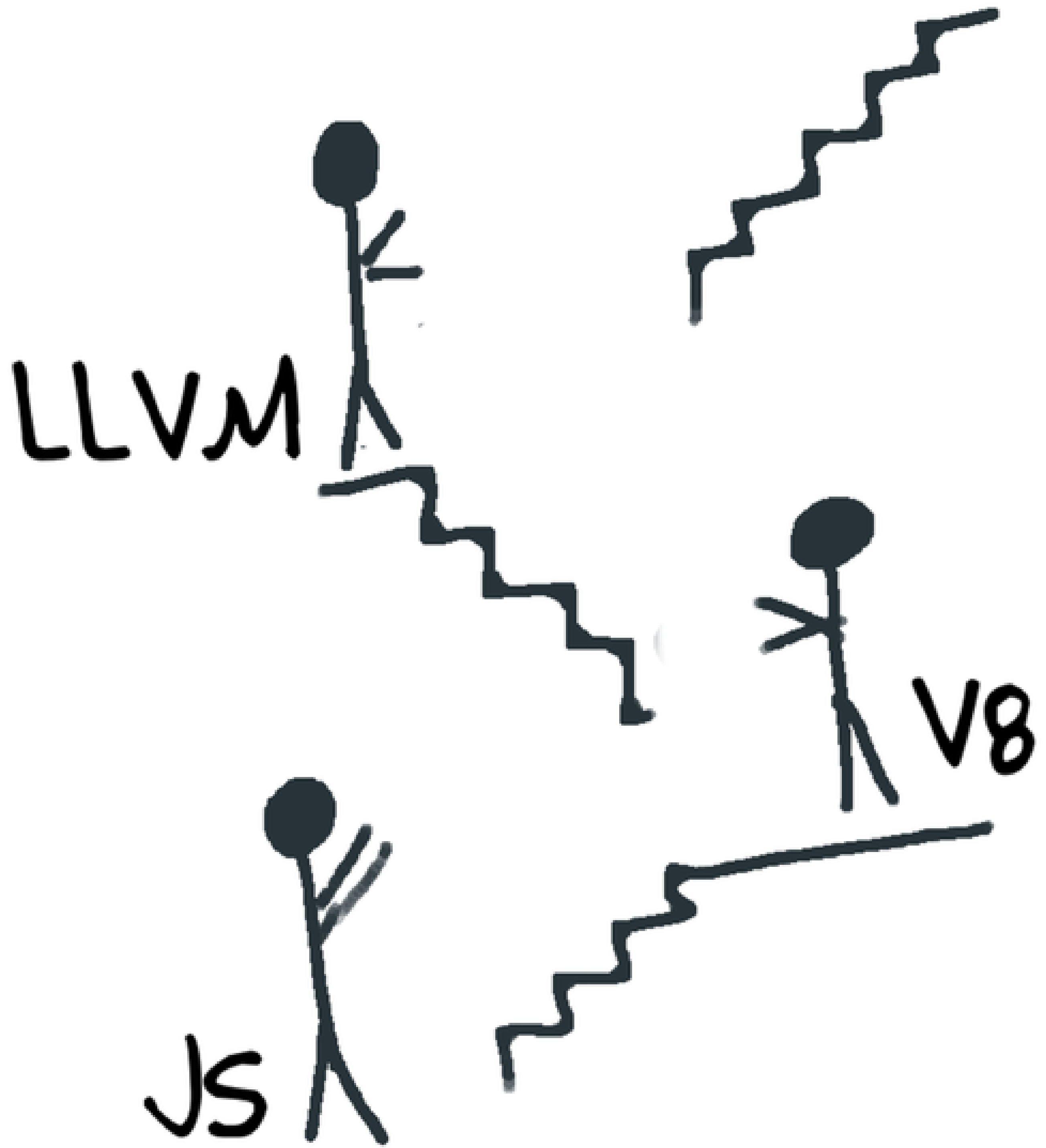
hand derived

based on the knowledge that

```
F(a) === F(b) ⇒ G(a) === G(b)
```

can be derived by tracing runtime

can we do this statically?


LLVM
V8
JS

«In the end, we are self-perceiving, self-inventing, locked-in mirages that are little miracles of self-reference.»

Douglas Hofstadter