Building High-Performance Language Implementations With Low Effort

Stefan Marr
FOSDEM 2015, Brussels, Belgium
January 31st, 2015

@smarr
http://stefan-marr.de

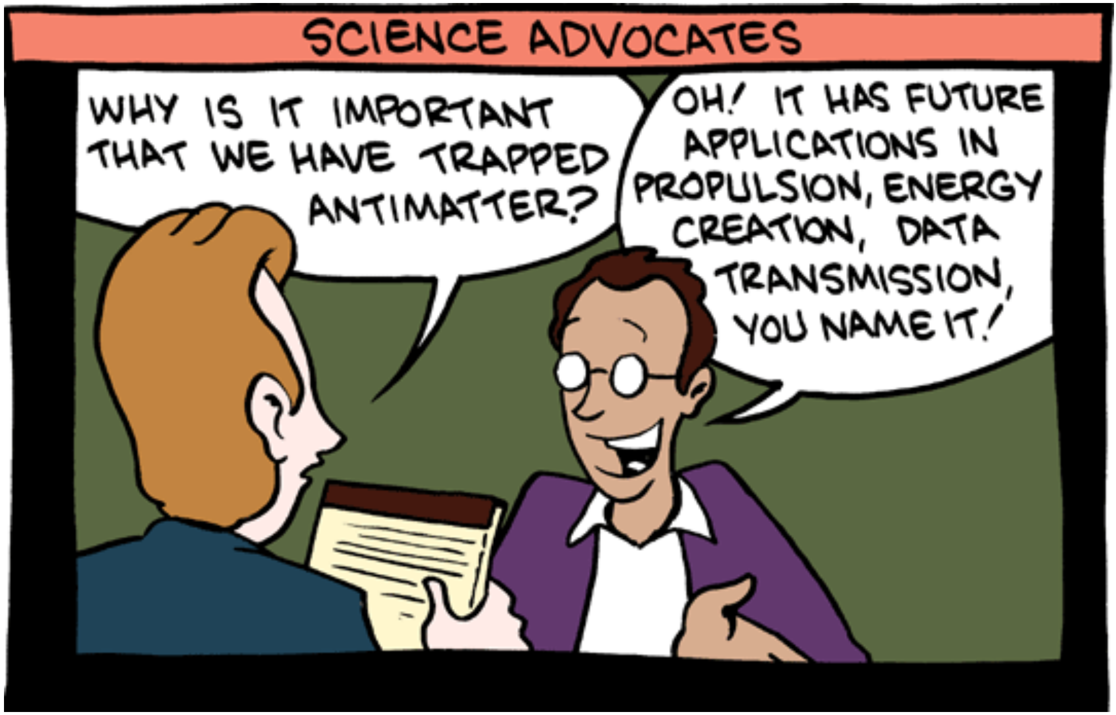
INVENTORS FOR THE DIGITAL WORLD

Why should you care about how Programming Languages work?

SCIENTISTS
WHY IS IT IMPORTANT THAT WE HAVE TRAPPED ANTIMATTER?
BECAUSE IT'S FUCKING AWESOME.

SMBC: http://www.smbc-comics.com/?id=2088

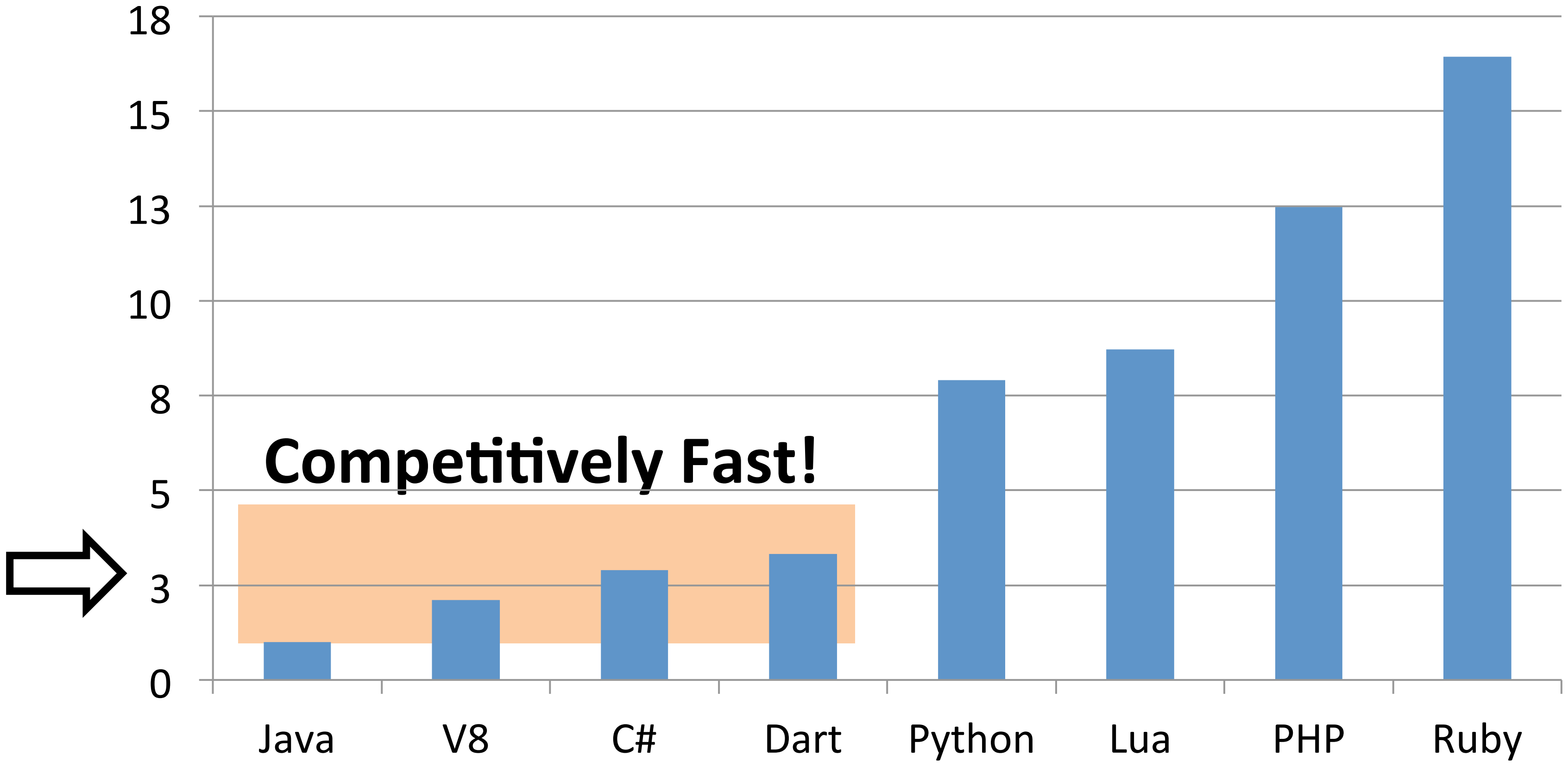
Why should you care about how Programming Languages work?


SCIENCE ADVOCATES
WHY IS IT IMPORTANT THAT WE HAVE TRAPPED ANTIMATTER?
OH! IT HAS FUTURE APPLICATIONS IN PROPULSION, ENERGY CREATION, DATA TRANSMISSION, YOU NAME IT!

- Performance isn't magic
- Domain-specific languages
  - More concise
  - More productive
- It's easier than it looks
  - Often open source
  - Contributions welcome

SMBC: http://www.smbc-comics.com/?id=2088

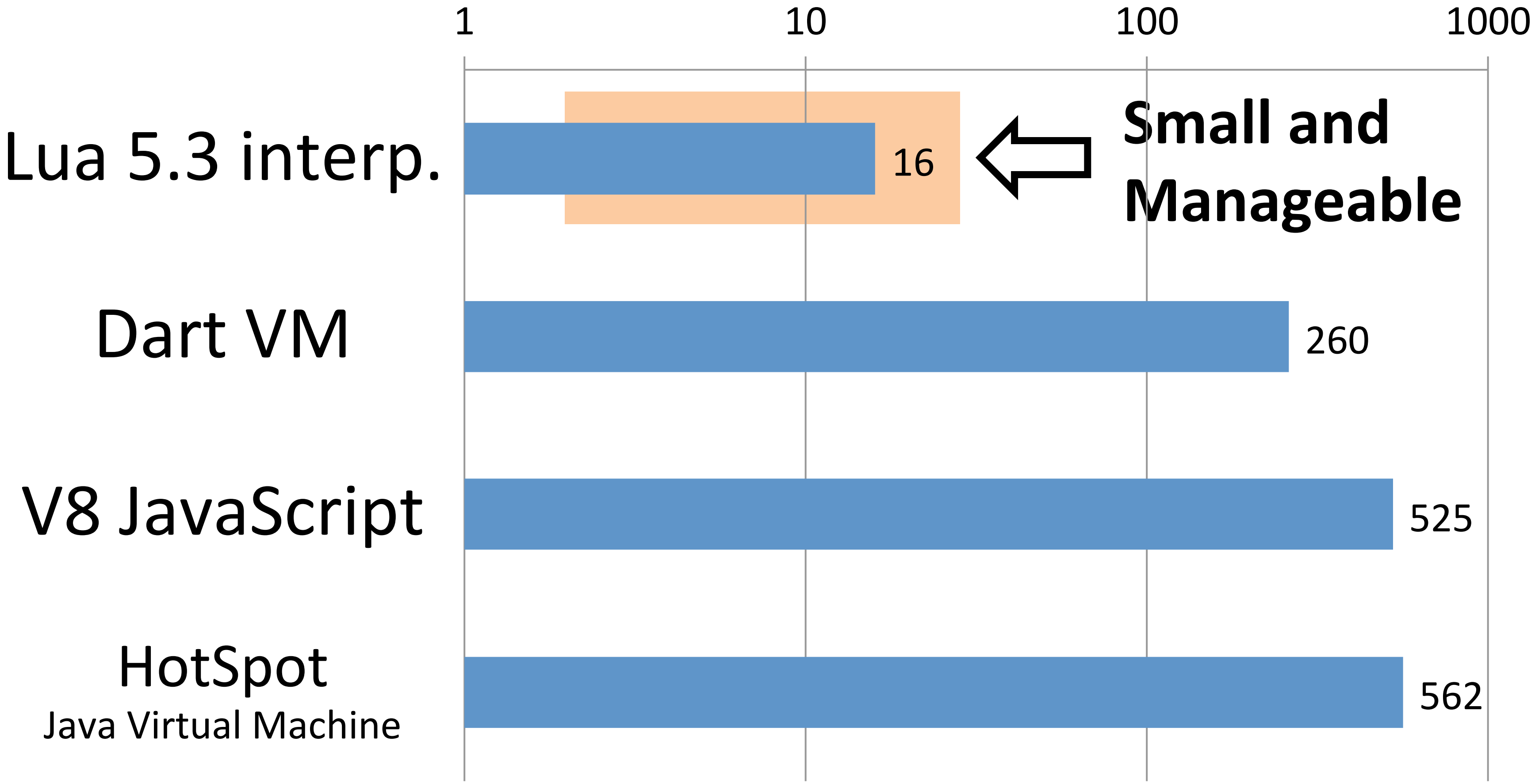
What's "High-Performance"?



Based on latest data from http://benchmarksgame.alioth.debian.org/
Geometric mean over available benchmarks.

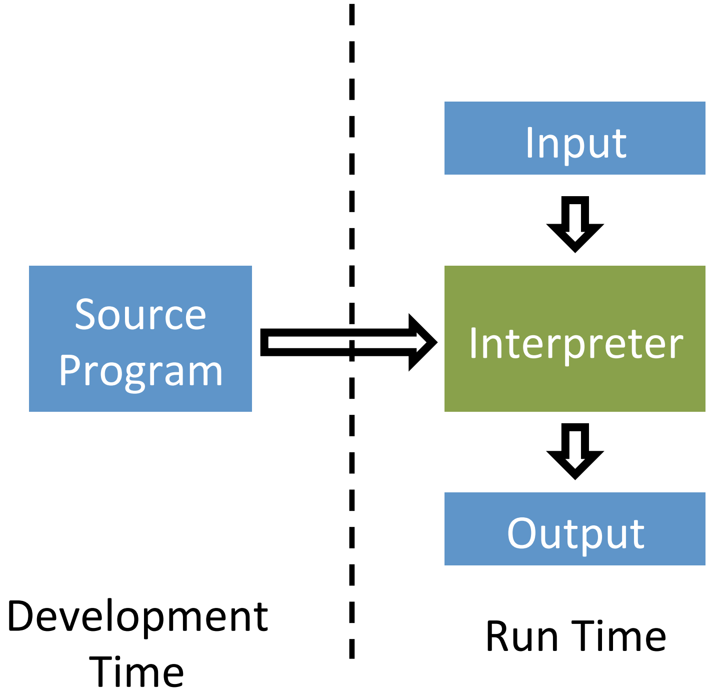
Disclaimer: Not indicate for application performance!

What’s “Low Effort”?



KLOC: 1000 Lines of Code, without blank lines and comments

Language Implementation Approaches



Source Program

Input

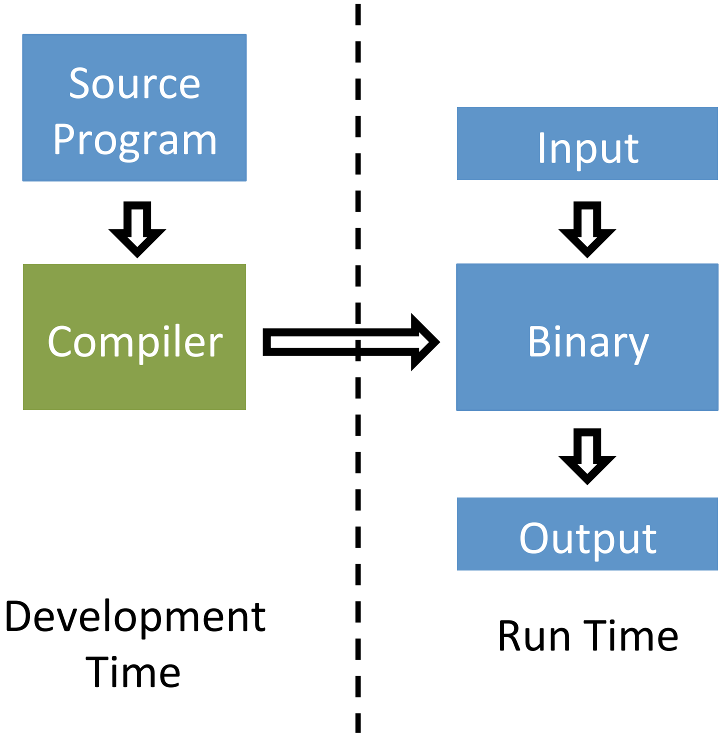
Interpreter

Output

Development Time

Run Time

Simple, but often slow



Source Program

Compiler

Input

Binary

Output

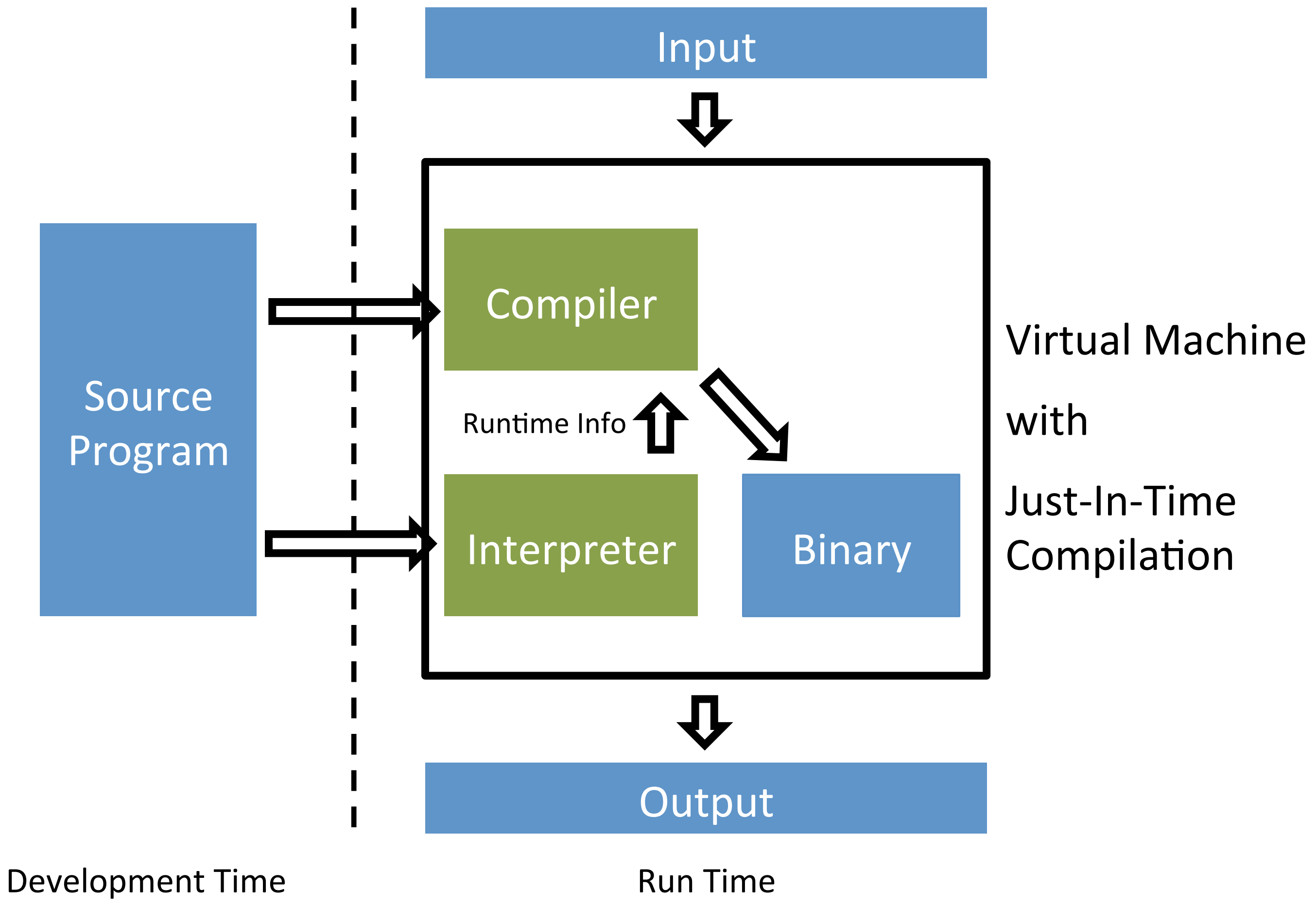
Development Time

Run Time

More complex, but often faster

Not ideal for all languages.

Modern Virtual Machines

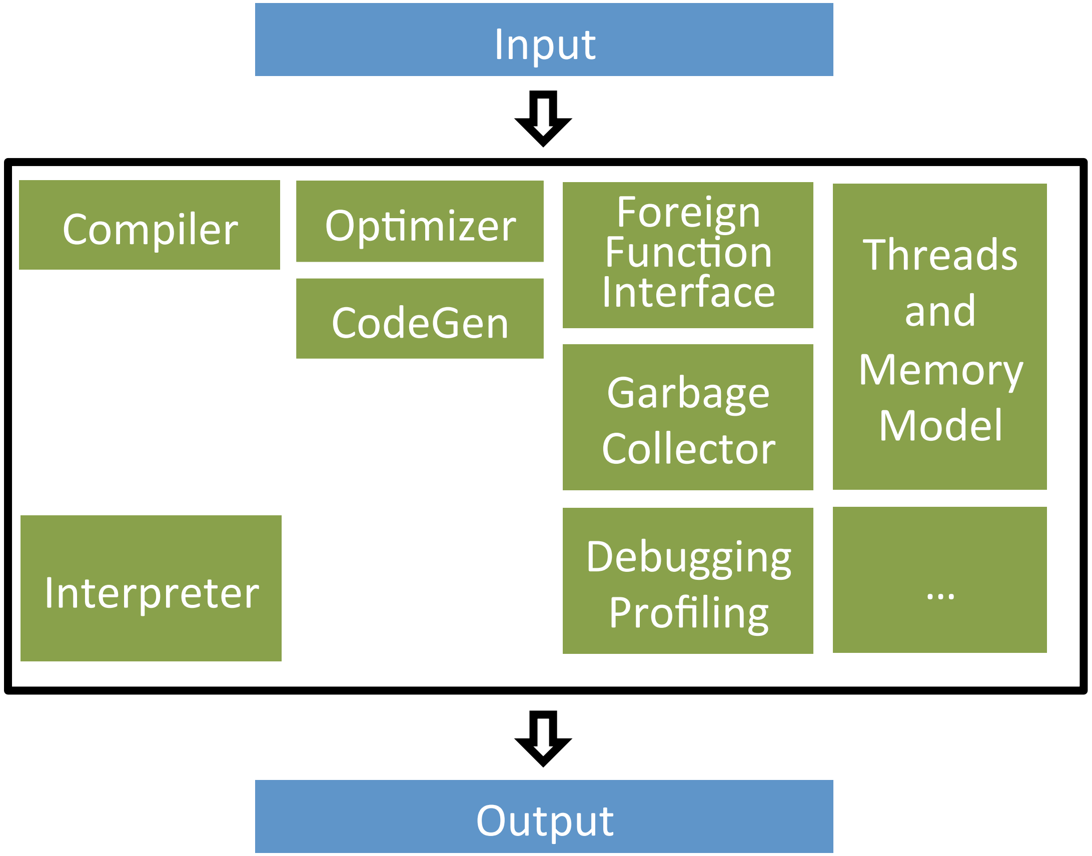


Virtual Machine

with

Just-In-Time Compilation

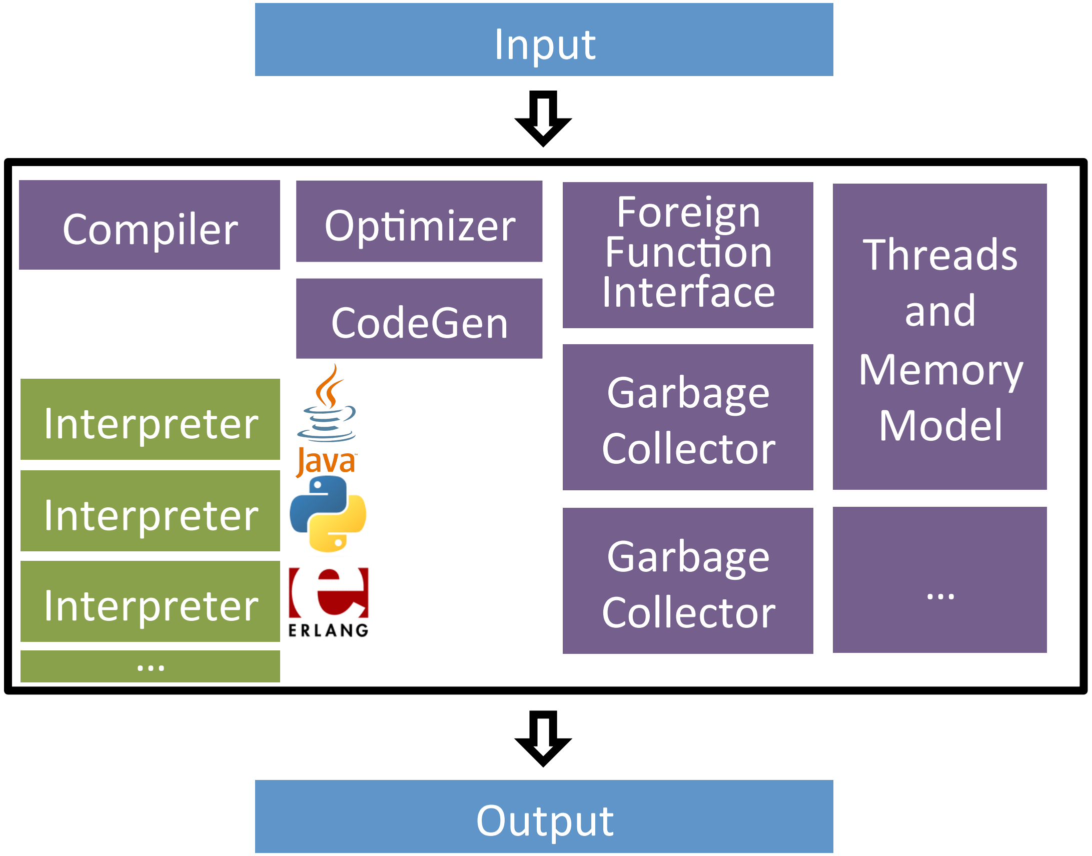
VMs are Highly Complex



Easily
500 KLOC

How to reuse most parts
for a new language?

How to reuse most parts for a new language?



Make Interpreters Replaceable Components!

Interpreter-based Approaches

RPython
with Meta-Tracing

pypy

Truffle + Graal
with Partial Evaluation

Oracle Labs

[2] Bolz et al., Tracing the Meta-level: PyPy's Tracing JIT Compiler, ICOOOLPS Workshop 2009, ACM, pp. 18-25.

[3] Würthinger et al., One VM to Rule Them All, Onward! 2013, ACM, pp. 187-204.

A Simple Technique for Language Implementation and Optimization

SELF-OPTIMIZING TREES

[1] Würthinger, T.; Wöß, A.; Stadler, L.; Duboscq, G.; Simon, D. & Wimmer, C. (2012), Self-Optimizing AST Interpreters, in 'Proc. of the 8th Dynamic Languages Symposium' , pp. 73-82.

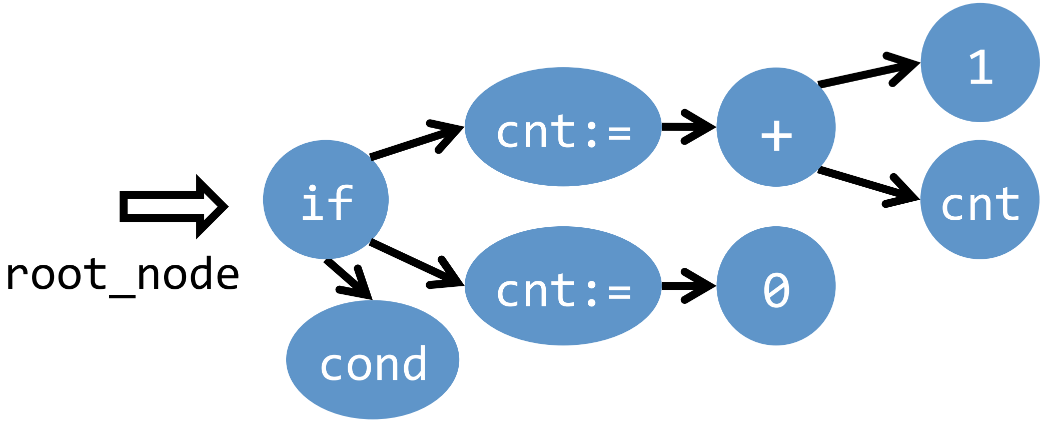
Code Convention

Java-ish → Application Code

Python-ish → Interpreter Code

A Simple
Abstract Syntax Tree Interpreter

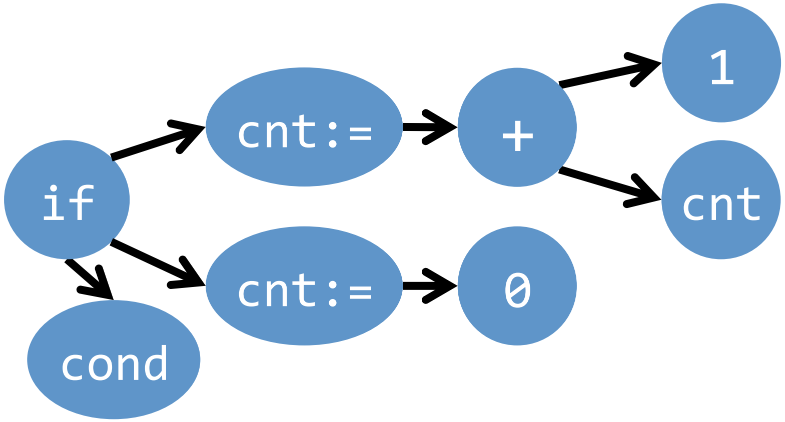
```
if (condition) {
  cnt := cnt + 1;
} else {
  cnt := 0;
}
```



```
root_node = parse(file)

root_node.execute(Frame())
```

Implementing AST Nodes

```
if (condition) {
  cnt := cnt + 1;
} else {
  cnt := 0;
}
```



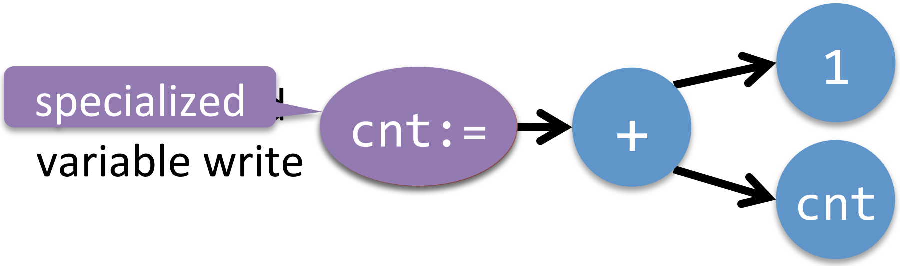
```
class Literal(ASTNode):
  final value
  def execute(frame):
    return value
```

```
class VarRead(ASTNode):
  final idx
  def execute(frame):
    return frame.local_obj[idx]
```

```
class VarWrite(ASTNode):
  child sub_expr
  final idx
  def execute(frame):
    val := sub_expr.execute(frame)
    frame.local_obj[idx]:= val
    return val
```

Self-Optimization by Node Specialization

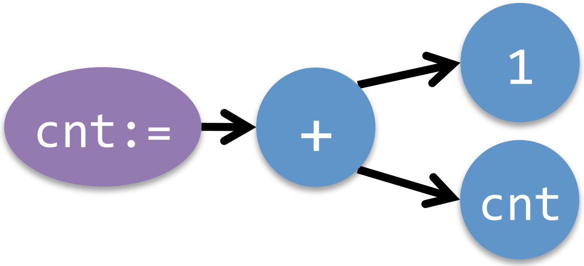
```
cnt := cnt + 1
```


specialized variable write

```
def UninitVarWrite.execute(frame):
  val := sub_expr.execute(frame)
  return specialize(val).
    execute_evaluated(frame, val)

def UninitVarWrite.specialize(val):
  if val instanceof int:
    return replace(IntVarWrite(sub_expr))
  elif …:
     …
  else:
    return replace(GenericVarWrite(sub_expr))
```

Self-Optimization by Node Specialization

```
cnt := cnt + 1
```



int
variable write

```
def IntVarWrite.execute(frame):
  try:
    val := sub_expr.execute_int(frame)
    return execute_eval_int(frame, val)
  except ResultExp, e:
    return respecialize(e.result).
      execute_evaluated(frame, e.result)

def IntVarWrite.execute_eval_int(frame, anInt):
  frame.local_int[idx] := anInt
  return anInt
```

Some Possible Self-Optimizations

- Type profiling and specialization

- Value caching
- Inline caching
- Operation inlining

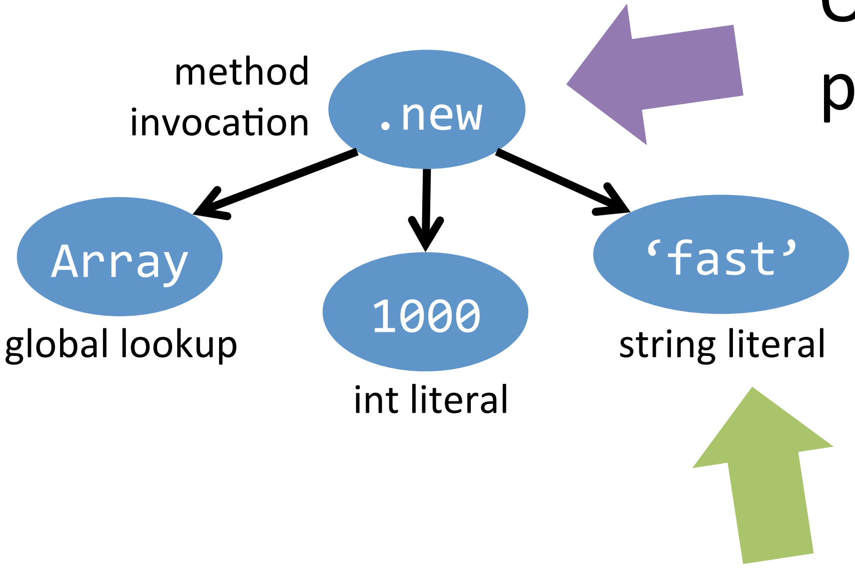
- Library Lowering

Library Lowering for Array class

```
createSomeArray() { return Array.new(1000, ‘fast fast fast’); }

class Array {
  static new(size, lambda) {
    return new(size).setAll(lambda);
  }

  setAll(lambda) {
    forEach((i, v) -> { this[i] = lambda.eval(); });
  }
}

class Object {
  eval() { return this; }
}
```

Optimizing for Object Values

```
createSomeArray() { return Array.new(1000, ‘fast fast fast’); }
```



method invocation

.new

Array

global lookup

1000

int literal

‘fast’

string literal

Optimization potential

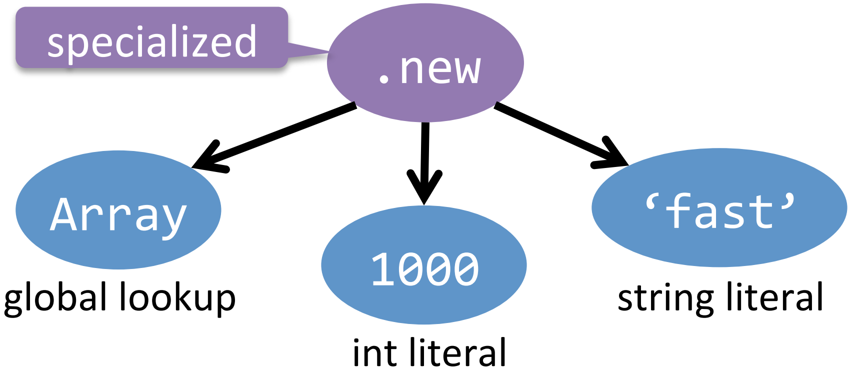
Object, but not a lambda

Specialized new(size, lambda)

```
createSomeArray() { return Array.new(1000, ‘fast fast fast’); }
def UninitArrNew.execute(frame):
  size := size_expr.execute(frame)
  val  := val_expr.execute(frame)
  return specialize(size, val).
    execute_evaluated(frame, size, val)

def UninitArrNew.specialize(size, val):
  if val instanceof Lambda:
    return replace(StdMethodInvocation())
  else:
    return replace(ArrNewWithValue())
```

Specialized new(size, lambda)

```
createSomeArray() { return Array.new(1000, ‘fast fast fast’); }
```



specialized

.new

Array

global lookup

1000

int literal

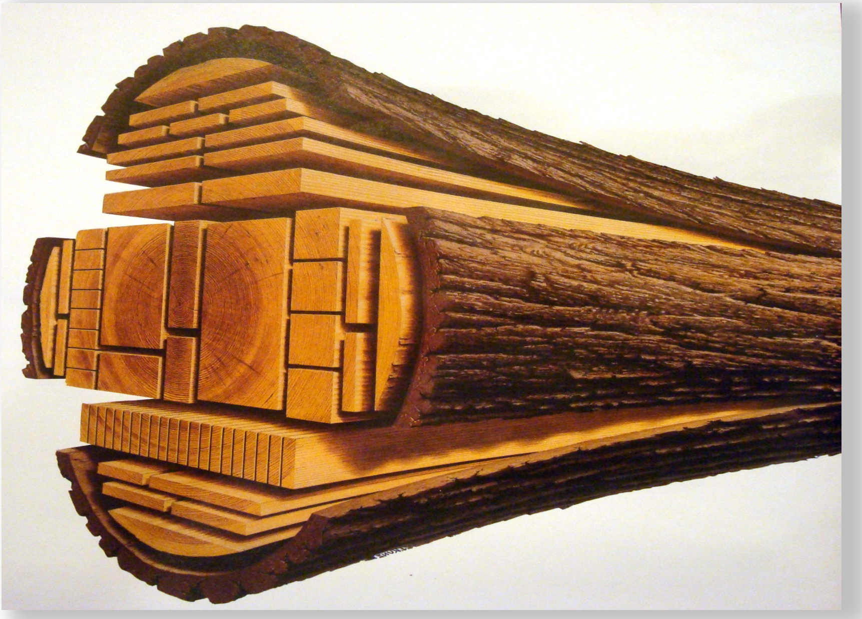
‘fast’

string literal

```
def ArrNewWithValue.execute_evaluated(frame, size, val):
  return Array([val] * 1000)
```

1 specialized node vs.

1000x `this[i] = lambda.eval()`
1000x `eval() { return this; }`



Generating Efficient Native Code

Just-in-Time Compilation for Interpreters

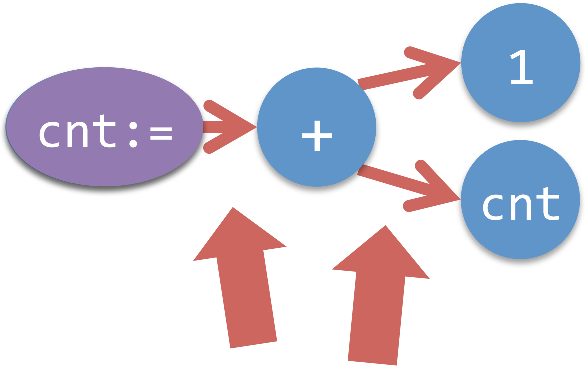
How to Get Fast Program Execution?

Standard Compilation: 1 node at a time

```
VarWrite.execute(frame)                  ..VW_execute()    # bin
IntVarWrite.execute(frame)               ..IVW_execute()   # bin
VarRead.execute(frame)                   ..VR_execute()    # bin
Literal.execute(frame)                   ..L_execute()     # bin
ArrayNewWithValue.execute(frame)         ..ANWV_execute() # bin
```

Minimal Optimization Potential

Problems with Node-by-Node Compilation



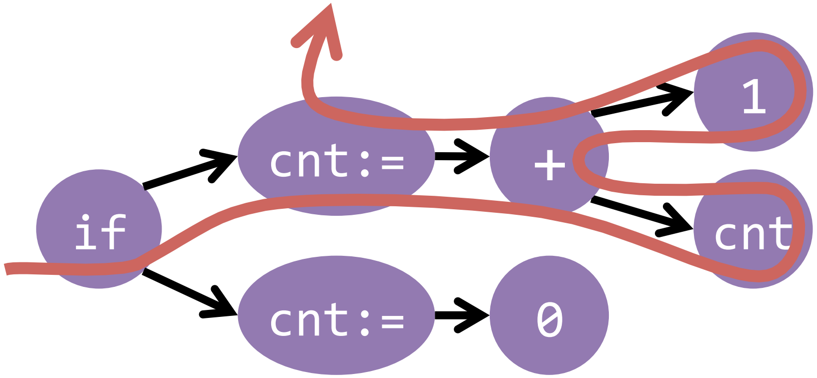
Slow Polymorphic Dispatches

```
def IntVarWrite.execute(frame):
 try:
  val := sub_expr.execute_int(frame)
  return execute_eval_int(frame, val)
 except ResultExp, e:
  return respecialize(e.result).
   execute_evaluated(frame, e.result)
```

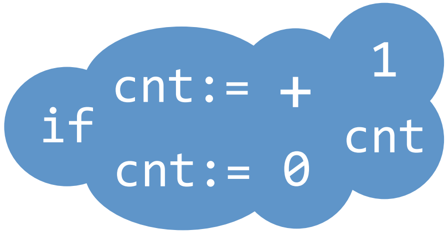
Runtime checks in general

Compilation Unit based on User Program

Meta-Tracing



Partial Evaluation Guided By AST



[2] Bolz et al., Tracing the Meta-level: PyPy's Tracing JIT Compiler, ICOOOLPS Workshop 2009, ACM, pp. 18-25.

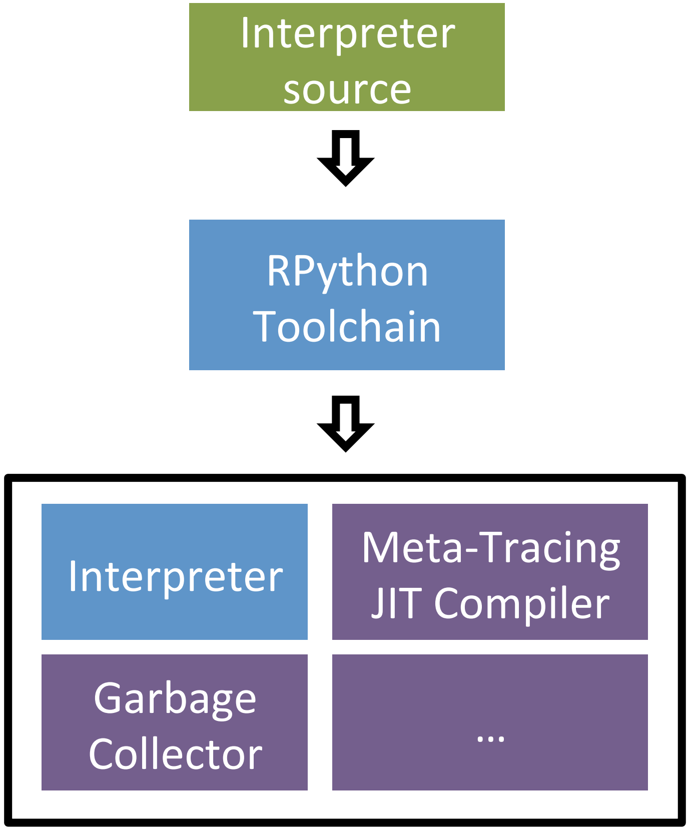
[3] Würthinger et al., One VM to Rule Them All, Onward! 2013, ACM, pp. 187-204.

Just-in-Time Compilation with Meta Tracing

RPython

pypy

RPython



Interpreter source

RPython Toolchain

Interpreter

Meta-Tracing JIT Compiler

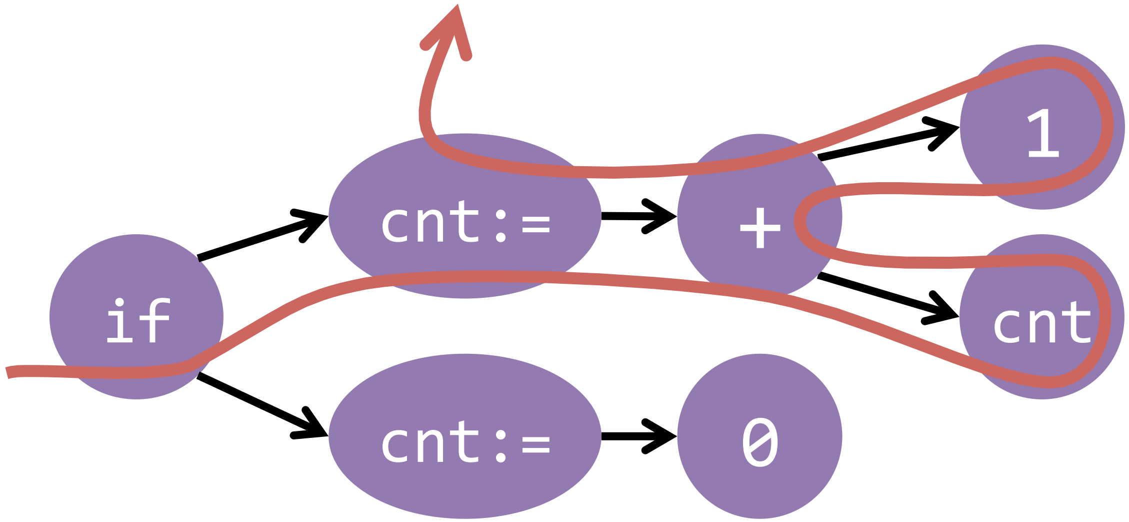
Garbage Collector

…

- Subset of Python
  - Type-inferenced
- Generates VMs

http://rpython.readthedocs.org/

Meta-Tracing of an Interpreter



[2] Bolz et al., Tracing the Meta-level: PyPy's Tracing JIT Compiler, ICOOOLPS Workshop 2009, ACM, pp. 18-25.

Meta Tracers need to know the Loops

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      jit_merge_point(node=self)

      cond = cond_expr.execute_bool(frame)
      if not cond:
        break
      body_expr.execute(frame)
```

Trace

```
guard(cond_expr == Const(IntLessThan))
```

Tracing Records one Concrete Execution

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r:
      ...
    return left < right
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
```

Tracing Records one Concrete Execution

```
class IntVarRead(ASTNode):
  final idx

  def execute_int(frame):
    if frame.is_int(idx):
      return frame.local_int[idx]
    else:
      new_node = respecialize()
      raise UnexpectedResult(new_node.execute())
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
```

`i1 := left_expr.idx # Const(1)`

Tracing Records one Concrete Execution

```
class IntVarRead(ASTNode):
  final idx

  def execute_int(frame):
    if frame.is_int(idx):
      return frame.local_int[idx]
    else:
      new_node = respecialize()
      raise UnexpectedResult(new_node.execute())
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
```

Tracing Records one Concrete Execution

```
class IntVarRead(ASTNode):
  final idx

  def execute_int(frame):
    if frame.is_int(idx):
      return frame.local_int[idx]
    else:
      new_node = respecialize()
      raise UnexpectedResult(new_node.execute())
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
```

Tracing Records one Concrete Execution

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r:
      ...
    return left < right
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult)
```

Tracing Records one Concrete Execution

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r:
      ...
    return left < right
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult)
```
`guard(right_expr == Const(IntLiteral))`

Tracing Records one Concrete Execution

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r
      ...
    return left < right
```

```
while (cnt < 100) {
   cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult)
guard(right_expr == Const(IntLiteral))
i5 := right_expr.value # Const(100)
```

Tracing Records one Concrete Execution

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r:
      ...
    return left < right
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult)
guard(right_expr == Const(IntLiteral))
i5 := right_expr.value # Const(100)
guard_no_exception(Const(UnexpectedResult)
```

Tracing Records one Concrete Execution

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r:
      ...
    return left < right
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult)
guard(right_expr == Const(IntLiteral))
i5 := right_expr.value # Const(100)
guard_no_exception(Const(UnexpectedResult)
b1 := i4 < i5
```

Tracing Records one Concrete Execution

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      jit_merge_point(node=self)

      cond = cond_expr.execute_bool(frame)
      if not cond:
        break
      body_expr.execute(frame)
```

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult
guard(right_expr == Const(IntLiteral))
i5 := right_expr.value # Const(100)
guard_no_exception(Const(UnexpectedResult
b1 := i4 < i5
guard_true(b1)
```

Tracing Records one Concrete Execution

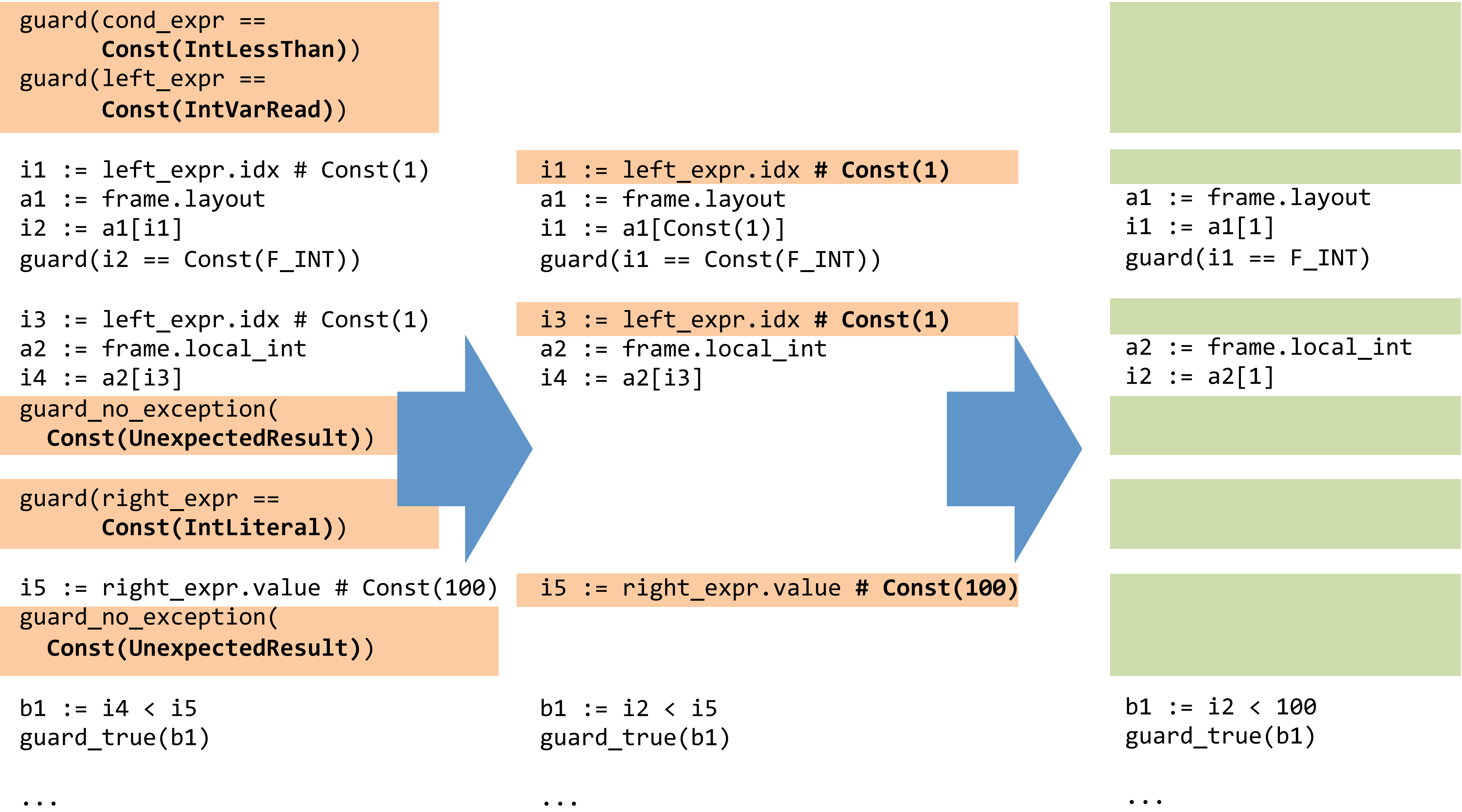
```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      jit_merge_point(node=self)

      cond = cond_expr.execute_bool(frame)
      if not cond:
        break
      body_expr.execute(frame)
```

Trace

```
guard(cond_expr == Const(IntLessThan))
guard(left_expr == Const(IntVarRead))
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))
i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(Const(UnexpectedResult)
guard(right_expr == Const(IntLiteral))
i5 := right_expr.value # Const(100)
guard_no_exception(Const(UnexpectedResult)
b1 := i4 < i5
guard_true(b1)
...
```

Traces are Ideal for Optimization



```
guard(cond_expr ==
      Const(IntLessThan))
guard(left_expr ==
      Const(IntVarRead))

i1 := left_expr.idx # Const(1)
a1 := frame.layout
i2 := a1[i1]
guard(i2 == Const(F_INT))

i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]
guard_no_exception(
  Const(UnexpectedResult))

guard(right_expr ==
      Const(IntLiteral))

i5 := right_expr.value # Const(100)
guard_no_exception(
  Const(UnexpectedResult))

b1 := i4 < i5
guard_true(b1)

...
```

```
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i1 := a1[Const(1)]
guard(i1 == Const(F_INT))

i3 := left_expr.idx # Const(1)
a2 := frame.local_int
i4 := a2[i3]

i5 := right_expr.value # Const(100)

b1 := i2 < i5
guard_true(b1)

...
```

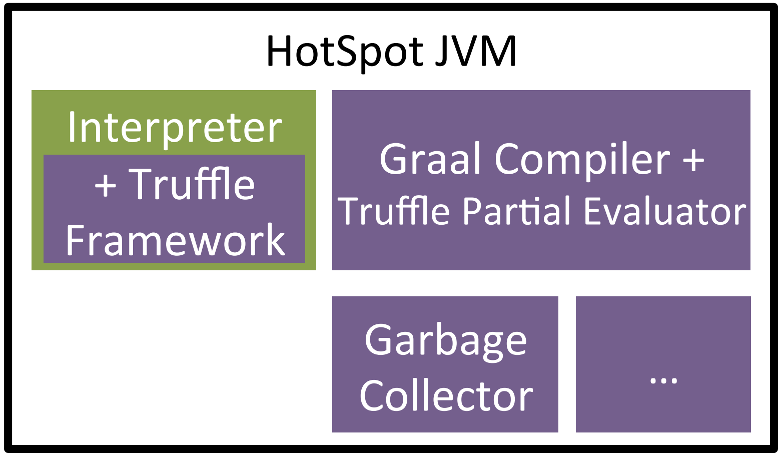
```
a1 := frame.layout
i1 := a1[1]
guard(i1 == F_INT)

a2 := frame.local_int
i2 := a2[1]

b1 := i2 < 100
guard_true(b1)

...
```

Just-in-Time Compilation with Partial Evaluation

Truffle + Graal

Oracle Labs

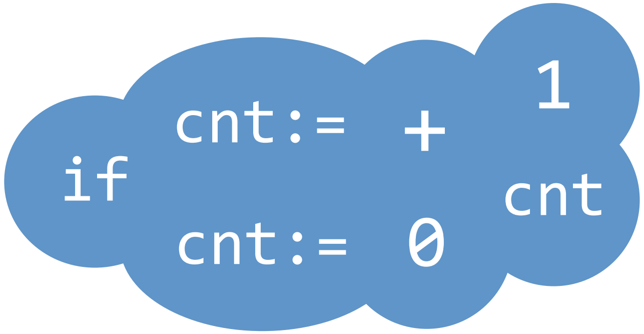
Truffle+Graal



- Java framework
  - AST interpreters
- Based on HotSpot JVM

http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html
http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html

Partial Evaluation Guided By AST



[3] Würthinger et al., One VM to Rule Them All, Onward! 2013, ACM, pp. 187-204.

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      cond = cond_expr.execute_bool(frame)
      if not cond:
        break
      body_expr.execute(frame)
```

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      cond = cond_expr.execute_bool(frame)
      if not cond:
        break
      body_expr.execute(frame)
```

```
class IntLessThan(ASTNode):
  child left_expr
  child right_expr

  def execute_bool(frame):
    try:
      left = left_expr.execute_int()
    except UnexpectedResult r:
      ...
    try:
      right = right_expr.execute_int()
    expect UnexpectedResult r:
      ...
    return left < right
```

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      try:
        left = cond_expr.left_expr.execute_int()
      except UnexpectedResult r:
        ...
      try:
        right = cond_expr.right_expr.execute_int()
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      try:
        left = cond_expr.left_expr.execute_int()
      except UnexpectedResult r:
        ...
      try:
        right = cond_expr.right_expr.execute_int()
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

```
class IntVarRead(ASTNode):
  final idx

  def execute_int(frame):
    if frame.is_int(idx):
      return frame.local_int[idx]
    else:
      new_node = respecialize()
      raise UnexpectedResult(new_node.ex
```

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      try:
        if frame.is_int(1):
          left = frame.local_int[1]
        else:
          new_node = respecialize()
          raise UnexpectedResult(new_node.execute())
      except UnexpectedResult r:
        ...
      try:
        right = cond_expr.right_expr.execute_int()
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
```

Optimize Optimistically

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      try:
        if frame.is_int(1):
          left = frame.local_int[1]
        else:
          new_node = respecialize()
          raise UnexpectedResult(new_node.execute())
      except UnexpectedResult r:
        ...
      try:
        right = cond_expr.right_expr.execute_int()
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
```

Optimize Optimistically

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      if frame.is_int(1):
        left = frame.local_int[1]
      else:
        __deopt_return_to_interp()
      try:
        right = cond_expr.right_expr.execute_int()
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      if frame.is_int(1):
        left = frame.local_int[1]
      else:
        __deopt_return_to_interp()
      try:
        right = cond_expr.right_expr.execute_int()
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

```
class IntLiteral(ASTNode):
  final value
  def execute_int(frame):
    return value
```

Partial Evaluation inlines based on Runtime Constants

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      if frame.is_int(1):
        left = frame.local_int[1]
      else:
        __deopt_return_to_interp()
      try:
        right = 100
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

```
class IntLiteral(ASTNode):
  final value
  def execute_int(frame):
    return value
```

Classic Optimizations:

Dead Code Elimination

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      if frame.is_int(1):
        left = frame.local_int[1]
      else:
        __deopt_return_to_interp()
      try:
        right = 100
      expect UnexpectedResult r:
        ...
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

```
class IntLiteral(ASTNode):
  final value
  def execute_int(frame):
    return value
```

Classic Optimizations:

Constant Propagation

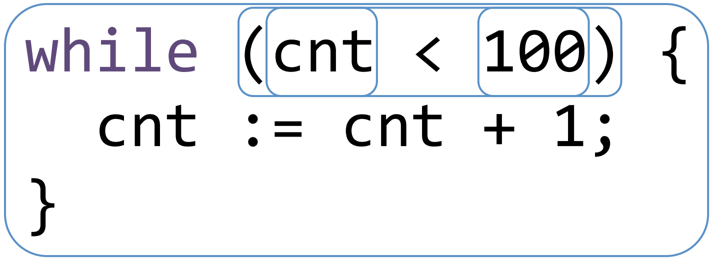
```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      if frame.is_int(1):
        left = frame.local_int[1]
      else:
        __deopt_return_to_interp()
      right = 100
      cond = left < right
      if not cond:
        break
      body_expr.execute(frame)
```

```
class IntLiteral(ASTNode):
  final value
  def execute_int(frame):
    return value
```

Classic Optimizations:

Loop Invariant Code Motion



```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    while True:
      if frame.is_int(1):
        left = frame.local_int[1]
      else:
        __deopt_return_to_interp()

      if not (left < 100):
        break
      body_expr.execute(frame)
```

Classic Optimizations:

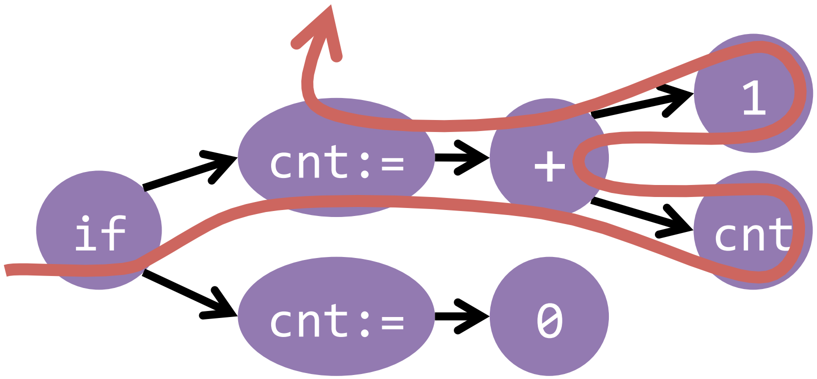
Loop Invariant Code Motion

```
while (cnt < 100) {
  cnt := cnt + 1;
}
```

```
class WhileNode(ASTNode):
  child cond_expr
  child body_expr

  def execute(frame):
    if not frame.is_int(1):
      __deopt_return_to_interp()

    while True:
      if not (frame.local_int[1] < 100):
        break
      body_expr.execute(frame)
```

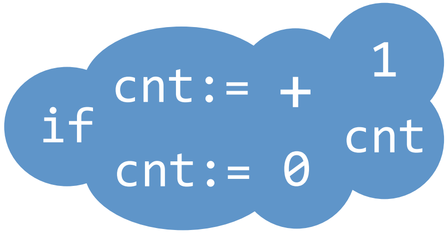
Compilation Unit based on User Program

Meta-Tracing



Partial Evaluation

Guided by AST



[2] Bolz et al., Tracing the Meta-level: PyPy's Tracing JIT Compiler, ICOOOLPS Workshop 2009, ACM, pp. 18-25.

[3] Würthinger et al., One VM to Rule Them All, Onward! 2013, ACM, pp. 187-204.

Results

What’s possible for a simple interpreter?

SOM

SimpleObjectMachine

http://som-st.github.io

Designed for Teaching:

- Simple
- Conceptual Clarity
- An Interpreter family
  - in C, C++, Java, JavaScript, RPython, Smalltalk

Used in the past by:

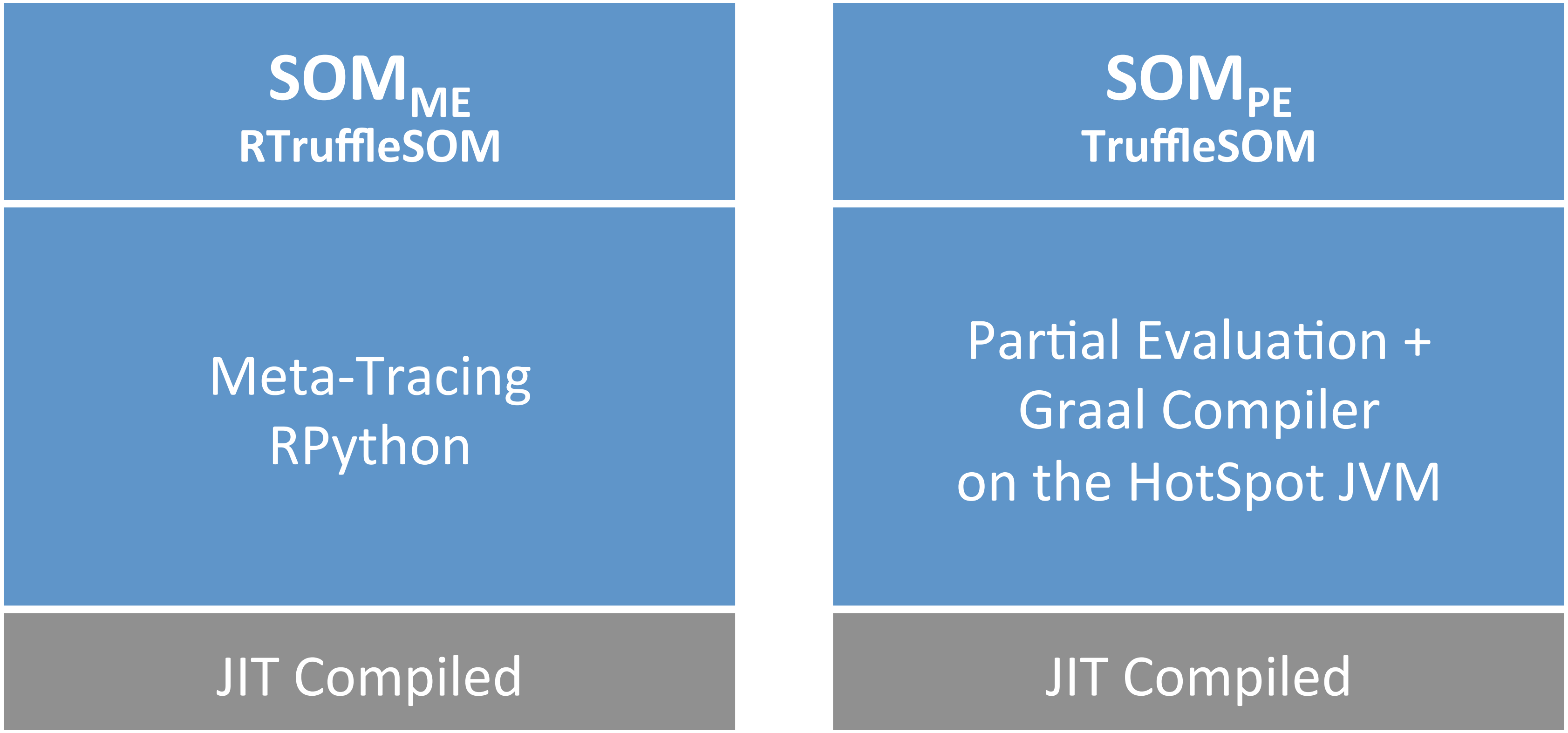
AARHUS UNIVERSITY

HPI Hasso Plattner Institut
IT Systems Engineering | Universität Potsdam

HEINRICH HEINE
UNIVERSITÄT DÜSSELDORF

Software
Languages.Lab

Self-Optimizing SOMs



SOM_{ME}

RTruffleSOM

Meta-Tracing

RPython

JIT Compiled

SOM_{PE}

TruffleSOM

Partial Evaluation +

Graal Compiler

on the HotSpot JVM

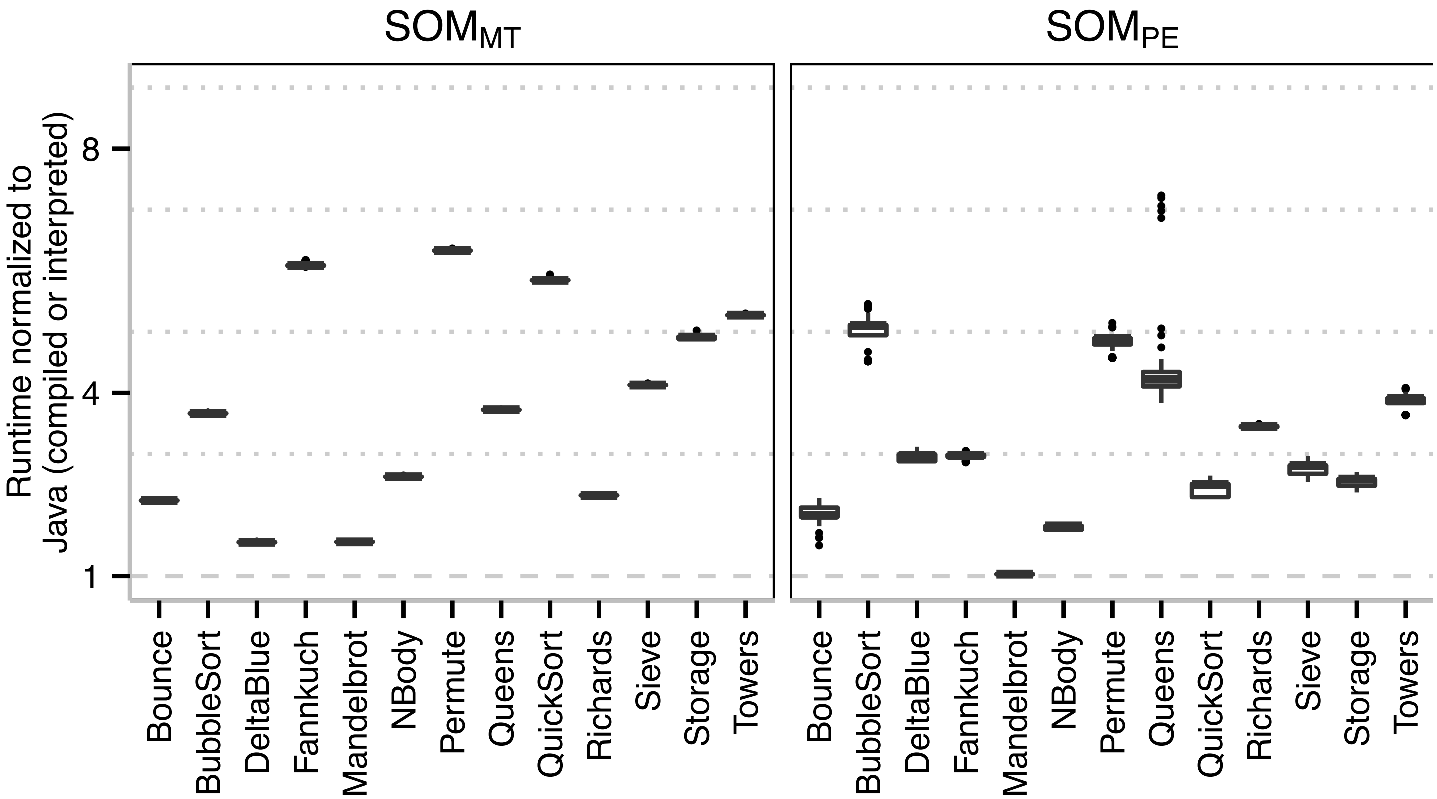
JIT Compiled

github.com/SOM-st/RTruffleSOM

github.com/SOM-st/TruffleSOM

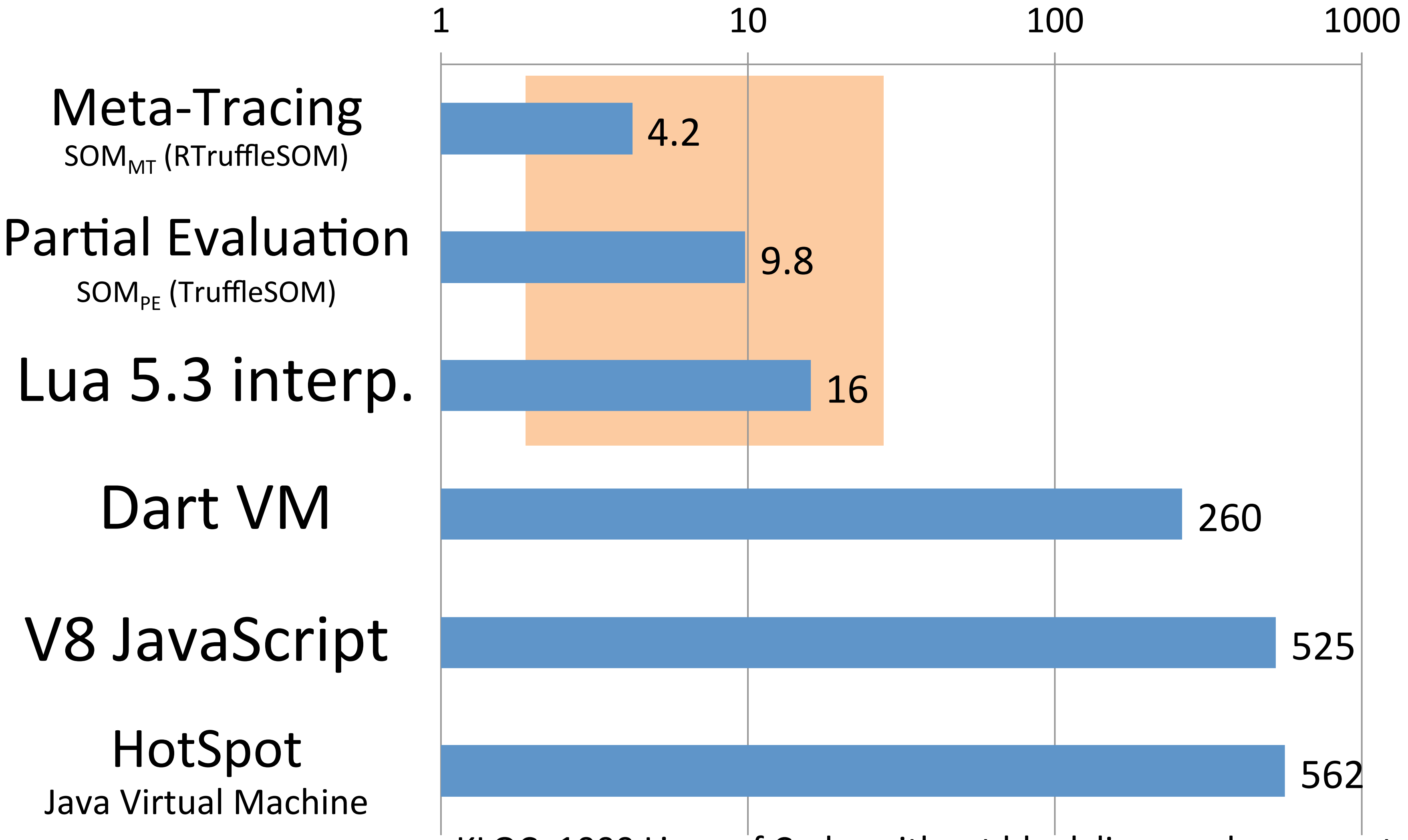
Java 8 `-server` *vs.* SOM+JIT

JIT-compiled Peak Performance



3.5x slower	2.8x slower
(min. 1.6x, max. 6.3x)	(min. 3%, max. 5x)
RPython	Truffle+Graal

Implementation: Smaller Than Lua

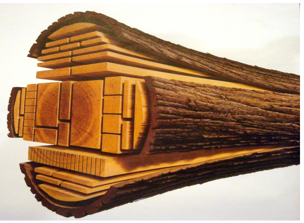


KLOC: 1000 Lines of Code, without blank lines and comments

CONCLUSION

Simple and Fast Interpreters are Possible!

Self-optimizing AST interpreters



RPython or Truffle for JIT Compilation

Literature on the ideas:

[1] Würthinger et al., Self-Optimizing AST Interpreters, Proc. of the 8th Dynamic Languages Symposium, 2012, pp. 73-82.

[2] Bolz et al., Tracing the Meta-level: PyPy's Tracing JIT Compiler, ICOOOLPS Workshop 2009, ACM, pp. 18-25.

[3] Würthinger et al., One VM to Rule Them All, Onward! 2013, ACM, pp. 187-204.

[4] Marr et al., Are We There Yet? Simple Language Implementation Techniques for the 21st Century. IEEE Software 31(5):60—67, 2014

Big Thank You!

to both communities,
for help, answering questions, debugging support, etc…!!!

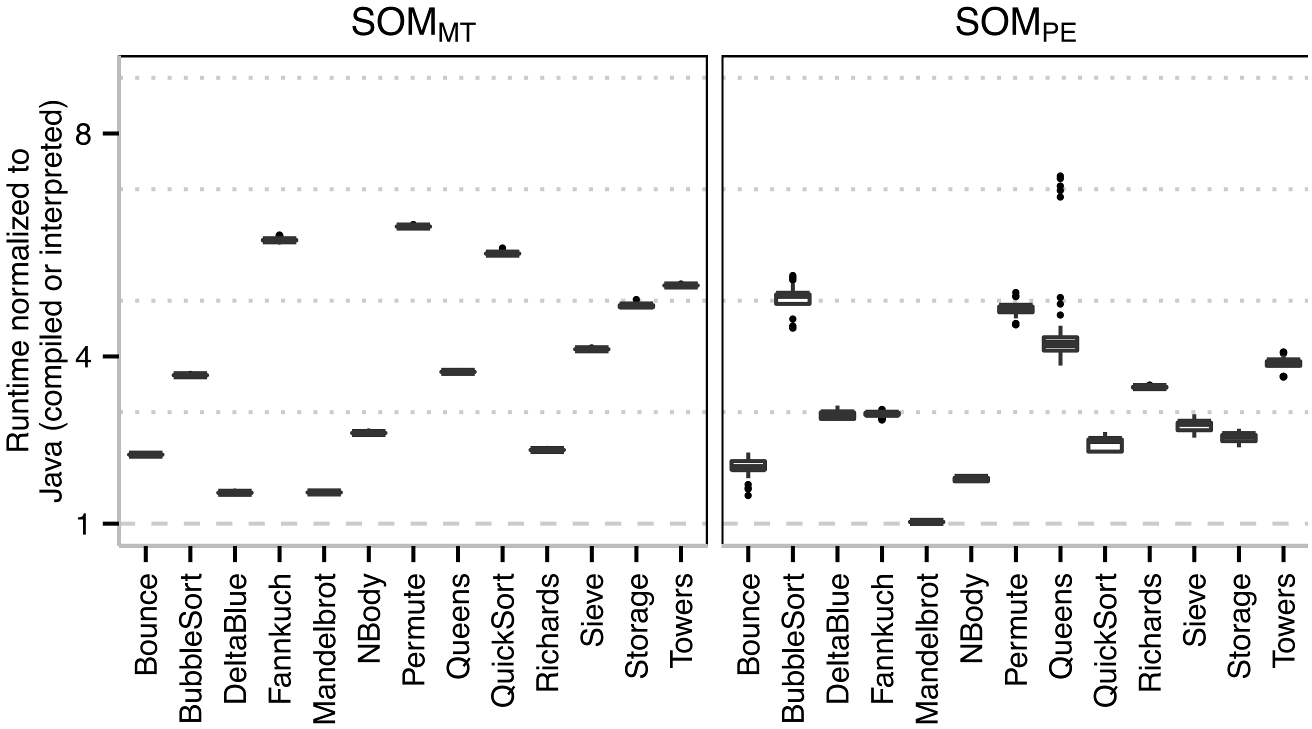
RPython

- #pypy on irc.freenode.net
- rpython.readthedocs.org
- Kermit Example interpreter
  https://bitbucket.org/pypy/example-interpreter
- A Tutorial
  http://morepypy.blogspot.be/2011/04/tutorial-writing-interpreter-with-pypy.html
- Language implementations
  https://www.evernote.com/shard/s130/sh/4d42a591-c540-4516-9911-c5684334bd45/d391564875442656a514f7ece5602210

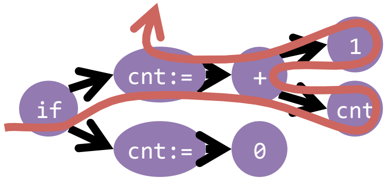
Truffle

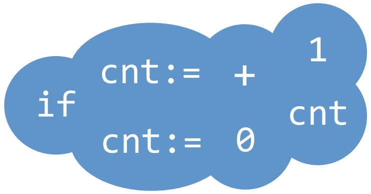
- http://mail.openjdk.java.net/mailman/listinfo/graal-dev
- SimpleLanguage interpreter
  https://github.com/OracleLabs/GraalVM/tree/master/graal/com.oracle.truffle.sl/src/com/oracle/truffle/sl
- A Tutorial
  http://cesquivias.github.io/blog/2014/10/13/writing-a-language-in-truffle-part-1-a-simple-slow-interpreter/
- Project
  - http://www.ssw.uni-linz.ac.at/Research/Projects/JVM/Truffle.html
  - http://www.oracle.com/technetwork/oracle-labs/program-languages/overview/index-2301583.html

Languages: Small, Elegant, and Fast!



3.5x slower	2.8x slower
(min. 1.6x, max. 6.3x)	(min. 3%, max. 5x)
4.2 KLOC	9.8 KLOC
RPython	Truffle+Graal


if
cnt:=
+
1
cnt
cnt:=
0


if
cnt:=
+
1
cnt
cnt:=
0