A Brief Introduction to Just-in-Time Compilation

Creative Commons Attribution-ShareAlike 4.0 License

Stefan Marr

PLISS, May 2026

Got a Question?
Please Interrupt Me!

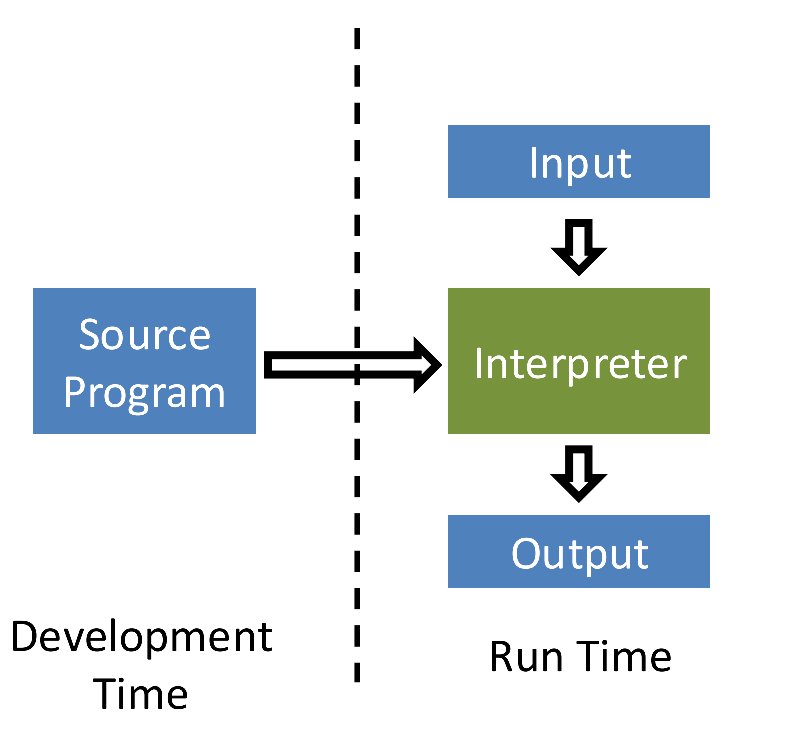
Topics Discussed Today

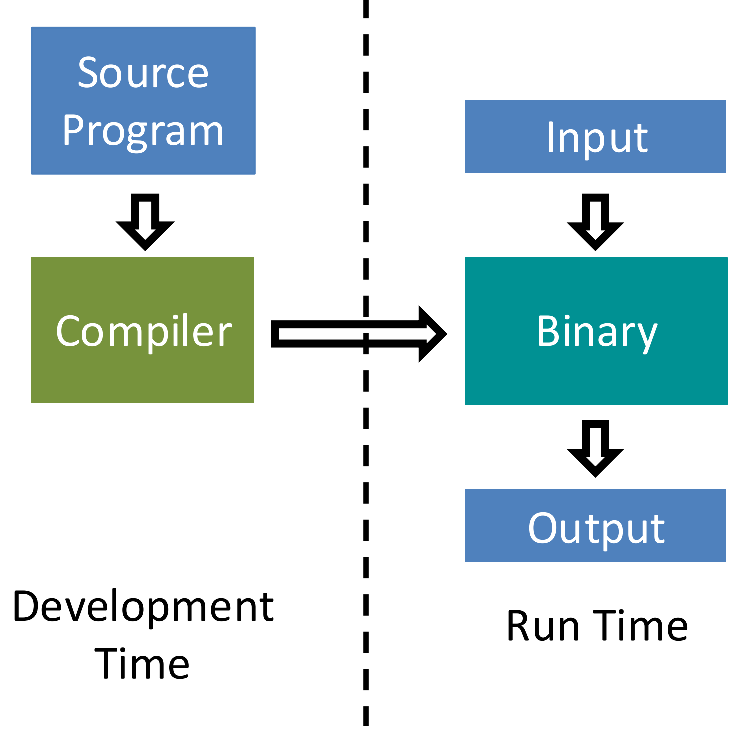
- Just-in-time compilation
  - Basic assumptions and application behavior
  - Selection of compilation units
  - Executing Dynamic Languages
  - Using Run-Time Feedback
  - Metacompilation
- Efficient Data Representation
  - Maps, hidden classes, shapes
  - Storage strategies

I am not using it today, but highly recommended: https://craftinginterpreters.com/

HOW TO IMPLEMENT A PROGRAMMING LANGUAGE?

Language Implementation Approaches



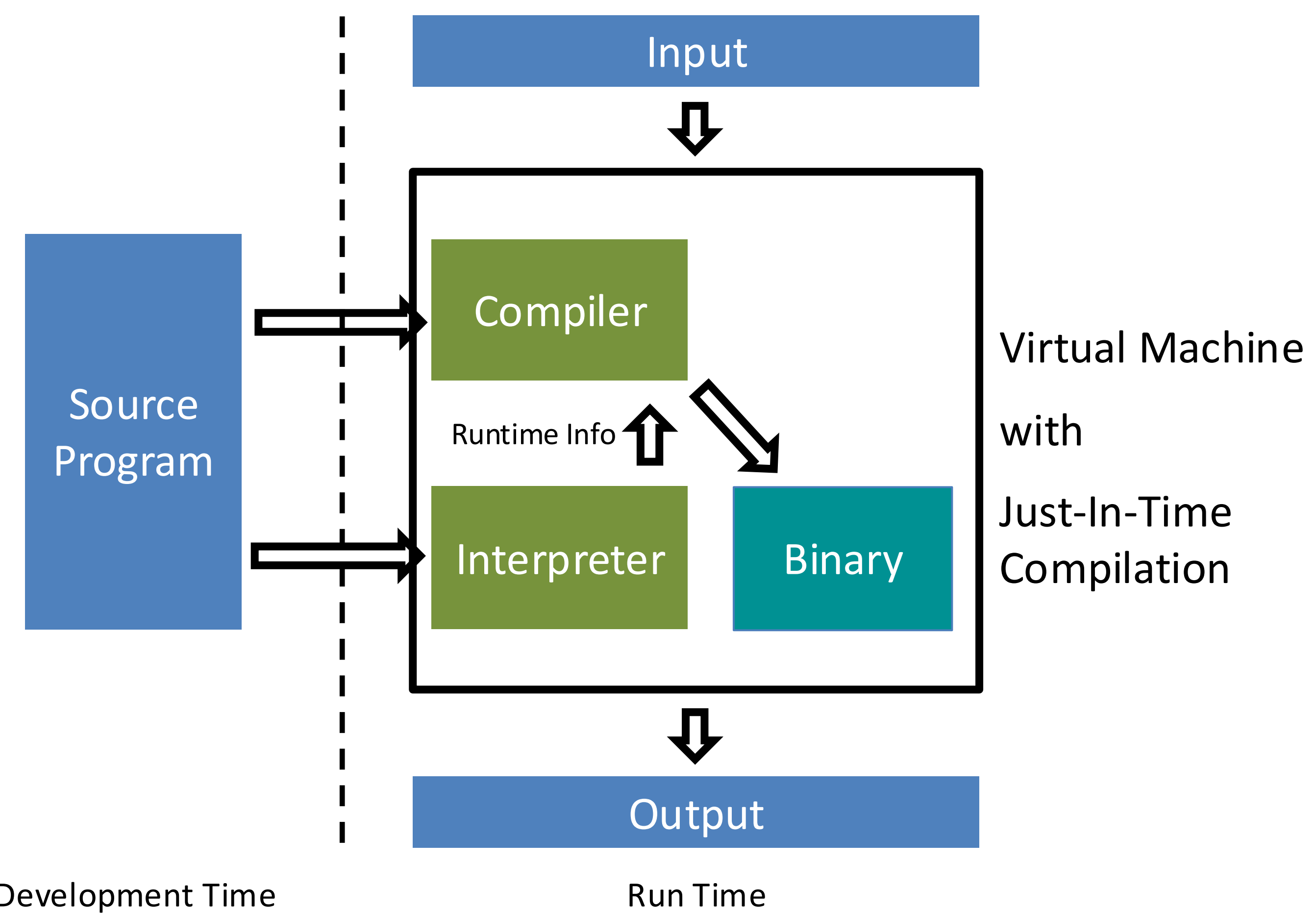


Simple, but often slow

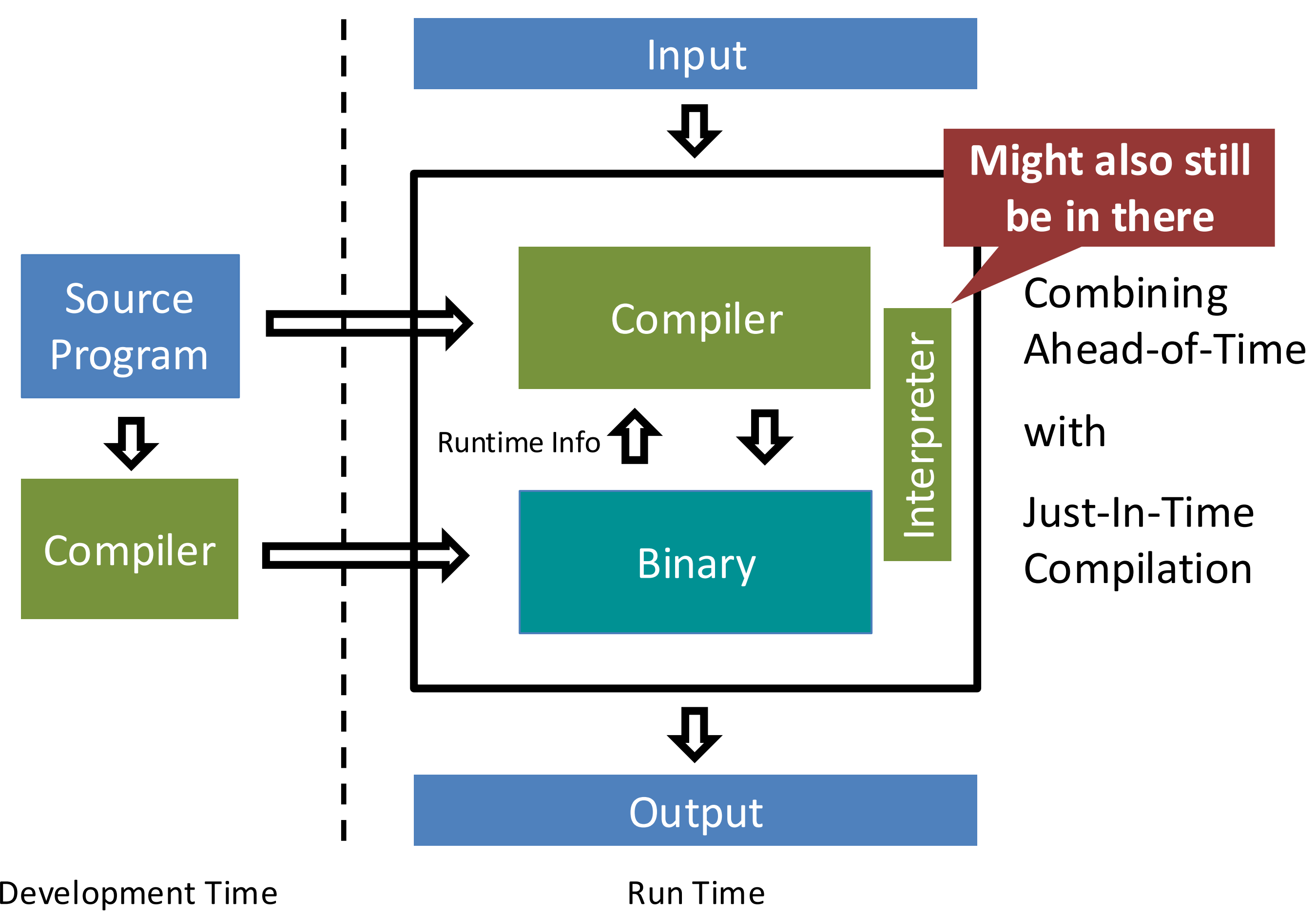
More complex, but often faster

Not ideal for all languages.

Modern Virtual Machines



Virtual Machines with Ahead-of-Time Compilation



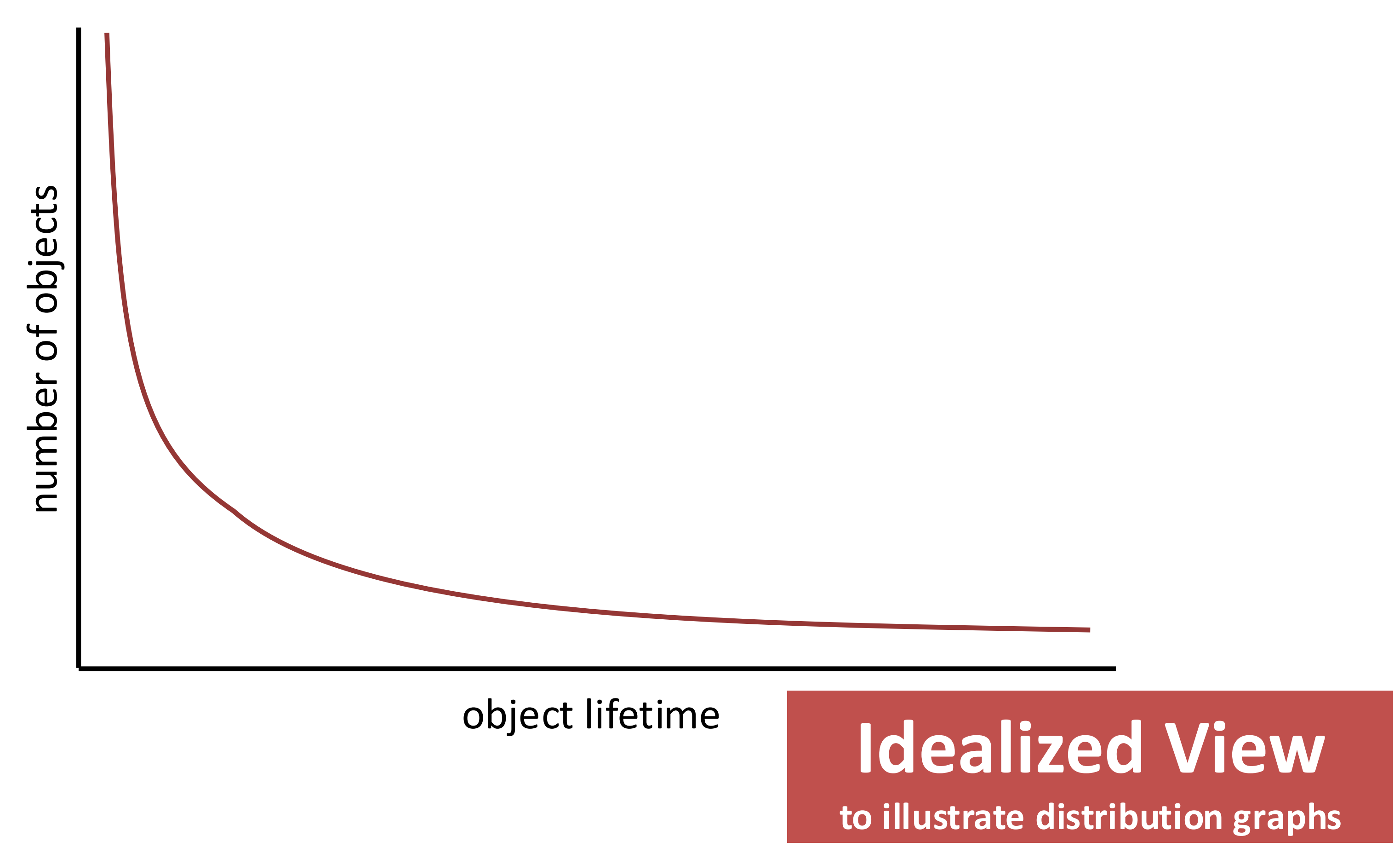
Combining Ahead-of-Time

with

Just-In-Time Compilation

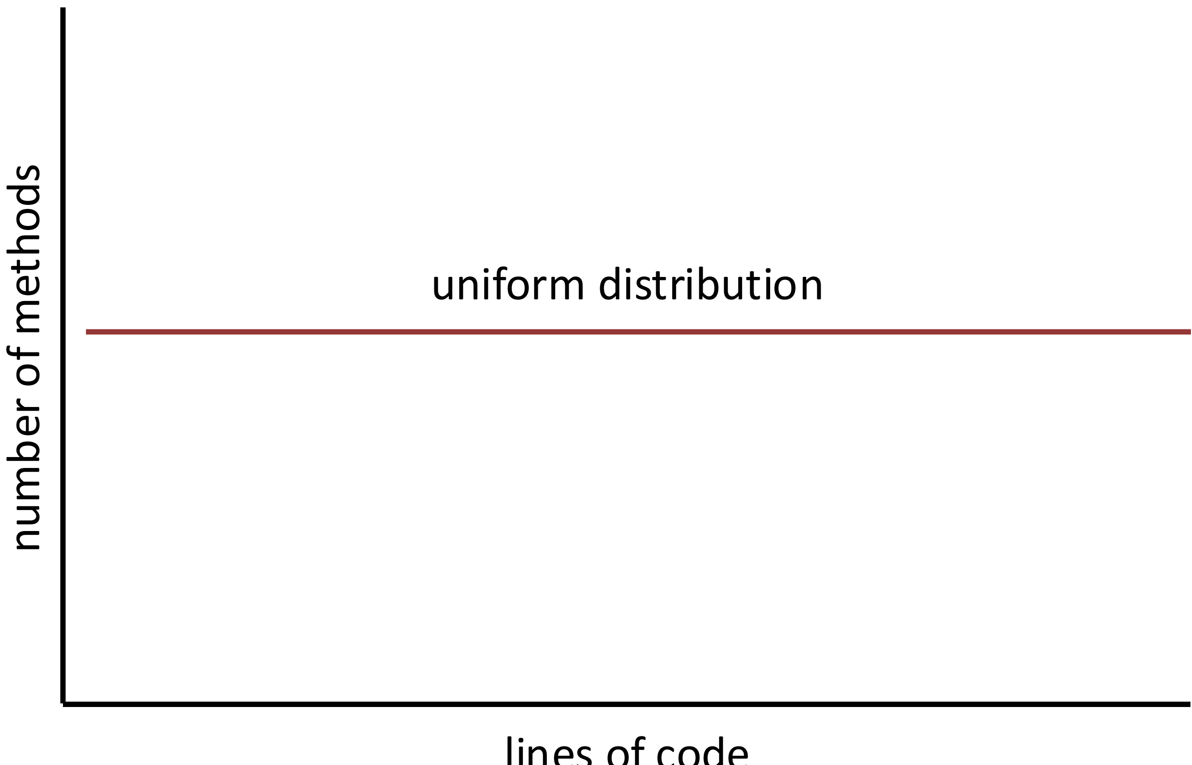
BASIC ASSUMPTIONS AND APPLICATION BEHAVIOR

"Most Objects Die Young" [1]

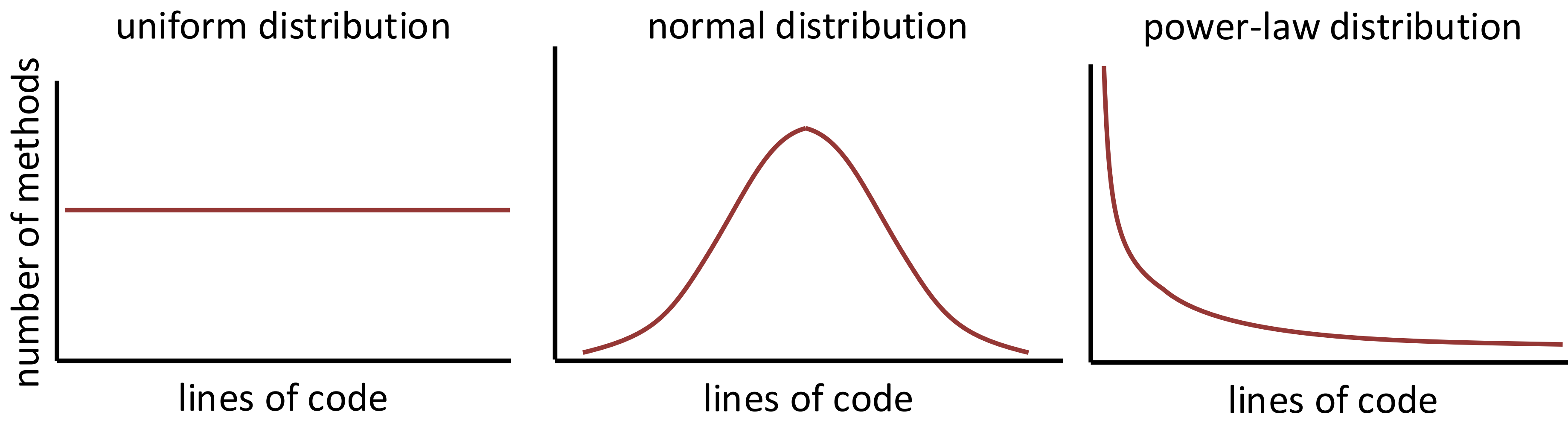


[1] 'Infant Mortality' and Generational Garbage Collection, Henry G. Baker, ACM SIGPLAN Notices, Volume 28, Issue 4, April 1993.

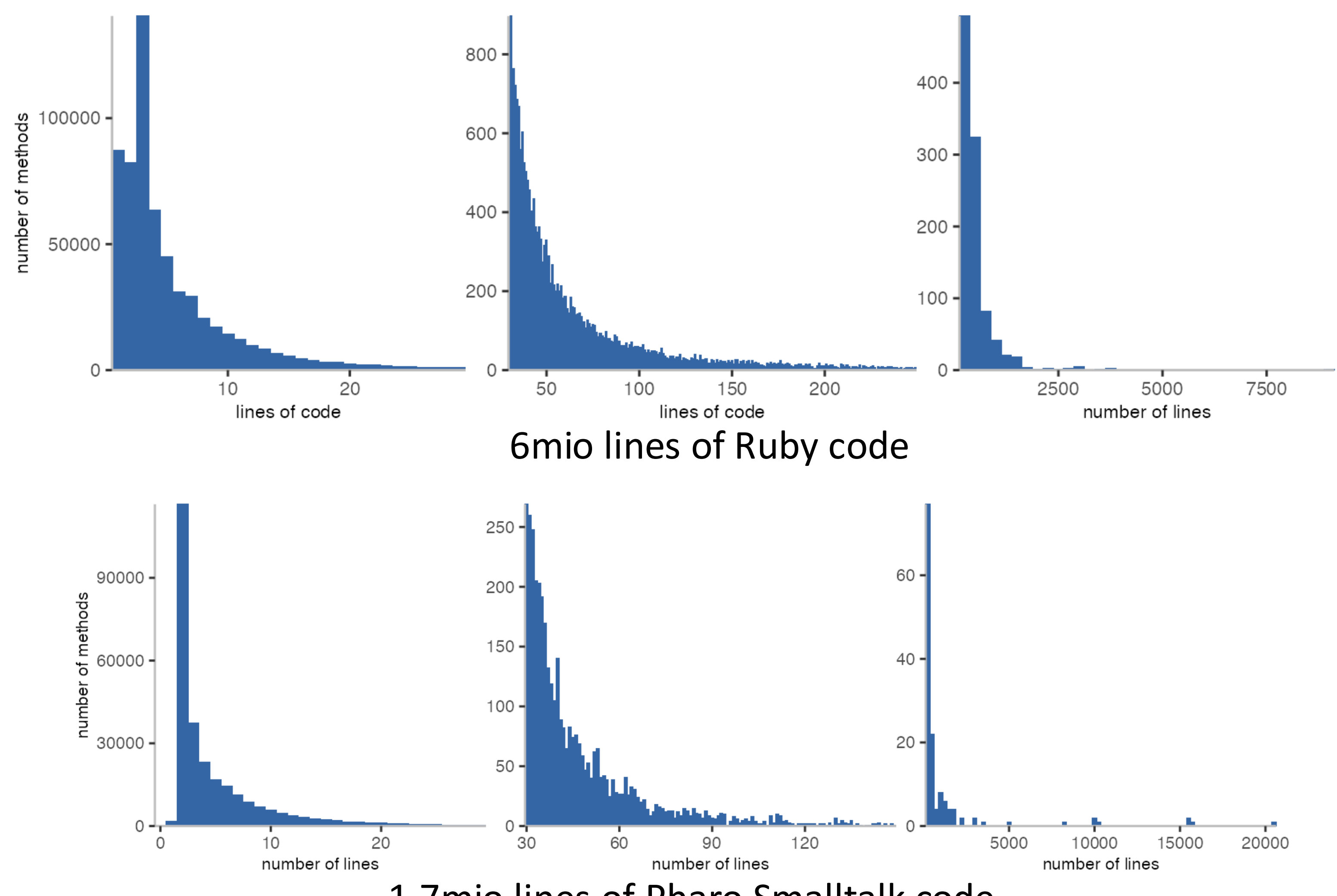
How Big are Methods Usually?



How Big are Methods Usually?

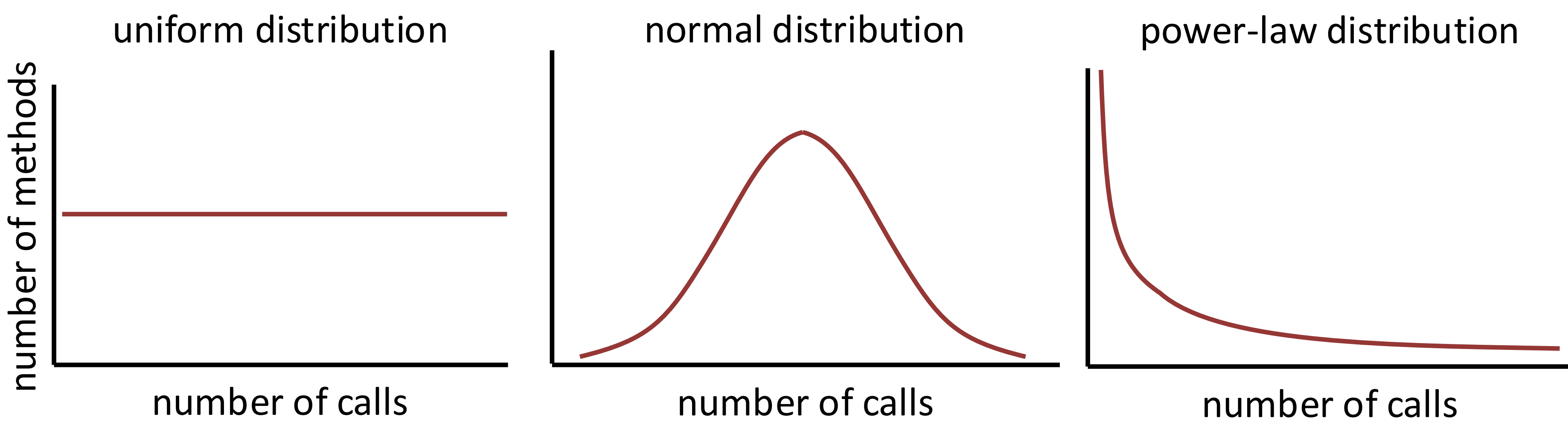


Size of Methods



https://stefan-marr.de/2020/12/shape-of-large-source-code/
https://stefan-marr.de/2020/12/shape-of-large-source-code-ruby/

How Often is a Method Typically Called?



Warning!!!!

You Can’t Trust Me

Always Measure!

It’s Application-Specific

JUST-IN-TIME COMPILATION

Definition

just-in-time compilation,
or dynamic compilation

is

compilation that happens
after a program started

John Aycock. 2003. A brief history of just-in-time. ACM Comput. Surv. 35, 2 (June 2003),

very broad

First Questions

- When does it start?

- What does it compile?

When do we start compiling?

- On 1st activation
- On kth activation, some constant(s)
- On nth activation
  - Dynamic thresholds, based on:
    - system load
    - tasks in compiler queue

- longer execution means more feedback/knowledge about execution
- later compilation: it's fast later, it's slow longer
- different amount of time spent compiling

What could be tradeoffs?

What to Compile?

(what is useful unit of compilation?)

SELECTION OF COMPILATION UNITS

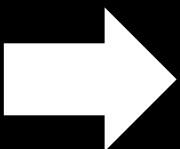
Selection of Compilation Units

- Method-based (Java, C#, JavaScript, Racket, Smalltalk, …)
- Trace-based (PyPy for Python, LuaJIT2)
- Lazy Basic Block Versioning (Ruby YJIT)
- Region-based (PHP, Hack)

- For “dynamic” and ”static” “languages”

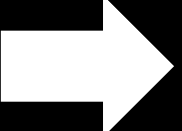
Code Convention

JavaScript-ish



Application Code

Python-ish
with implicit typing



Interpreter Code

Running Example

```
class Widget {
  fitsInto(width) {
    return this.width <= width;
  }
}

function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

Let's assume simplified JavaScript

Which Lines Are Hot (executed often)?

```
1   class Widget {
2     fitsInto(width) {
3       return this.width <= width;
4     }
5   }
8
9   function findAllThatFit(arr, width) {
10    const result = [];
11    for (const w of arr)
12      if (w.fitsInto(width))
13        result.push(w)
14    return result;
15  }
```

Hot: > 10 executions
for an arr
with 100 elements

?

Which Lines Are Hot (executed often)?

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
8
9  function findAllThatFit(arr, width) {
10   const result = [];
11   for (const w of arr)
12     if (w.fitsInto(width))
13       result.push(w)
14   return result;
15 }
```

Now, a VM can know what to focus on. Important: run time should be spent on the application

Method-based Compilation

```
class Widget {
  fitsInto(width) {
    return this.width <= width;
  }
}
```

Compilation unit
made of “method bodies”

```
function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

Method inlining controlled by heuristics, enables many optimizations

Method-based Compilation

```
class Widget {
  fitsInto(width) {
    return this.width <= width;
  }
}
```

Compilation unit made of "method bodies"

```
funct
  cor
  for
    i

  return result;
}
```

```
function compUnit(arr, width) {
  const result = [];
  for (const w of arr)
    if (isWidget(w))
      if (w.width <= width)
        result[result.length] = w;
  return result;
}
```

Method inlining, controlled by heuristics, enables many optimizations

Definition: Method-based Compilation

- Compilation unit is selected from a root method
- Inlining, based on heuristics, decides when to include other methods into the compilation unit
- Typically, all control flow paths are included

Trace-based Compilation

```
class Widget {
  fitsInto(width)
    return this.width <= width;
  }
}

function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

Starts at loop

Compilation unit is instruction trace

```
v1 = arr[i]
isWidget(v1)
v2 = v1.width
v3 = v2 <= width
isTrue(v3)
v4 = result.length
result[v4] = v1
```

Only a single path.
Control flow decisions checked by guards.
Easy to optimize, because traces contain no control flow

Definition: Trace-based Compilation

- Compilation unit is selected by following a single concrete execution
  - Only a single path is included
  - Starting from the top of a loop or function
- Tracing continues until returning to starting point, or aborting when limit reached
- If execution leaves the recorded path, it leaves the compilation unit

Lazy Basic Block Versioning

```
class Widget {
  fitsInto(width) {
    return this.width <= width;
  }
}

function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

What's a basic block?

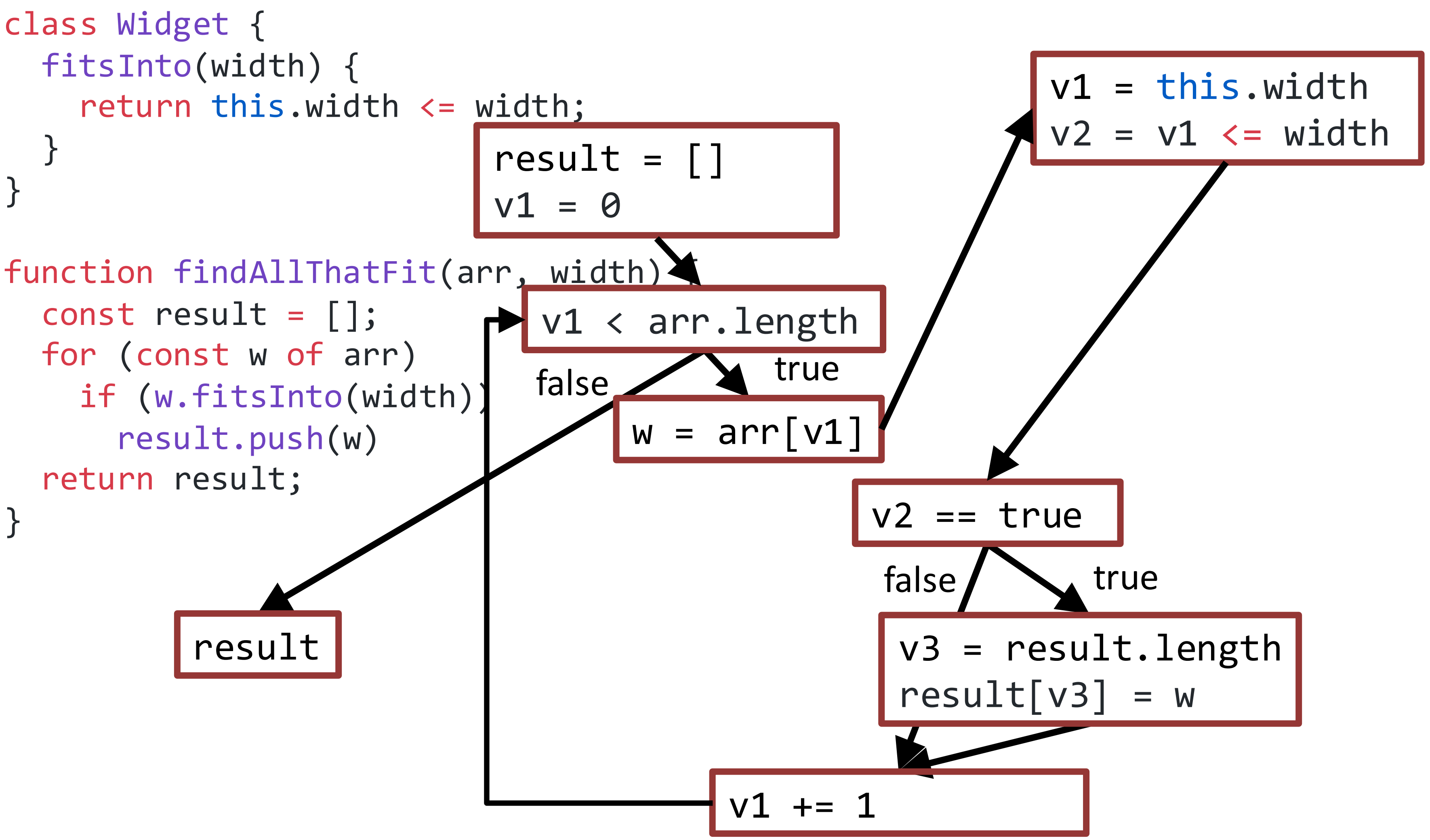
Basic Block

- One entry point
- One exit point

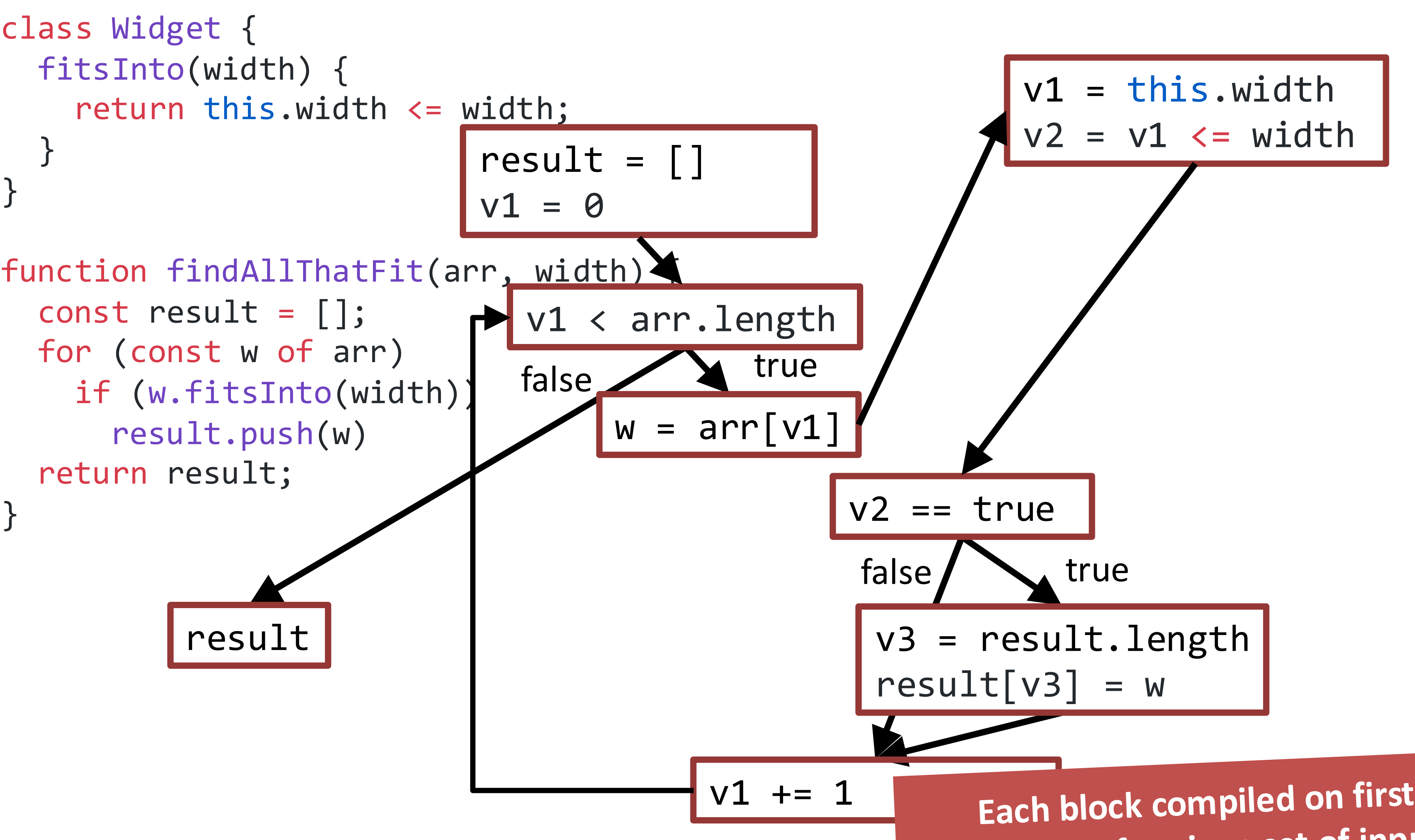
```
v1 = this.width
v2 = v1 <= width
```

- A straight line of code
- No jumps into the middle of it

Basic Blocks for Our Example



Lazy Basic Block Versioning

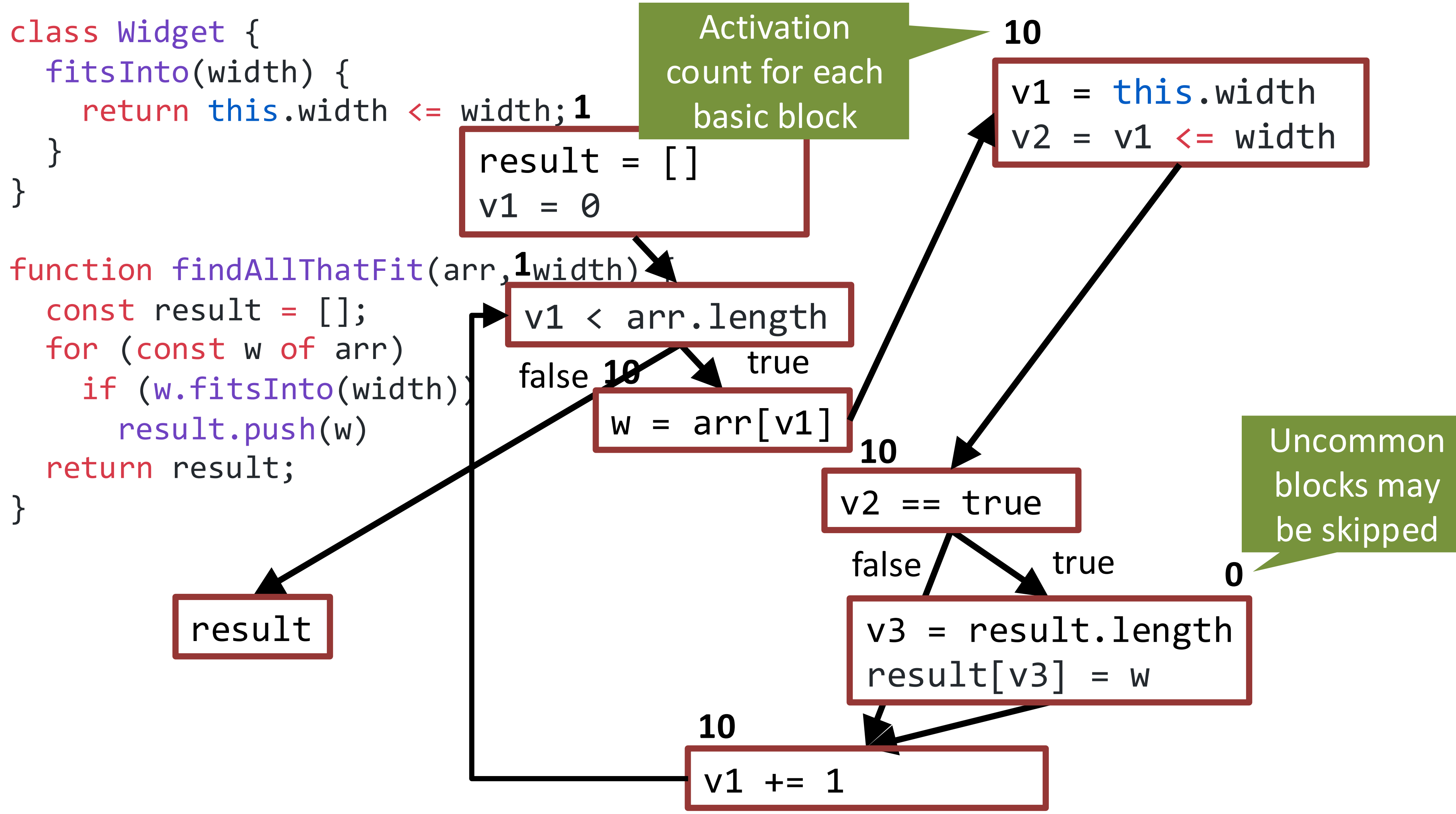


Each block compiled on first execution for given set of input types. Specialized to input variables

Definition: Lazy Basic Block Versioning

- Compilation unit is single basic block
- Can have multiple specializations
  - i.e., versioning based on types or values
- No control flow included
  (because it's a basic block)

Region-based Compilation



Definition: Region-based Compilation

- Compilation unit is selected based on heuristics, starting from an initial basic block
  - Typically, the start of a function
- Heuristics decide which basic blocks to include
  - Prefer “hot”/frequently executed blocks
  - Selection can cross function boundaries, i.e., do (partial) inlining
- Can include control flow, but may not include whole methods

Open Research Questions

Are there other sensible units of compilation?

Do we need baseline JIT compilers, or can we make interpreters fast enough?

Research and Literature

- **Method-based**
  - **An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes.** Chambers, C., Ungar, D. & Lee, E. (1989). OOPSLA'89
  - **The Jalapeño Dynamic Optimizing Compiler for Java.** Burke, M. G., Choi, J.-D., Fink, S., Grove, D., Hind, M., Sarkar, V., Serrano, M. J., Sreedhar, V. C., Srinivasan, H. & Whaley, J. (1999). *JAVA'99*
- **Region-based**
  - **HHVM JIT: A Profile-guided, Region-based Compiler for PHP and Hack.** Ottoni, G. (2018). *PLDI'18*
- **Trace-based**
  - **Dynamo: A Transparent Dynamic Optimization System.** Bala, V., Duesterwald, E. & Banerjia, S. (2000). *PLDI'00*
  - **Trace-based Just-in-time Type Specialization for Dynamic Languages.** Gal, A., Eich, B., Shaver, M., Anderson, D., Mandelin, D., Haghighat, M. R., Kaplan, B., Hoare, G., Zbarsky, B., Orendorff, J., Ruderman, J., Smith, E. W., Reitmaier, R., Bebenita, M., Chang, M. & Franz, M. (2009). *PLDI'09*
- **Basic Block Versioning**
  - Interprocedural Type Specialization of JavaScript Programs Without Type Analysis. Chevalier-Boisvert, M. & Feeley, M. (2016). *ECOOP'16*

USING RUN-TIME FEEDBACK

Which other details can we know at run time?

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
6  class Button extends Widget {}
7  class RadioButton extends Button {}
8
9  function findAllThatFit(arr, width) {
10   const result = [];
11   for (const w of arr)
12     if (w.fitsInto(width))
13       result.push(w)
14   return result;
15 }
```

?

Which other details can we know at run time?

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
6  class Button extends Widget {}
7  class RadioButton extends Button {}
8
9  function findAllThatFit(arr, width) {
10   cons                 [];
11   for (const w of arr)
12     if (w.fitsInto(width))
13       result.push(
14   return result;
15 }
```

[]{width: int, fitsInto()}

int

{width: int, fitsInto()}

In large programs, things are not guaranteed: speculate and hope for the best

Applying Optimizations

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
8
9  function findAllThatFit(arr, width) {
10   const result = [];
11   for (const w of arr)
12     if (w.fitsInto(width))
13       result.push(w)
14   return result;
15 }
```

Examples for illustration, details will vary

Applying Optimizations: Loop Boundary

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
8
9  function findAllThatFit(arr, width) {
10   const arrLen = arr.length;
11   const result = [];
12   for (const w of arr)
13     if (w.fitsInto(width))
14       result.push(w)
15   return result;
16 }
```

Applying Optimizations: Simplify Loop

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
8
9  function findAllThatFit(arr, width) {
10   const arrLen = arr.length;
11   const result = [];
12   for (let i = 0; i < arrLen; i++)
13     if (w.fitsInto(width))
14       result.push(w)
15   return result;
16 }
```

JavaScript's for..of loop is complicated

Applying Optimizations: Preallocate Array

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
8
9  function findAllThatFit(arr, width) {
10   const arrLen = arr.length;
11   const result = [null, … /* arrLen elements */];
12   for (let i = 0; i < arrLen; i++)
13     if (w.fitsInto(width))
14       result.push(w)
15   return result;
16 }
```

Applying Optimizations: Inline

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
8
9  function findAllThatFit(arr, width) {
10   const arrLen = arr.length;
11   const result = [null, … /* arrLen elements */];
12   for (let i = 0; i < arrLen; i++)
13     if (isWidget(w) && w.w      <= width)
14       result.push(
15   return result;
16 }
```

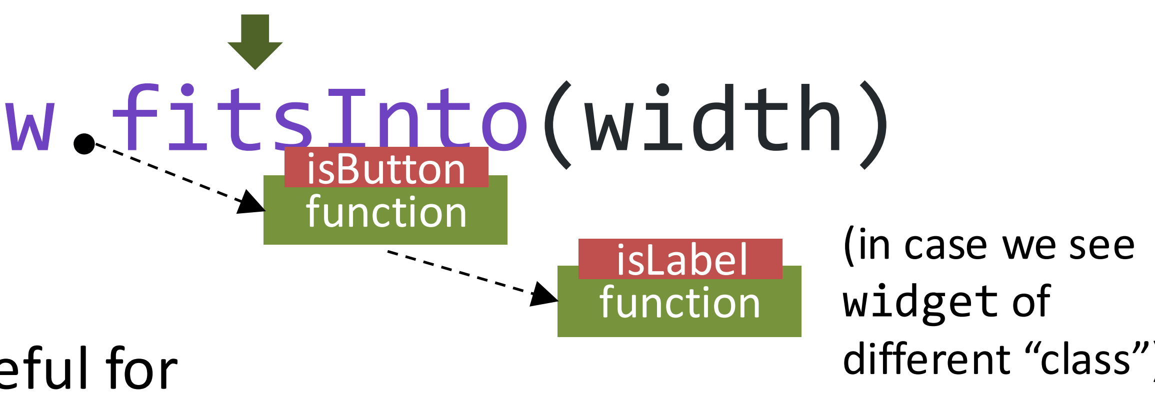
Inlining: often most important optimization, enabling further optimization

How can we know the method for inlining?

```
w.fitsInto(width)
```

Solution: Lookup Caching

could be various functions,
but we don't need to do the same lookup repeatedly



Useful for

- Method/function/operator lookup
- Field access
- Getters, setters, method returning constants, other "trivial" methods
- Objec
- Reflec
  - cac

Essential for any form of run-time polymorphism

Programs Utilize Potential Dynamicity only in Few Cases

Main Insight

Philosophical Question:
Should languages limit dynamicity to simplify implementation?

See for example:

Hölzle, U., Chambers, C. & Ungar, D. (1991). Optimizing Dynamically-Typed Object-Oriented Languages With Polymorphic Inline Caches. *ECOOP '91: European Conference on Object-Oriented Programming.*

a few interesting papers

- **Method-based**
  - **ShareJIT: JIT code cache sharing across processes and its practical implementation.** Xiaoran Xu, Keith Cooper, Jacob Brock, Yan Zhang, and Handong Ye. *OOPSLA'18*
  - **Reusing Just-in-Time Compiled Code.** Meetesh Kalpesh Mehta et al. *OOPSLA'23*
- **Region-based**
  - **HHVM jump-start: boosting both warmup and steady-state performance at scale.** Guilherme Ottoni and Bin Liu. *CGO '21.*
- **Trace-based**
  - **Amalgamating different JIT compilations in a meta-tracing JIT compiler framework.** Yusuke Izawa and Hidehiko Masuhara. *DLS'20*
- **Basic Block Versioning**
  - **Evaluating YJIT's Performance in a Production Context: A Pragmatic Approach.** Chevalier-Boisvert, M. et al. *MPLR'23*

Tangent/Recap

EXECUTING DYNAMIC LANGUAGES

Dynamic Languages

```
cnt + 1
```

What needs to happen to add one to some value (in JavaScript)?

?

Dynamic Languages

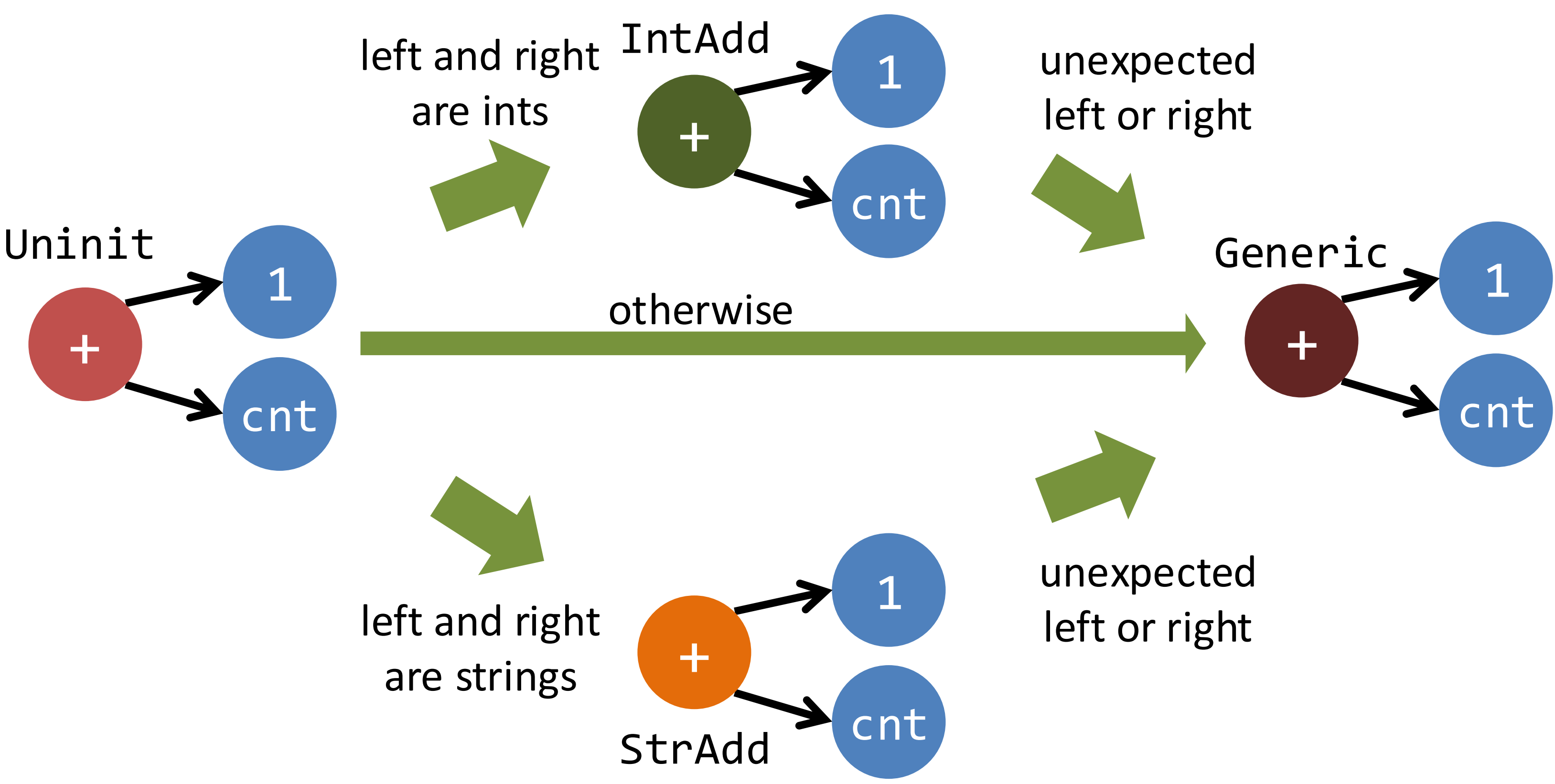
```
cnt + 1
```

ECMAScript Specification Sec. 11.6.1

```
left = ToPrimitive(GetValue(cnt))
right = ToPrimitive(GetValue(1))

if (IsString(left) || IsString(right)) {
  return ToString(left).concat(ToString(right))
}

return ToNumber(left) + ToNumber(right)
```

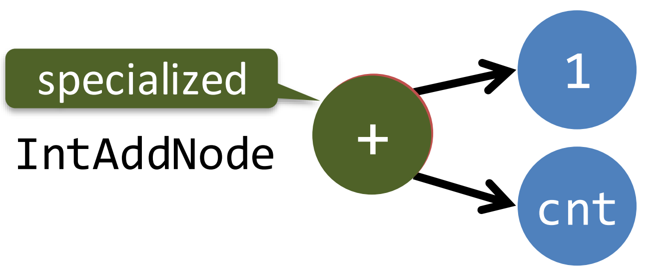
SPECULATIVE OPTIMIZATION IN INTERPRETERS

Self Optimization and Speculation



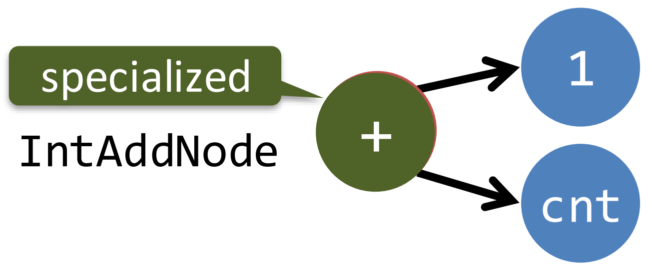
observe run-time value
select optimization of first execution

Self Optimization and Speculation




```
class UniniAdditionNode(BinaryNode):
def execute(frame):
  lVal = left.execute(frame)
  rVal = right.execute(frame)
  if type(lVal) == int and type(rVal) == int:
    return replace(IntAddNode(self)).
                  execute_evaluate(lVal, rVal)
```

Self Optimization and Speculation




```
class IntAddNode(BinaryNode):
  def execute(frame):
    lVal = left.execute(frame)
    rVal = right.execute(frame)
    if type(lVal) == int and type(rVal) == int:
      return lVal + rVal
    else:
      return deoptimize(lVal,
```

Simplified Version

Self Optimization and Speculation

specialized

IntAddNode

\+

1

cnt

```
class IntAddNode(BinaryNode):
  def execute(frame):
    try:
      lVal = left.execute_int(frame)
    except UnexpectedResult, exp:
      return deoptimize(exp)
    try:
      rVal = right.execute_i
    except UnexpectedResult,
      return deoptimize(lVal
    return lVal + rVal
```

Check moved down tree, and potentially eliminated*

Some Possible Self-Optimizations

- Type profiling and specialization

`cnt + 1`

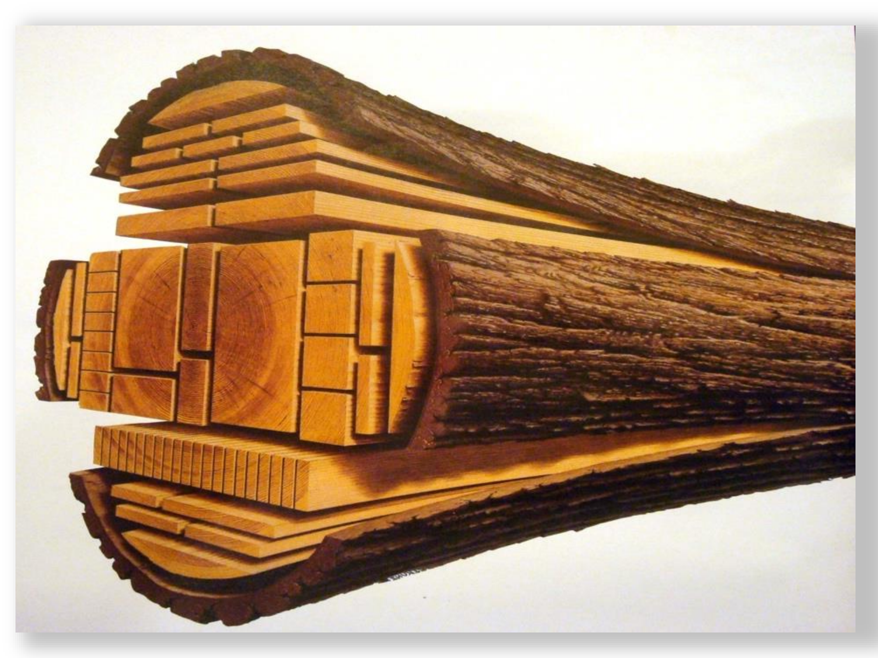
- Lookup caching

- Value caching
- Operation inlining

- Library Lowering

Research and Literature

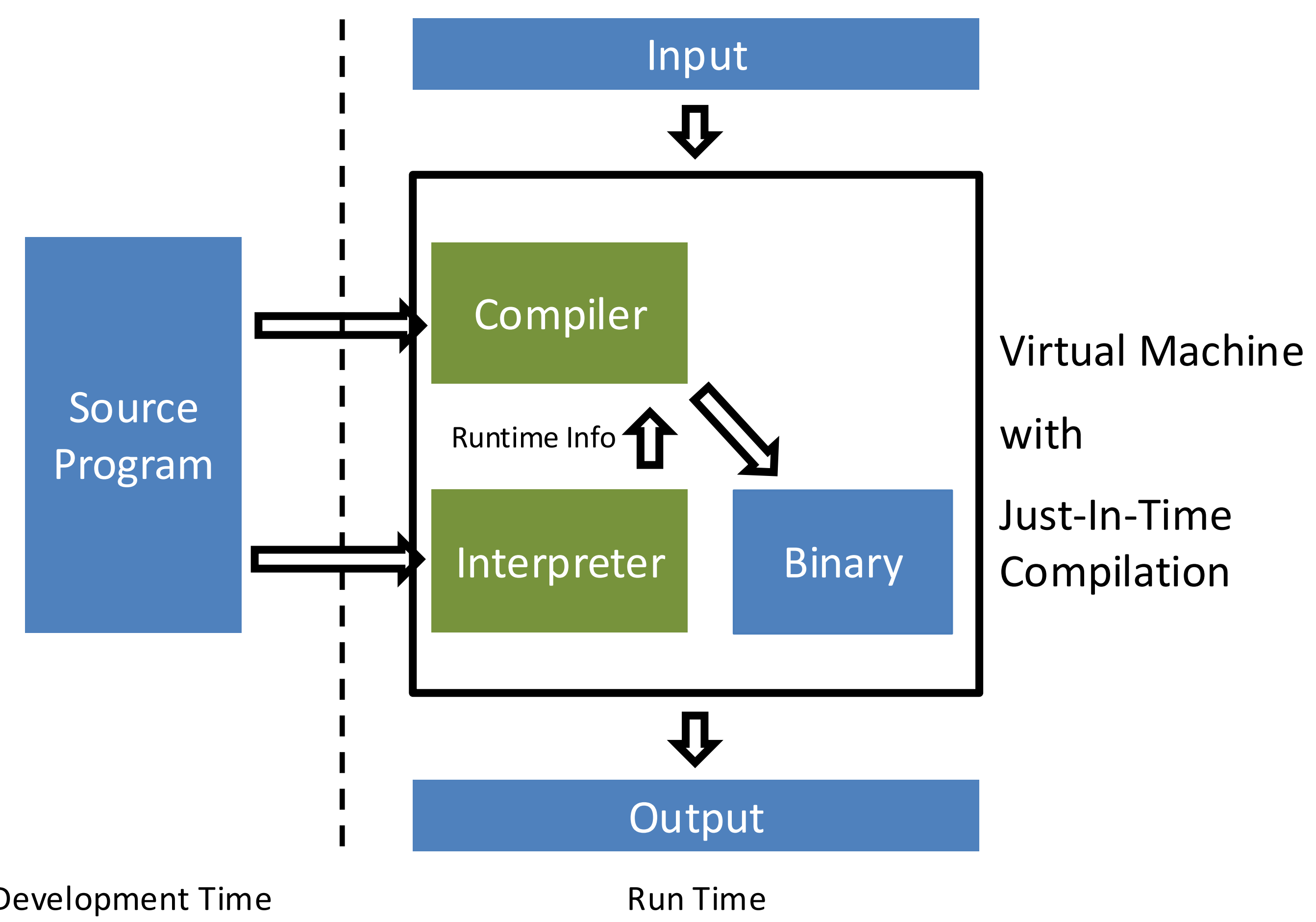
- **AST interpreters**
  - **Self-Optimizing AST Interpreters**
    Würthinger, T., Wöß, A., Stadler, L., Duboscq, G., Simon, D. & Wimmer, C. (2012). *DLS'12*
- **Bytecode interpreters**
  - **Efficient Interpretation Using Quickening**
    Brunthaler, S. (2010). *DLS'10*
  - **Optimizing Indirect Branch Prediction Accuracy in Virtual Machine Interpreters**
    Casey, K., Ertl, M. A. & Gregg, D. (2007). *ACM Trans. Program. Lang. Syst.*, 29, 37.
- **JIT compilers, deoptimization**
  - **An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes.**
    Chambers, C., Ungar, D. & Lee, E. (1989). OOPSLA'89



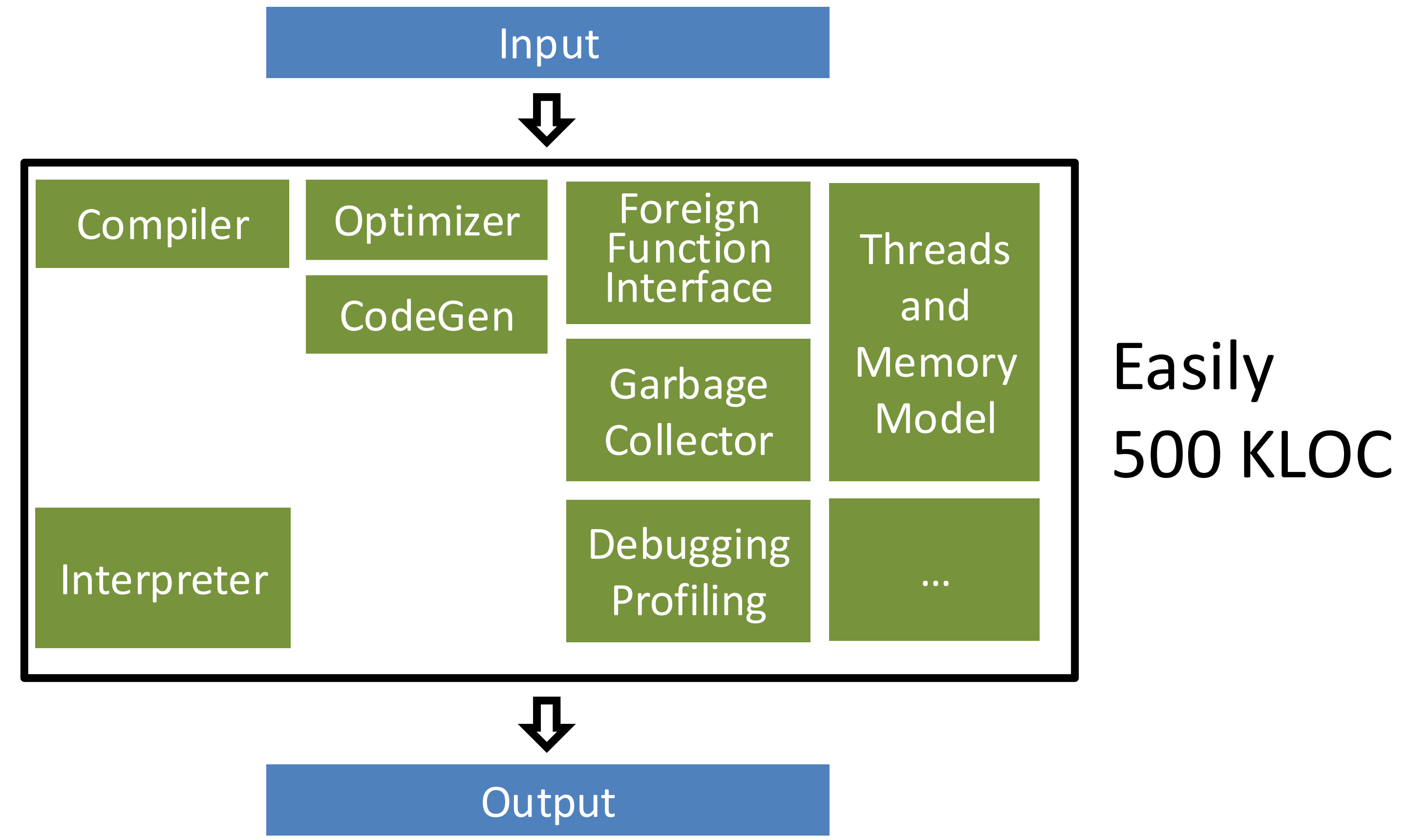
Generating Efficient Native Code

JUST-IN-TIME COMPILATION FOR INTERPRETERS

Modern Virtual Machines

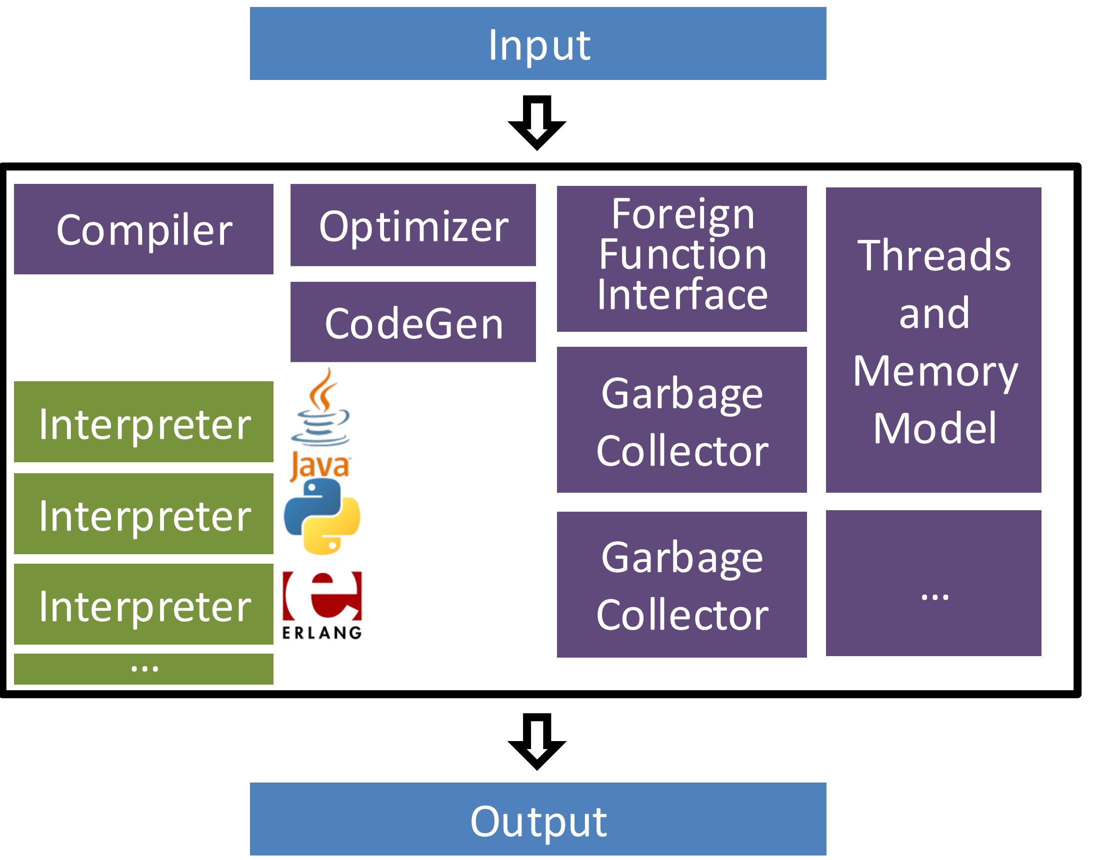


VMs are Highly Complex



How to reuse most parts for a new language?

How to reuse most parts for a new language?

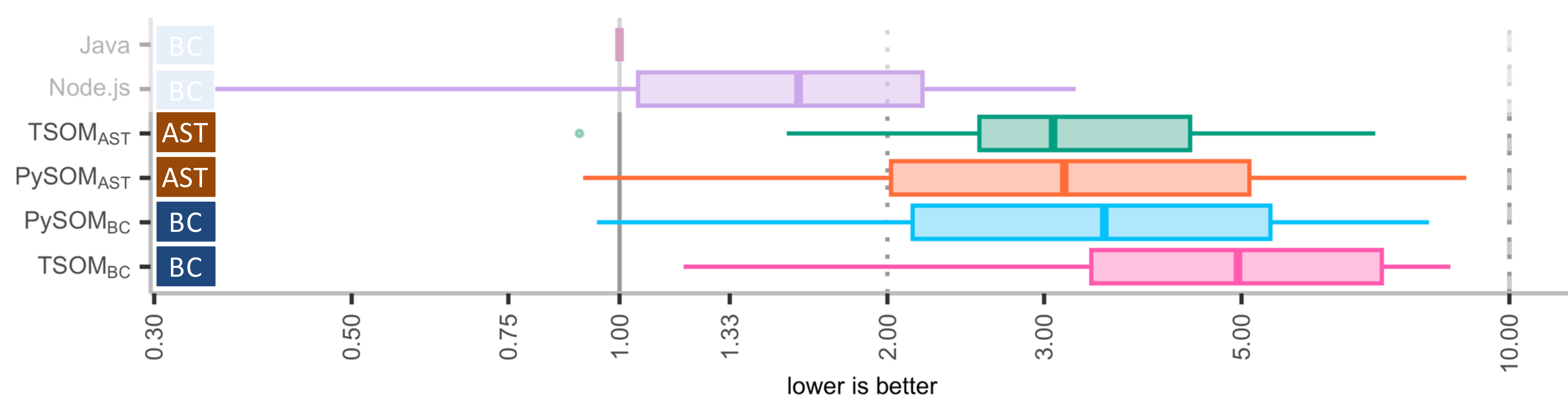


Make Interpreters Replaceable Components!

Does it give us performance?

Bing Image Creator

AST vs Bytecode Interpreters for Metacompilation Systems

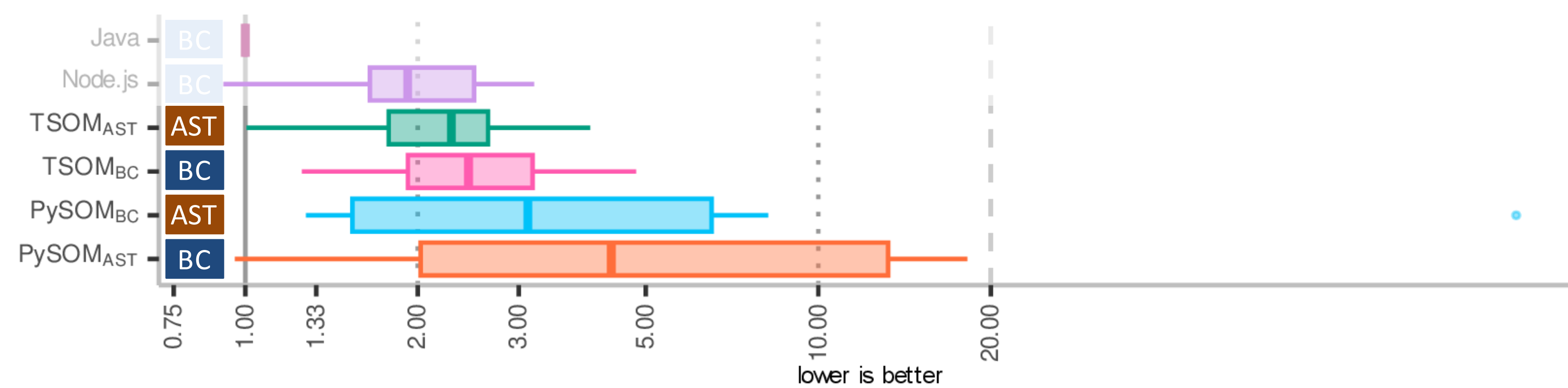


AST interpreters can be surprisingly fast!
Though, bytecodes are much more compact in memory.

Pure interpretation, no compilation

AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. O. Larose, S. Kaleba, H. Burchell, S. M
https://stefan-marr.de/downloads/oopsla23-larose-et-al-ast-vs-bytecode-interpreters-in-the-age-of-

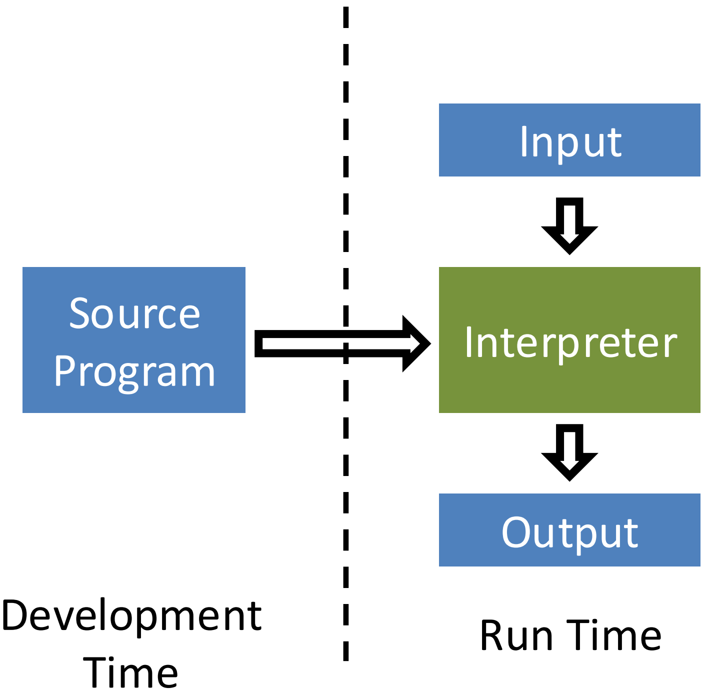
Peak Performance on Metacompilation Systems



We reach into the range of state-of-the-art, custom-built systems!

AST vs. Bytecode: Interpreters in the Age of Meta-Compilation. O. Larose, S. Kaleba, H. Burchell, S. M
https://stefan-marr.de/downloads/oopsla23-larose-et-al-ast-vs-bytecode-interpreters-in-the-age-of-

How to get Fast Program Execution?



Basic Abstract Syntax Tree Interpreter
One execute() method per AST Node

```
log("Hello");
```

CallFn
Constant "Hello"
LoadGlobal log

```
class Constant:
    final value
    def execute(frame):
        return value


class LoadGlobal:
    final binding
    # ...
    def execute(frame):
        return binding.value
```

```
class CallFn:
    node fn
    node argument
    def execute(frame):
      fn = fn.execute(frame)
      arg = argument.execute(frame)
      return call(fn, arg)
```

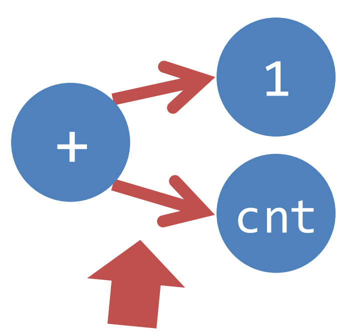
How to Get Fast Program Execution?

Standard Compilation:
1 interpreter method at a time

```
Constant.execute(frame)               ..Const_execute() # bin
LoadGlobal.execute(frame)             ..LG_execute()    # bin
CallGlobal.execute(frame)             ..CG_execute()    # bin
IntAddNode.execute(frame)             ..IAN_execute()   # bin
ArrayNewWithValue.execute(frame)      ..ANWV_execute()  # bin
```

Minimal Optimization Potential

Problems with Compiling an AST Interpreter



Slow Polymorphic Dispatches

```
class IntAddNode(BinaryNode):
  def execute(frame):
    try:
      lVal = left.execute_int(frame)
    except Unexp
      return c
    # ...
```

Runt

For bytecode interpreters, problem moves to bytecode loop, which is often huge

For Fast Execution

Code of interpreter

```
Constant.execute(frame)
LoadGlobal.execute(frame)
CallGlobal.execute(frame)
IntAddNode.execute(frame)
ArrayNewWith.execute(frame)
```

Interpreter

```
function findAllThatFit(arr, width) {
  const result = [];
  for (const w of arr)
    if (w.fitsInto(width))
      result.push(w)
  return result;
}
```

Application Program

Compile the Program,
not the Interpreter!

Meta Compilation

RPython
with Meta-Tracing

Truffle + Graal
with Partial Evaluation

GraalVM™

Main Approaches

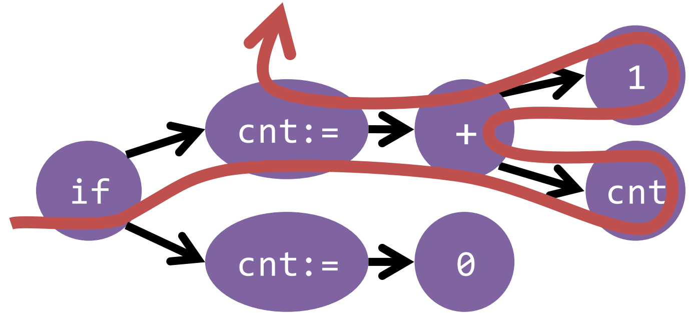
Method-based

- Monitors hot methods and loops
- Compiles these *methods*
  - Includes all control flow

if
cnt:= +
1
cnt
cnt:= 0

Trace-based

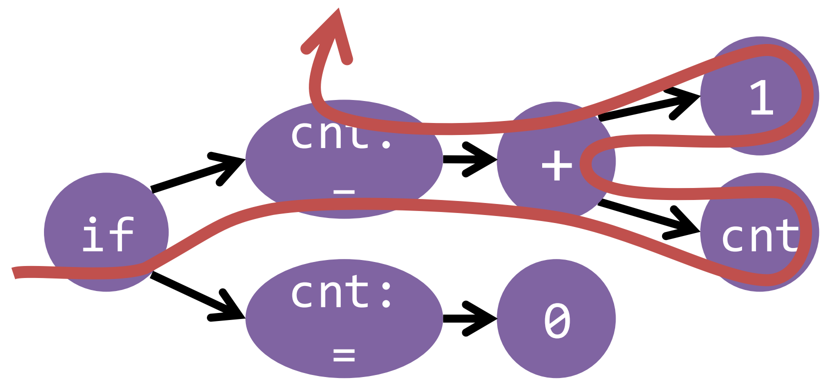
- Monitors hot loops
- Records concrete hot paths ('traces')
  - ignoring method boundaries



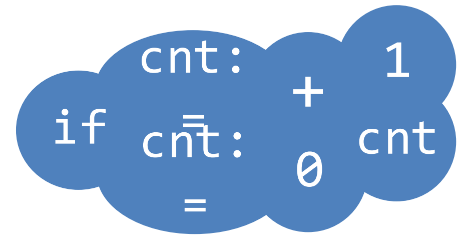
- compiles *traces* with exit points in case execution leaves recorded path
  - e.g. expected an integer but got a string

Compilation Unit based on User Program

Meta-Tracing



Partial Evaluation Guided By AST



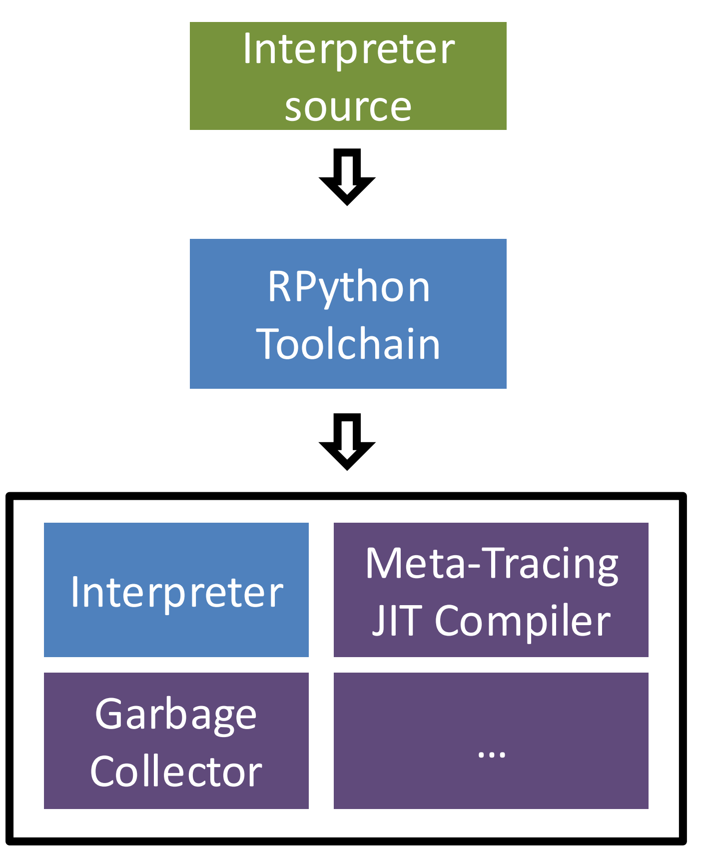
What do we include in a compilation?

- run time is precious
- larger units take longer to compile
- might give more benefits

Just-in-Time Compilation with Meta Tracing

RPython

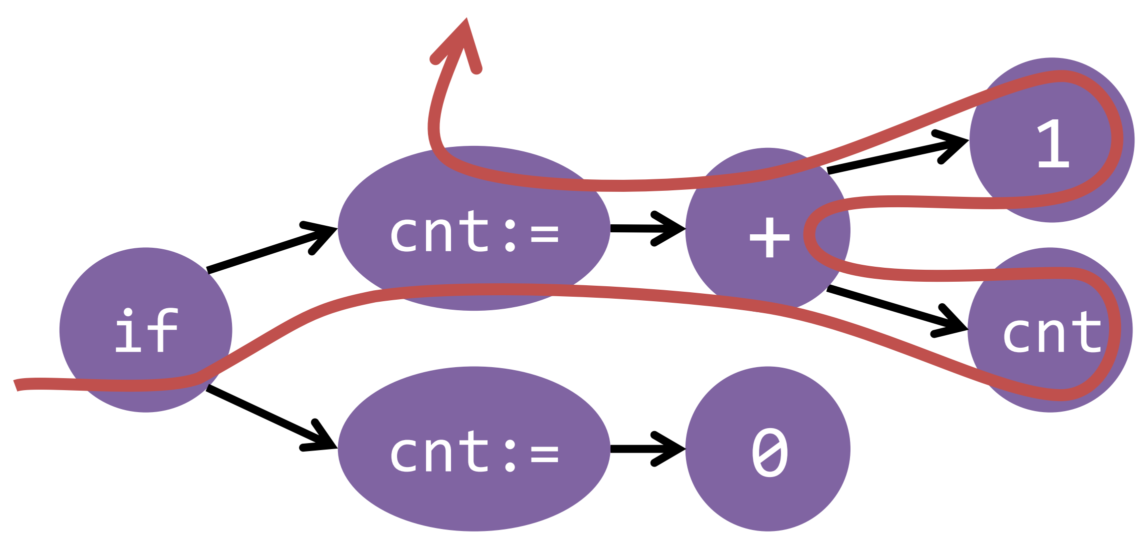
RPython



- Subset of Python
  - Type-inferenced
- Generates VMs

http://rpython.readthedocs.org/

Meta-Tracing of an Interpreter



Meta Tracers need to know the Loops

```
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))
```

Tangent

Guards

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

- Document control-flow choices
- Trace is linear, no branches
- On failure:
  - Return to interpreter, deoptimize
  - Or, if frequen

Unsolved research issue:
How to avoid trace explosion, or reintegrate control flow

Meta Tracers need to know the Loops

```
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))
```

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))
```

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()
    except 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

```
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))
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

```
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)
...
```

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)

...
```

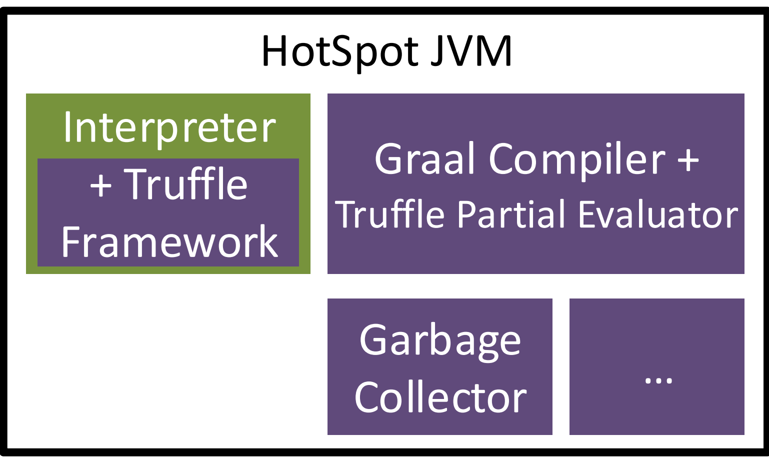
```
i1 := left_expr.idx # Const(1)
a1 := frame.layout
i1 := a1[Const(1)]
guard(i1 == Const(F_INT))

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

Truffle+Graal



- Java framework
  - AST interpreters
- Based on HotSpot JVM

http://graalvm.org

Partial Evaluation Guided By AST

if
cnt:=
+
1
cnt
cnt:=
0

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)
```

Tangent

Deoptimization of Compiled Code

`__deopt_return_to_interp()`

- Equivalent to failing guard
- Needs to keep metadata to reconstruct state for unoptimized execution
  - Variables might have been optimized out
  - Objects may not have been allocated
  - …

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:

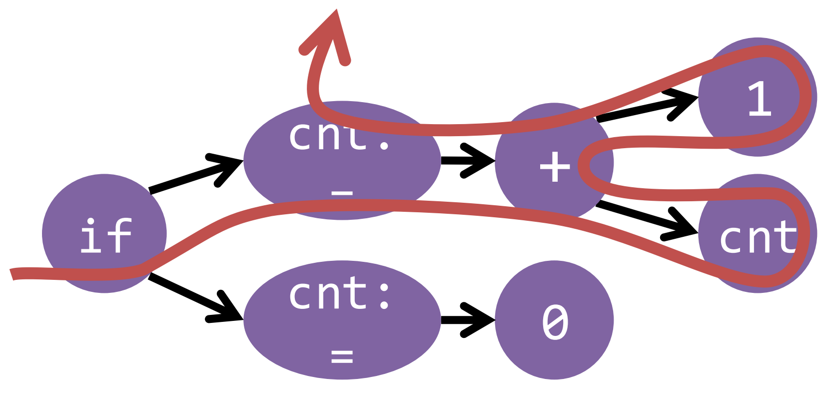
Loop Invariant Code Motion

```
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

if
cnt:
=
cnt:
=
+
0
1
cnt

- Two different approaches, with different tradeoffs
- Both enable compilation of the program, instead of interpreter

Open Research Questions

Metacompilation is very resource intensive!

(Huge amounts of interpreter code for the application code, compilation takes much more time and memory than direct compilation)

Can we utilize higher Futamura Projections?
Compiling the Compiler?

Research and Literature

- **Meta Tracing**
  - **Tracing the Meta-level: PyPy's Tracing JIT Compiler.** Bolz, C. F., Cuni, A., Fijalkowski, M. & Rigo, A. (2009). *ICOOOLPS'09*
- **Partial Evaluation**
  - **Practical Partial Evaluation for High-performance Dynamic Language Runtimes.** Würthinger, T., Wimmer, C., Humer, C., Wö A., Stadler, L., Seaton, C., Duboscq, G., Simon, D. & Grimmer, M. (2017). *PLDI'17*
- **Tracing vs. Partial Evaluation: Comparing Meta-Compilation Approaches for Self-Optimizing Interpreters.** Marr, S. & Ducasse, S. (2015). *OOPSLA'15*
- Speculation Without Regret: Reducing Deoptimization Meta-data in the Graal Compiler. Duboscq, G., Würthinger, T. & Mössenböck, H. (2014). *PPPJ'14*

Taming Dynamic Languages

EFFICIENT DATA REPRESENTATION

Dynamic Languages Can Do This

```
o = {foo: 33}
```

Object with 1 field

```
o.bar = new Object()
```

Object with 2 fields

```
o.float = 4.2
```

Object with 3 fields

```
o.float = "string"
```

And you can
store anything

Data Representation for Objects

```
o = {foo: 33}
o.bar = new Object()
o.baz = "string"
o.float = 4.2
```


Full Power of Dynamic Languages: **Rarely Used**

Again, Programs Utilize Potential Dynamicity only in Few Cases

“Magic” is hard to understand.
It’s easier to build complex systems from simple structures.

Sometimes,
dynamicity brings simplicity!

Efficient Data Representation
Maps/Shapes/Hidden Classes

```
o = {foo: 33}
o.bar = new Object()
o.baz = "string"
o.float = 4.2
```

Obj

33

4.2

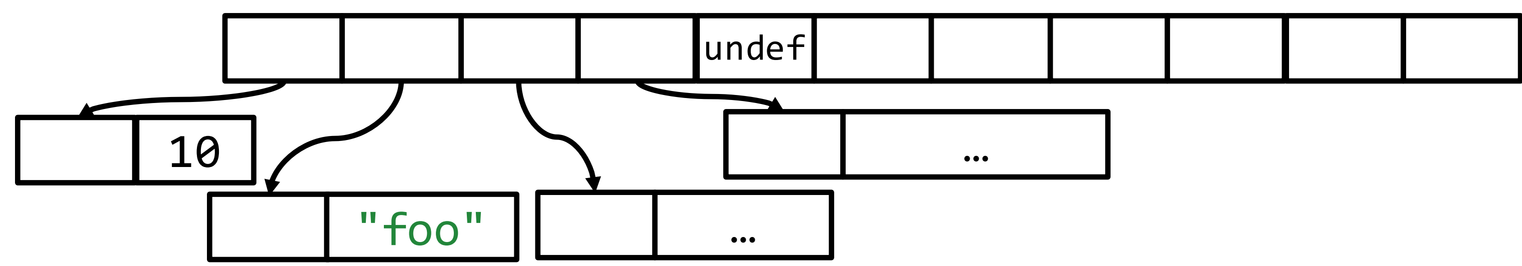
"string"

Shape
1: foo(int)
2: baz(ptr)
3: float(float)
4: bar(ptr)

Combined with inline caches, a field access is a simple access at memory offset

Data Representation for Arrays

```
arr = [10, "foo", new Object(), [2, 3, 4], undefined]
arr[100] = 49
```

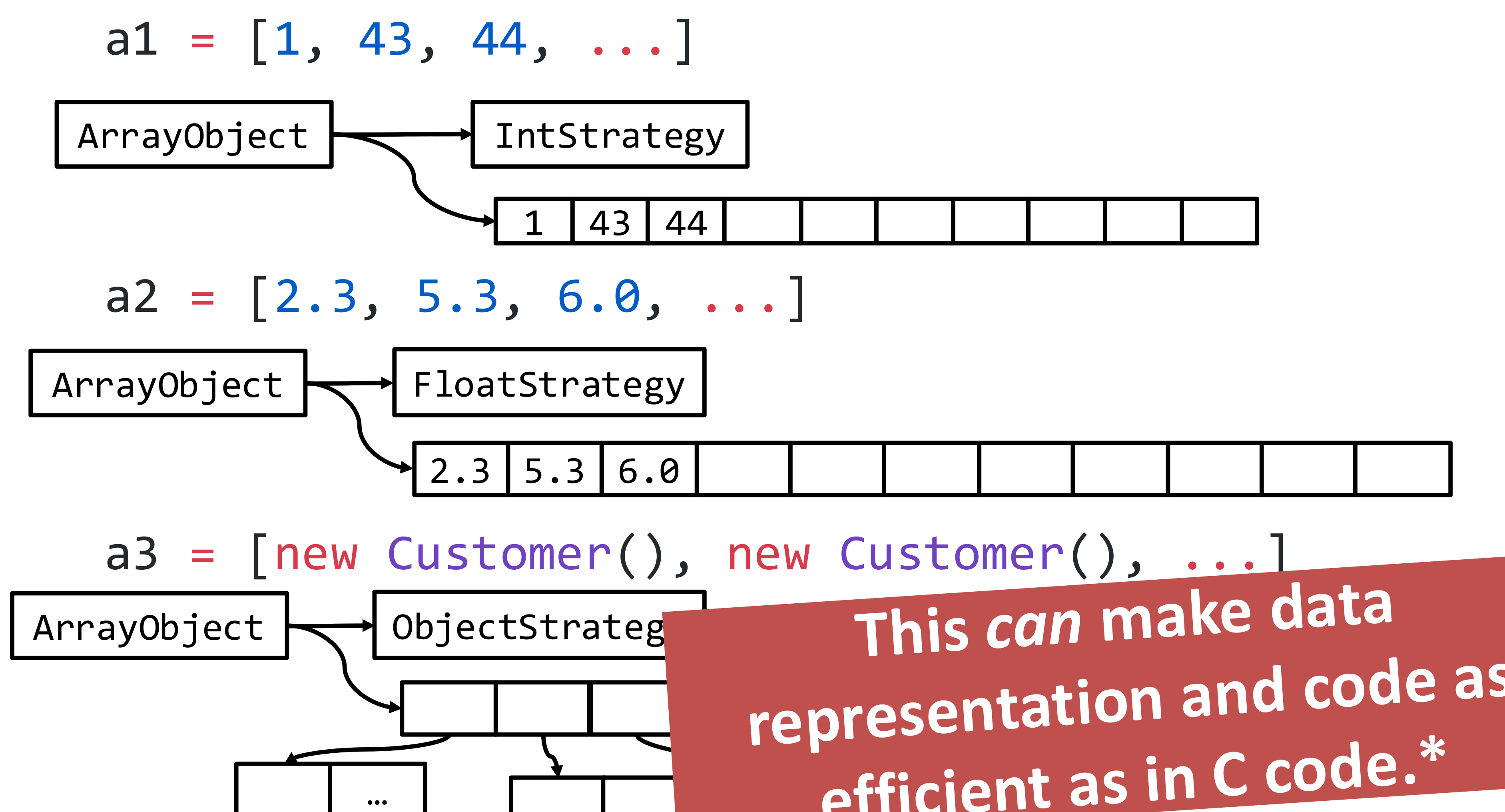


Full Power of Dynamic Languages: **Rarely Used**

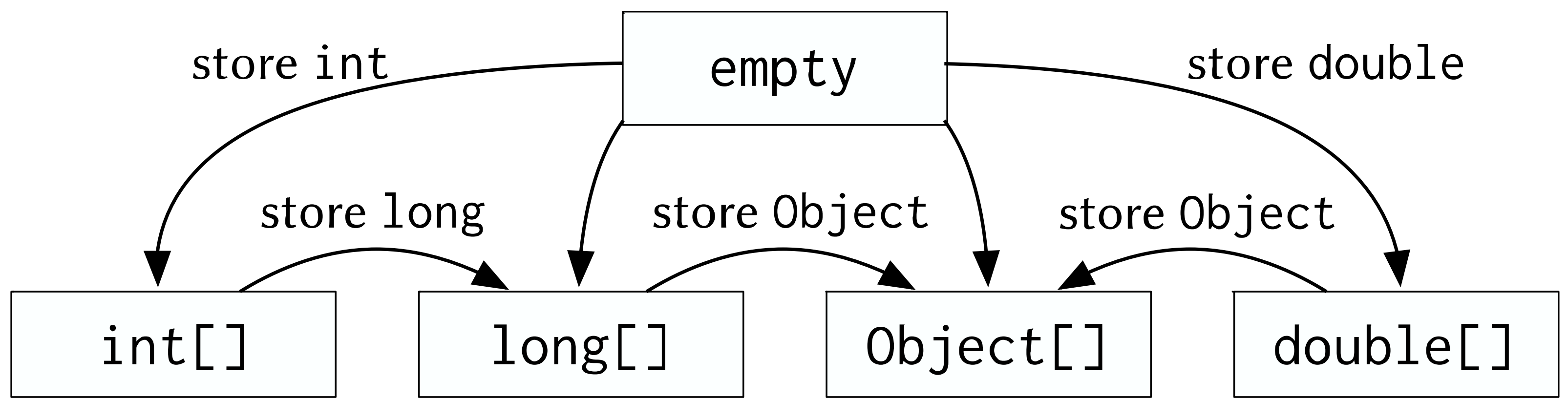
```
a1 = [1, 43, 44, ...]
a2 = [2.3, 5.3, 6.0, ...]
a3 = [new Customer(), new Customer(), ...]
```

Very frequently: homogeneous elements

Efficient Data Representation
Storage Strategies for Collections



Transitions between Storage Strategies



Does our example benefit from these optimizations?

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
6  class Button extends Widget {}
7  class RadioButton extends Button {}
8
9  function findAllThatFit(arr, width) {
10   var result = [];
11   for (var w of arr)
12     if (w.fitsInto(width))
13       result.push(w)
14   return result;
15 }
```

?

Does our example benefit from these optimizations?

```
1  class Widget {
2    fitsInto(width) {
3      return this.width <= width;
4    }
5  }
6  class Button extends Widget {}
7  class RadioButton extends Button {}
8
9  function findAllThatFit(arr, width) {
10   var result = [];
11   for (var w of arr)
12     if (w.fitsInto(width))
13       result.push(w);
14   return result;
15 }
```

[{}…]

Mementos: feedback of strategy to allocation site

{width: int, fitsInto()}

Shapes are the key/check in Lookup Caches

?

Open Research Questions

Can we reduce the memory use of all these optimizations?

Embedded, mobile devices,…

Research and Literature

- **An Efficient Implementation of SELF a Dynamically-Typed Object-Oriented Language Based on Prototypes.** Chambers, C., Ungar, D. & Lee, E. (1989). *OOPSLA'89*
- **An Object Storage Model for the Truffle Language Implementation Framework.** A. Wöß, C. Wirth, D. Bonetta, C. Seaton, C. Humer, and H. Mössenböck. *PPPJ'14.*
- **Storage Strategies for Collections in Dynamically Typed Languages.** C. F. Bolz, L. Diekmann, and L. Tratt. *OOPSLA'13*.
- **Memento Mori: Dynamic Allocation-site-based Optimizations.** Clifford, D., Payer, H., Stanton, M. & Titzer, B. L. (2015). ISMM'15

Optimizing prototypes in V8

- https://mathiasbynens.be/notes/prototypes
- https://mathiasbynens.be/notes/shapes-ics

WRAP UP

PLISS-Related Events at ECOOP

Mon 29 June - Fri 3 July 2026 Brussels, Belgium

ECOOP 2026 (series) / MPLR 2026 (series) /

MPLR 2026

About | Program | Accepted Papers | Call for Papers

The 23rd International Conference on Managed Programming Languages and Runtimes (MPLR 2026, formerly ManLang, originally PPPJ) is a premier forum for presenting and discussing novel results in all aspects of managed programming languages and runtime systems, which serve as building blocks for some of the most important computing systems around, ranging from small-scale (embedded and real-time systems) to large-scale (cloud-computing and big-data platforms) and anything in between (mobile, IoT, and wearable applications).

Papers accepted by MPLR 2026 describe original research results and have not been published anywhere else. Each submitted paper has received a minimum of three reviews by members of the program committee. Papers have been selected based on their originality, relevance, technical clarity, and quality of presentation. At least one author of each accepted paper must register for the MPLR 2026 symposium and present the paper.

Important Dates	AoE (UTC-12h)
Tue 30 Jun 2026 Conference day	

ICOOOLPS 2026

About | Program | Accepted Papers | Call for Papers

The ICOOOLPS workshop series brings together researchers and practitioners working in the field of language implementation and optimization.

The goal of the workshop is to discuss emerging problems and research directions as well as new solutions to classic and novel implementation challenges. The topics of interest for the workshop include implementation and optimization strategies for a wide range of programming languages, including but not limited to object-oriented ones. Compiler retargeting, virtual machine implementations, and generative programming approaches are welcome too.

Keynote

Pragmatic Approaches to Improving Compiler Correctness

CF Bolz-Tereick

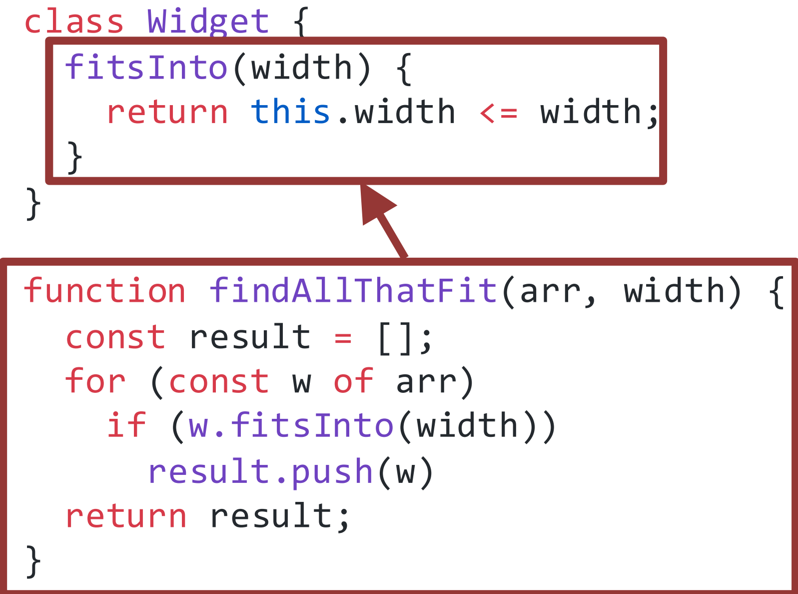
Important Dates	AoE (UTC-12h)
Mon 29 Jun 2026 **ICOOOLPS takes place**	

Human Generated Code Often Looks and Behaves like a Power Law Distribution

number of XYZ this size

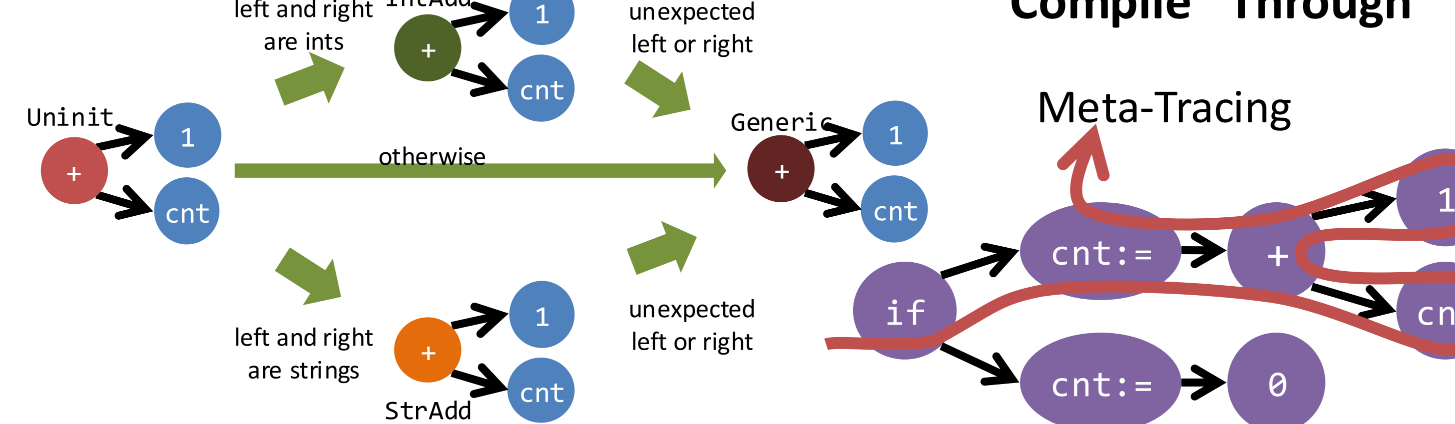
size of XYZ

Method-based Compilation
A Good Solution for Average Code



Self-Optimize and Speculate

Meta-Compilation
Compile "Through" The Interpreter



observe run-time value
select optimization of first execution

Partial Evaluation
Guided by AST

if
cnt:=
+
1
cnt
cnt:=
0

Compilation Unit based on User Program