Bringing Performance and Scalability to Dynamic Languages

Mario Wolczko
Architect
Virtual Machine Research Group
Oracle Labs

The following is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract.
It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

Dynamic Languages

Ruby

JavaScript

python

php

Julia

MATLAB

The Language of Technical Computing

R

J

SMALLTALK

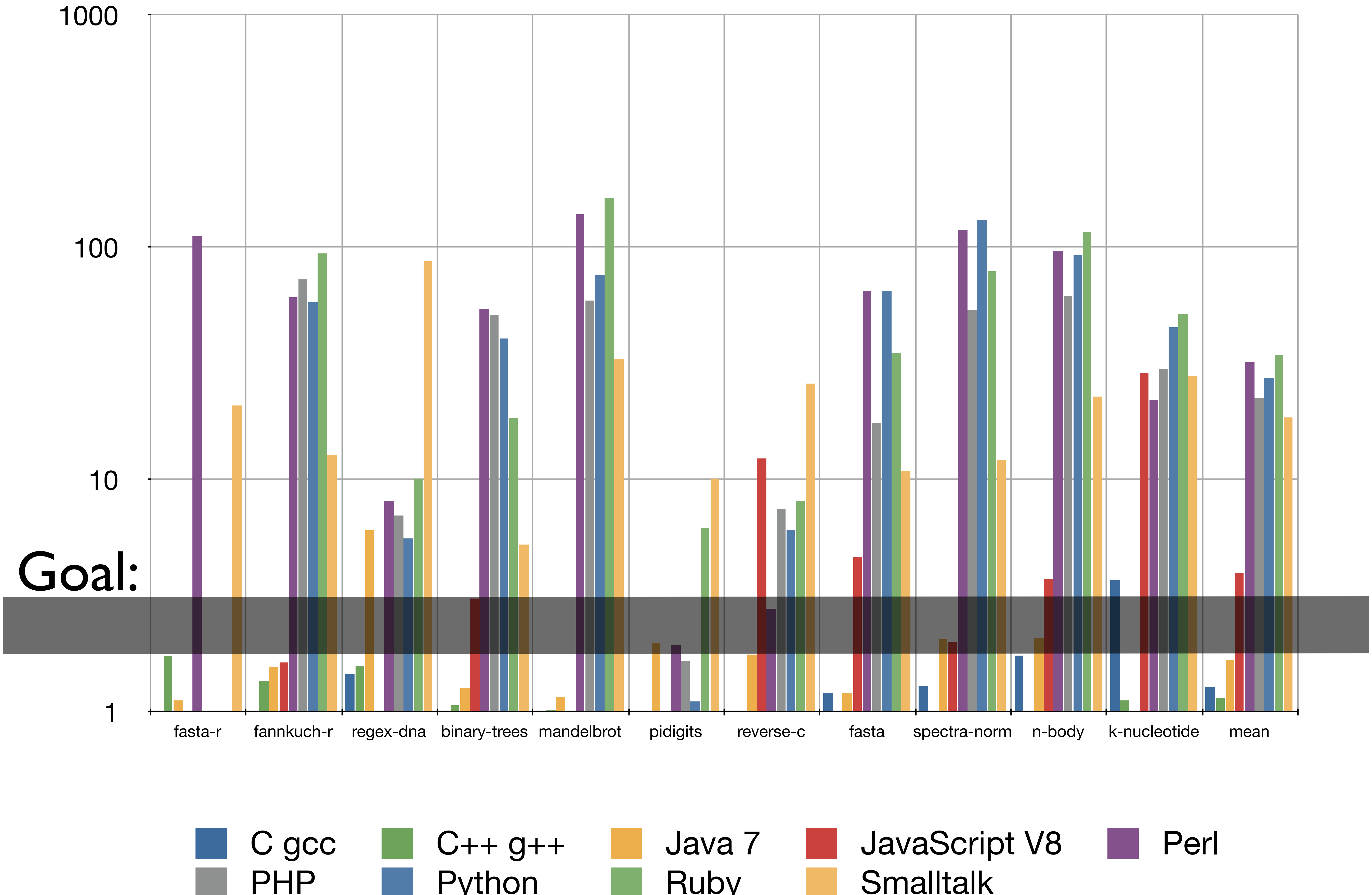
APL

perl

Common Characteristics of Dynamic Languages

- Modest adoption (~10^4–10^6 users/developers)
  - Easy to get started
  - Well adapted to a niche
    - Web: JavaScript, Perl, PHP, Python, Ruby
    - Technical: R, Julia, MATLAB, APL, J
- Features:
  - Absence of type declarations
  - Objects and/or vectors/strings at the core
  - Lexical scoping, closures, reflection, eval
- Fairly slow implementations (10–100x off optimal)

Relative speeds of various languages (Language Shootout benchmarks)



Why are they slow?

- Interpretation is easy to implement, but highly inefficient
- Consider the execution of a simple expression, `a+b:`
  - Find out the type of `a`
  - Find out the type of `b`
  - Find out what + means
  - Check that the operation is applicable to the data types, throw error if not
  - Prepare the data (e.g, strip tags)
  - **Invoke the operation**
  - Convert the result to canonical form (add tags)

VM implementation history, part 1 of 3:
Bytecodes plus simple Just-In-Time compilation: dragging performance out of the mud (1983–89)

- Instead of interpreting a parse tree, create an instruction set for a *virtual* machine (often a stack machine) and interpret those instructions
  - BCPL O-code, 1960s, UCSD Pascal (1978), Smalltalk (1976)
  - Somewhat denser representation, using bytecode

```
push a
push b
add
```

- Faster still: “macro-expand” instructions into machine code, just-in-time (ParcPlace Smalltalk, 1983)
  - Trades interpreter decode and dispatch overhead for compilation cost
  - 10x faster

VM implementation history, part 2:
Feedback-driven, adaptive compilation: Respectable performance (1989–95)

- Q. High performance comes from compiled code with known types–but where is the type information?
- A. The types are in the data, not in the code.
  The data are only available at run time.
- Thankfully, most expressions in real programs are *monomorphic* (i.e., use only one type combination).
- Solution: interpret for a short while, observe the actual types, and then compile code with optimistic assumptions and aggressive inlining (Self, Sun Labs, 1992).
- If assumptions are wrong, take slow path, or discard compiled code, revert to interpretation (*deoptimization*), recompile again later.
- Another 5x performance gain

VM implementation history, part 3:
Reduction to practice, deployment in production (1996–present)

- Java, even though typed, can benefit:
  - No ahead-of-time compilation
  - Types can drive method resolution, inlining
  - Large inlined regions present plenty of opportunity for optimization (another 3x?)
  - Current HotSpot JVM is ~50x faster than JDK1.0 (interpreted bytecodes)
- Similar techniques adopted by other production VMs: IBM J9 JVM, Microsoft CLR, JavaScript V8.

The state of the art: performance, at a price

- High performance, at a high price in complexity:
  - 1–2MLOC? 200–500 engineer years? $100M?
- Responses: research JVMs, written in Java, to reduce the high price (safer language, better tools, better factoring).
  - Jikes (IBM), Maxine (Sun/Oracle): Neither is in production.
- None of the dynamic languages have had this kind of investment (except JavaScript, partially).

ORACLE

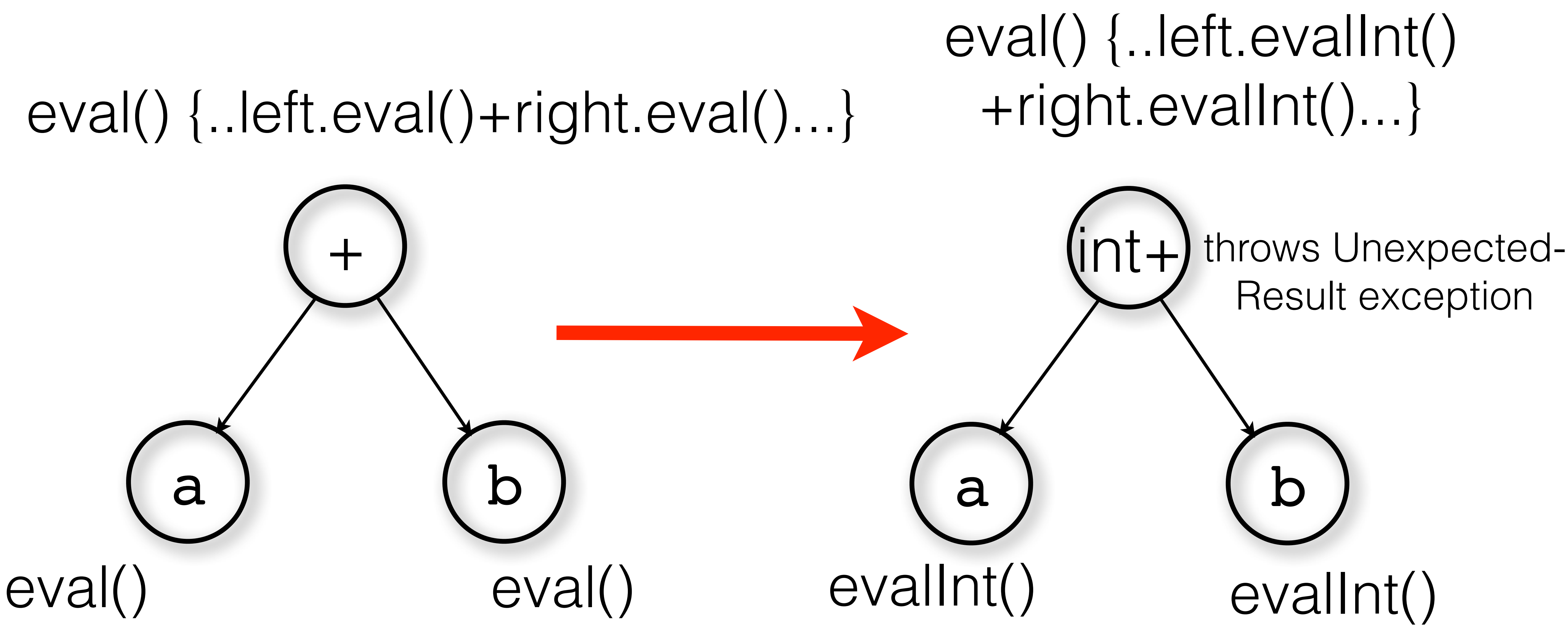
Alphabet Soup: Fusing interpretation and compilation

- What does the compiler need?
  1. The region of code being compiled
  2. The types of the operands
  3. The semantics of the operations, as code
- In a conventional VM, #3 is implemented twice:
  1) in the interpreter (easy) or JIT compiler (moderate)
  2) in the optimizing compiler (hard)

Can we do better?

ORACLE

AST rewriting during interpretation to gather types



Rewritten node does less work in subsequent evaluations

Before:

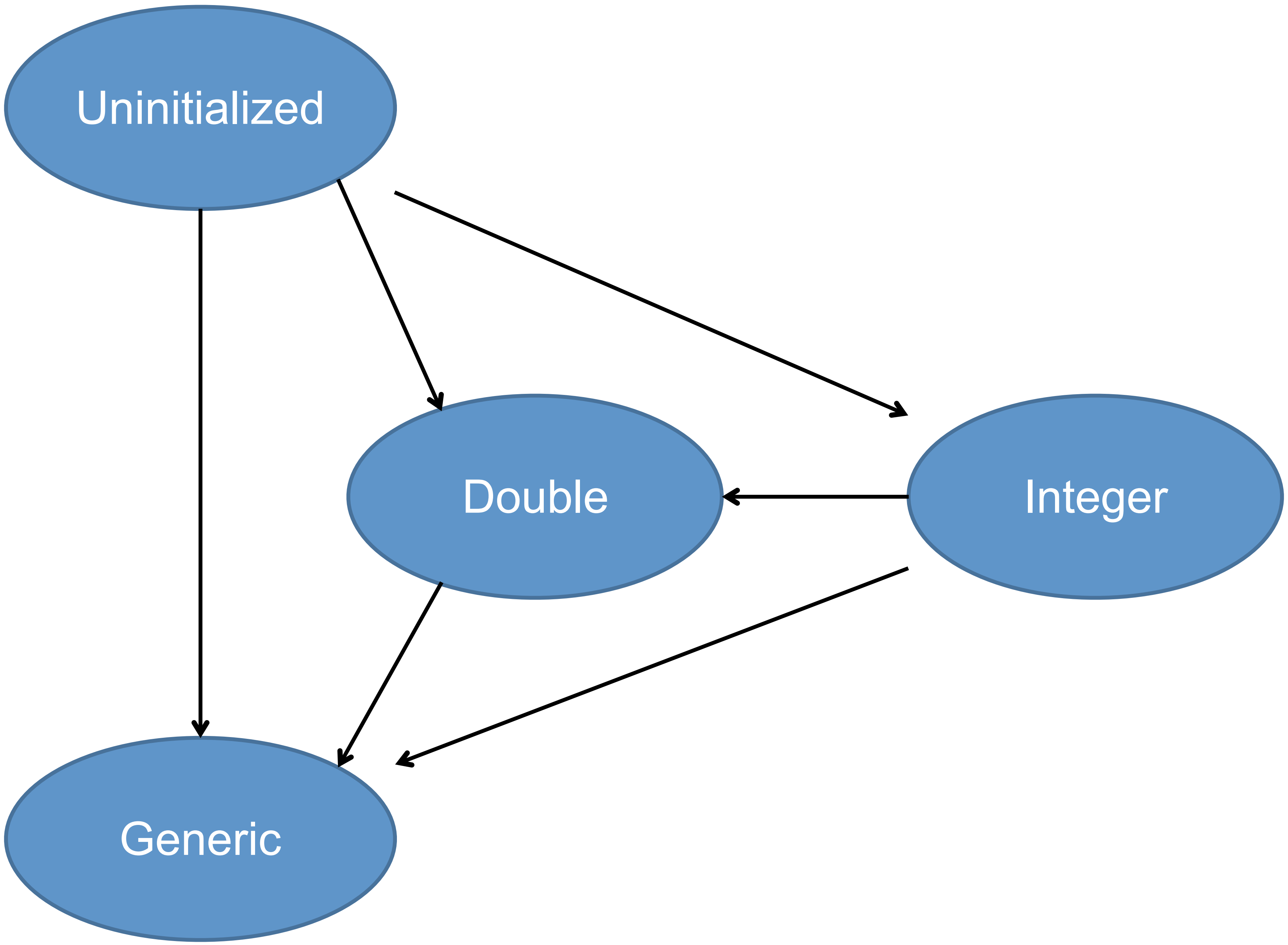
- Find out the type of `a`
- Find out the type of `b`
- Find out what + means
- Check that the operation is applicable to the data types, throw error if not
- Prepare the data (e.g, strip tags)
- **Invoke the operation**
- Convert the result to canonical form (add tags)

Rewritten node does less work in subsequent evaluations

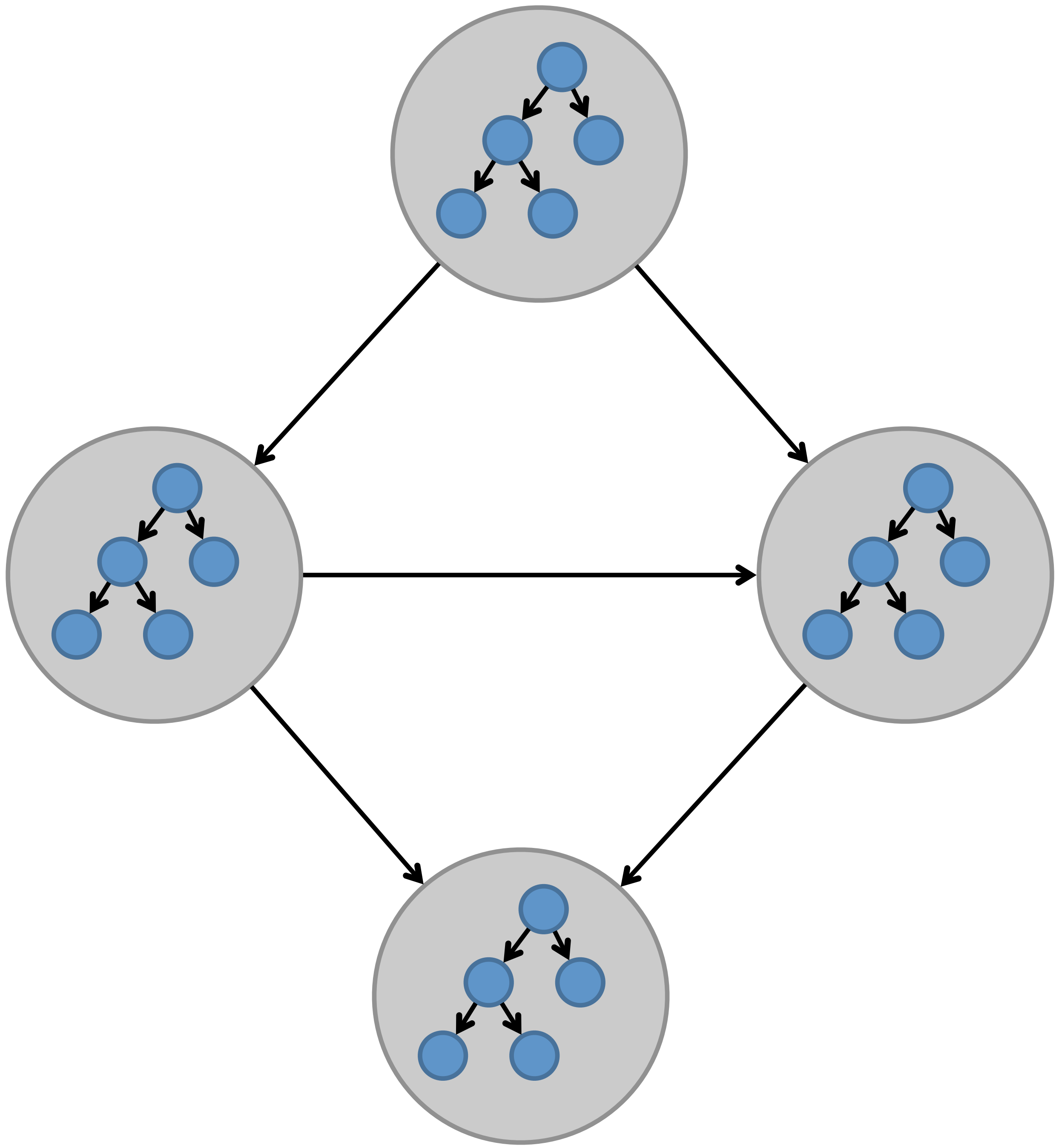
After:

- ~~Find out the type of `a`~~
- ~~Find out the type of `b`~~
- ~~Find out what + means~~
- Check that the operation is applicable to the data types, throw error if not
- ~~Prepare the data (e.g, strip tags)~~
- **Invoke the operation**
- ~~Convert the result to canonical form (add tags)~~

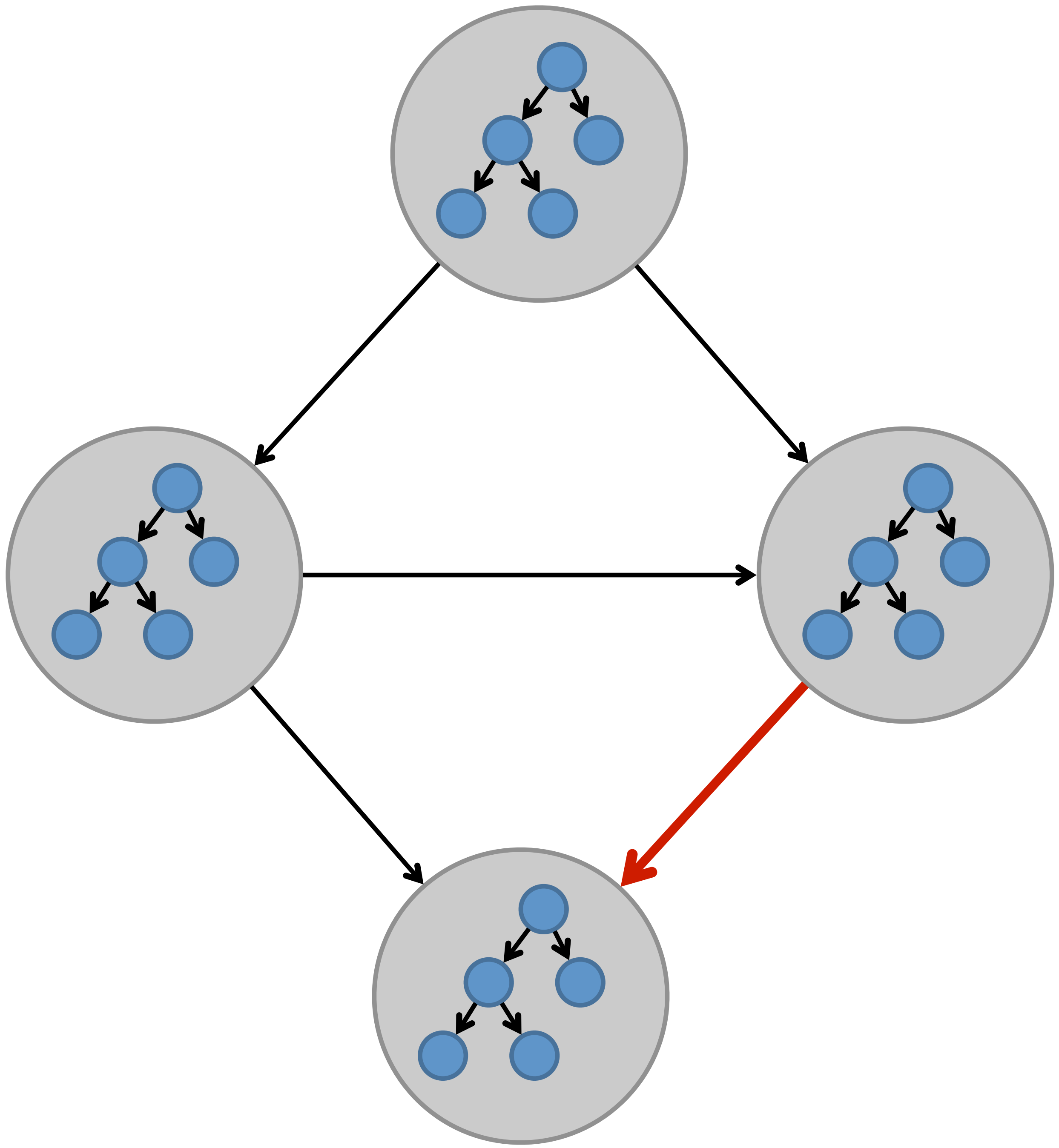
JavaScript Arithmetic Operations: Node State Graph



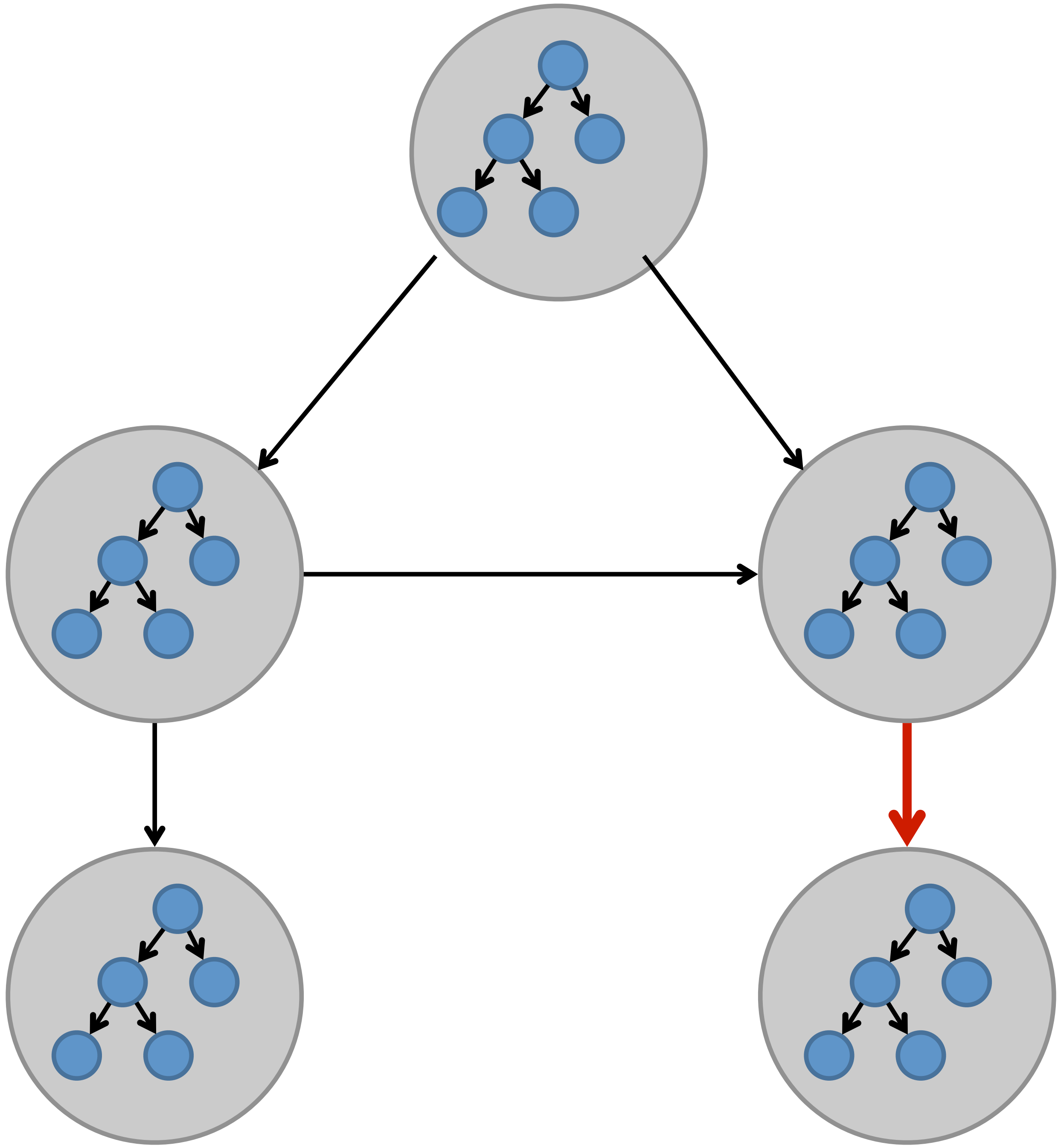
Optimization of programs during interpretation



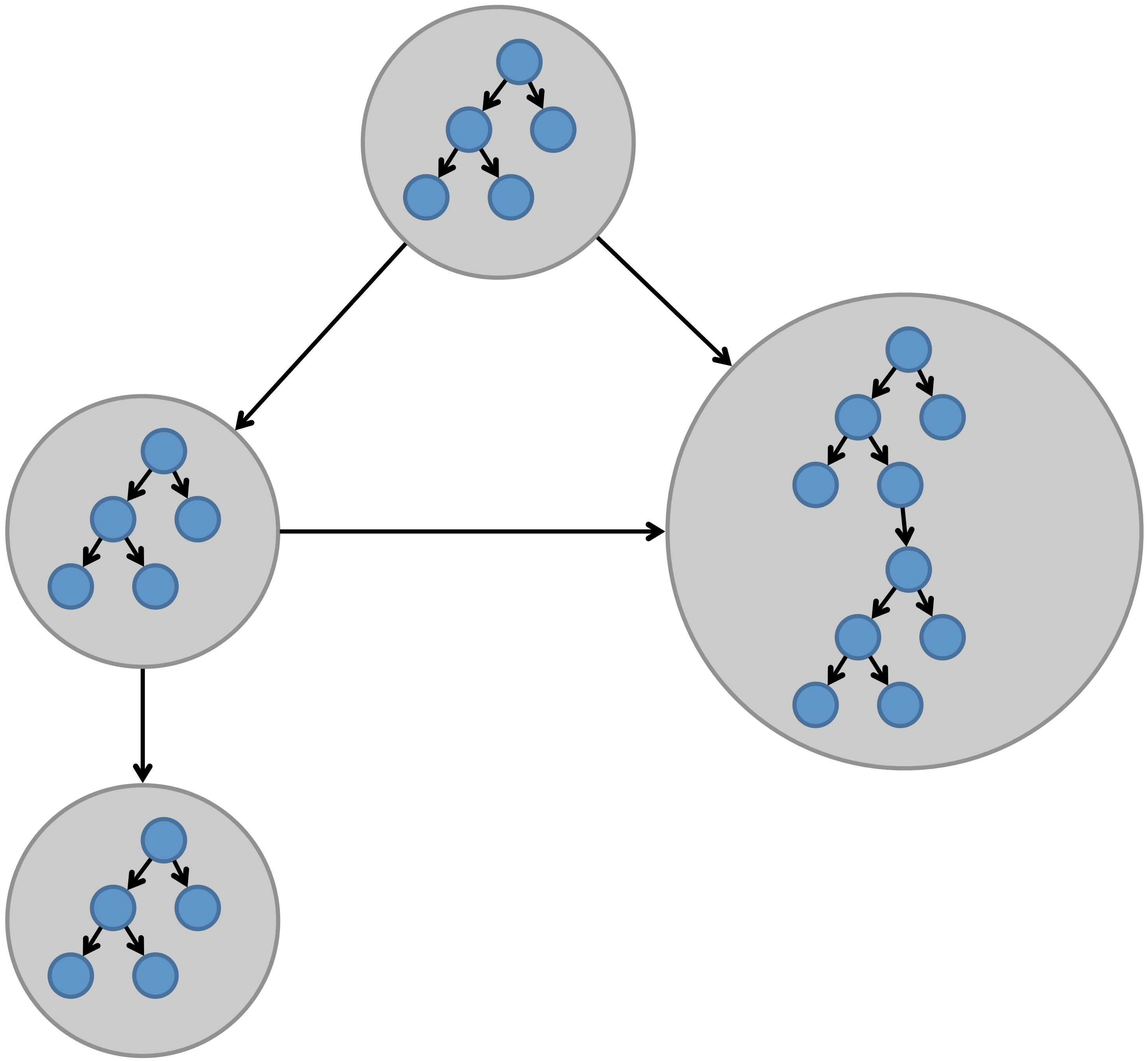
Call Graph: Hot Links



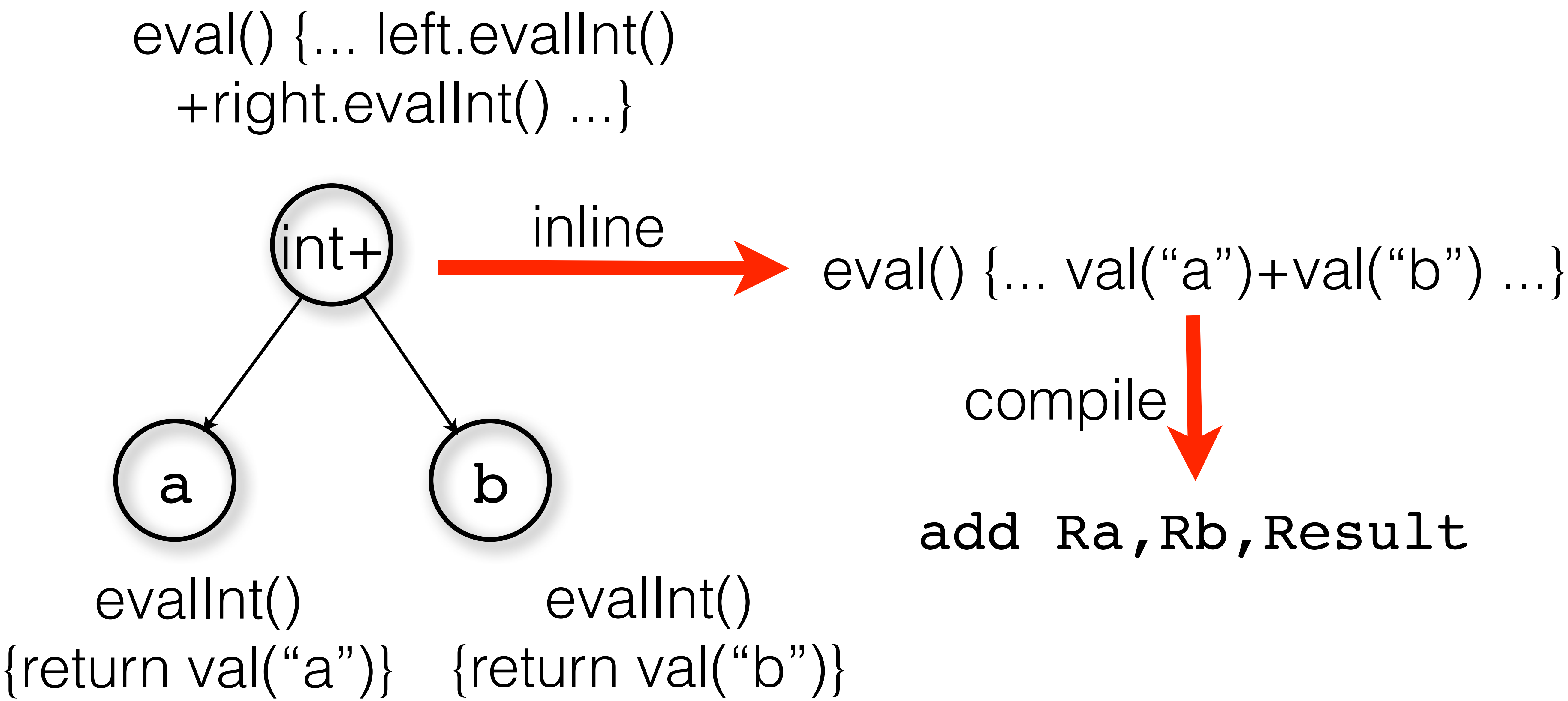
Call Graph: Duplication from hot call site



Call Graph: Inlining



Compiling the specialized ASTs



Rewritten node does less work in subsequent evaluations

After compilation:

checks
hoisted above all uses of
the data within the
compilation

- Check that the operation is applicable to the data types, throw error if not
- **Invoke the operation**
- **Invoke the operation**
- **Invoke the operation**
- **....**

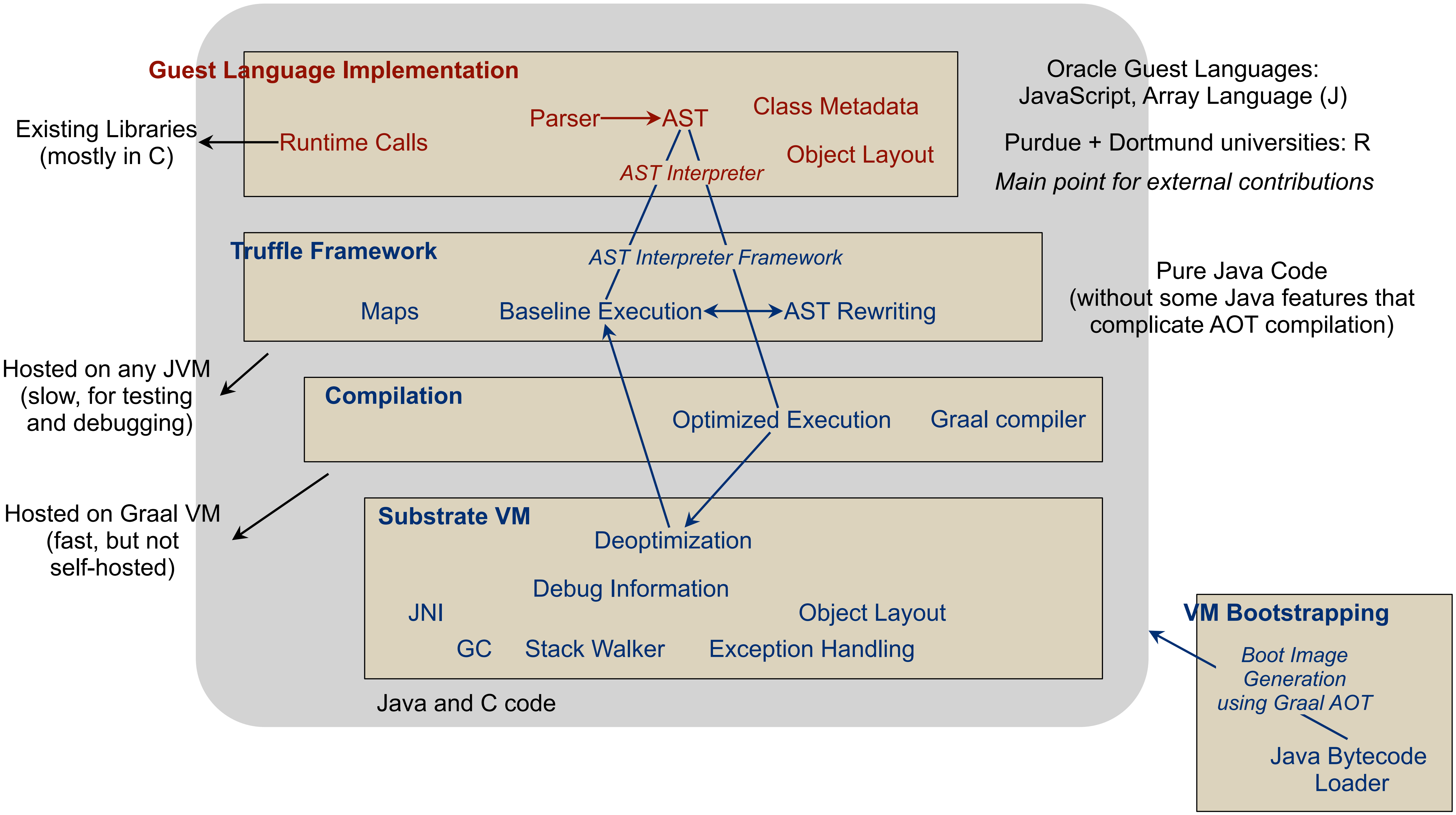
ORACLE

Putting it all together: repurposing the compiler

By traversing the recursive code of the interpreter, guided by the AST and the type information, we can compile the original expression/statement/method/region **without having written a compiler for that language.**

ORACLE

Alphabet Soup HLVM architecture



Parallelization opportunities in technical computing languages

- The technical computing languages have an array-processing core which is amenable to parallel execution:
  - Regular, bulk vector operations
  - Functional
- Once the interpretation overhead has been removed, these are attractive targets for further optimization when used on “big data”.
- High-level optimizations can remove intermediate computations.
- Vectorization and multi-core or GPU parallelism can exploit regularity.

Languages we are implementing

- To demonstrate the viability of the system for the “web” and “technical” languages we are implementing one of each: JavaScript and J.
- Longer term, we hope to build a community of language implementors via open source, and implement other languages. http://openjdk.java.net/projects/graal/

Status

- We are well along with interpreters for JavaScript and J (written in Java).
  - JS: 5x faster than Rhino interpreter
- We already have an extensible optimizing compiler, Graal, written in Java, for Java:
  - Compiles Java bytecode to machine code via a graph IR, for a JVM.
  - Extension in progress for AST inputs.
- To come:
  - Vectorization and parallelization
  - Design and implement a substrate VM (don’t need all the complexity of a full JVM)

Hardware and Software

Engineered to Work Together