
ORACLE®

ORACLE

Truffle: A Self-Optimizing Runtime System

Christian Wimmer, Thomas Würthinger
Oracle Labs

“Write Your Own Language”

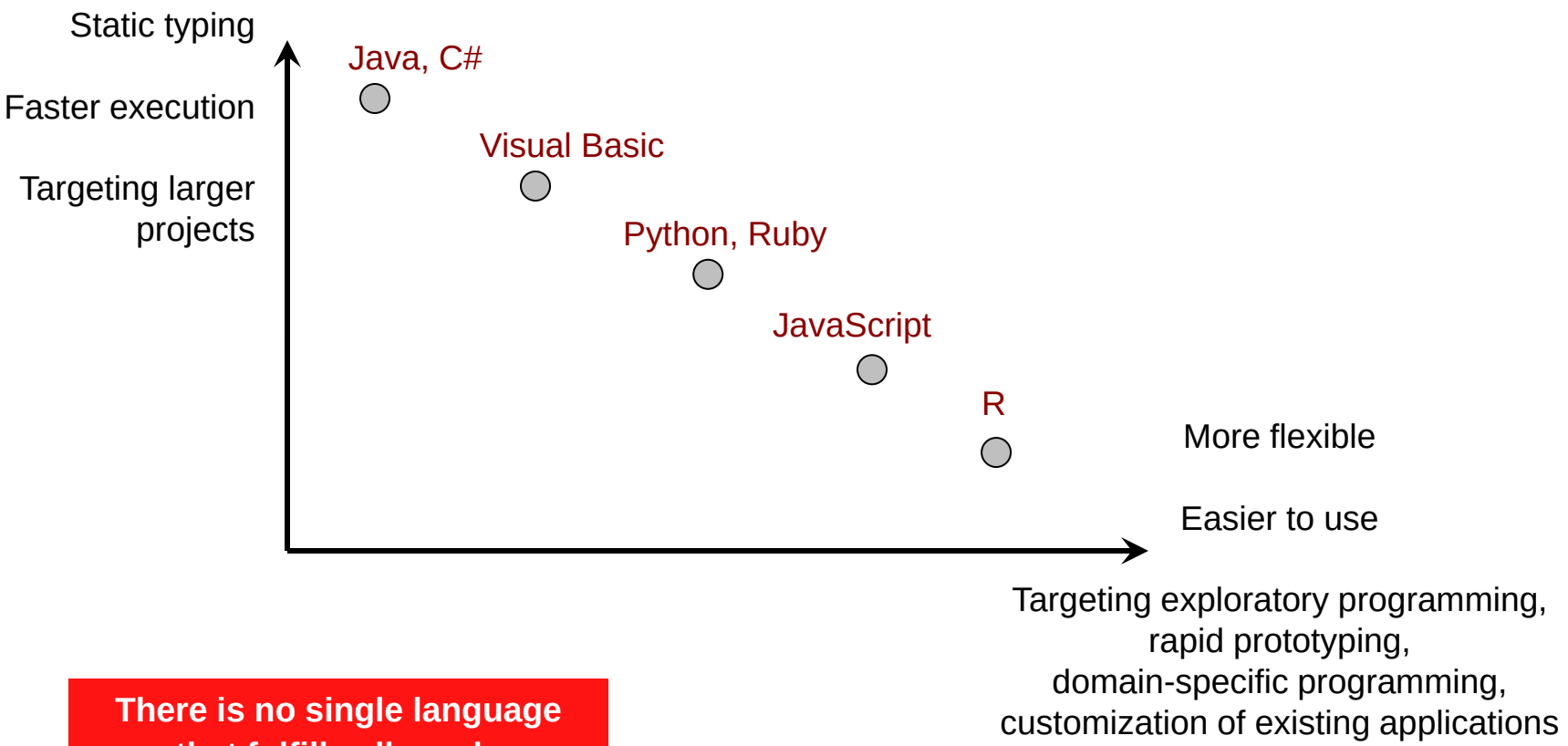
Current situation

- Prototype a new language
  - Parser and language work to build syntax tree (AST), AST Interpreter
- Write a “real” VM
  - In C/C++, still using AST interpreter, spend a lot of time implementing runtime system, GC, …
- People start using it
- People complain about performance
  - Define a bytecode format and write bytecode interpreter
- Performance is still bad
  - Write a JIT compiler
  - Improve the garbage collector

How it should be

- Prototype a new language in Java
  - Parser and language work to build syntax tree (AST)
  - Execute using AST interpreter
- Integrate with VM-building framework
  - Integrate with Modular VM
  - Add small language-specific parts
- People start using it
  - And it is already fast

A Spectrum of Programming Languages



There is no single language that fulfills all needs

Truffle Requirements

Java, Python, Ruby,
JavaScript, Groovy,
Clojure, Scala,
J, R, ...

Generality

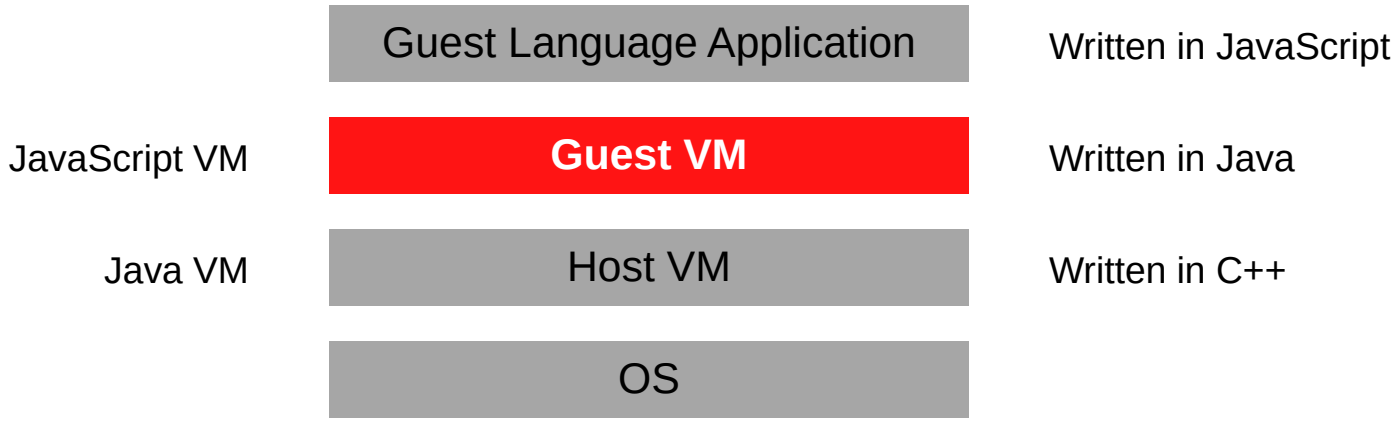
+

Performance

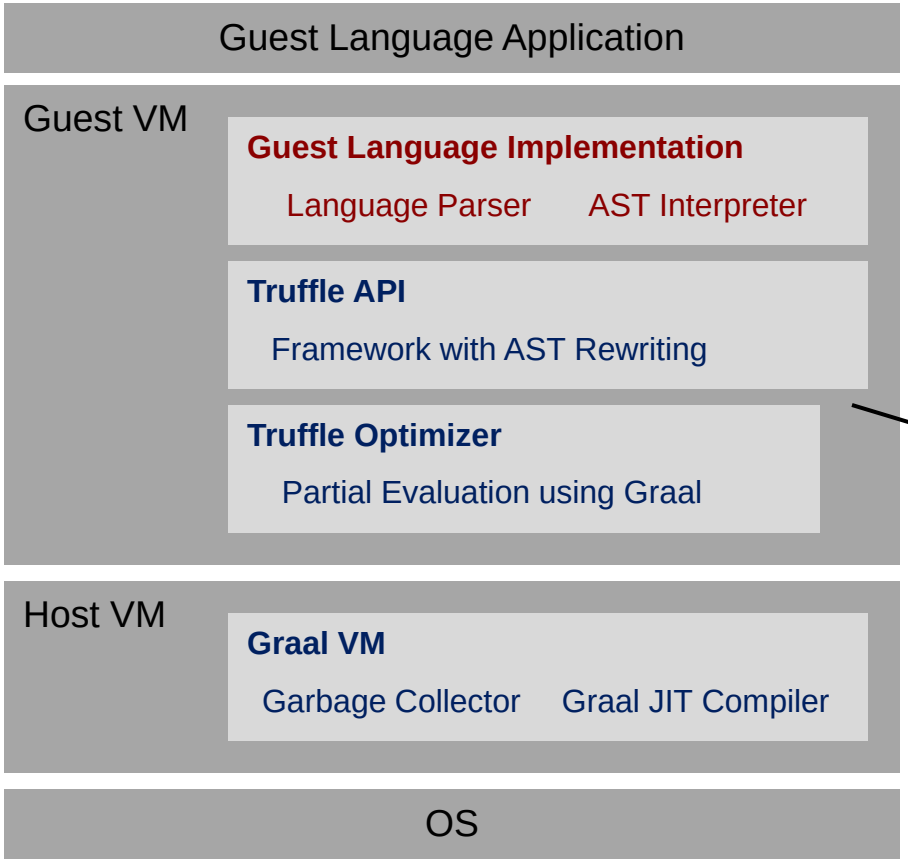
```
function f(a, n) {
   var x = 0;
   while (n-- > 0) {
      x = x + a[n];
   }
   return x;
}
```

```
L1: decl rax
jz L2
movl rcx, rdx[16+4*rax]
cvtsi2sd xmm1, rcx
addsd xmm0, xmm1
jmp L1
L2:
```

Overall System Structure



Detailed System Structure



Oracle Guest Languages: JavaScript, J

Main point for external contributions

Baseline execution on any JVM
(slow, for testing and debugging)

A modified version of the Java HotSpot VM
with Graal as the JIT compiler

Syntax Tree Interpreter

```
class IfNode extends StatementNode {
  ConditionNode condition;
  StatementNode thenPart, elsePart;

  void execute(Frame frame) {
    if (condition.execute(frame)) {
      thenPart.execute(frame);
    } else {
      elsePart.execute(frame);
    }
  }
}
```

```
class LessThanNode
      extends ConditionNode {
  TypedNode left, right;

  boolean execute(Frame frame) {
    Object l = left.execute(frame);
    Object r = right.execute(frame);
    return lessThan(l, r);
  }
}
```

```
class BlockNode extends StatementNode {
  StatementNode[] statements;

  void execute(Frame frame) {
    for (StatementNode s: statements) {
      s.execute(frame);
    }
  }
}
```

```
class AddNode extends TypedNode {
  TypedNode left, right;

  Object execute(Frame frame) {
    Object l = left.execute(frame);
    Object r = right.execute(frame);
    return addExact(l, r);
  }
}
```

```
class WhileNode extends StatementNode {
  ConditionNode condition;
  StatementNode body;

  void execute(Frame frame) {
    while (condition.execute(frame)) {
      body.execute(frame);
    }
  }
}
```

```
class ReadLocalNode extends TypedNode {
  FrameSlot slot;

  Object execute(Frame frame) {
    return frame.getValue(slot);
  }
}
```

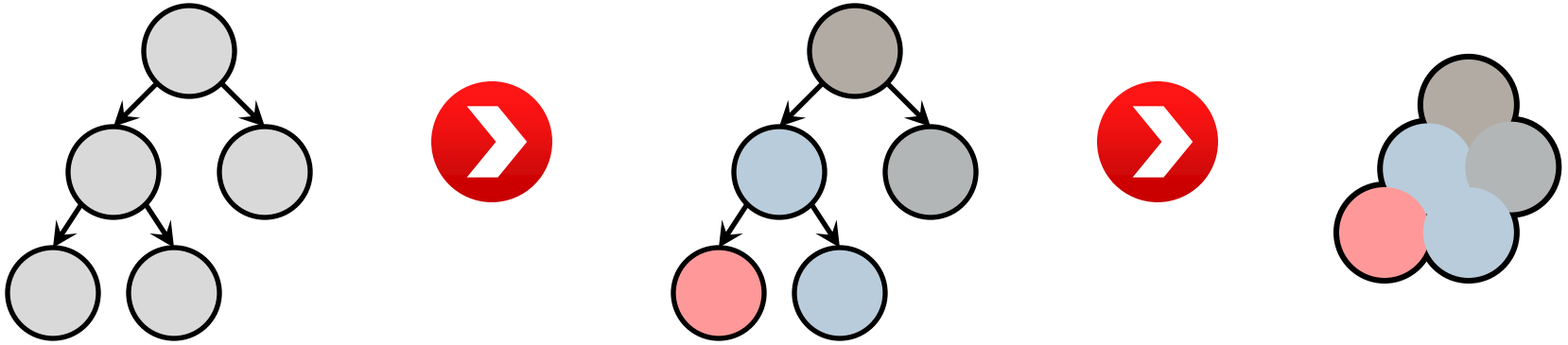
High run-time overhead

- Dispatch overhead: virtual method calls
- Heap-based frame: memory access for locals
- Primitive values passed as Object: boxing

Truffle Approach

AST Rewriting for Type Feedback

Automatic Partial Evaluation



AST Interpreter
Untyped Nodes

AST Interpreter
Typed Nodes

Compiled Code

- Eliminate boxing of primitive values
- Eliminate dynamic type checks
- AST Inlining

- Syntax tree nodes are “stable”
- Aggressive constant folding, method inlining, escape analysis
- Deoptimize compiled code on tree rewrite

A Simple Language

- Control flow
  - `BlockNode` (Sequence of statements), `IfNode`, `WhileNode`, `ReturnNode`
- Data types
  - Number (arbitrary precision integer number), Boolean, String
  - Implementation types: `BigInteger`, `boolean`, `String`
  - Dynamically typed: no explicit type declarations
  - Strongly typed: runtime error if operation performed on unsupported operand types
- Operations
  - Constant literals, `AddOp`, `MulOp`, `LessThanOp`, `LogicalAndOp`
- Restrictions
  - No objects, arrays, function calls
- Language parser
  - Generated from attributed grammar using Coco/R (http://ssw.jku.at/coco/)

Typed Arithmetic Operations

```
class AddNode extends TypedNode {
  TypedNode left, right;

  Object execute(Frame frame) {
    Object l = left.execute(frame);
    Object r = right.execute(frame);

    if (l instanceof BigInteger && r instanceof BigInteger) {
      return ((BigInteger) l).add((BigInteger) r);
    } else if (l instanceof String || r instanceof String) {
      return left.toString() + right.toString();
    } else {
      throw new RuntimeException("type error");
    }
  }
}
```

Problem: `BigInteger` is slow

Solution: Use primitive type `int` when possible, automatic overflow to `BigInteger`

Typed Arithmetic Operations

```
class AddNode extends TypedNode {
  TypedNode left, right;

  Object execute(Frame frame) {
    Object l = left.execute(frame);
    Object r = right.execute(frame);

    if (l instanceof Integer && r instanceof Integer) {
      try {
        return Math.addExact((int) l, (int) r);
      } catch (ArithmethicException ex) {
        return BigInteger.valueOf((int) l).add(BigInteger.valueOf((int) r));
      }
    } else if (l instanceof BigInteger && r instanceof BigInteger) {
      return ((BigInteger) l).add((BigInteger) r);
    } else if (l instanceof String || r instanceof String) {
      return left.toString() + right.toString();
    } else {
      throw new RuntimeException("type error");
    }
  }
}
```

In `java.lang.Math` for JDK 8

Problem: `int` boxed to `Integer`; many dynamic type checks

Solution: type-specialized nodes

Typed Arithmetic Operations

```
class IntegerAddNode extends TypedNode {
  TypedNode left, right;

  int executeInteger(Frame frame) {
    int l = left.executeInteger(frame);
    int r = right.executeInteger(frame);

    try {
      return Math.addExact(l, r);
    } catch (ArithmethicException ex) {
      return ??
    }
  }
}
```

```
class StringAddNode extends TypedNode {
  TypedNode left, right;

  String executeString(Frame frame) {
    Object l = left.executeGeneric(frame);
    Object r = right.executeGeneric(frame);

    return left.toString() + right.toString();
  }
}
```

```
class BigIntegerAddNode extends TypedNode {
  TypedNode left, right;

  BigInteger executeBigInteger(Frame frame) {
    BigInteger l = left.executeBigInteger(frame);
    BigInteger r = right.executeBigInteger(frame);
    return l.add(r);
  }
}
```

```
class GenericAddNode extends TypedNode {
  TypedNode left, right;

  Object executeGeneric(Frame frame) {
    Object l = left.executeGeneric(frame);
    Object r = right.executeGeneric(frame);

    return doGeneric(l, r);
    // doGeneric: same code as execute method on
    // previous slide, with all type checks
  }
}
```

Problem: dynamic typing and overflow check

Solution: Replace a node if the operands have unexpected types or values

Typed Arithmetic Operations

```
class IntegerAddNode extends TypedNode {
  TypedNode left, right;

  int executeInteger(Frame frame) throws UnexpectedResultException {
    int l;
    try {
      l = left.executeInteger(frame);
    } catch (UnexpectedResultException ex) {
      Object r = right.executeGeneric(frame);
      replace(createNewSpecialization(ex.getResult(), r));
      throw new UnexpectedResultException(doGeneric(ex.getResult(), r));
    }
    int r;
    try {
      r = right.executeInteger(frame);
    } catch (UnexpectedResultException ex) {
      replace(createNewSpecialization(l, ex.getResult()));
      throw new UnexpectedResultException(doGeneric(l, ex.getResult()));
    }
    try {
      return Math.addExact(l, r);
    } catch (ArithmeticException ex) {
      replace(createNewSpecialization(l, r));
      throw new UnexpectedResultException(doGeneric(l, r));
    }
  }
}
```

Contains a result of generic type `Object`

Create a new specialization based on the observed argument types

Replace this node with new specialization

Propagate generic result to caller (which might then change its specialization too)

Problem: lots of boilerplate code

Solution: Code generation based on Java Annotations

Node Infrastructure

API base class for `Node`:

```
public abstract class Node implements Cloneable {

  private Node parent;

  protected final Node updateParent(Node newChild) { ... }
  public final Node replace(Node newNode) { ... }

  public final Iterable<Node> getChildren() { ... }
  public Node copy() { ... }
}
```

Language specific base class:

```
public abstract class TypedNode extends Node {

  abstract boolean    executeBoolean(Frame frame)    throws UnexpectedResultException;
  abstract int        executeInteger(Frame frame)    throws UnexpectedResultException;
  abstract BigInteger executeBigInteger(Frame frame) throws UnexpectedResultException;
  abstract String     executeString(Frame frame)     throws UnexpectedResultException;

  abstract Object     executeGeneric(Frame frame);
}
```

- Framework requires a unique `Node` parent, provides tree rewriting
- Language defines type-specialized execution methods
- `UnexpectedResultException` to return a result of unexpected type

Typed Arithmetic Operations

```
@Operation(typeSystem = SLTypes.class, values = {"left", "right"})
public final class AddOp {

  @Specialization
  @SpecializationThrows(javaClass = ArithmeticException.class, transitionTo = "doBigInteger")
  public static int doInteger(int l, int r) {
    return Math.addExact(l, r);
  }

  @Specialization
  public static BigInteger doBigInteger(BigInteger l, BigInteger r) {
    return l.add(r);
  }

  @Specialization
  @SpecializationGuard(methodName = "isString", dynamic = true)
  public static String doString(Object l, Object r) {
    return l.toString() + r.toString();
  }

  @Generic
  public static Object doGeneric(Object l, Object r) {
    throw new RuntimeException("type error");
  }
}
```

Marks a type specialized operation

Tree rewriting on integer overflow

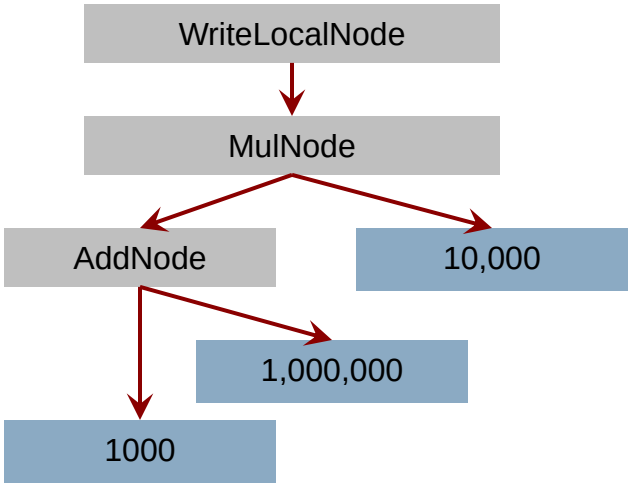
A custom type guard

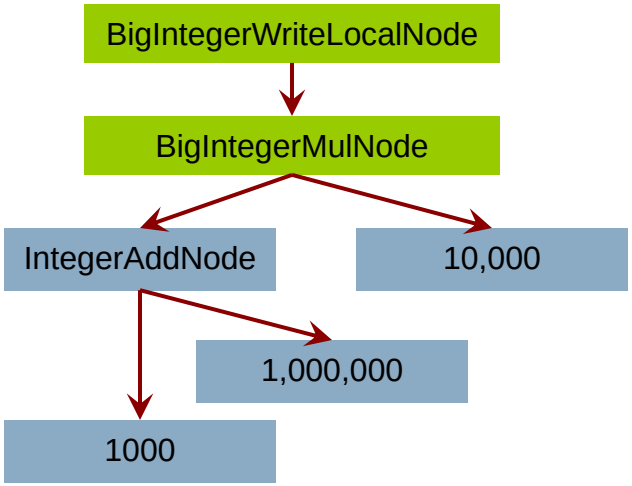
Marks the generic operation, called when no specialization matches

```
@TypeSystem(types = {int.class, BigInteger.class, boolean.class, String.class, Object.class},
    nodeBaseClass = TypedNode.class)
abstract class SLTypes {
  public boolean isString(Object l, Object r) {
    return l instanceof String || r instanceof String;
  }
}
```

Specialization Example

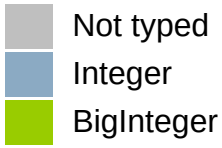
Before first execution → **After execution**



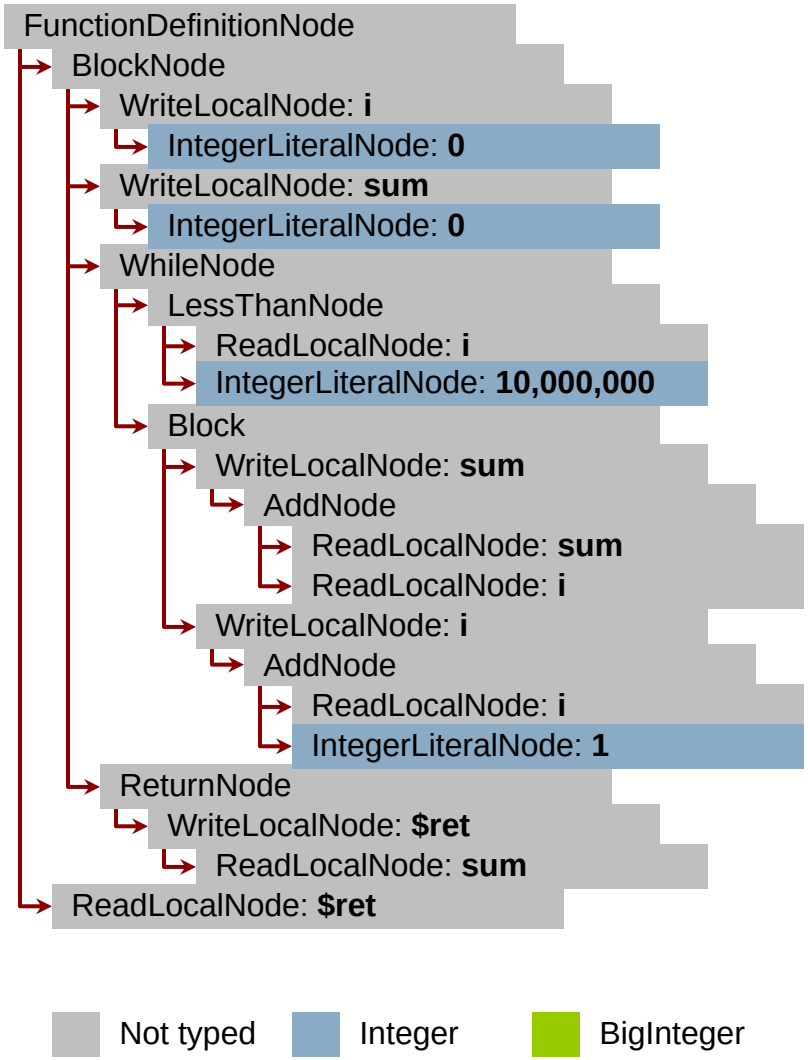


Literal constants are always typed and never change their type

Types are propagated from the leaf nodes



Local Variable Specialization



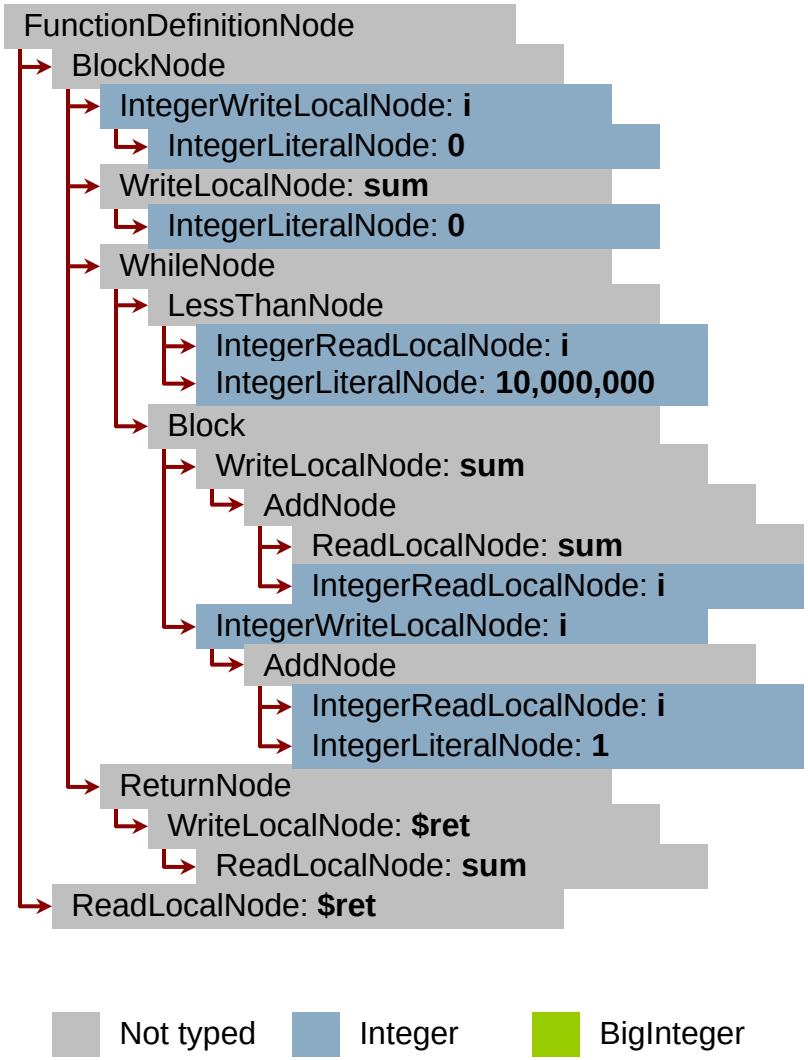
Simple Language Code:

```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Local Variables:

```
i:    <not initialized>
sum:  <not initialized>
$ret: <not initialized>
```

Local Variable Specialization



Simple Language Code:

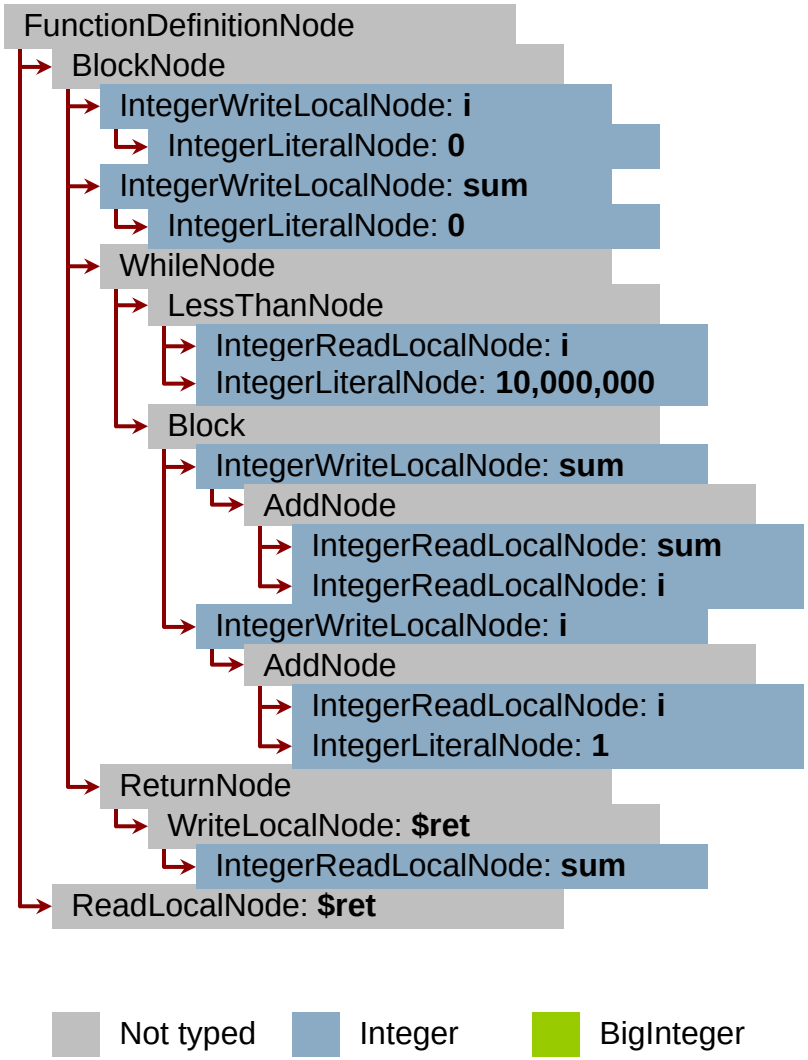
```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Local Variables:

```
i:    int 0
sum:  <not initialized>
$ret: <not initialized>
```

All reads and writes of a local variable change the type together

Local Variable Specialization



Simple Language Code:

```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

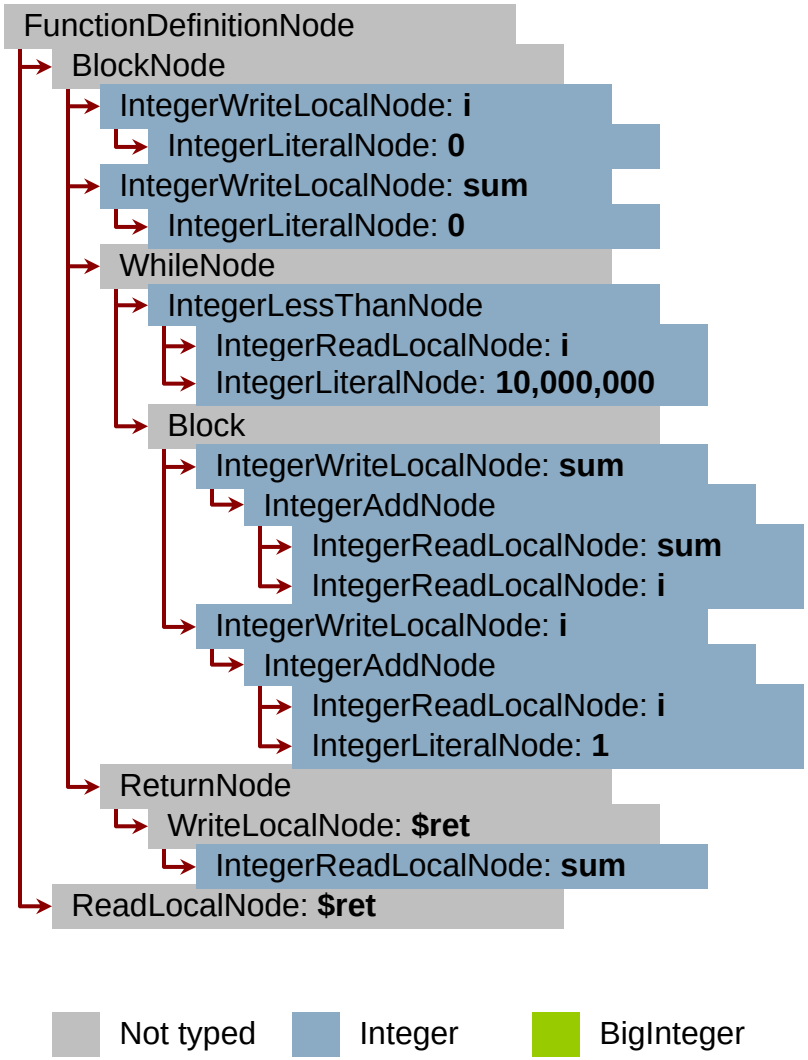
Local Variables:

```
i:    int 0
sum:  int 0
$ret: <not initialized>
```

All reads and writes of a local variable change the type together

Not typed Integer BigInteger

Local Variable Specialization



Not typed Integer BigInteger

Simple Language Code:

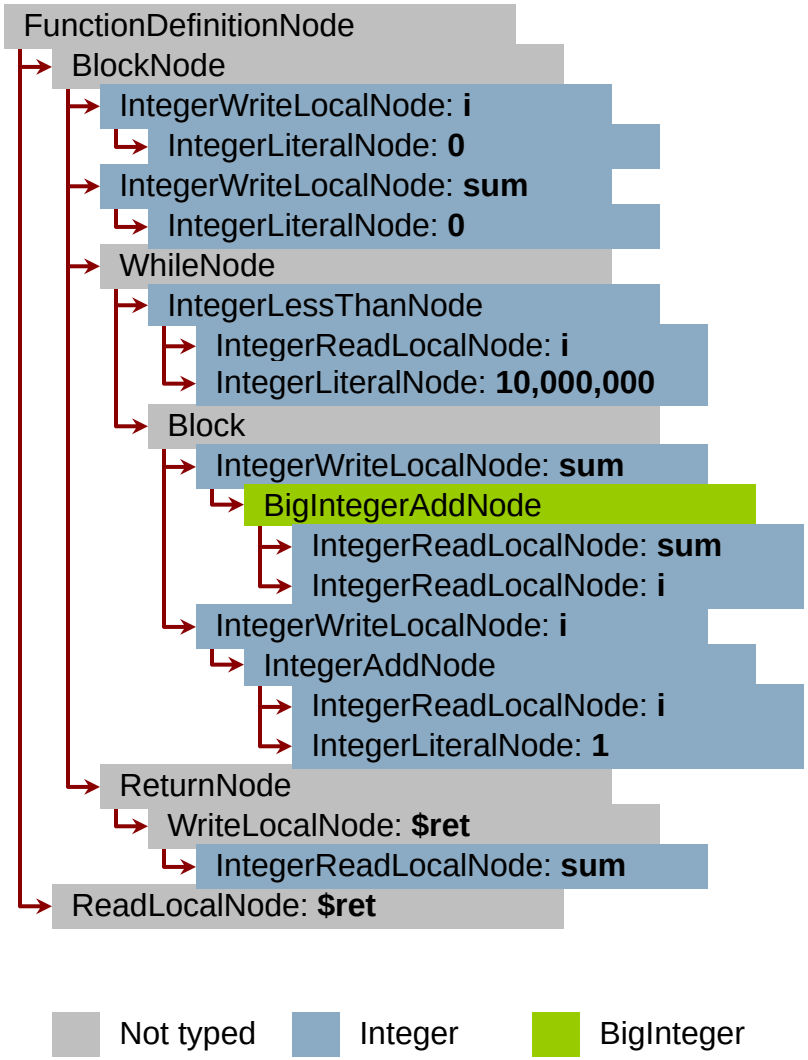
```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Local Variables:

```
i:    int 1
sum:  int 0
$ret: <not initialized>
```

All reads and writes of a local variable change the type together

Local Variable Specialization



Simple Language Code:

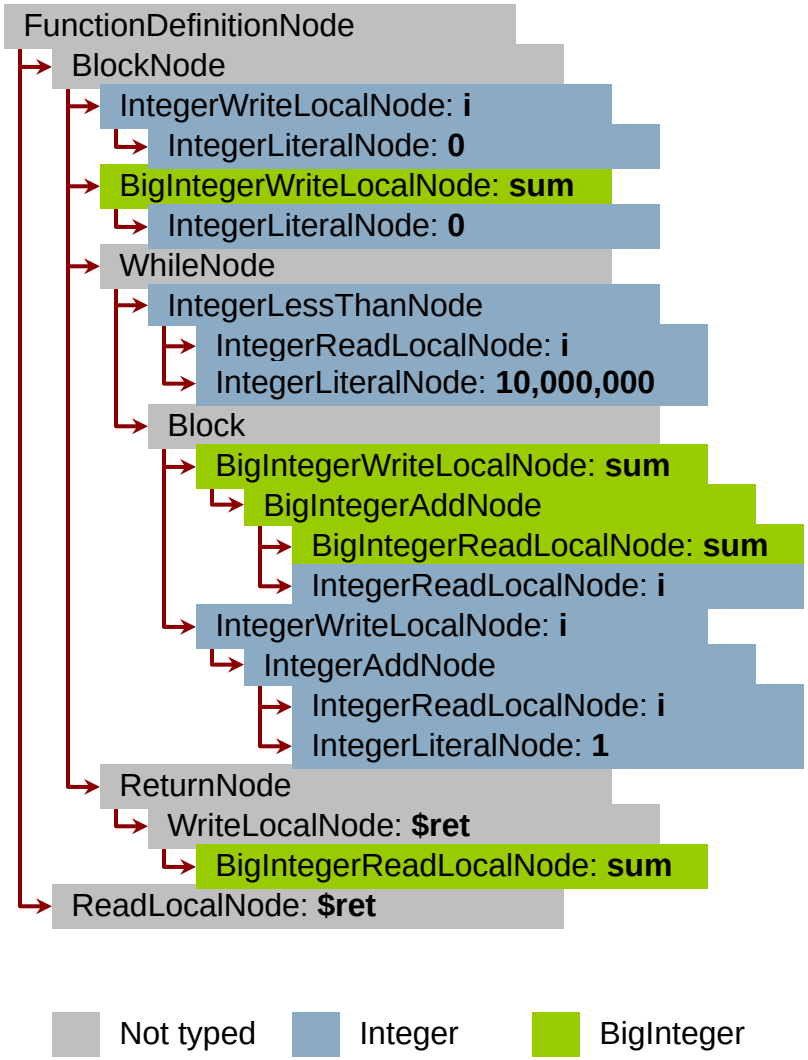
```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Local Variables:

```
i:    int 65536
sum:  int 2147450880
$ret: <not initialized>
```

All reads and writes of a local variable change the type together

Local Variable Specialization



Simple Language Code:

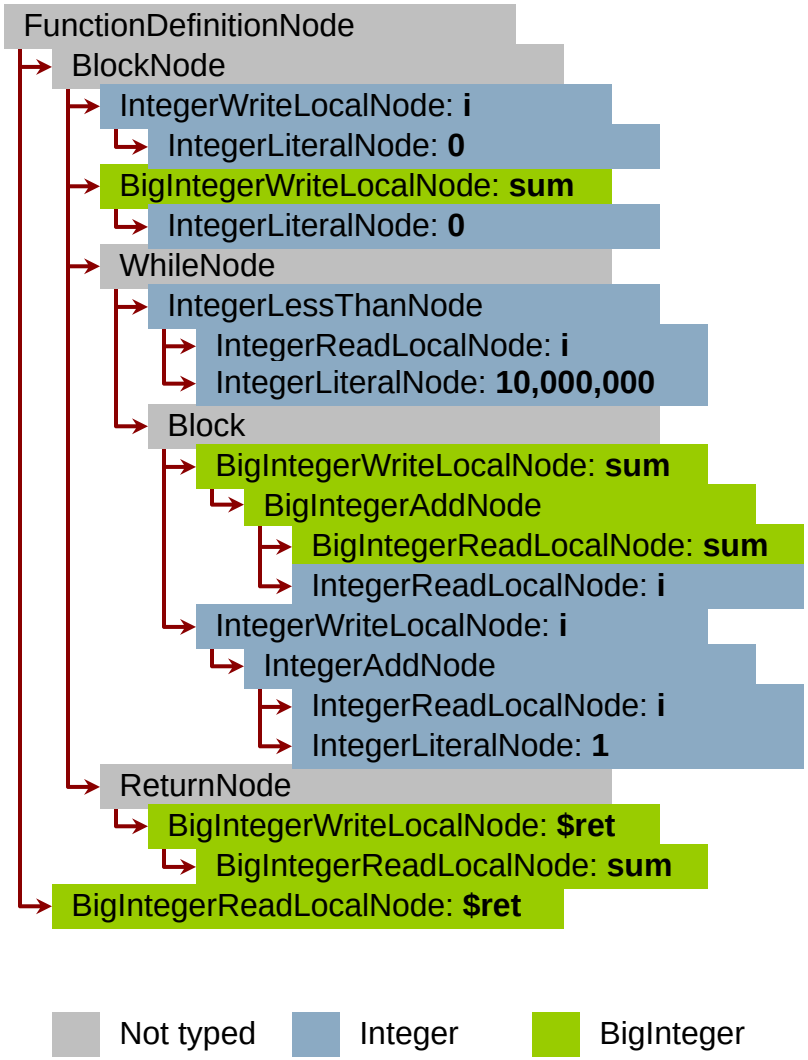
```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

Local Variables:

```
i:    int 65536
sum:  BigInteger 2147516416
$ret: <not initialized>
```

All reads and writes of a local variable change the type together

Local Variable Specialization



Simple Language Code:

```
function main {
  i = 0;
  sum = 0;
  while (i < 10000000) {
    sum = sum + i;
    i = i + 1;
  }
  return sum;
}
```

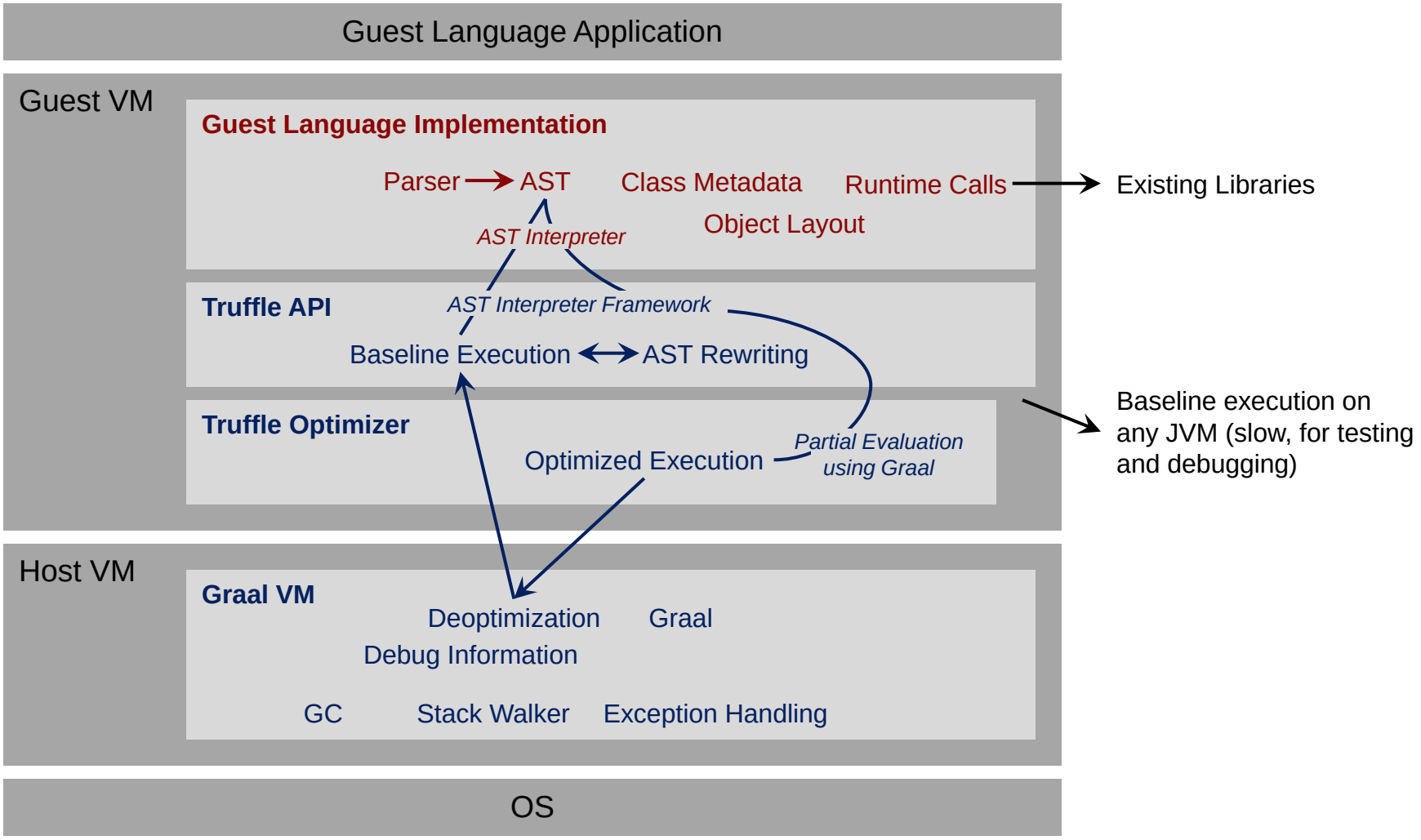
Local Variables:

```
i:    int 10000000
sum:  BigInteger 49999995000000
$ret: BigInteger 49999995000000
```

All reads and writes of a local variable change the type together

Optimization: operations start as *Integer* instead of *Not Typed* to avoid first rewrite

Detailed System Structure

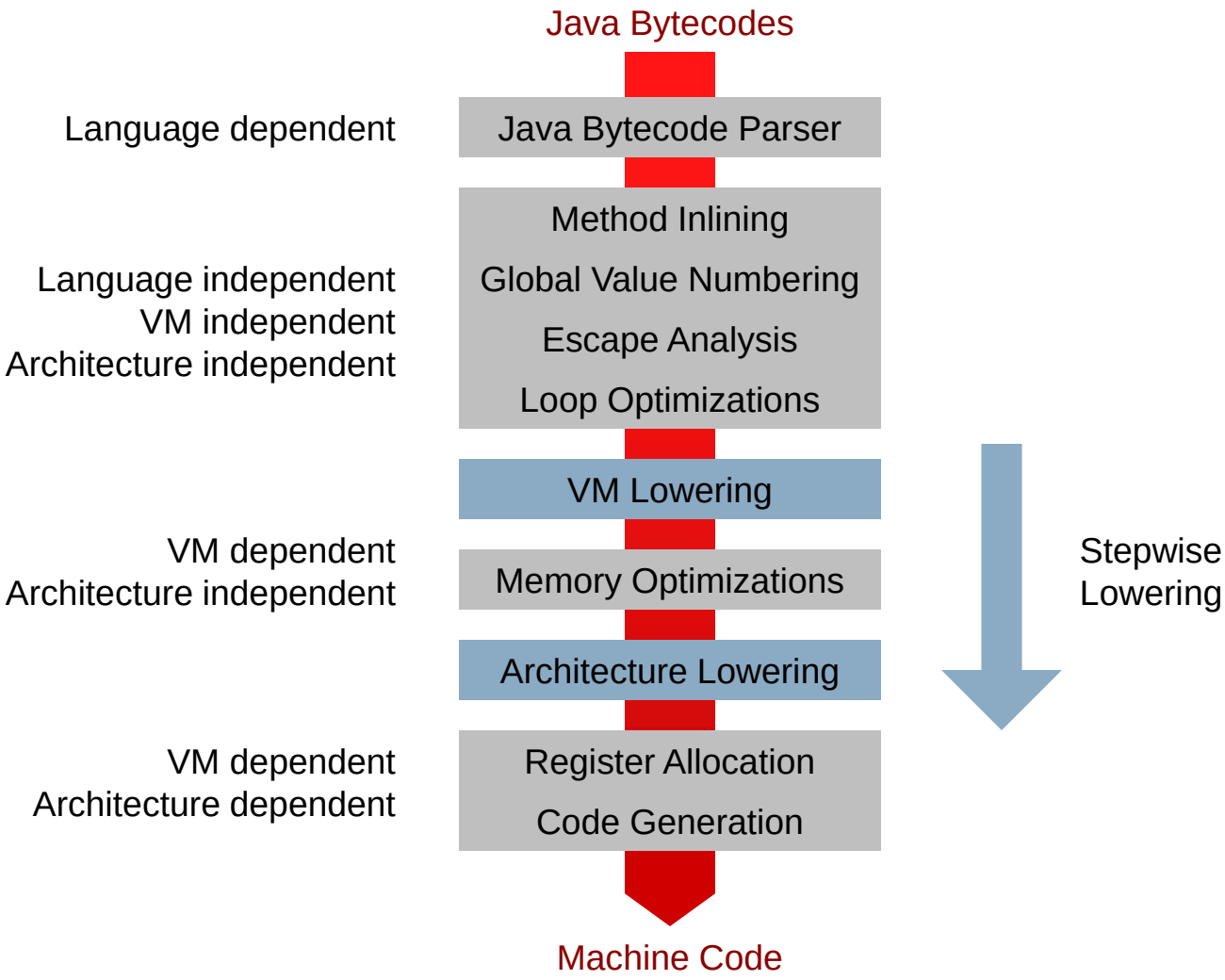


Graal Compiler Vision Statement

"Create an **extensible**, **modular**, **dynamic**, and **aggressive** compiler using **object-oriented** and **reflective** Java programming, a **graph-based** and **visualizable** intermediate representation, and Java **snippets**."

Thomas Würthinger

Graal Compiler Structure



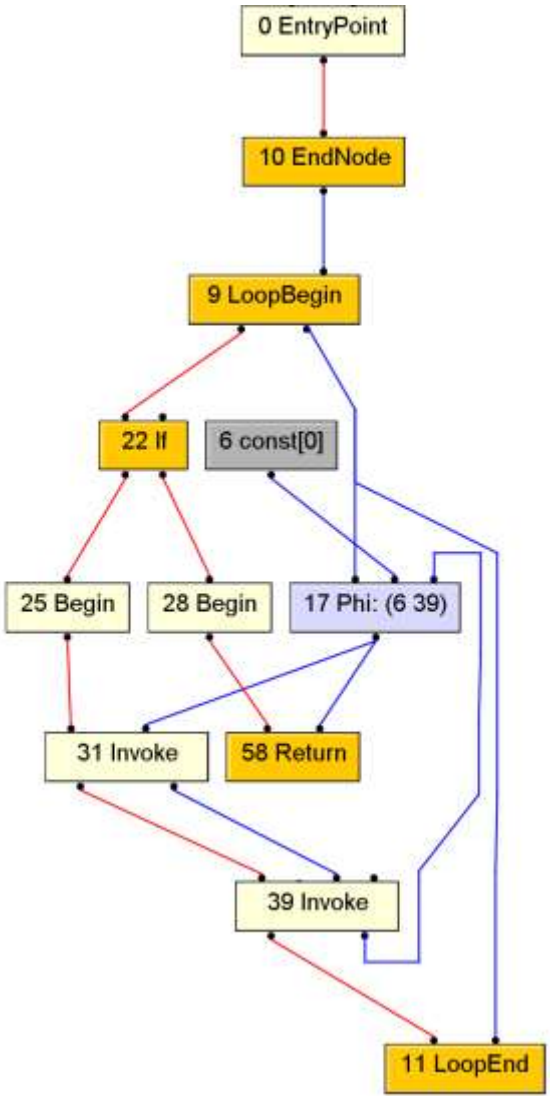
Graal Intermediate Representation

- Hybrid view on inputs: iterable and named fields
- Automatic def-use edges: efficient node substitution
- Automates output to visualization tools

```
class CompareNode extends BooleanNode
                  implements Canonicalizable,
                             LIRLowerable {
  @Input private ValueNode x;
  @Input private ValueNode y;

  private final Condition condition;
  private final boolean unorderedIsTrue;
  ...
}
```

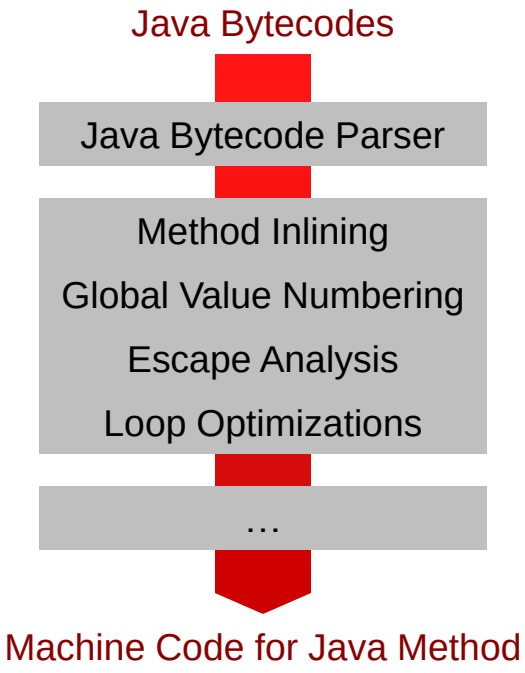
Graal Intermediate Representation



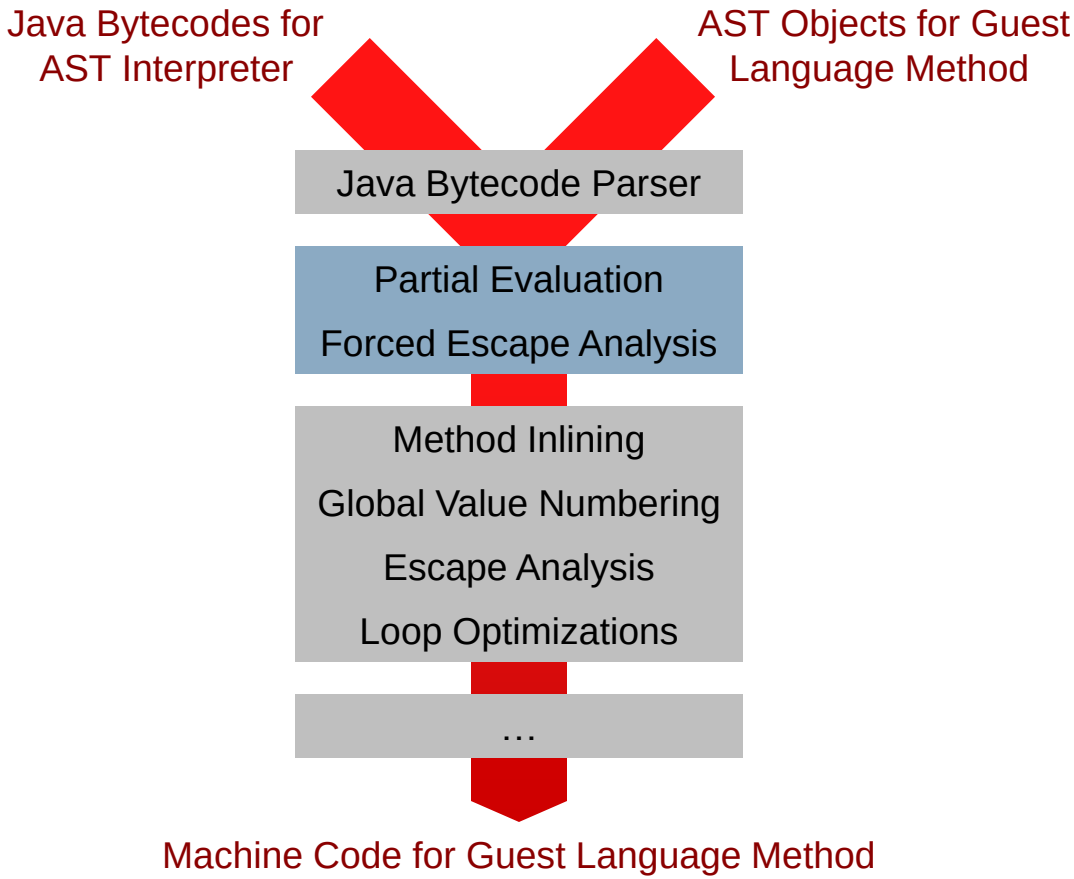
- Visualization tools
  - Graph logged after every phase.
  - Difference view between two states.
  - Plotting works automatically for newly added nodes.
  - Different coloring of control flow and data flow edges.

Partial Evaluation

Graal configured for Java



Graal configured for Truffle



Performance

Simple Language:

```
function main {
  i = 0;
  while (i < 10000000) {
    i = i + 1;
  }
  return i;
}
```

Java int:

```
static int main() {
  int i = 0;
  while (i < 10000000) {
    i = i + 1;
  }
  return i;
}
```

Java BigInteger:

```
static BigInteger main() {
  BigInteger one = BigInteger.valueOf(1);
  BigInteger end = BigInteger.valueOf(10000000);
  BigInteger i = BigInteger.valueOf(0);
  while (i.compareTo(end) < 0) {
    i = i.add(one);
  }
  return i;
}
```

	HotSpot Server VM	Graal VM
Java int	0.03 ms (*)	3.55 ms
Java BigInteger	357 ms	263 ms
Simple Language Interpreted	225 ms	165 ms
Simple Language Compiled	N/A	13.3 ms

Peak performance after warmup runs, so that all relevant methods are compiled

(*) The HotSpot server compiler is clever enough to eliminate the entire loop and just return a constant

Machine Code Comparison

Simple Language:

```
    [...]
    mov   esi, 1
    jmp   L2
L1: mov   eax, esi
    addl  eax, 1
    jo    L3
    mov   r10, {safepoint}
    mov   esi, eax
L2: cmpl  esi, 10000000
    jl    L1
    [...]
L3: mov   r10d, 0xFFFFFFA5
    call  {deoptimizeStub}
```

Input of addition needs to be preserved for deoptimization (`esi` is in debug information for deoptimize)

Integer overflow check

The VM can pause the thread here

Deoptimize: discard machine code and rewrite AST

Java int, compiled with Graal:

```
    [...]
    mov   eax, 1
    jmp   L2
L1: mov   r10, {safepoint}
    addl  eax, 1
L2: cmpl  eax, 10000000
    jl    L1
    [...]
```

Acknowledgments

Oracle Labs
Laurent Daynès
Michael Haupt
Peter Kessler
David Leibs
Doug Simon
Michael Van De Vanter
Christian Wimmer
Mario Wolczko
Thomas Würthinger
Shams Imam (Intern)
Helena Kotthaus (Intern)
Gregor Richards (Intern)
Laura Hill (Manager)

Purdue University
Tomas Kalibera
Floreal Morandat
Prof. Jan Vitek

JKU Linz
Gilles Duboscq
Matthias Grimmer
Christian Häubl
Josef Haider
Christian Humer
Christian Huber
Manuel Rigger
Lukas Stadler
Andreas Wöß
Prof. Hanspeter Mössenböck

Summary

- The Truffle approach for language implementation
  - AST interpreter for language semantics
  - Tree rewriting for type specialization
  - Compilation by partial evaluation of interpreter
    - No need for language implementer to think about the compiler
  - Deoptimization and recompilation for type changes after compilation
- Prototyped languages
  - Dynamic languages: JavaScript
  - Languages for technical computing: J
- Source code availability
  - We intend to make the source code of Truffle, Graal and the language implementations available to bona fide academic researchers on request, under a non-commercial-use license

Hardware and Software

ORACLE

Engineered to Work Together


ORACLE®