Deoptimization for Dynamic Language JITs on Typed, Stack-based Virtual Machines

Madhukar Kedlaya[1], Behnam Robatmili[2], Calin Cascaval[2], Ben Hardekopf[1]

University of California, Santa Barbara[1]
Qualcomm Research Silicon Valley[2]

Motivation

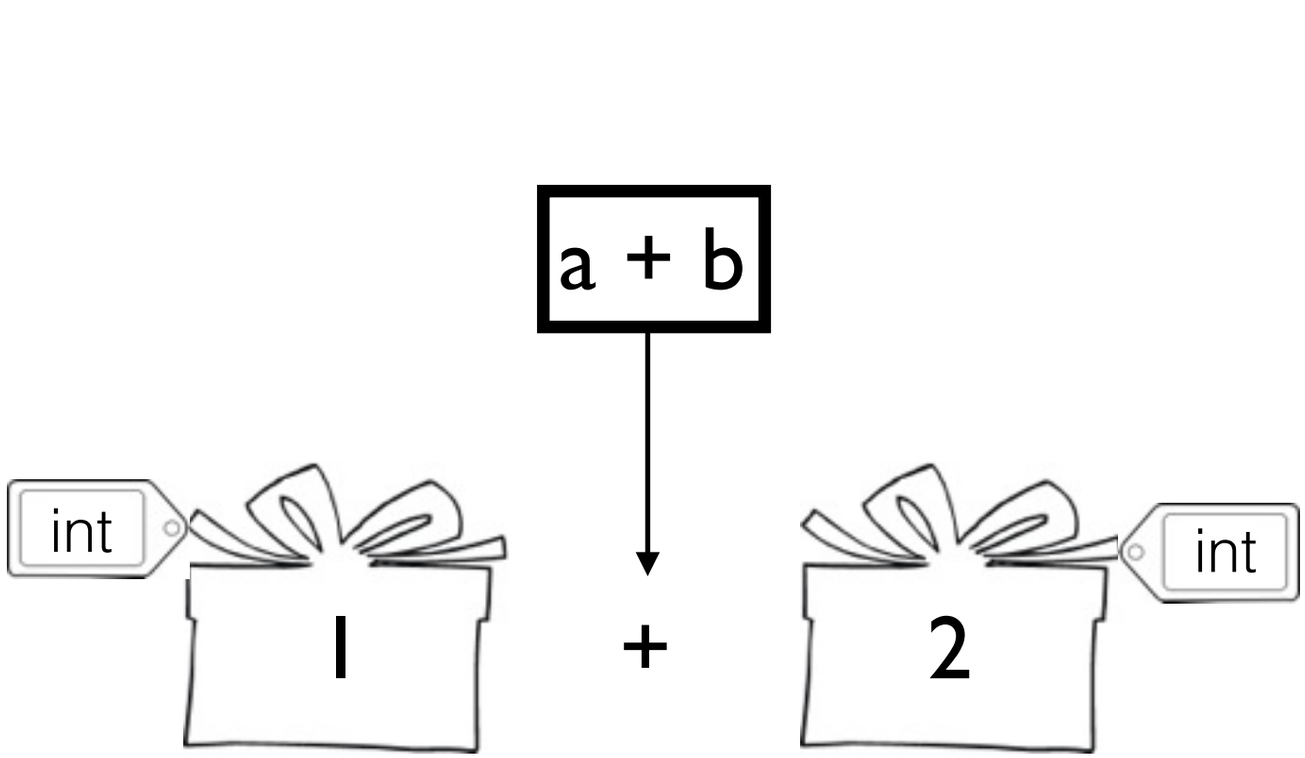
- Implement a performant **dynamic language** implementation **on** a language-level **Virtual Machine** (VM).

- VMs like JVM and CLR are attractive targets for implementing dynamic language runtimes.

- One of the key optimizations is **type specialization**.

Type Specialization

- Dynamic languages operate on dynamic values which are wrapper objects enclosing concrete values.

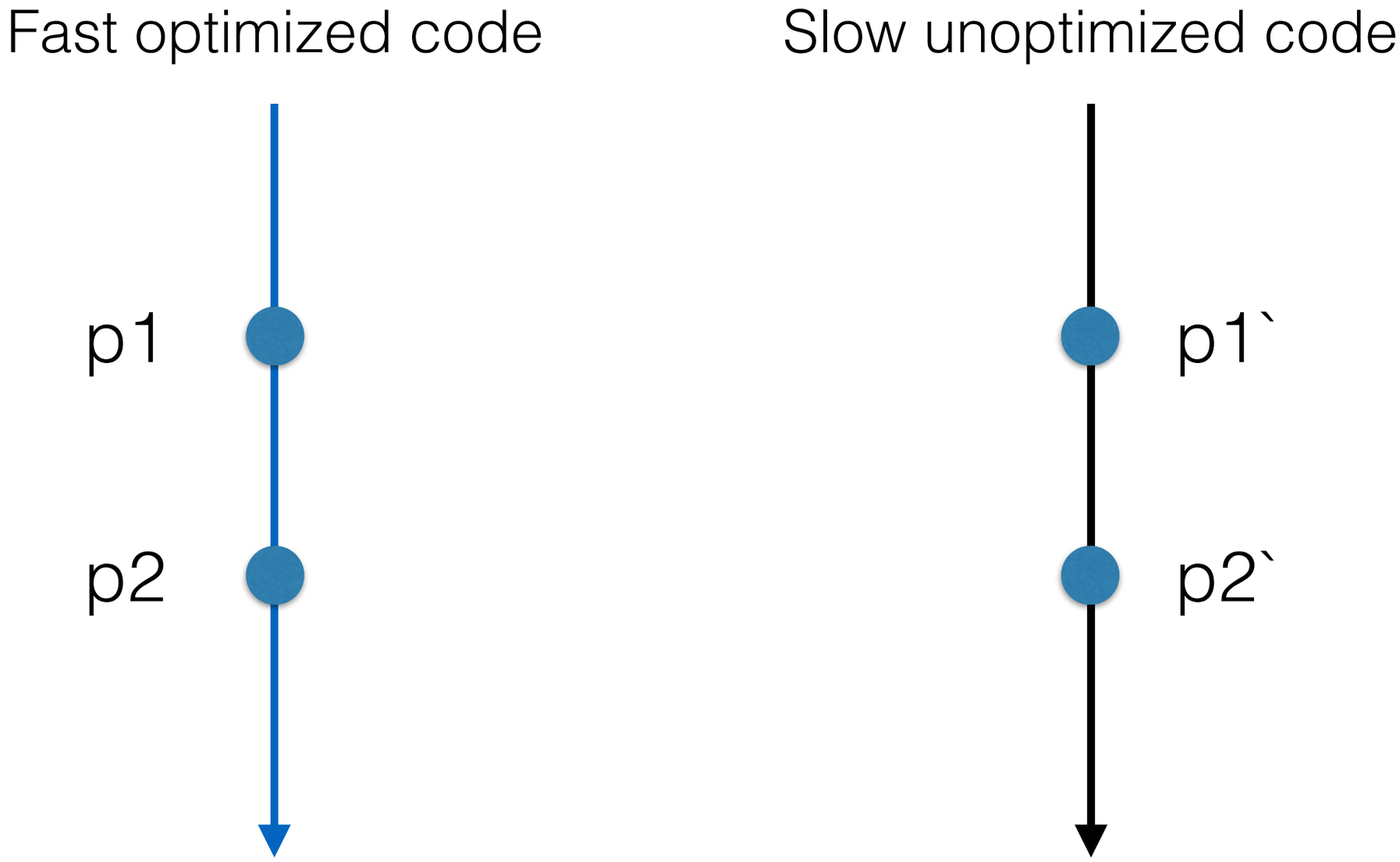


Runtime

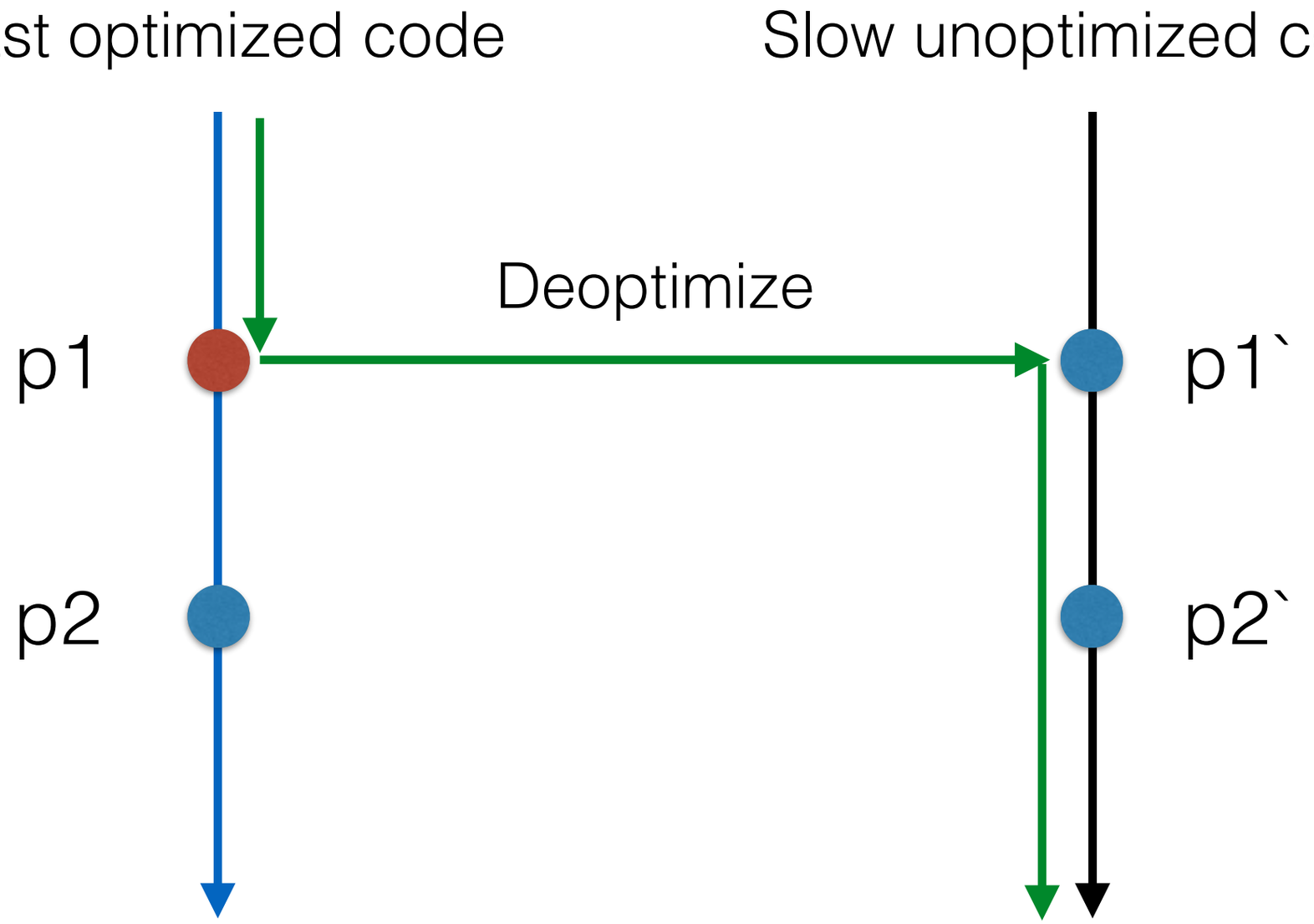
Profile based Type Specialization

```
if (unbox(variable).type == expected type T) {
  T variable’ = unbox(variable)
  // fast path: specialized code for type T
  // computes the result using variable’
}
else {
  // jump to equivalent program point in
  // unspecialized code
}
// use result
```

Deoptimization



Deoptimization



Common Deoptimization Techniques (that do not work on top of VMs)

- On-stack replacement/Code patching.
  - ✳ Cannot modify generated bytecode code during execution.
- Long jumps to unoptimized code.
  - ✳ Violates bytecode verification rules.

Our Approach

- Novel deoptimization approach of code generation without modifying the underlying VM.

- Control transfer —> Exception handling

- State transfer —> Bytecode verifier

- Deoptimization target is a subroutine threaded interpreter.

Control Transfer

```
try {
  if (GetType(variable) != ProfiledType) {
    /* capture state here */
    throw new GuardFailureException(subroutineIndex);
  }
  /* fast Path */
}

catch (GuardFailureException e) {
  /* capture state here */
  SubroutineThreadedInterp(e.subroutineIndex, state);
}
```

State of Execution

- **state** data structure captures the current state of execution of the function.

- Two parts.
  - ✳ Values of **local variables**.
  - ✳ Values in **operand stack**.

State Transfer

```
try {
  if (GetType(variable) != ProfiledType) {
    for (value in operandStack) {
      state.stack.enqueue(value);
    }
    throw new GuardFailureException(subroutineIndex);
  }
  /* Fast Path */
  …
}
catch (GuardFailureException e) {
  for (variable in localVariables) {
    state.variables[variable] = GetValue(variable);
  }
  SubroutineThreadedInterp(e.subroutineIndex, state);
}
```

State Transfer

```
for (value in operandStack) {
  state.stack.enqueue(value);
}
```

Q: Which enqueue method does the runtime call?
enqueue(int)?
enqueue(string)?
enqueue(double)?
…

A: Depends on the types of values present in the operand stack.

State Transfer

- Bytecode verifier checks type-safety of the Common Intermediate Language (CIL) code while generating it.

- Bytecode verifier uses a **type stack** to track the types of values in operand stack

- Code generator uses the type stack to generate the calls to proper enqueue methods at each of the deoptimization points.

Results

- Implemented in MCJS, a research JavaScript engine running on top of Mono 3.2.3.

- Benchmarks: Sunspider, V8, and JS1k web application benchmark suites.

- MCJS with deoptimization is on an average **1.18x** (up to 2.6x) faster than MCJS with fast path + slow path.

- MCJS with deoptimization is on an average **3.18x** (up to 9.9x) faster than IronJS (DLR).