Practical Second Futamura Projection

Partial Evaluation for High-Performance Language Interpreters

Florian Latifi
GraalVM Meetup
August 13, 2020

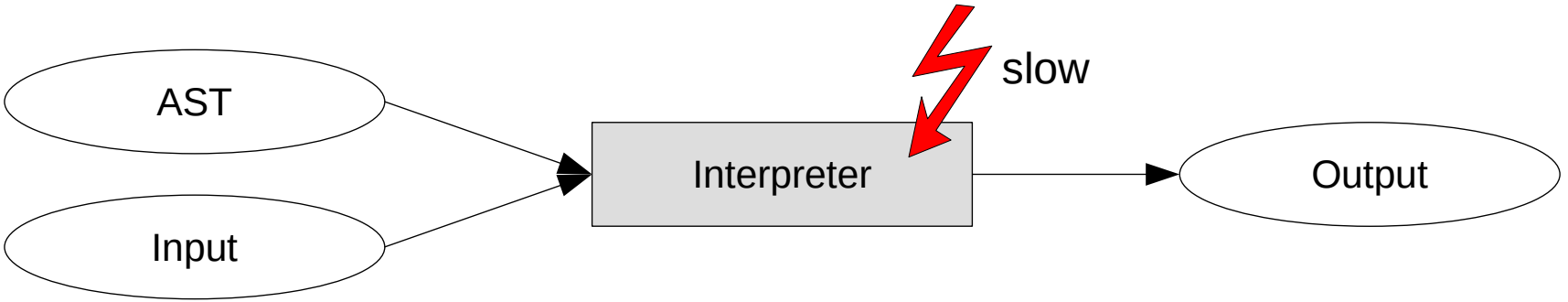
Contents

- Truffle Compilation
- Problem & Motivation
- Approach
- Live Demo

- Current Status
- First Results
- Next Steps

Truffle Compilation

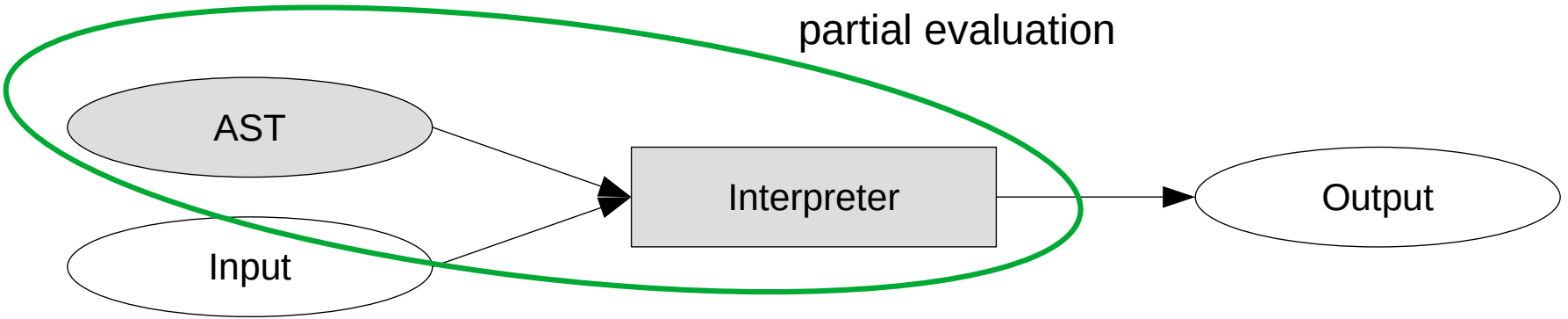
1st Futamura Projection



- Problem:
  - execution of AST interpreter is slow
  - virtual dispatch between AST nodes
- Goal:
  - speed up execution through compilation

Truffle Compilation

1st Futamura Projection



- 1st Futamura Projection:
  - assumption: AST is constant input to interpreter
  - specialize interpreter with respect to AST
  - result: compiled version of AST

Problem & Motivation

General

- Truffle has bad start-up performance:
  - slow first-tier AST interpreter & last-tier optimizing compiler
  - no fast “middle-tier” compiler
- Graal can serve as fast middle-tier compiler:
  - by disabling expensive optimization phases
- But partial evaluation is slow too:
  - takes ~50% of overall compile time for large ASTs
  - making Graal infinitely fast only gives 2x speedup

- Goal:
  - speed up partial evaluation

Problem & Motivation

Why PE is slow

```
graph = parse("OptimizedCallTarget#call(Object[] args)");
pushWorklist(graph.startNode);

do {
  node = popWorklist();
  node = simplify(graph, node);

  if (node instanceof Invoke) {
    inlineeGraph = parse(node.targetMethod);
    pushWorklist(inlineeGraph.startNode);
  } else {
    pushWorklist(node.successors):
  }
} while (worklistNotEmpty());
```

- Generic inlining algorithm:
  - allocates IR graphs for given target methods
  - iterates IR & simplifies encountered nodes
  - kind of abstract interpretation
- What we want:
  - fast PE code that emits target programs directly

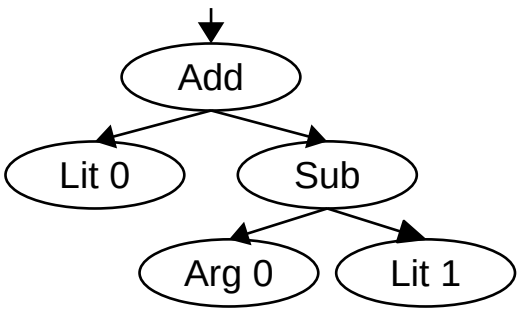
Problem & Motivation

Example

User-Level Function

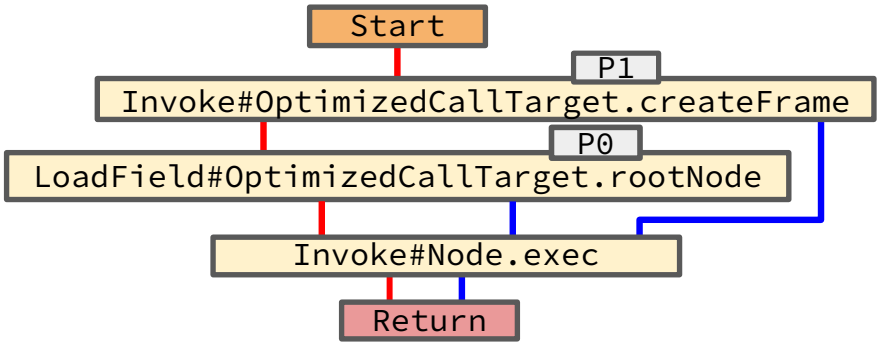
```
int foo(int n) {
  return 0 + (n - 1);
}
```

Call Target



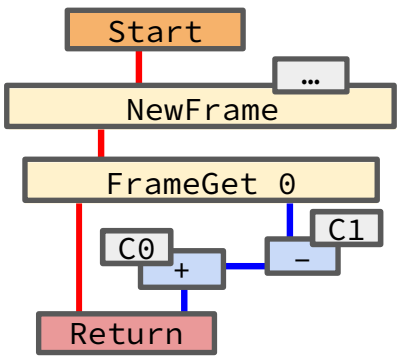
① after parse

`OptimizedCallTarget#call(Object[] args)`



as fast as possible

② after PE



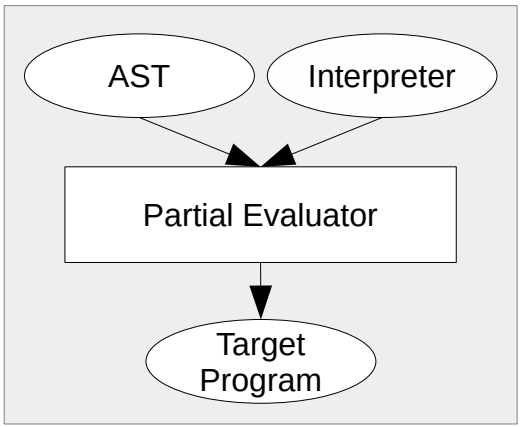
Approach

2nd Futamura Projection

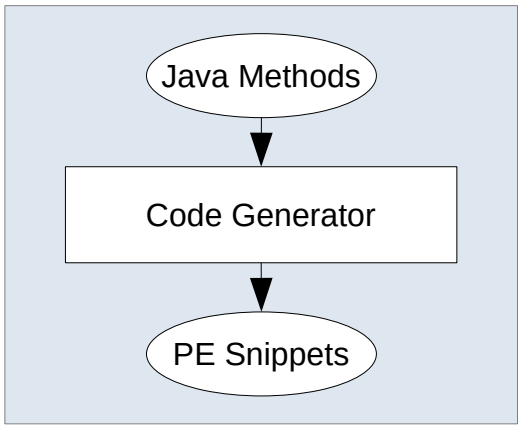
- Ahead of time:
  - parse Java methods of an AST interpreter (e.g., all `Node#execute` methods)
  - generate fast PE snippets that emit partially evaluated interpreter instructions for invocations of these methods
- At run time:
  - given: an AST that represents a user program
  - assumption: AST is constant input to the interpreter
  - dispatch between PE snippets and emit partially evaluated interpreter instructions for the whole user program

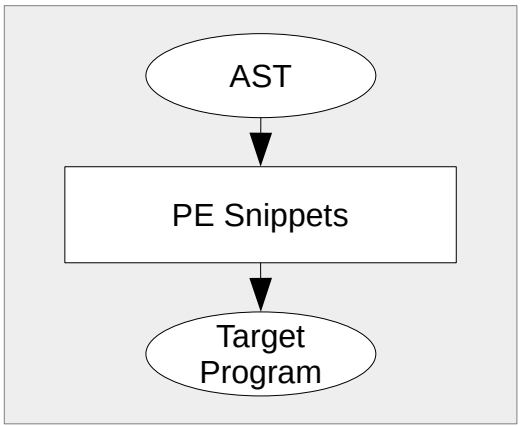
Approach

F1 vs. F2



1st Futamura Projection

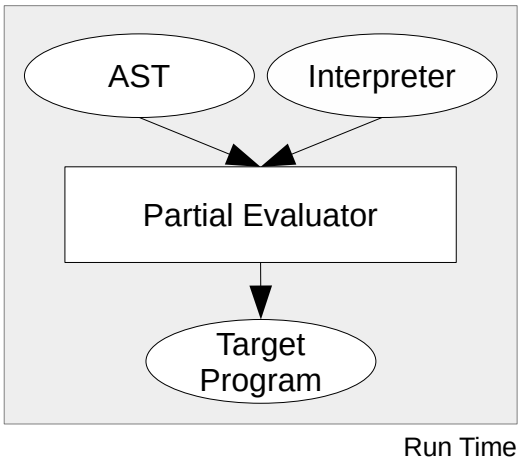




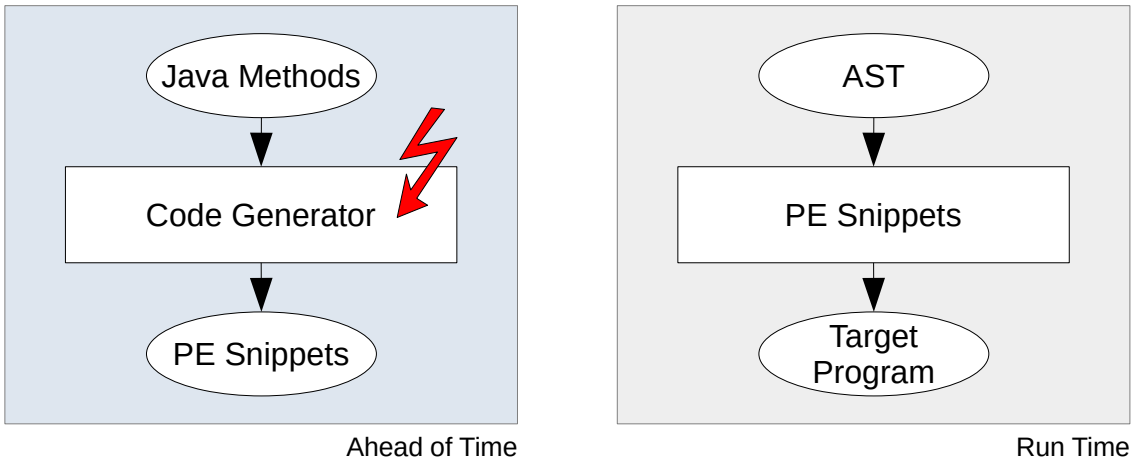
2nd Futamura Projection

Approach

F1 vs. F2



1st Futamura Projection

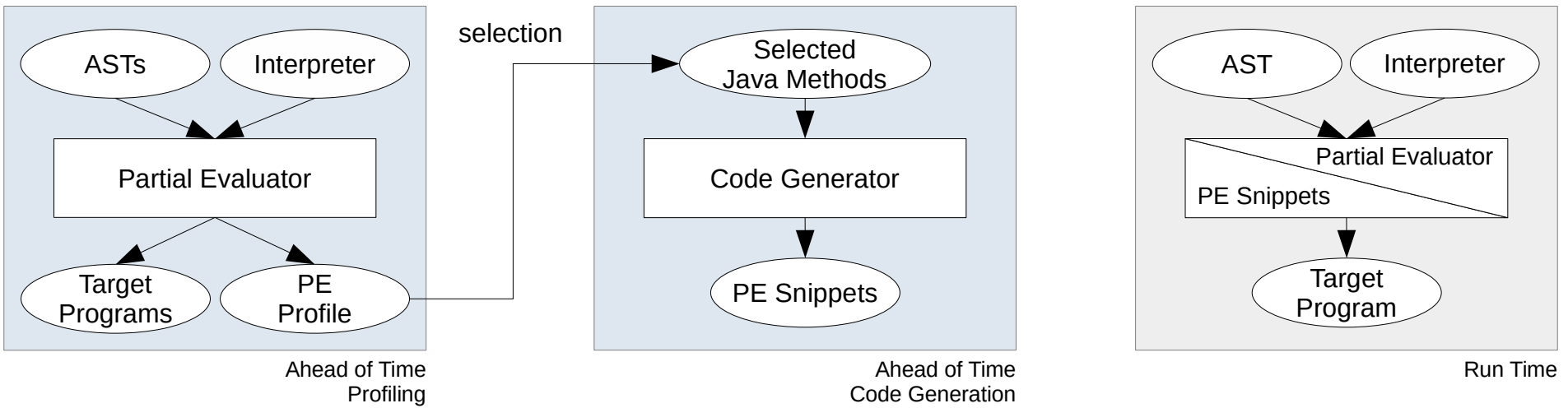


2nd Futamura Projection

- Code generation:
  - quickly leads to code-size explosion
  - we can't generate PE code for all interpreter methods:
    - full F2 is not practical

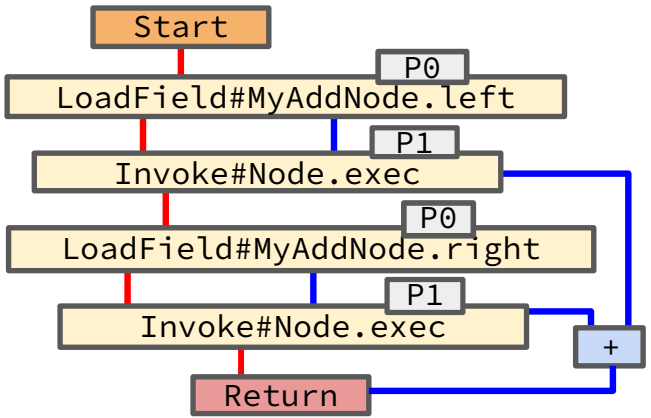
Approach
Practical F2

- Generate PE snippets only for **important** interpreter methods
- Dispatch between snippets & fall back to generic F1 if necessary
- Avoid JIT-compilation of snippets (libgraal, native image)



Approach
Example

MyAddNode#exec(VirtualFrame frame)




```
void decode(PEGraphBuilderContext ctx) {
  ValueNode lField = tryConstantFold("MyAddNode#left", ctx, ctx.scope.args[0]);
  ValueNode lExec = trySimplifyInvoke("Node#exec", ctx, lField, ctx.scope.args[1]);
  ValueNode rField = tryConstantFold("MyAddNode#right", ctx, ctx.scope.args[0]);
  ValueNode rExec = trySimplifyInvoke("Node#exec", ctx, rField, ctx.scope.args[1]);
  ValueNode retVal = ctx.add(AddNode.create(lExec, rExec);

  registerNode(ctx.scope.callerLoopScope, ctx.scope.invokeOrderId, retVal);
  registerNode(ctx.scope.callerLoopScope, ctx.scope.nextOrderId, ctx.current());
}
```

Approach

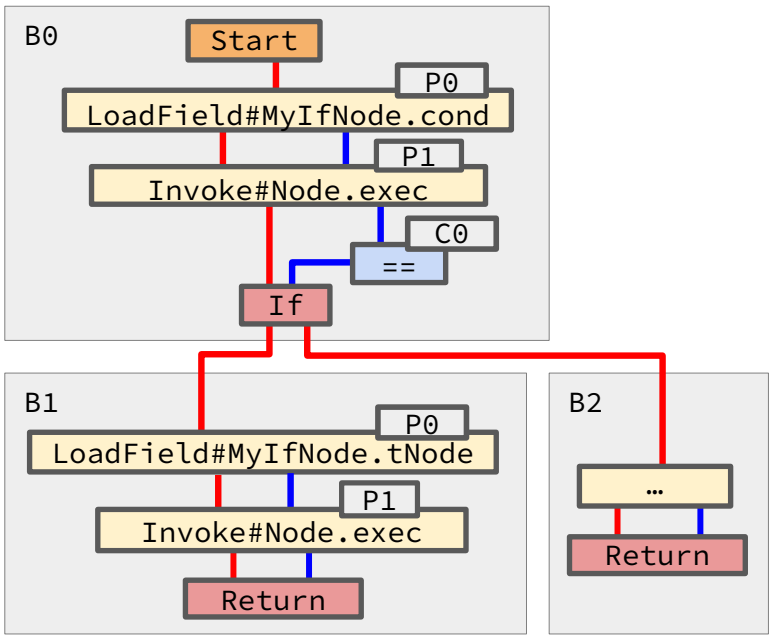
Complex Constructs

- What about more complex constructs?
  - splits & merges
  - return & unwind paths
  - dead-end paths (fixed guards, deopts)
  - loops & loop explosion
  - …

- Idea:
  - one method per basic block
  - intermediate value nodes & fixed anchor nodes
  - flags to indicate if a block must be emitted
  - store this state in a loop scope object

Approach
Example

MyIfNode#exec(VirtualFrame frame)




```
class F2LoopScope extends LoopScope {
  boolean genB1; FixedWithNextNode anchorB1;
  boolean genB2; FixedWithNextNode anchorB2;
}

void decode(PEGraphBuilderContext ctx) {
  F2LoopScope ls = new F2LoopScope();
  genB0(ctx, ls);
  if (ls.genB1) genB1(ctx, ls);
  if (ls.genB2) genB2(ctx, ls);
}

void genB0(PEGraphBuilderContext ctx, F2LoopScope ls) {
  ValueNode cField = tryConstantFold("MyIfNode#cond", ctx, ctx.scope.args[0]);
  ValueNode cExec = trySimplifyInvoke("Node#exec", ctx, cField, ctx.scope.args[1]);
  ValueNode eqNode = ctx.add(EqualsNode.create(cExec, ConstantNode.for(0)));

  if (eqNode.isTautology()) {
    ls.genB1 = true;
    ls.anchorB1 = ctx.current();
  } else if (eqNode.isContradiction()) {
    ls.genB2 = true;
    ls.anchorB2 = ctx.current();
  } else {
    ls.genB1 = true;
    ls.genB2 = true;
    ls.anchorB1 = ctx.add(new BeginNode());
    ls.anchorB2 = ctx.add(new BeginNode());
    ctx.append(IfNode.create(eqNode, ls.anchorB1, ls.anchorB2);
  }
}

void genB1(PEGraphBuilderContext ctx, F2LoopScope ls) { … }
void genB2(PEGraphBuilderContext ctx, F2LoopScope ls) { … }
```

Live Demo

- A few notes before we start:
  - work is based on very old Graal version – rebase ongoing
  - F1+F2 integration not fully finished – bugs & performance regressions

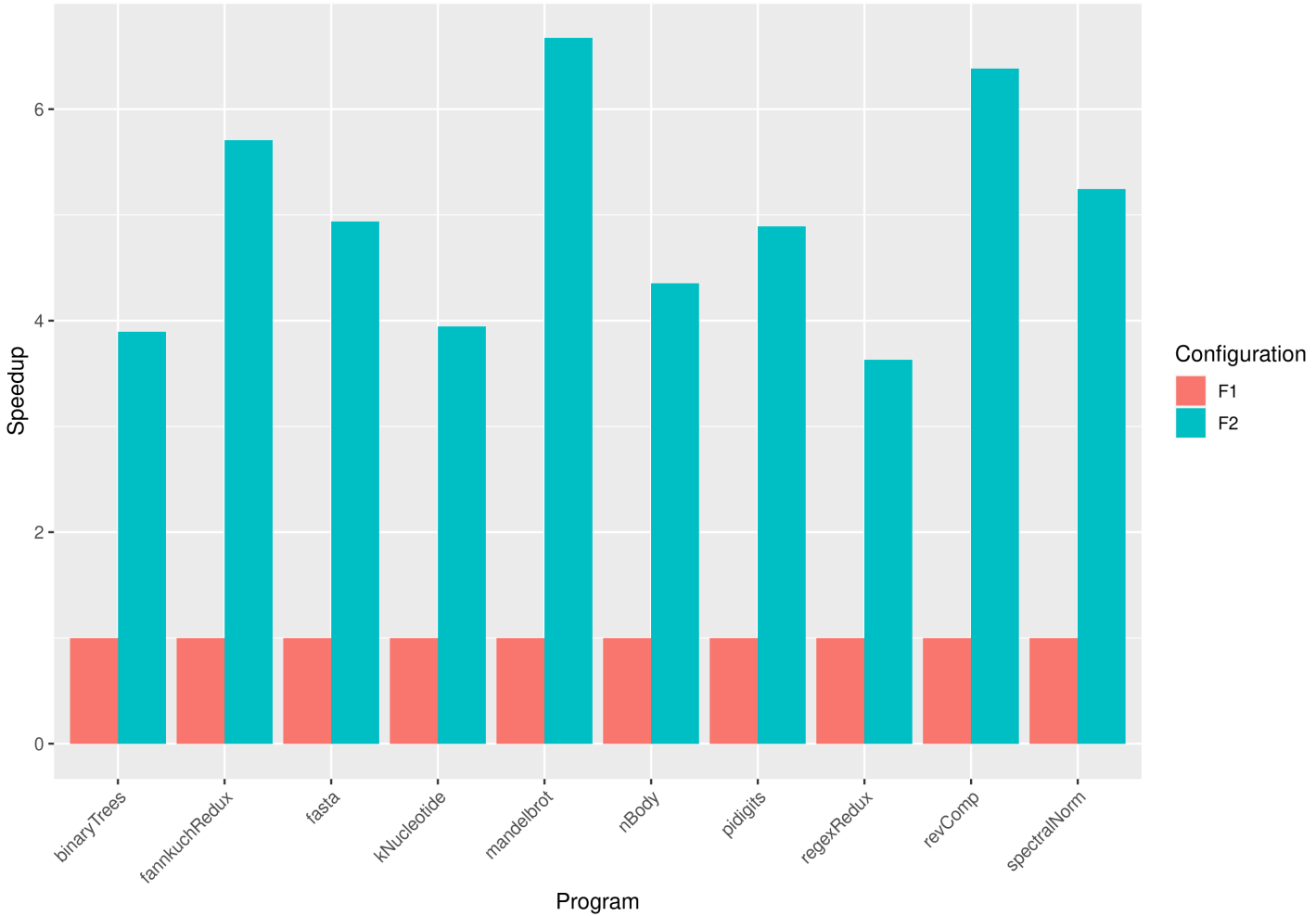
Current Status

- F2 code generation works:
  - full compilation pipeline (PE + Graal compilation)
  - we produce same graphs as F1 with similar run-time performance
- We handle:
  - canonicalization / constant folding / simplification
  - complex control flow
  - loop explosion (full unroll, full explode, full explode until return)
  - frame states
- F1+F2 integration in principle works:
  - we can also fall back to F1 for constructs we don’t support yet (lambdas, merge explode)
  - bugs & performance regressions that need to be fixed

First Results

F1 vs. F2 – no integration

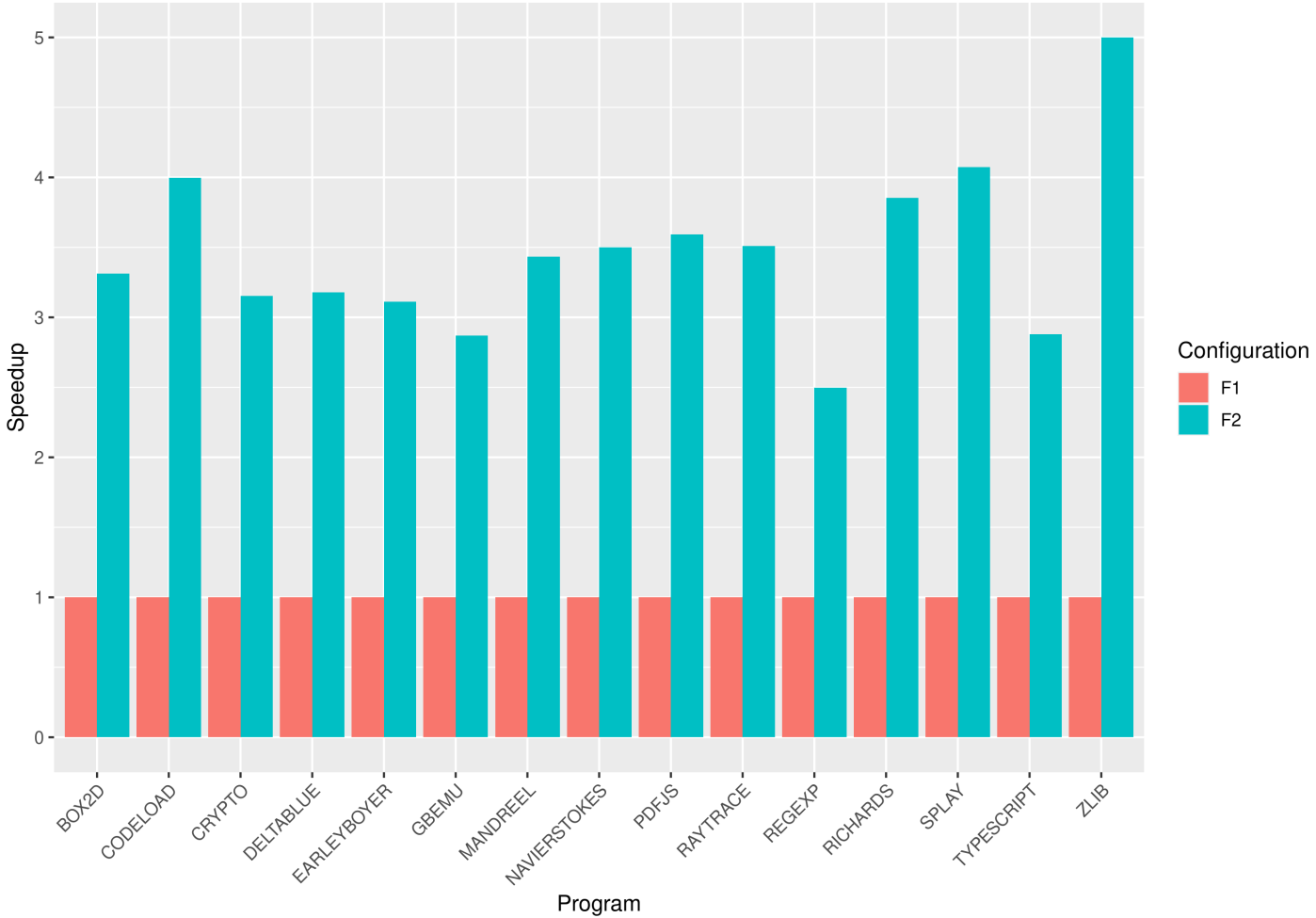
GraalJS CLBG



First Results

F1 vs. F2 – no integration

GraalJS Octane



Next Steps

- Rebase on top of Graal master / current release
- Bug & performance regression fixing in F1+F2
- Performance optimizations in F2
- Experiment with PE profiles & method selections

- Write main paper

Discussion / QA