UNIVERSITY OF WATERLOO
FACULTY OF MATHEMATICS
David R. Cheriton School
of Computer Science

Implementing a Functional Language for Flix

Master's Thesis Presentation

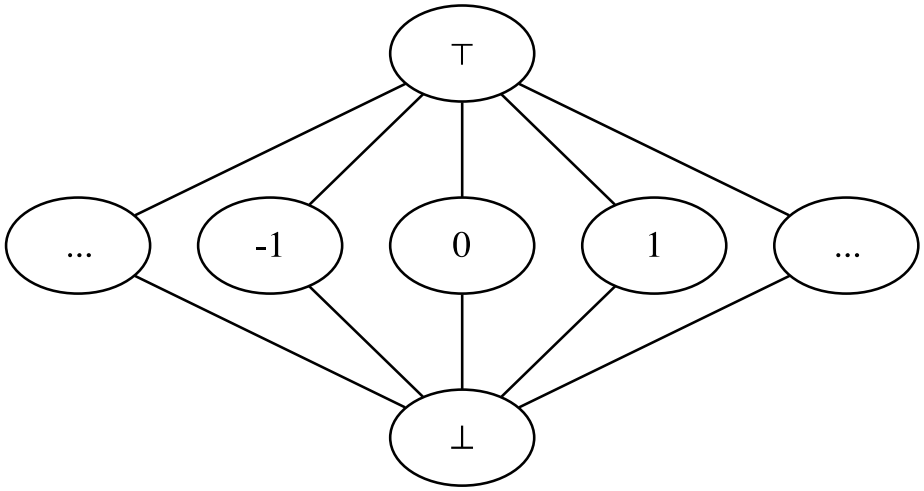
Ming-Ho Yee

Supervisor: Ondřej Lhoták

September 1, 2016

Static Analysis

- Analyze software without executing it
- Model abstract program state with lattice elements



Constant propagation lattice

Datalog

- Datalog is a declarative programming language
  - “What not how”
  - Has been used for pointer analyses
- But Datalog has limitations:
  - No lattices
  - No functions
  - Poor interoperability

A Language for Static Analysis

- Flix extends Datalog with lattices and functions
  - Logic language
  - Functional language
- Flix is implemented on the JVM

Constant Propagation in Flix (1/2)

```
enum Constant {
  case Top, case Cst(Int), case Bot
}

def leq(e1: Constant, e2: Constant): Bool =
  match (e1, e2) with {
    case (Bot, _)            => true
    case (Cst(n1), Cst(n2)) => n1 == n2
    case (_, Top)            => true
    case _                   => false
  }

def lub(e1: Constant, e2: Constant): Constant = …
def glb(e1: Constant, e2: Constant): Constant = …

def sum(e1: Constant, e2: Constant): Constant = …
```

Constant Propagation in Flix (2/2)

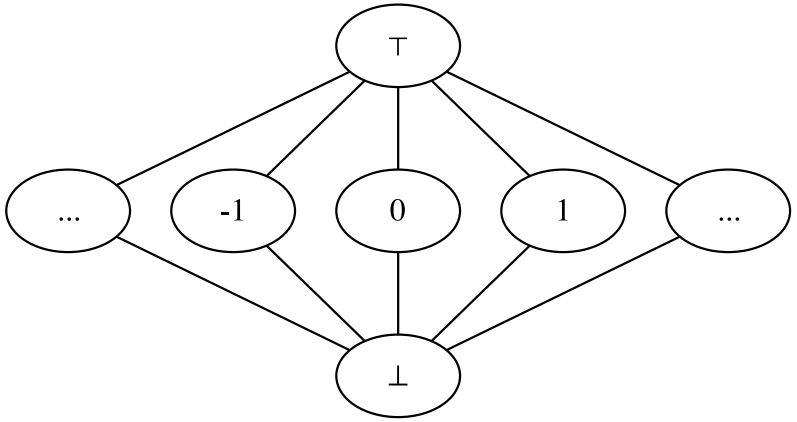
```
// analysis inputs
rel AsnStm(r: Str, c: Int)
rel AddStm(r: Str, x: Str, y: Str)

// analysis outputs
lat LocalVar(k: Str, v: Constant)

// rules
LocalVar(r, Cst(c)) :- AsnStm(r, c).

LocalVar(r, sum(v1, v2)) :- AddStm(r, x, y),
                            LocalVar(x, v1),
                            LocalVar(y, v2).
```

Constant Propagation Example

```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

```
// output facts
```

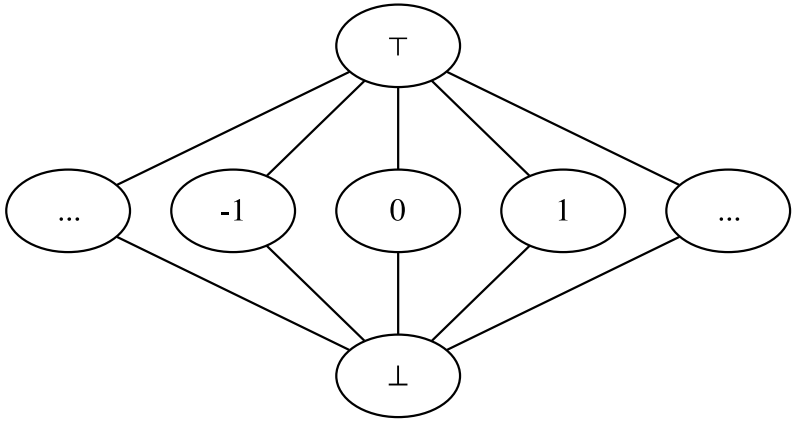


Constant Propagation Example

```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

```
// output facts
LocalVar("x", Cst(0)).
```

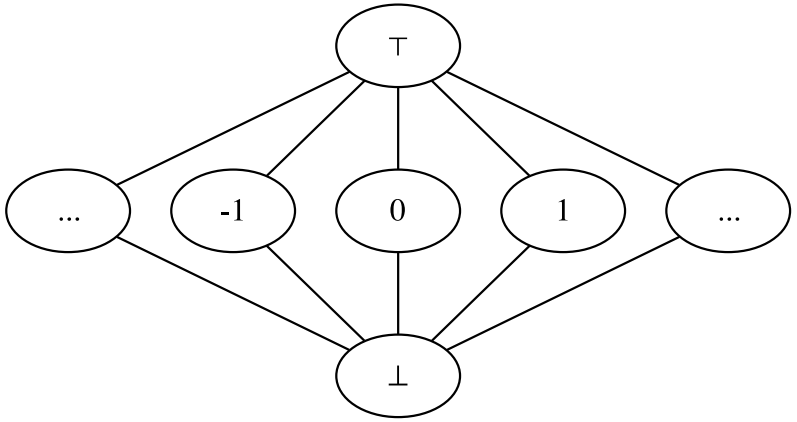


Constant Propagation Example

```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

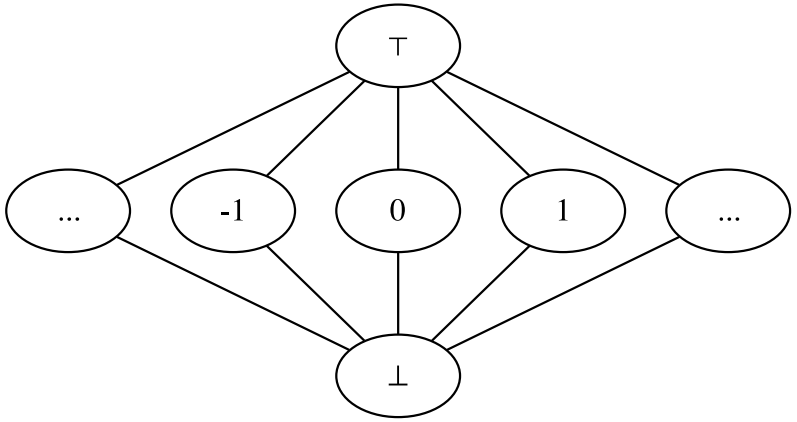
```
// output facts
LocalVar("x", Cst(0)).
LocalVar("x", Cst(1)).
```



Constant Propagation Example

```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

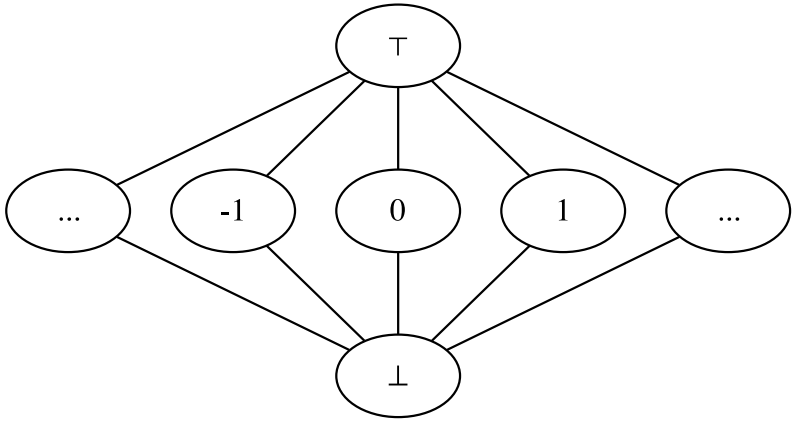



```
// output facts
~~LocalVar("x", Cst(0)).~~
~~LocalVar("x", Cst(1)).~~
```

Constant Propagation Example

```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```



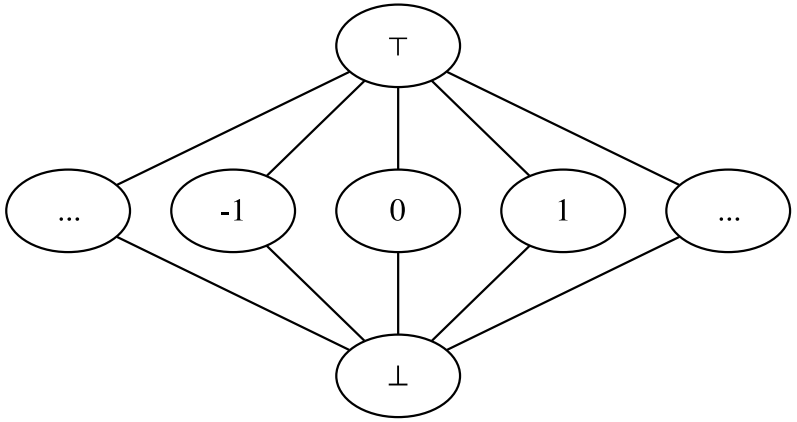

```
// output facts
LocalVar("x", Cst(0)).
LocalVar("x", Cst(1)).
LocalVar("x", lub(Cst(0), Cst(1))).
```

Constant Propagation Example

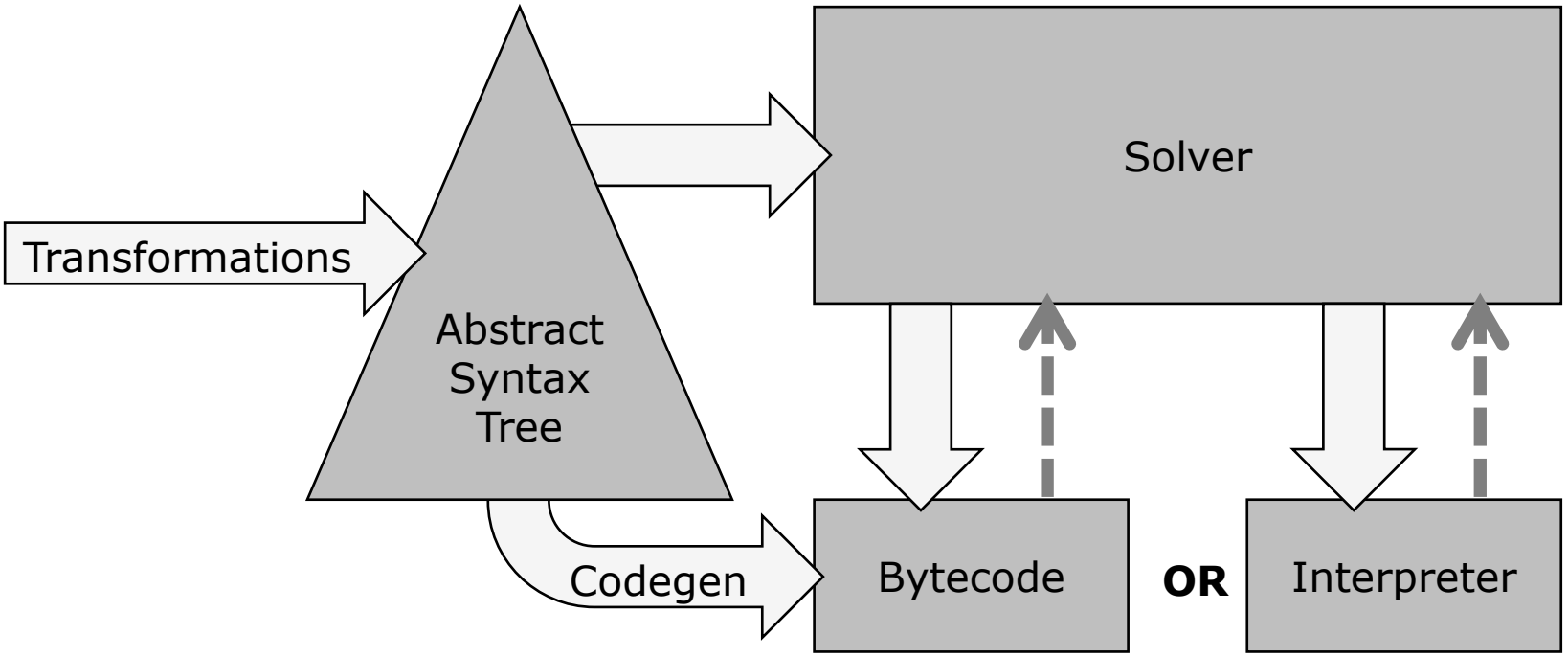
```
LocalVar(r, Cst(c)) :- AsnStm(r, c).
```

```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```

```
// output facts
~~LocalVar("x", Cst(0)).~~
~~LocalVar("x", Cst(1)).~~

LocalVar("x", Top).
```



Back-end Architecture



AST Transformations

- Pattern-matching compilation
- Closure conversion and lambda lifting
- Variable numbering
- Optimizations (future work)

AST Transformations

- **Pattern-matching compilation**
- **Closure conversion and lambda lifting**
- Variable numbering
- Optimizations (future work)

Compiling Pattern Matching

```
// before
match x with {
  case PAT1 => EXP1
  case PAT2 => EXP2
  case _    => ERROR // implicit default case
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_2 = λ() if (PAT2 succeeds) EXP2 else err() in
let e_1 = λ() if (PAT1 succeeds) EXP1 else e_2() in
  e_1()
```

Wildcard Pattern

```
// before
match x with {
  case _ => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() true in
  e_1()
```

Variable Pattern

```
// before
match x with {
  case n => n + 1
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() (let n = v_0 in n + 1) in
  e_1()
```

Literal Pattern

```
// before
match x with {
  case 42 => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() if (v_0 == 42)
                true
              else
                err() in
  e_1()
```

Tag Pattern

```
enum E { case A(Int), case B(Str) }

// before
match x with {
  case E.A(42) => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() if (CheckTag(A, v_0))
                let n_0 = GetTagValue(v_0) in
                  if (n_0 == 42) true else err()
              else
                err() in
  e_1()
```

Tuple Pattern

```
// before
match x with {
  case (4, 2) => true
}

// after
let v_0 = x in
let err = λ() ERROR in
let e_1 = λ() let n_0 = GetTupleIndex(v_0, 0) in
              let n_1 = GetTupleIndex(v_0, 1) in
                if (n_0 == 4)
                  if (n_1 == 2) true else err()
                else
                  err() in
  e_1()
```

Lambda Functions

- Functions are first-class
  - Can be nested, stored in variables, passed as arguments, returned from functions…
- No nested methods in bytecode
- Target of a call must be a method reference

Lambda Lifting (1/3)

```
// before
def f() = let g = λ(x, y) x+y in
              g(1, 2)

// after lifting
def f() = g(1,2)
def g(x, y) = x+y
```

Lambda Lifting (2/3)

```
// before
def f(a) = let g = λ(x, y) a+x+y in
               g(1, 2)

// after lifting
def f(a) = g(1,2)
def g(x, y) = a+x+y // what is a?
```

Lambda Lifting (3/3)

```
// before
def f(a) = let g = λ(x, y) a+x+y in
               g(1, 2)

// after rewriting
def f(a) = let g = λ(a', x, y) a'+x+y in
               g(a, 1, 2)

// after lifting
def f(a) = g(a, 1, 2)
def g(a', x, y) = a'+x+y
```

Lambda Lifting…?

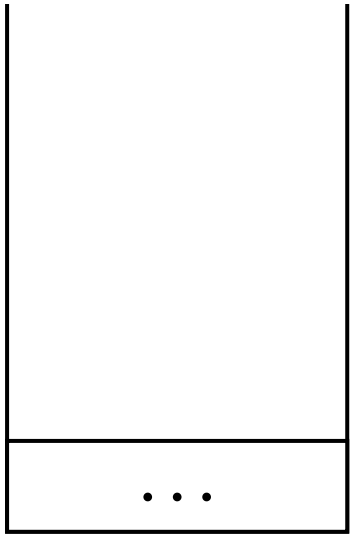
```
def f(a) = let g = λ(x, y) a+x+y in
               h(g, 1, 2)
def h(g', x, y) = g'(x, y) // how to rewrite g'?
```

Closure Conversion

```
// after closure conversion
def f(a) = let g = MkClosure(λ(a', x, y) a'+x+y, a) in
               h(g, 1, 2)
def h(g', x, y) = ApplyClosure(g', x, y)

// after lifting
def f'(a', x, y) = a'+x+y
def f(a) = let g = MkClosure(f', a) in
               h(g, 1, 2)
def h(g', x, y) = ApplyClosure(g', x, y)
```

Code Generation

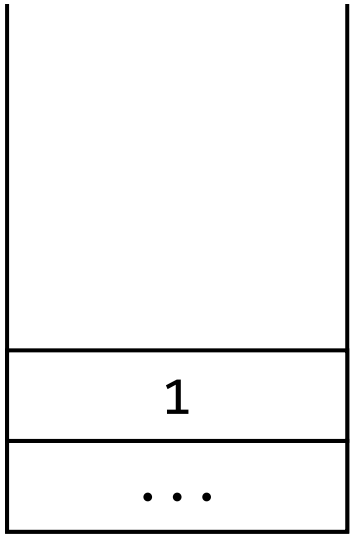
- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Code Generation

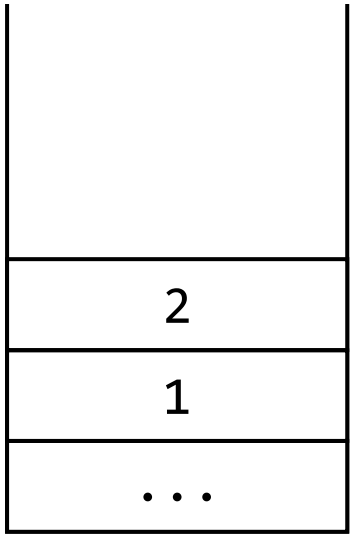
- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Code Generation

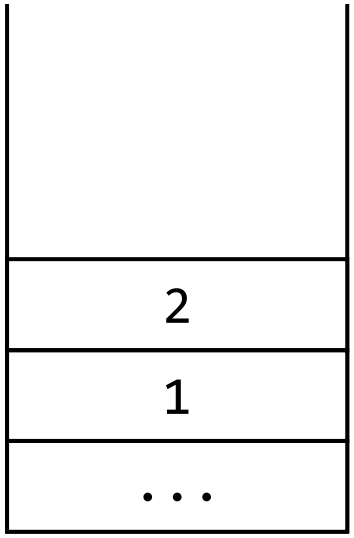
- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Code Generation

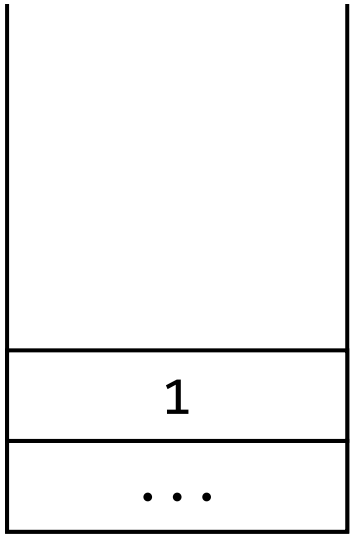
- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Code Generation

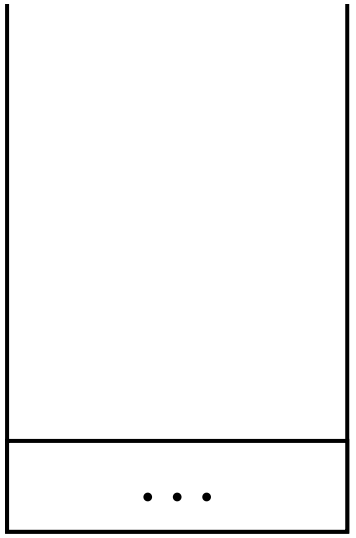
- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Code Generation

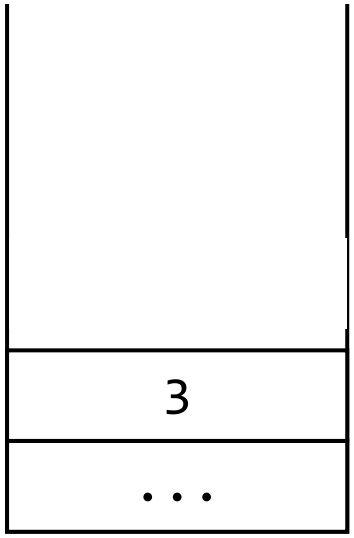
- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Code Generation

- Interpreter is easy to understand and maintain
- Code generator is better for performance
  - Targets the JVM
- JVM is a stack machine
  - All operands and intermediate values placed on stack




```
ICONST_1
ICONST_2
IADD
```

Loading and Executing Bytecode

- How to call generated bytecode?
- Use reflection and Java's `ClassLoader`
  - Represent bytecode method as a `Method` object
  - Call with `m.invoke()`

Representing Flix Values

Flix Type	**JVM Type**	
	Primitive	Reference
`Int8`	`byte`	`java.lang.Byte`
`Int16`	`short`	`java.lang.Short`
`Int32`	`int`	`java.lang.Integer`
`Str`		`java.lang.String`
`Tag`		`Value.Tag`
`Tuple`		`Value.Tuple`

Integer Semantics

- JVM and Flix support integers with 8, 16, 32, 64 bits
  - Two's complement representation

- JVM sign-extends 8-bit and 16-bit integers
- Flix does not sign-extend integers
  - Different overflow semantics

Integer Overflow (1/2)

$$
\begin{array}{rll}
 & 01000000_2 & = \quad 64_{10} \\
+ & 01000000_2 & = \quad 64_{10} \\
\hline
\end{array}
$$

Integer Overflow (1/2)

$$
\begin{array}{rll}
 & 01000000_2 & = \quad 64_{10} \\
+ & 01000000_2 & = \quad 64_{10} \\
\hline
 & 10000000_2 &
\end{array}
$$

Integer Overflow (1/2)

$$
\begin{array}{rl}
 & 01000000_2 = \ \ 64_{10} \\
+ & 01000000_2 = \ \ 64_{10} \\
\hline
 & 10000000_2 = -128_{10}
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rll}
 & 01000000_2 & = \quad 64_{10} \\
+ & 01000000_2 & = \quad 64_{10} \\
\hline
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
\hline
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 01000000_2 = \quad 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 = \quad 64_{10} \\
\hline
 & 00000000\ 00000000\ 00000000\ 10000000_2
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
\hline
 & 00000000\ 00000000\ 00000000\ 10000000_2 = 128_{10}
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
\hline
 & 00000000\ 00000000\ 00000000\ 10000000_2 = 128_{10} \\
 & \\
 & 10000000_2
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rll}
 & 00000000\ 00000000\ 00000000\ 01000000_2 & = \quad 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 & = \quad 64_{10} \\
\hline
 & 00000000\ 00000000\ 00000000\ 10000000_2 & = \ 128_{10} \\
 & & \\
 & 11111111\ 11111111\ 11111111\ 10000000_2 &
\end{array}
$$

Integer Overflow (2/2)

$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
\hline
 & 00000000\ 00000000\ 00000000\ 10000000_2 = 128_{10} \\
 & \\
 & 11111111\ 11111111\ 11111111\ 10000000_2 = -128_{10}
\end{array}
$$

Implementing Closures…?

```
// Scala
val a = 10
val f = (x: Int, y: Int) => a + x + y
f(1, 2) // 13
```

```
// Compiled Scala
class anon$fun(a$0: Int) extends Function2 {
  def apply(x: Int, y: Int) = a$0 + x + y
}
val a = 10
val f = new anon$fun(a)
f.apply(1, 2) // 13
```

Using `invokedynamic`

- Flix uses the same strategy as Java 8 and Scala 2.12
  - Create closure object with `invokedynamic`
- `invokedynamic` represents a dynamic call site
  - Initially, target method is unknown
  - `invokedynamic` calls bootstrap method to link target
  - Subsequent calls skip bootstrap and directly call target

Implementing Closures

- Closure creation (`MkClosure`)
  - `invokedynamic` call to Java's `LambdaMetafactory`
  - Static arguments: functional interface, method handle
  - Dynamic arguments: captured values

- Closure call (`ApplyClosure`)
  - Emit an interface call

Generating Functional Interfaces

- A closure object implements a functional interface
  - Interface is provided by the implementation
- Flix generates its own functional interfaces
- Before code generation, traverse AST to collect type signatures of closures
  - Generate the interfaces

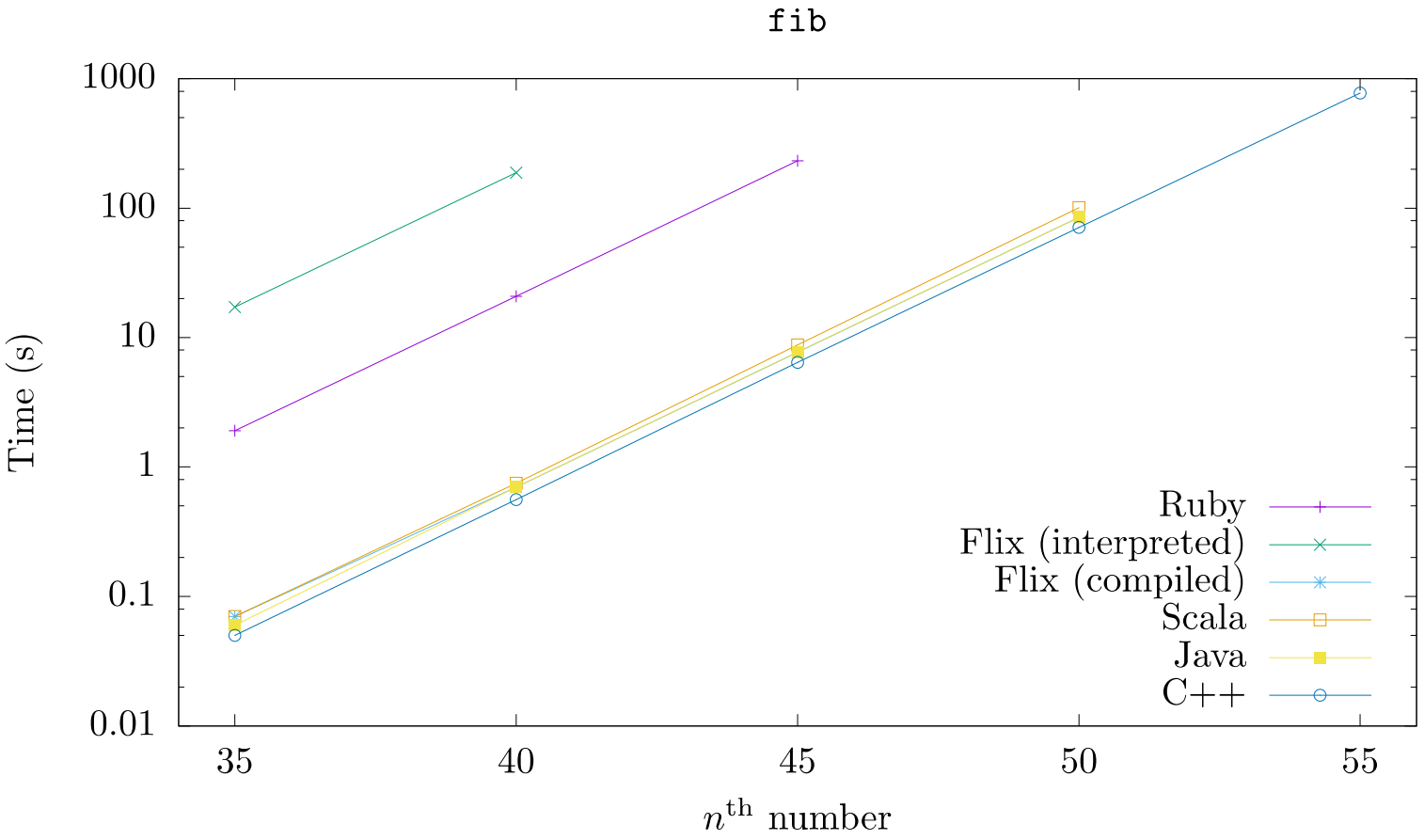
Evaluation – Correctness

- Implemented in ScalaTest
  - Over 500 tests, each a small Flix program
- Strong Update analysis
  - Points-to analysis for C programs
  - Compare Flix versions with pure Datalog version
  - Use SPEC CPU200 and CPU2006 integer benchmarks as analysis inputs

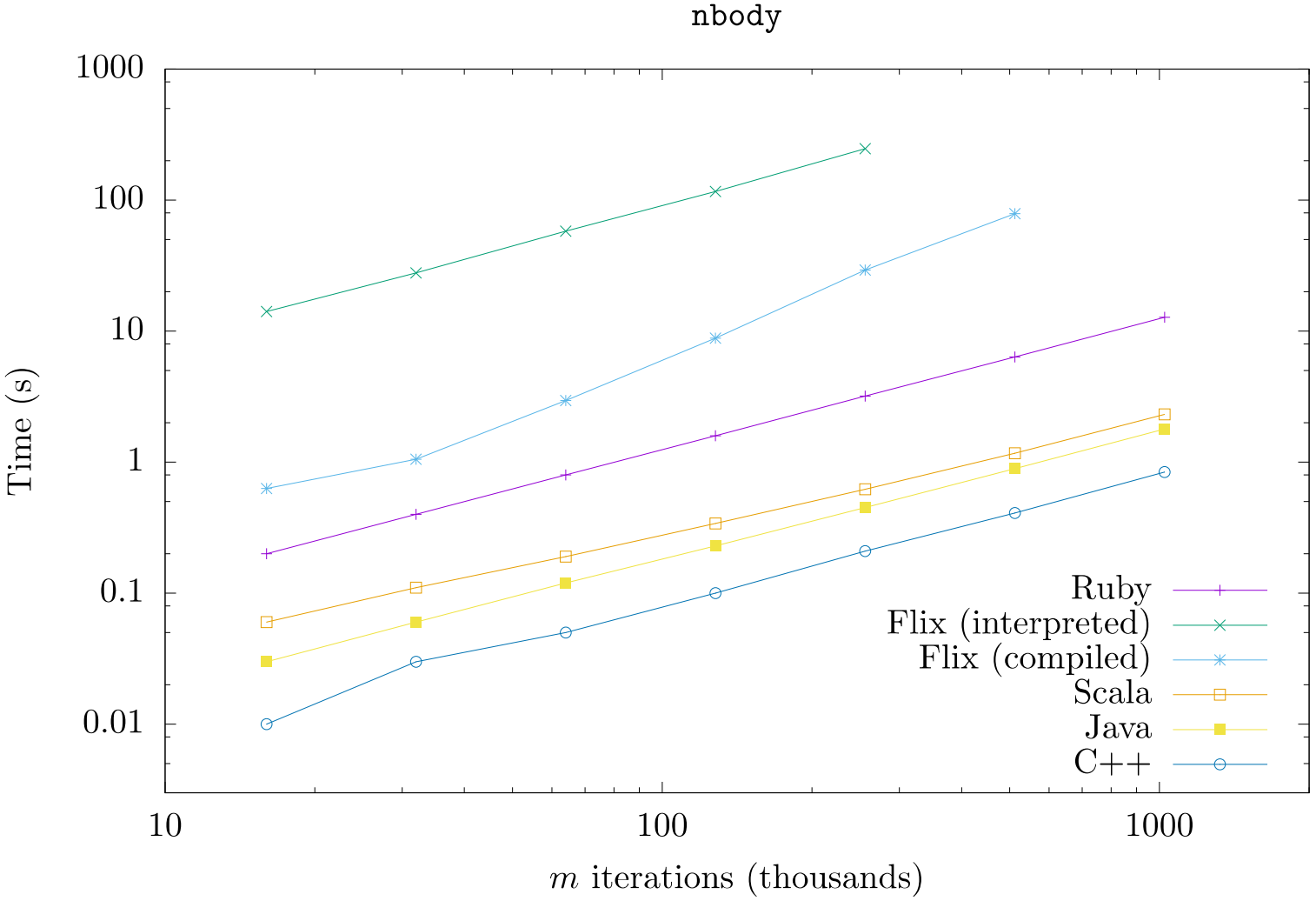
Evaluation - Performance

- Benchmarks:
  - fib
  - nbody
  - pidigits
  - matrixmult
  - shortestpaths
  - strongupdate
- Languages:
  - Flix
  - Ruby
  - Scala
  - Java
  - C++

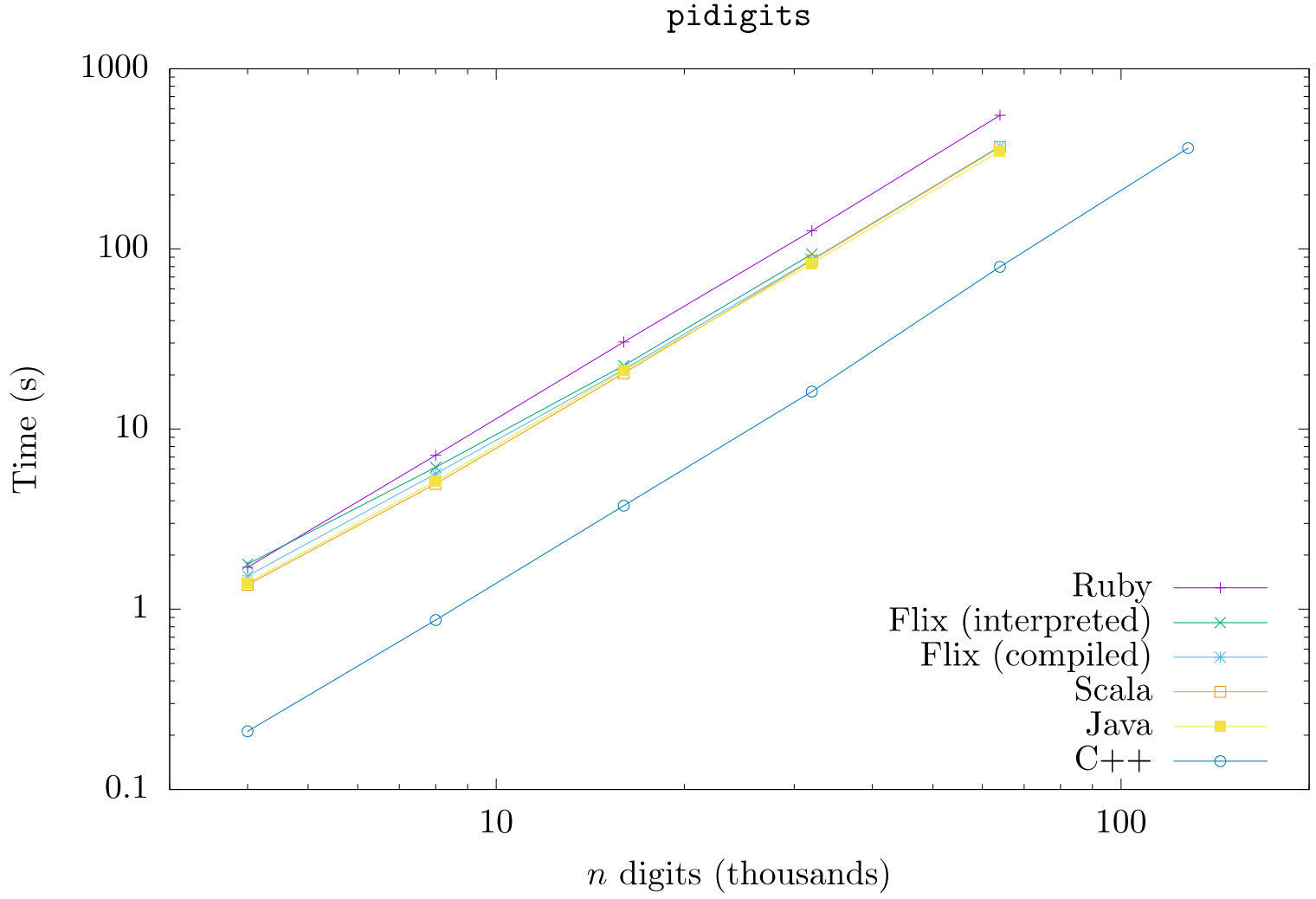
Evaluation – fib



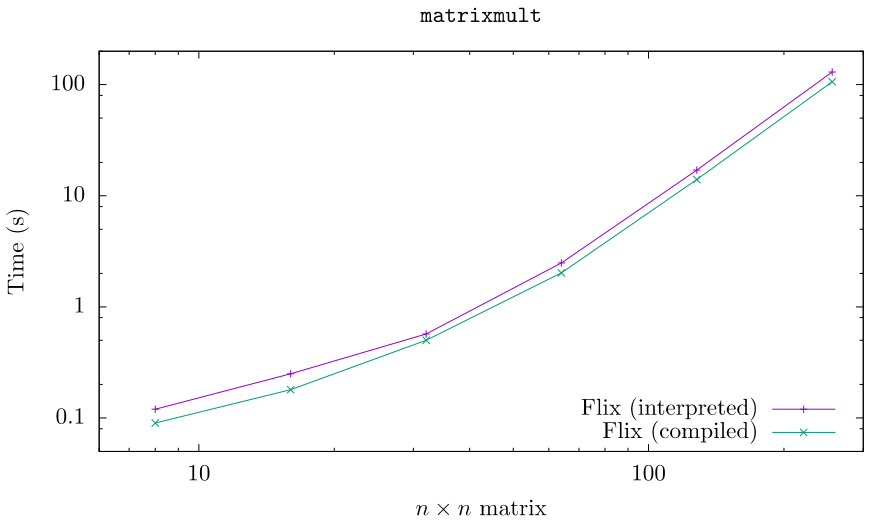
Evaluation – nbody

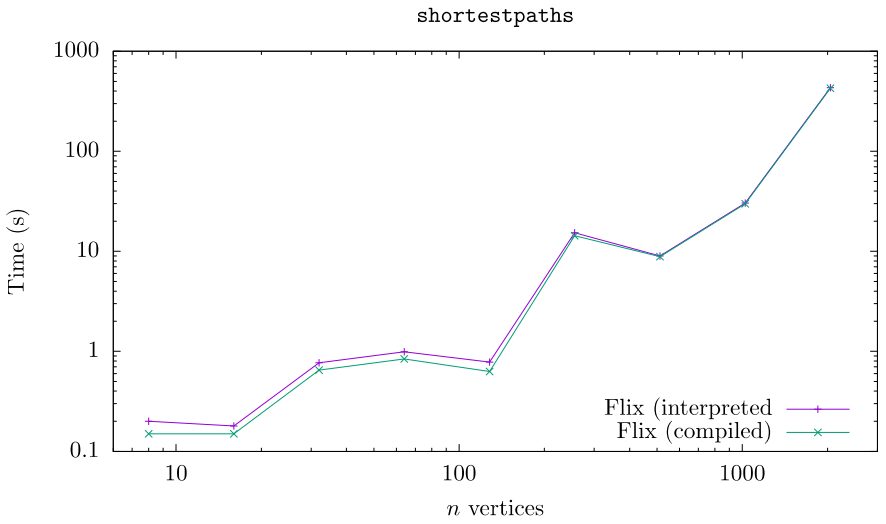


Evaluation – pidigits

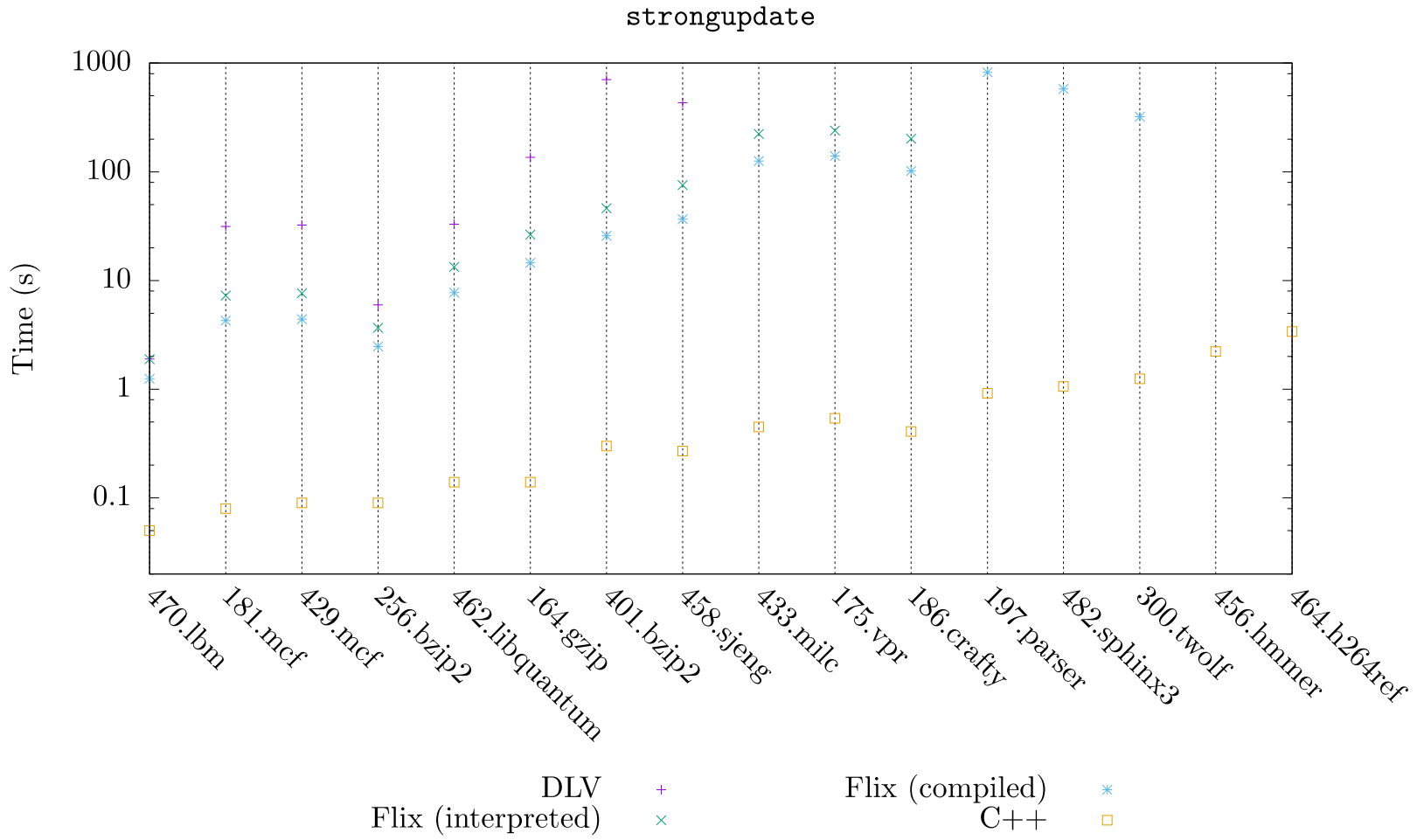


matrixmult and shortestpaths





Evaluation – strongupdate



Future Work

- Language is still evolving
  - New features to implement
- Performance
  - Improve pattern matching
  - AST optimizations
  - Peephole optimizations
  - Tail call optimization

Conclusions

- Implementing the functional language of Flix
  - AST transformations, interpreter, code generator
- Evaluation
  - Compiled Flix is faster than interpreted Flix
  - Sometimes comparable to Java and Scala
- Bytecode generator is first step for performance
  - Much work remains to be done