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

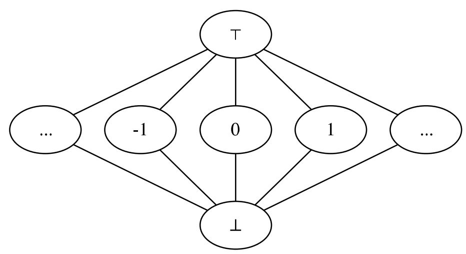
Supervisor: Ondřej Lhoták

September 1, 2016

- Today I'm going to be talking about what I worked on for my thesis.
- Flix is a project I've been working on with Ondřej and Magnus.
    - We've also had two undergraduates who have worked on Flix (Billy and Luqman).
    - My focus has been the functional language back-end.

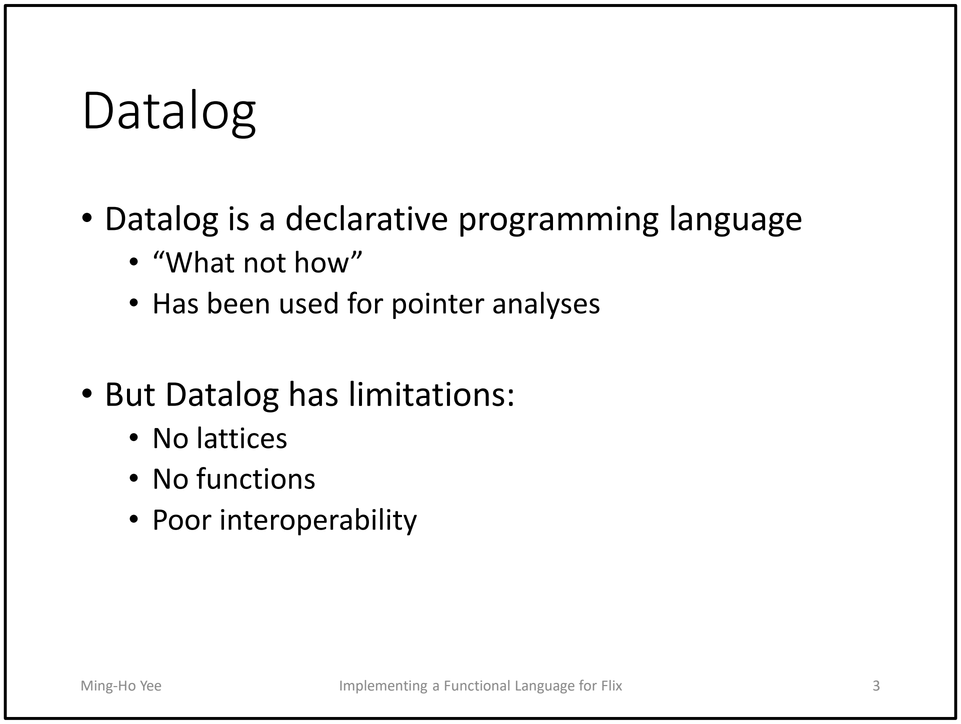
Static Analysis

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



Constant propagation lattice

- Static analysis: technique for analyzing software without executing it.
  - Applications: compiler optimizations, code refactoring, bug finding.
- A static analysis typically models abstract program state as elements of a lattice.
  - Ordering represents precision, lower = more precise.
  - Example: constant propagation lattice.



- One approach to implementing static analyses is to use Datalog.
    - Datalog is a declarative language: what not how.
    - Specify the constraints of the analysis, and a Datalog solver finds the solution.
    - Much easier to understand and maintain than using Java or C++
    - Many researchers have used Datalog to implement pointer analyses
- But Datalog has some limitations:
    - No user-defined lattices
    - No functions
    - Poor interoperability
- Some analyses cannot be expressed in Datalog.
- Using Datalog with existing tools and front-ends is difficult.

A Language for Static Analysis

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

- Flix extends Datalog with user-defined lattices and functions.
  - Specify analysis constraints in the logic language.
    - Based on Datalog and supports user-defined lattices.
  - Express user-defined functions in the functional language.
    - Pure and strict, supports let-bindings, first-class functions, pattern matching.
    - Supports the Java integer types, including BigInteger. Also supports tags and tuples.
- Flix is implemented on the JVM.
  - Interoperability with JVM languages.
  - Call Flix from a JVM language, call JVM code from Flix.

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 = …
```


- Here is what constant propagation looks like in Flix.
    - Some details are omitted for brevity.
- First, look at the functional code.
- We define a tagged union, Constant.
    - Represents elements of the constant propagation lattice.
- We define the three lattice operations:
    - leq, lub, glb
    - leq is an example of pattern matching.
- sum is a separate function, more on that later.

Constant Propagation in Flix (2/2)

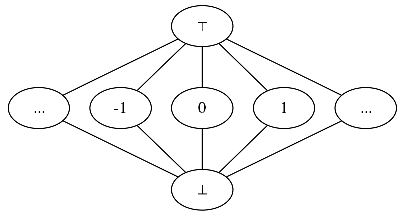
```
// analysis inputs
rel AsnStm(r: Str, c: Int)
rel AddStm(r: Str, x: Str, y: Str)

// 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).
```


- Now for the logic code.
- We define two relations, AsnStm and AddStm, as inputs.
    - Variable r is assigned the integer c
    - Variable r is the result of x + y
- We define the LocalVar lattice, which is the output the analysis computes.
    - Variable k has value v.
    - LocalVar is a map lattice, where k is the key and v is the value.
- First rule: if we assign c to r, then we know the variable r has value c.
- Second rule: if we're adding two variables and know their values, we can compute the value of the result, using the sum function.

Constant Propagation Example

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

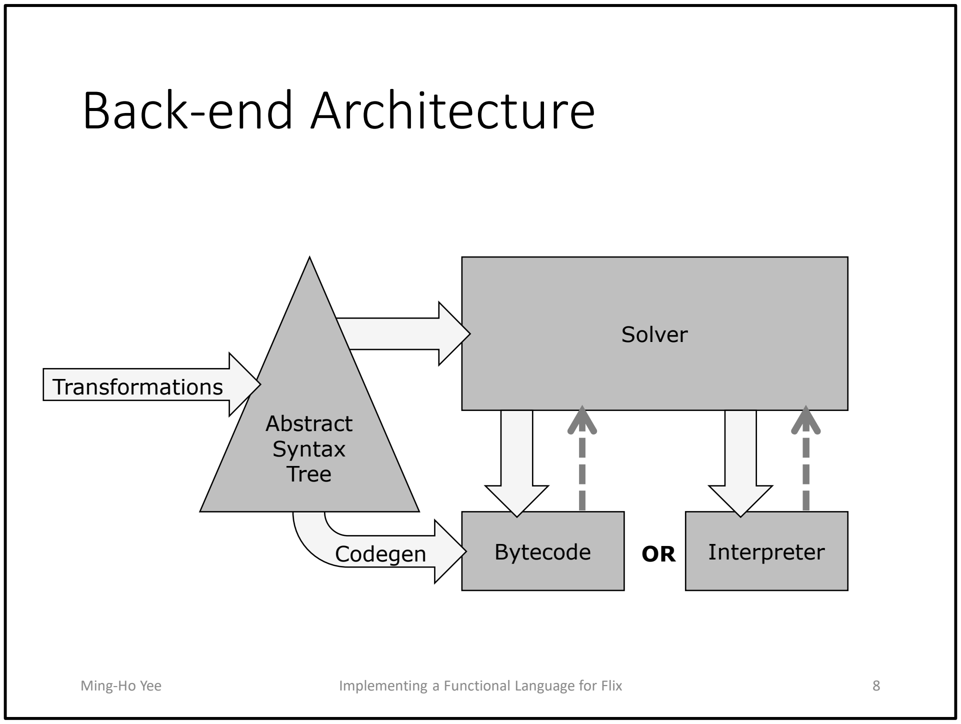
```
// input facts
AsnStm("x", 0).
AsnStm("x", 1).
```



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

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


- Here’s a small example of how Flix handles lattices.
  - We’ll look at the first rule, and two input facts.
- Evaluating the rule, we infer that the local variable ”x” has value 0 *and* 1.
  - But LocalVar is a lattice. We have two values for the same key.
  - We have to compress the values, using the lub operation.
  - This gives us Top.
- In the static analysis, we don’t know the exact value for “x”.
  - So we approximate by saying the value is Top.

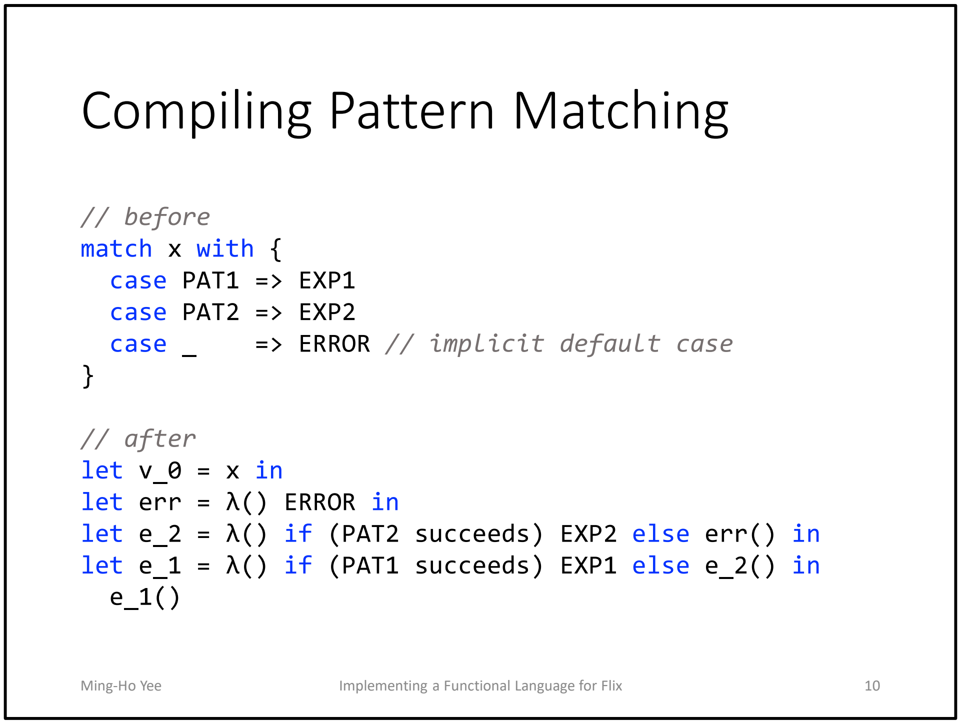


- After several phases, the front-end produces a TypedAst.
- The TypedAst goes through several transformations, becoming a SimplifiedAst and then an ExecutableAst.
    - Compiles higher-level constructs like pattern matching into lower-level primitives.
        - We'll discuss pattern matching and lambda functions.
- Execution starts in the solver, which evaluates rules of the logic language.
    - During this process, the solver may need to evaluate functional code.
        - i.e. lattice operation (lub), or an explicit function call (sum)
    - After evaluating the function, the result is returned to the solver.
- Two implementations of the functional language:
    - Interpreter was original, and is for debugging and prototyping.
    - JVM bytecode generator is newer, and for performance.
- This presentation will cove the code generator.

AST Transformations

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

- **10:00 to get here.**
- Before code generation, a number of AST transformations are required.
  - The interpreter doesn't really need any of this.
  - But to keep things consistent, the interpreter and code generator consume the same AST
- First is to compile pattern matching from a high-level representation into lower-level primitives.
- Then we need to do a few transformations to implement lambda functions.
- Variable numbering is self-explanatory.
  - Numbers are needed so the code generator can emit load/store instructions.
- Finally, AST transformations go here.
  - Currently none, but they could be constant propagation, copy propagation, dead code elimination.
- Today I'll discuss pattern-matching compilation and closure conversion.



- First transformation to discuss is pattern-matching compilation.
- This is all pseudocode.
- In the pattern match, the expression x is compared against the patterns.
    - If it matches, then the corresponding expression is evaluated.
    - If nothing matches, the expression throws an error.
- It’s straightforward to implement this in an interpreter, but not so for a code generator.
    - We want to transform to something simple. Let-expressions, if-expressions, functions, and other primitives.
- Two main steps: create the hierarchy of let-expressions, then transform the patterns.
    - Need to generate fresh names for the variable being matched (so we don’t evaluate it multiple times), each case, and the error case.
    - Construct the let-expressions inside-out, so a case can refer to the next one.
    - Finally, construct a call to the function representing the first case.

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()
```


- There are five types of patterns, and each needs to be handled specifically.
- A wildcard pattern matches everything and also succeeds.
- So the transformation is simply the body of the case.

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()
```


- A variable pattern always succeeds, but it binds the matched value to a name.
  - This is typically used in subpatterns to extract value from tags and tuples.
- The transformation is a let-expression in the body of the case.

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()
```


- A literal pattern succeeds if the value equals the literal in the pattern.
    - This pattern checks if x == 42.
- This translates to an if-expression and an equality check.
    - The true branch is the body of the case.
    - The false branch is a call to the next case.

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()
```

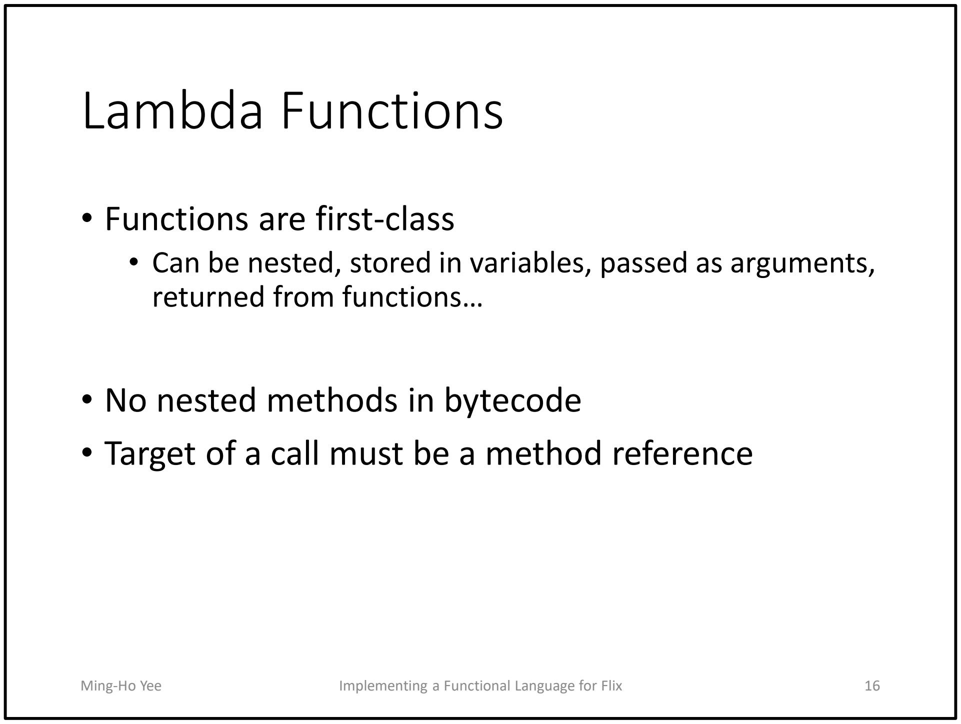

- A tag pattern succeeds if the enum and tag names match, and the subpattern also matches.
  - This pattern checks that x is the tag E.A, and its inner value == 42.
- In this example, E is a tagged union with tags A and B.
  - E is the enum name, A and B are the tag names.
- The pattern match uses the CheckTag primitive.
  - Note that the type checker guarantees that v_0 is a member of the E tagged union.
  - If the check succeeds, then we bind the inner tag value to a fresh name, using a let-expression and the primitive GetTagValue.
    - Then we have the transformed subpattern.
  - If the subpattern or the CheckTag fail, then we evaluate the error case.

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()
```

Ming-Ho Yee   Implementing a Functional Language for Flix   15

- A tuple pattern contains multiple subpatterns, each corresponding to a tuple element.
    - The type checker guarantees the type and arity of x matches the patterns.
- The transformed pattern extracts the tuple elements, using GetTupleIndex, and binds them to fresh names.
    - Then each subpattern is translated.
    - If everything succeeds, the body of the case is evaluated.
    - If anything fails, the next case is evaluated.



- In Flix, functions are first-class.
  - You can nest function definitions, store a function in a variable, pass it as an argument, and return from a function.
- This does not hold for bytecode.
  - All methods must be defined at the top-level. No nesting.
  - The target of a method call must be a method reference.
    - Cannot be an arbitrary expression that evaluates to a function.

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
```


- We solve the first problem (nested functions) with lambda lifting.
  - Self-explanatory name: we lift a nested lambda definition to the top level.
- In this example, within the definition of f, we bind a function to g, and then call g.
- The transformation simply lifts the inner definition.
  - g now refers to a function and not a local variable.

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?
```


- But what about free variables?
- If we naïvely lift, then we have a definition with an unbound variable.
- So lambda lifting must account for free variables.

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
```


- We rewrite the function to pass the free variable as an extra parameter.
    - Our convention is to prepend the free variables to the parameter list.
    - Note that the call site must also be rewritten.
- Now we can safely lift the lambda.

Lambda Lifting…?

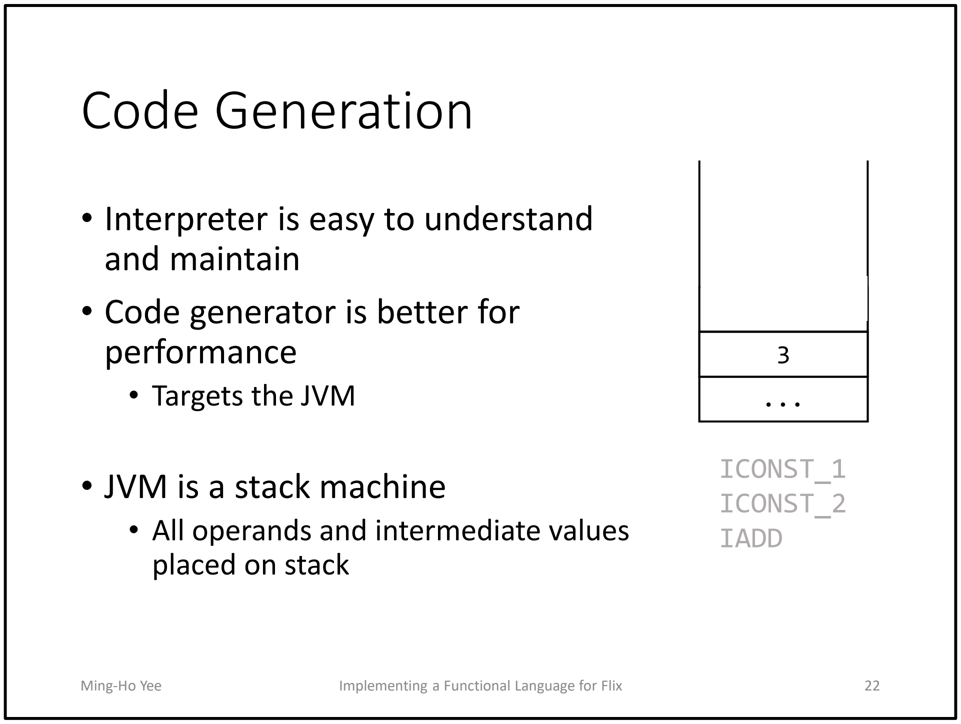
```
def f(a) = let g = λ(x, y) a+x+y in
             h(g, 1, 2)
def h(g’, x, y) = g’(x, y) // how to rewrite g’?
```


- Rewriting all the call sites is pretty annoying.
- But it gets worse: what if you can’t rewrite the call site?
  - What if we pass a function as an argument?
- In this example, g needs to be rewritten to take an extra parameter.
  - But how do we know that we need to rewrite g’?
  - What if some other function passes in a different g that doesn’t need to be rewritten?
- Real problem: variables are bound at different times.
  - a is bound when the lambda is created.
  - x and y are bound when the lambda is called.
- Solution: use closures.
  - Store the data (variables that are bound when closure is created)
  - Store the code (reference to function)

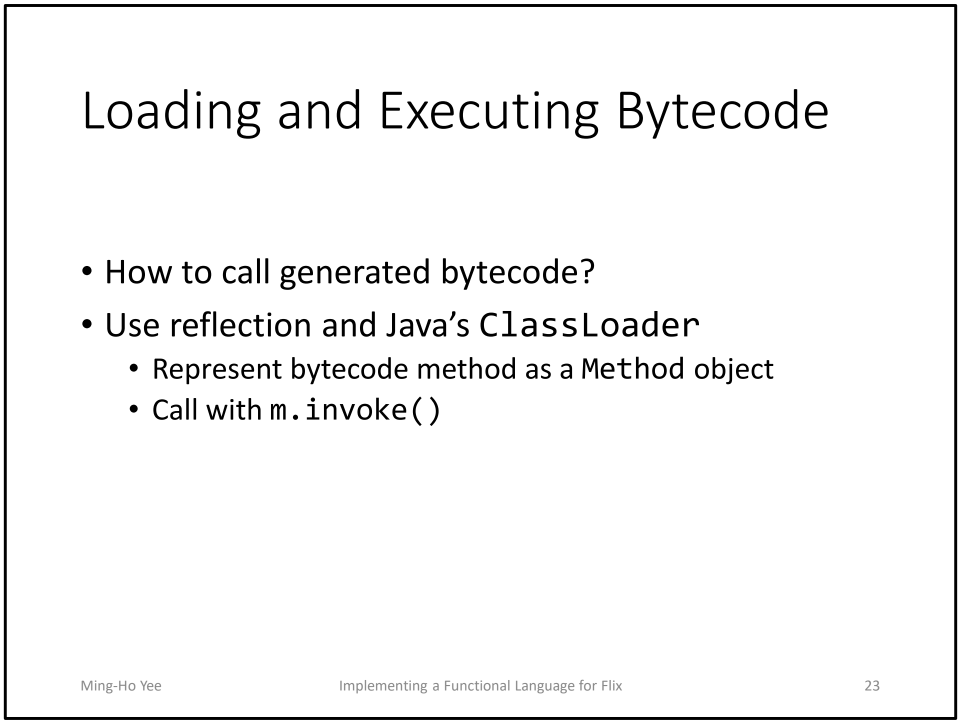
Closure Conversion

```
// 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)
```


- We convert all lambda functions into closures.
    - The closure contains an inner lambda function.
        - The original function is rewritten to pass the captured variables.
    - The closure contains the values that are captured.
- At closure creation, we save the values that need to be passed into the function.
- At closure call (ApplyClosure), the saved values ae combined with the closure arguments and passed to the implementing function.
    - Now we can safely lift the inner lambda, giving it a name.
- We’ll get back to MkClosure and ApplyClosure later.



- **13:00 (23:00 total) to get here**
- Two back-ends for the functional language:
    - Interpreter – original one, easy to understand and maintain, good for debugging and prototyping features.
    - Code generator – now the default back-end, more complicated, but better for performance.
- Since Flix is implemented on the JVM, the code generator targets the JVM.
    - Also good for portability.
- JVM is a stack machine, unlike x86, ARM, MIPS
    - Makes some things nicer: no register allocation since all operands and intermediate values go onto the stack.
    - Makes some things harder: need to compute maximum stack height, manage constant pool, compute metadata for bytecode verifier.
        - Use the ASM library to handle these tasks.
        - Scala 2.12 uses ASM.
- There's a few interesting codegen problems to discuss today.

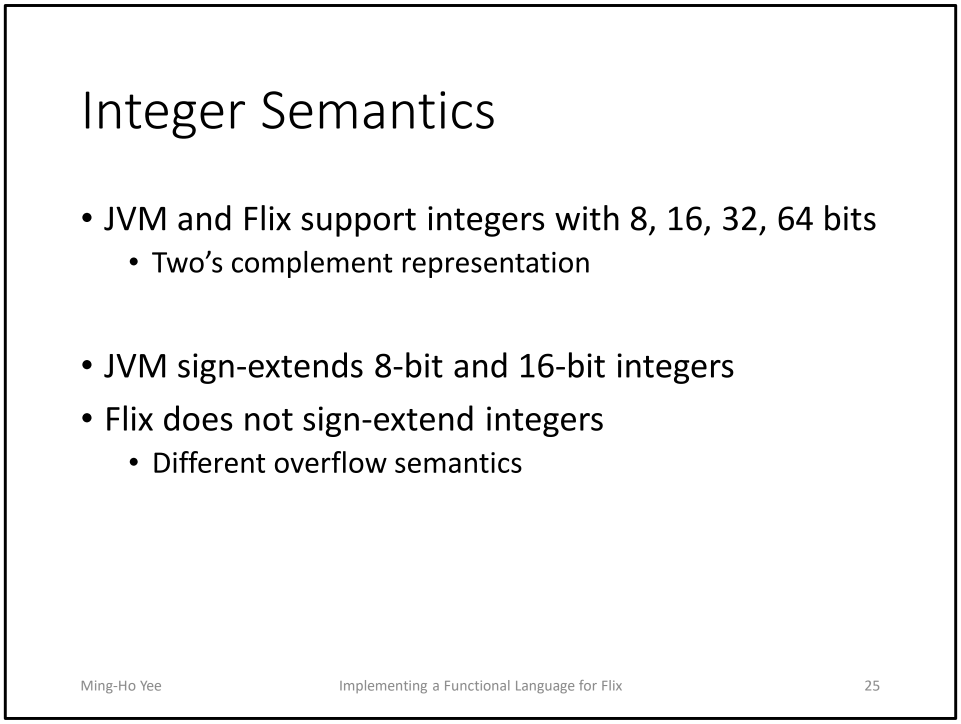


- ASM library produces bytecode as an array of bytes.
  - For convenience, we want to load it immediately, instead of writing to disk.
  - Then we need to be able to execute it.
- We use reflection and Java's ClassLoader.
  - Each bytecode method is represented by a reflection Method object.
  - Call a method with m.invoke().
- We store Method objects on the AST.
  - Each function has an expression AST and a Method object.

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`

- Flix values are represented as primitives whenever possible.
    - E.g. Int32 -> int
    - Sometimes values need to be boxed, e.g. java.lang.Integer.
- Some Flix values are represented as Java reference types.
    - E.g. Str -> java.lang.String
- Other Flix values have no corresponding class in the Java standard library, so Flix defines them.
    - Tag -> Value.Tag
    - Tuple -> Value.Tuple
    - Tags and tuples are generic in the values they contain, so Flix has to use reference types and type erasure.
        - Box Int32 as a java.lang.Integer and then store as java.lang.Object.
        - When extracting the value, need to cast or unbox.
        - Code generator emits the code to do this automatically.



- Both the JVM and Flix support signed integers, with 8, 16, 32, and 64 bits.
    - Use two’s complement representation.
- Implementation detail of the JVM: sign-extend 8-bit and 16-bit integers to 32 bits.
    - The JVM is designed for 32-bit integers and operations.
    - A “surprise” from compilers class: adding two 8-bit integers returns a 32-bit integer.
- Flix does not sign-extend integers.
    - This may be harder to implement, but conceptually, it’s easier to understand.
    - Leads to some differences, specifically with overflow.
- The problem is to implement Flix semantics on the JVM, which has different semantics.

Integer Overflow (1/2)

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

- Consider adding two 8-bit integers with value 64.
    - Mathematically, the result is 128.
- But this cannot be represented in two's complement with 8-bits.
    - The result we get is the 8-bit representation for -128.

Integer Overflow (2/2)

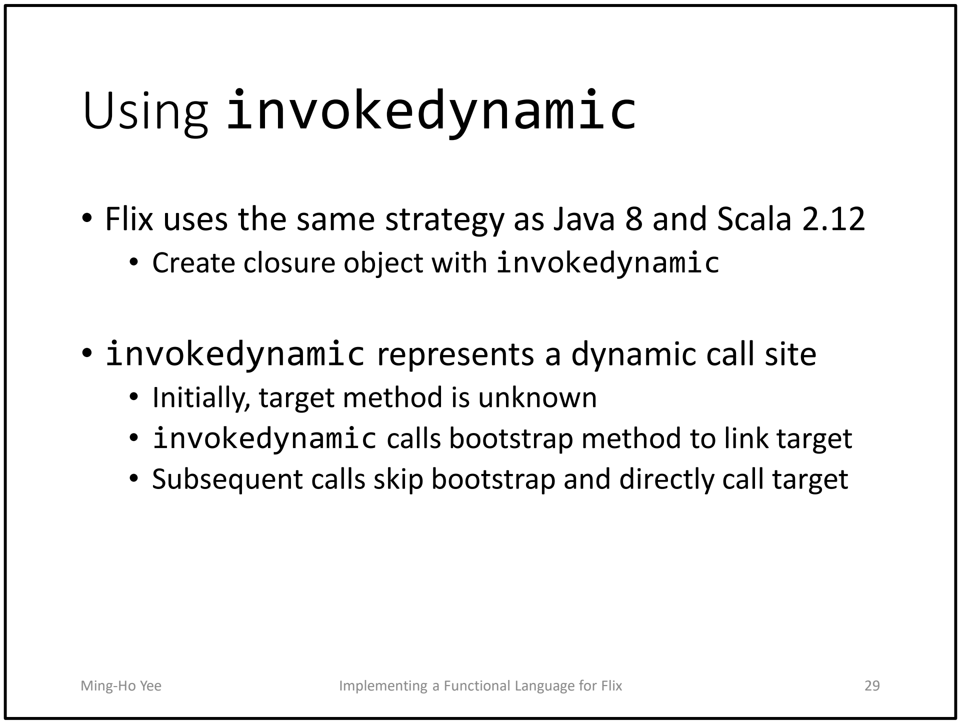
$$
\begin{array}{rl}
 & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
+ & 00000000\ 00000000\ 00000000\ 01000000_2 = 64_{10} \\
\hline
 & 00000000\ 00000000\ 00000000\ 10000000_2 = 128_{10} \\
 & \\
 & 11111111\ 11111111\ 11111111\ 10000000_2 = -128_{10}
\end{array}
$$

- On the JVM, the operands are sign-extended.
    - We get the 32-bit representation for the same value, 64.
- But the binary result now represents a different number, 128.
    - This is the correct mathematical result, but not the result according to Flix semantics.
- Fortunately, the JVM has a “truncate to 8 bits and sign-extend” instruction, I2B.
    - This gives us the 32-bit representation of -128, the desired result.
- This is something we have to do for integer expressions, including all arithmetic and some bitwise expressions.

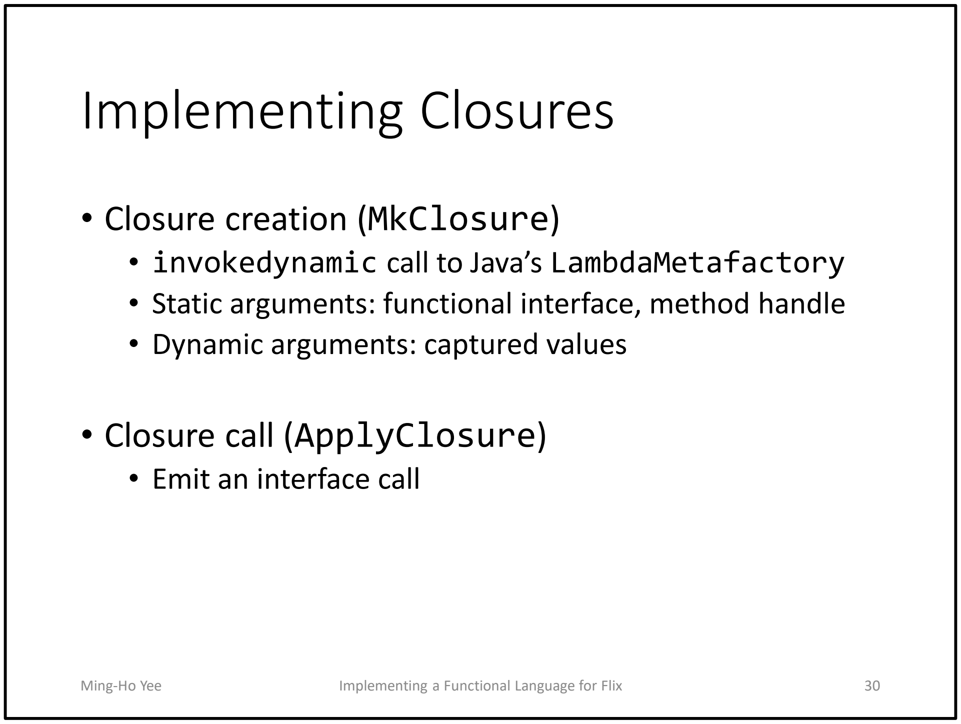
Implementing Closures…?

```
// 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
```


- In object-oriented languages, one way to implement closures is to use function objects.
    - C++, C#, and Scala 2.11 use this method.
- Every lambda function has an associated anonymous class.
    - The class stores captured variables, and defines a method that implements the lambda function.
- Creating a closure instantiates that class, with values of captured variables.
    - Here, a is passed to the constructor.
- Calling a closure is an interface call on the method.
- Problem with this approach: must generate an anonymous class for each lambda function. Increases code size.



- An alternate approach, used by Java 8 and Scala 2.12, is invokedynamic.
    - Instead of the code generator statically creating the classes, invokedynamic will dynamically create the classes.
- Initially, the invokedynamic instruction is a dynamic call site, and the target of the call is unknown
    - To determine the target, invokedynamic calls a bootstrap method, and then links it
    - Subsequent calls bypass the bootstrap and directly call the target
    - In other words, let the run time determine which method is called, but then permanently link it so future calls are “static”



- To create a closure, code generator emits an invokedynamic call to LambdaMetafactory, which is defined in the Java standard library.
    - Static arguments represent the functional interface implemented by the closure, and a handle to the method implementing the function.
    - Dynamic arguments represent the captured values.
- When a closure is created for the first time, invokedynamic calls the metafactory, which generates an anonymous class.
    - The class is instantiated with the captured values.
- Subsequent calls bypass the metafactory and directly instantiate the class.
- Closure call
    - Emit an interface call.
    - The closure will automatically supply the captured values to the implementing function.

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

- Each closure object needs to implement a functional interface.
  - Functional interface: interface with a single abstract method.
  - The interfaces must be provided by the implementation.
  - Java provides a very small selection.
    - If you're writing lambdas in Java and can't find the interface you need, you have to define your own.
  - Scala is the opposite extreme.
    - Very general interfaces, all generic
- Flix generates its own functional interfaces.
  - Traverse the AST, find every lambda function, and generate an interface for each unique type.
  - Generates only the interfaces that are needed.
  - Interfaces are specialized, so no generics and no boxing/unboxing.

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

- **15:00 (38:00 total) to get here**
- It's important to ensure the code generator produces the right code.
- Almost all of the tests are written in the ScalaTest framework.
  - Over 500 tests, each a small and complete Flix program.
- For a larger test, we use the Strong Update analysis, which is a real-world static analysis.
  - Exercises logic code as well as functional code.
  - A points-to analysis for C programs.
    - Achieves better precision by propagating singleton sets flow-sensitively.
    - Does not sacrifice performance by propagating non-singleton sets flow-insensitively.
  - Compare the two Flix implementations to a Datalog reference implementation.
    - Datalog implementation must simulate lattices.
    - Run on the DLV solver.
  - Use SPEC integer benchmarks as analysis inputs.

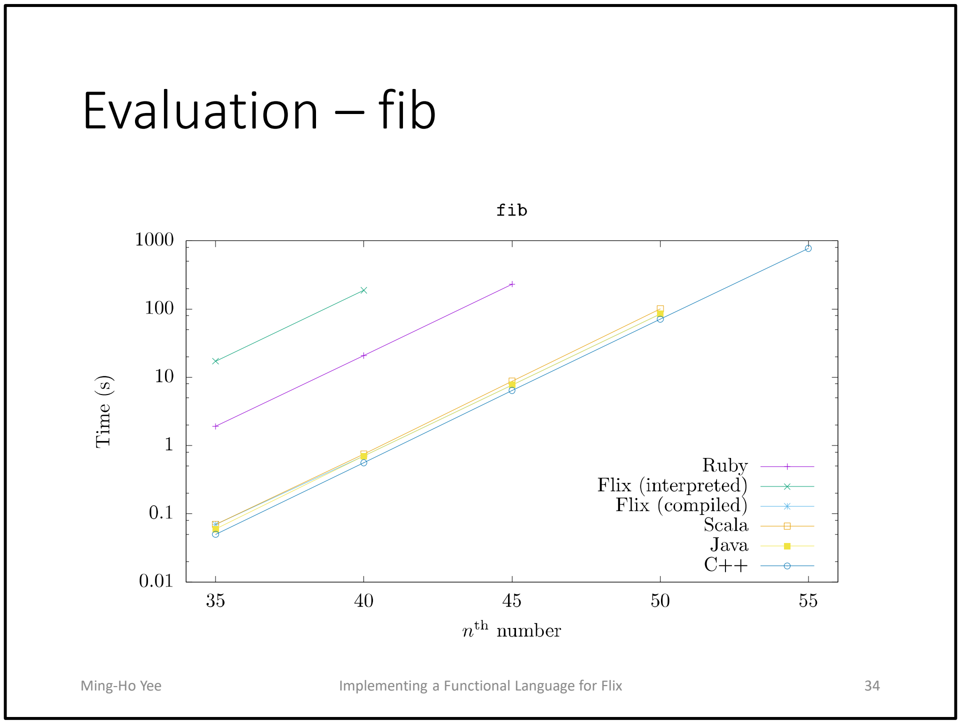
Evaluation - Performance

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

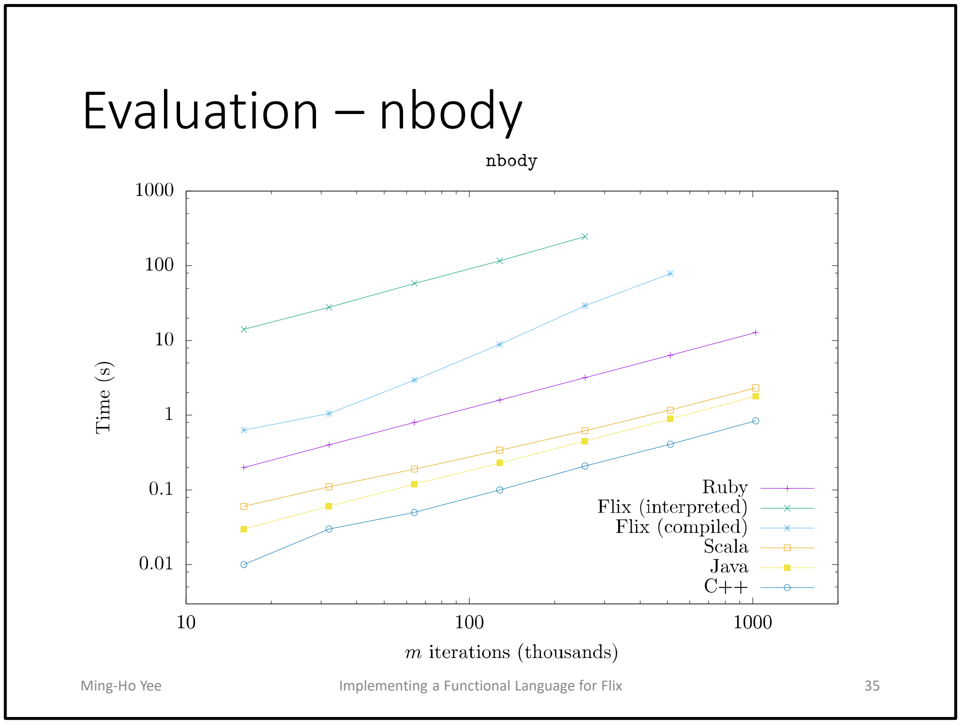
- Benchmarks:
  - Purely functional, so compared against implementations in other languages.
    - fib – naïve implementation of Fibonacci, taking exponential time.
    - nbody – Computer Languages Benchmarks Game; N-body simulation that models Sun and gas giants
    - pidigits – CLBG; computes the digits of pi one at a time; requires arbitrary-precision arithmetic
  - Require the solver, so only compiler vs interpreter
    - matrixmult – multiply two random matrices together, using cubic-time algorithm
    - shortestpaths – based on the Floyd-Warshall algorithm for all-pairs shortest paths
  - Real static analysis
    - strongupdate – the Strong Update analysis; also compared with Datalog and handwritten C++ analyzer
- Languages
  - Flix – both compiled and interpreted. Purely functional language, so data structures need to be copied and not mutated in place. No tail call

optimization.

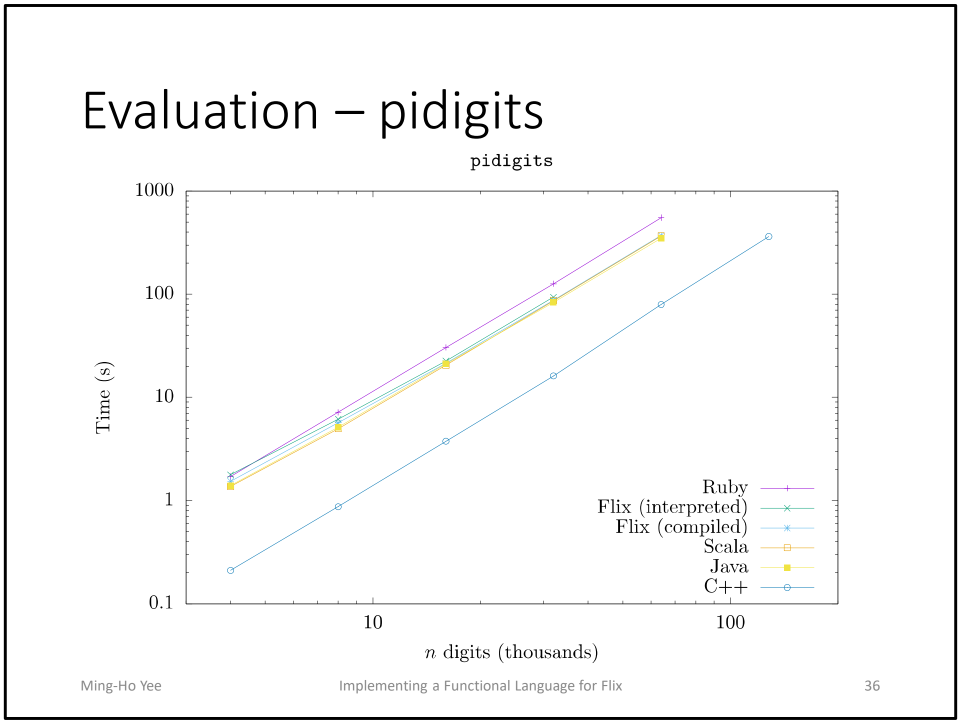
- Ruby – dynamic language with a bytecode interpreter.
- Scala – JVM language; benchmarks written in functional style.
- Java – benchmarks written on OO-style.
- C++ – benchmarks written in OO-stye.



- Compiled Flix is 250x faster than interpreted Flix
- Fibonacci is a simple function, so the compiled bytecode for Flix, Scala, and Java is very similar.
- Scala is slower than Flix
    - Scala convention put the methods in a singleton object, which get compiled to instance methods.
- Ruby is faster than Flix interpreter
    - Bytecode interpreter vs AST interpreter

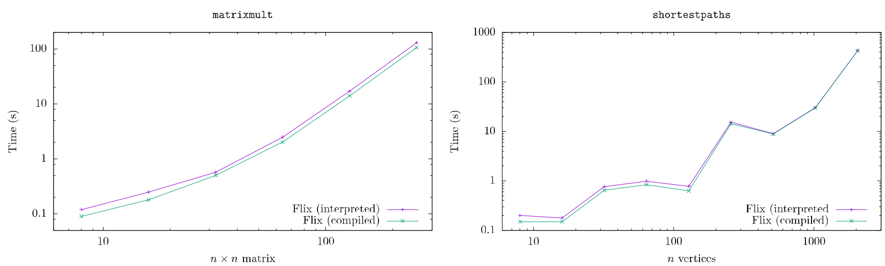


- Both Flix implementations are the slowest.
  - But compiled Flix is 17x faster than interpreted Flix.
- nbody is the most complicated functional program implemented in Flix, and highlights many inefficiencies.
  - No tail call optimization, so the stack memory usage increases until the stack overflows.
  - Interpreter needs to copy the environment for each call, which becomes expensive.
- C++ is the fastest
  - Compiler can emit vector instructions.

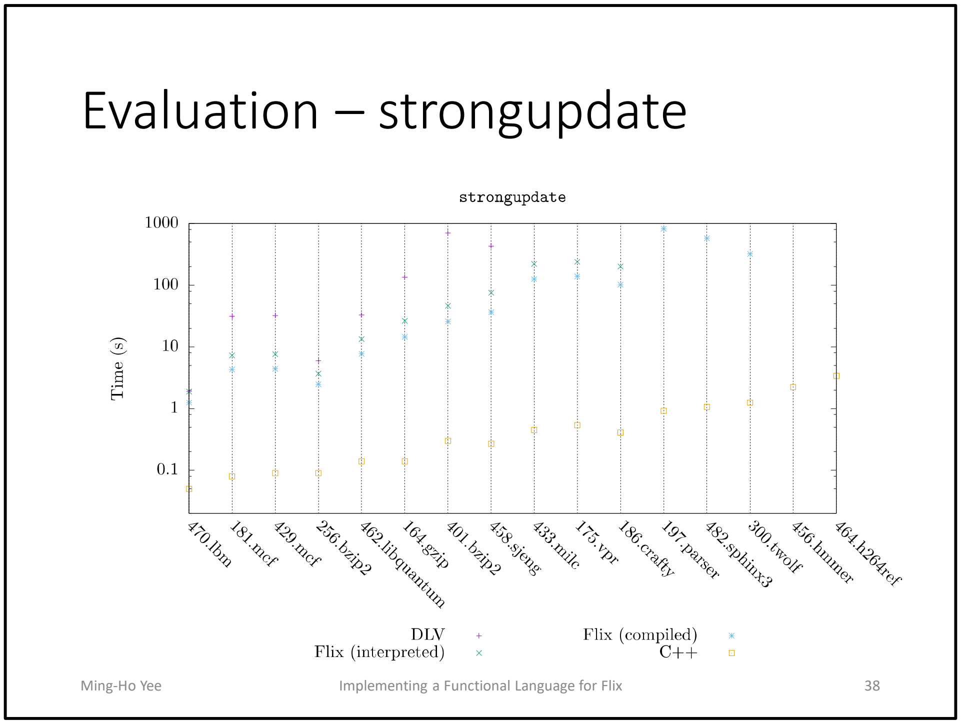


- Interestingly, Ruby, Flix, Scala, and Java are all very similar.
    - Though Ruby is a bit slower, and C++ is the fastest.
- The bottleneck is in the arbitrary-precision arithmetic.
    - Java, Scala, and Flix all use the same library (java.math.BigInteger).
    - Doesn't matter if other parts of the program are slower.
- Ruby and C++ use the same C library: GNU Multiple Precision Arithmetic Library.
    - But Ruby is much slower, probably because of overhead as a dynamic language.
    - Calling + involves a lot of overhead before the call makes it to the C library.

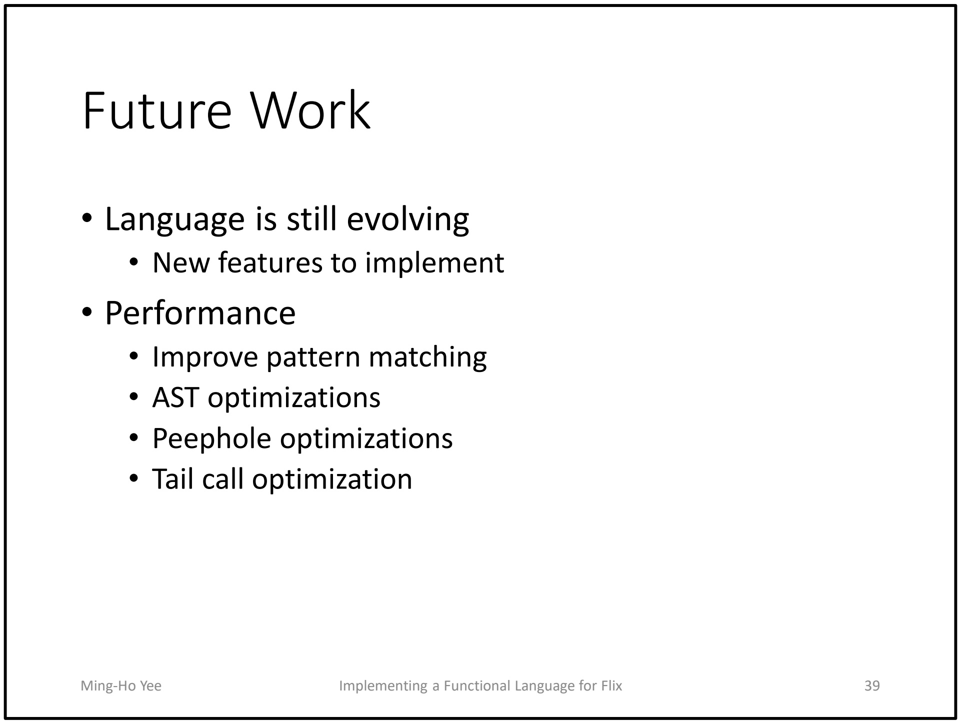
matrixmult and shortestpaths



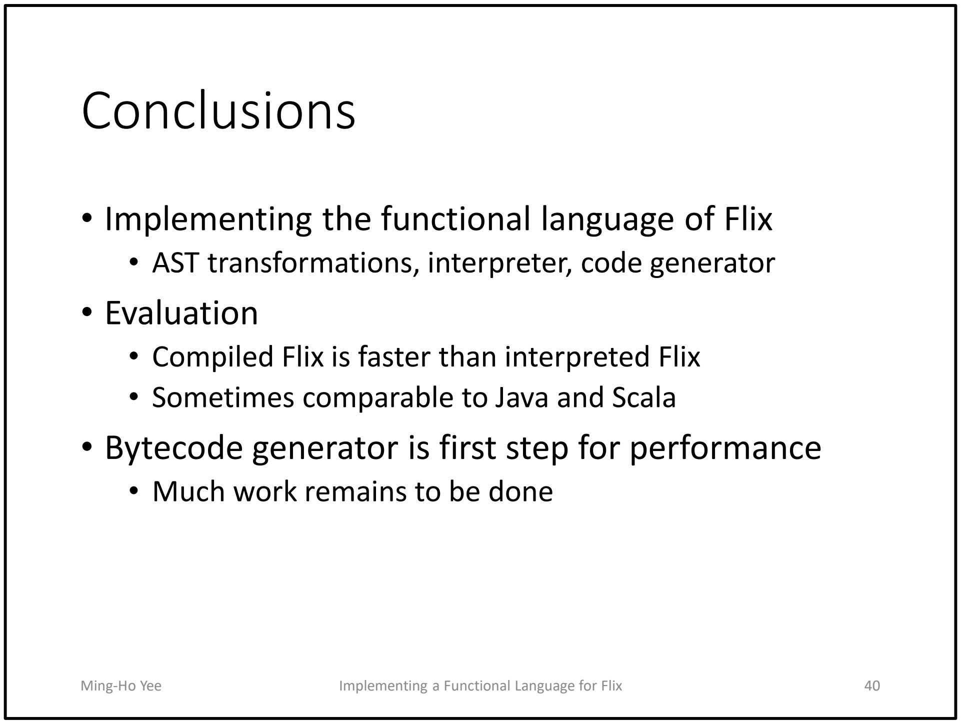
- matrixmult and shortestpaths is very expensive.
- Most of the time is spent in the solver, evaluating the logic code.
- Functional code has very little effect.
  - So very little difference between interpreter and code generator.



- Differences are consistent.
    - Datalog slower than interpreted Flix, slower than compiled Flix, slower than C++.
- The analysis requires a constant propagation lattice.
    - In Datalog, the lattice is simulated as a power set lattice, which is much more expensive.
    - In Flix, the lattice can be expressed directly.
    - So interpreted Flix is 3.7x faster than Datalog.
    - Compiled Flix is 1.7x faster than the interpreter.
- C++ is even faster, at 126x.
    - Flix is a general framework implemented in Scala, so already at a disadvantage compared to C++.
    - The C++ implementation also has a specific optimization to reduce memory usage.
        - Some elements of the lattice occur much more frequently.
        - The C++ analyzer uses a special data structure that can implicitly represent these elements.
        - But Flix must explicitly represent them.



- Two main areas for future work.
- New language features:
  - The language is still evolving.
  - Some features may be isolated to the logic code, or can be implemented in just the front-end.
  - Otherwise, features need to be implemented twice, in the interpreter and the code generator.
  - Interpreter should make it easier to prototype and test.
- Performance is the big category.
  - Hook calls could be more direct, rather than going through the Flix object and invoke method.
  - Pattern matching could be optimized.
  - AST optimizations: constant propagation, copy propagation, dead code elimination.
  - Peephole optimizations on the generated bytecode.
  - Tail call optimization, since we need recursion.
- And slightly unrelated: code generation for logic language.
  - This will probably have the most benefit, since the solver is a major bottleneck for performance.



- **5:00 (43:00 total) to get here.**
- To summarize:
  - This thesis concerned the implementation of the Flix functional language.
  - First the interpreter, then the code generator, and also common AST transformations.
- Evaluation finds that the compiled code is faster than the interpreted code.
  - Especially for benchmarks that spend most of the time in functional code
  - In some cases, Flix is comparable to Java and Scala.
  - However, Flix is still slower than a handwritten C++ static analyzer.
- The bytecode generator is only the first step for performance.
  - There is much work remaining.