Flix: A Language for Static Analysis

Magnus Madsen, **Ming-Ho Yee**, Ondřej Lhoták

October 20, 2016

- Before coming to Northeastern, was a master's student at Waterloo
  - Worked on Flix as part of my master's thesis project
- The Flix project is much larger than that
  - Joint work with Ondřej Lhoták and Magnus Madsen
  - We've also had undergraduates who have worked on Flix
- My focus has been the functional language back-end
  - But today I'll also talk more generally about Flix the language

Datalog

- A declarative programming language
  - Syntactic subset of Prolog, but different semantics
  - Every Datalog program terminates with a unique solution
  - [Ceri, Gottlob, and Tanca, TKDE 1989]
- Datalog has been used for points-to analyses
  - Separates specification from implementation
  - [Bravenboer and Smaragdakis, OOPSLA '09]

2

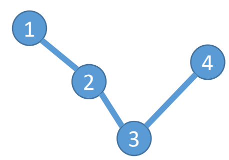
- Static analyses are usually very complicated and difficult to implement
- One approach to implementing static analyses is to use Datalog.
  - Datalog is a declarative language: what not how.
  - Syntactic subset of Prolog, but different semantics (declarative vs operational)
  - Specify the constraints of the analysis, and a Datalog solver finds the solution.
  - Much easier to understand and maintain than using Java or C++
  - Every Datalog program terminates with a unique solution (unlike Prolog)
  - Good intro: "What you always wanted to know about Datalog And Never Dared to Ask"
- Many researchers have used Datalog to implement pointer analyses
  - E.g. Doop framework by Bravenboer and Smaragdakis

Example: Transitive Closure

```
// Rules
Path(x, y) :- Edge(x, y).
Path(x, z) :- Path(x, y), Edge(y, z).
```

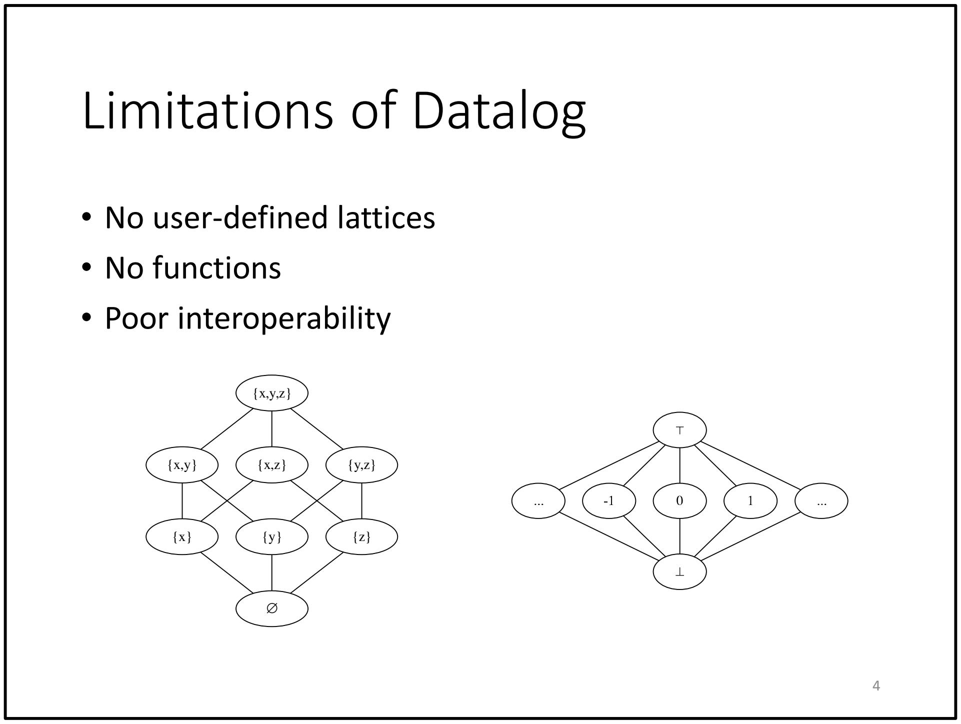
Head — Body

```
// Facts
Edge(1, 2).
Edge(2, 3).
Edge(3, 4).
```

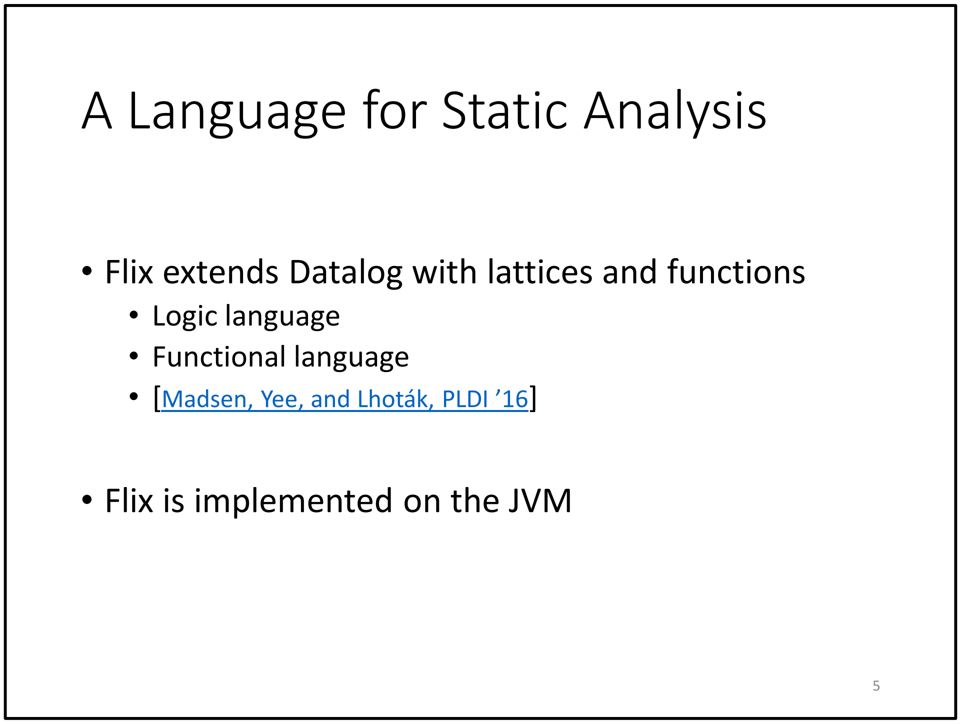


3

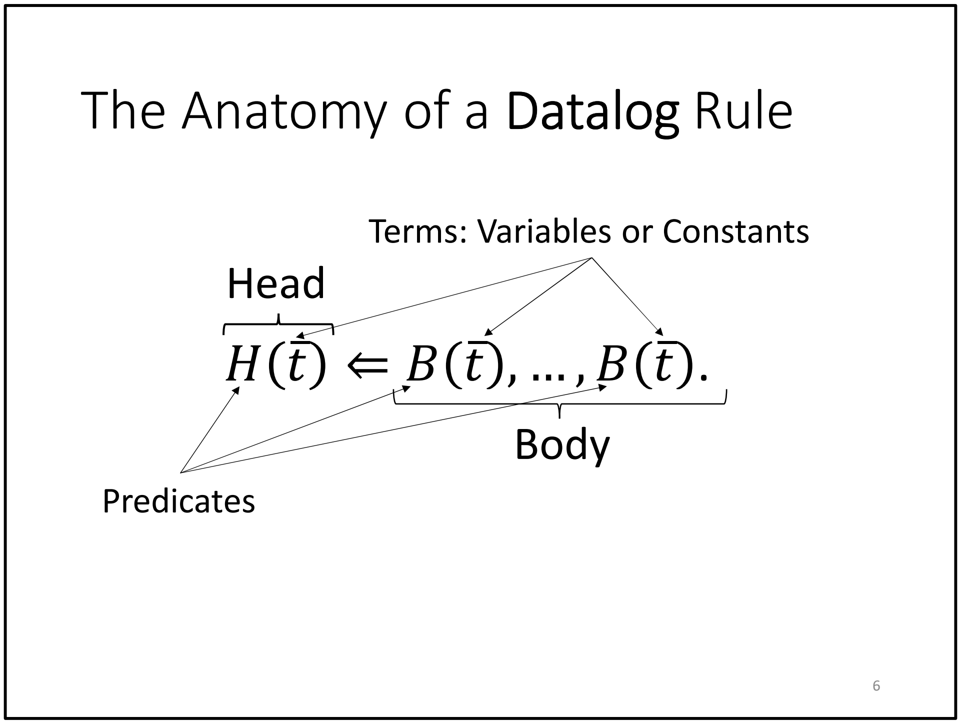
- Common example Datalog program
  - Computes the transitive closure (i.e. reachability) of a graph.
- Path and Edge are relations
  - Edge relation is input, we start with known inputs
  - Path relation is output, we want to compute the paths
- In a Datalog program, we use rules to infer new facts.
- If the body of a rule is true, then the head must also be true.
  - "If *Edge(x, y)* holds, then so must *Path(x, y)*"
  - "If *Path(x, y)* and *Edge(y, z)* hold, then so must *Path(x, z)*"
- This is a very elegant way of expression the problem (and notice the recursion in the second rule)



- But Datalog has some limitations:
  - No user-defined lattices (you have the powerset lattice)
  - No functions
  - Poor interoperability
- Some analyses cannot be expressed in Datalog.
  - It’s possible to work around some of these limitations, but performance suffers
  - And the workarounds fail if the domain is infinite
- Using Datalog with existing tools and front-ends is difficult.
  - Typically extract input facts from program under analysis, and save as text file
  - Datalog communicates with other tools through a textual interface



- Flix extends Datalog with user-defined lattices and monotone 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 (in Scala).
  - Interoperability with JVM languages.
  - Call Flix from a JVM language, call JVM code from Flix.



- Let’s look at how Flix differs from Datalog
- Here’s what a Datalog rule looks like, but with math syntax
    - The right-hand side is the body.
    - If the body is satisfied, then the left-hand side, the head, must also be satisfied.
    - The head and body are composed of atoms.
    - Each atom is a predicate symbol with variable or constant terms.

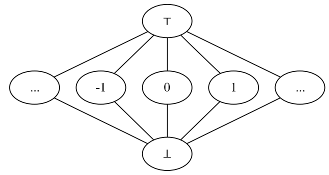
The Anatomy of a **Flix** Rule

$$H_{\ell}\big(\bar{t}, \underbrace{f(\bar{t})}_{\text{Transfer Function}}\big) \Leftarrow \overbrace{\varphi(\bar{t})}^{\text{Filter Function}}, B_{\ell}(\bar{t}), \dots, B_{\ell}(\bar{t}).$$

7

- Flix rules are based on Datalog rules.
    - We still have a head and a body.
    - But each predicate symbol is associated with a lattice.
    - The body may contain a list of filter functions.
        - If the body is satisfied *and* the filter functions evaluate to true, then the head must be satisfied
    - The head atom may contain transfer functions.
        - These functions map lattice elements to lattice elements.
- Note: filter and transfer functions must be monotone and lattices must have finite height to guarantee termination

Constant Propagation

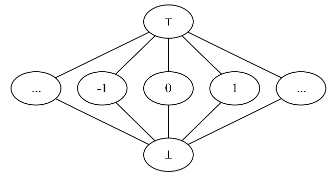



```
enum Constant {
  case Top, case Cst(Int), case Bot
}

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 = …
```

8

- **13:00 to get here.**
- 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.

Constant Propagation

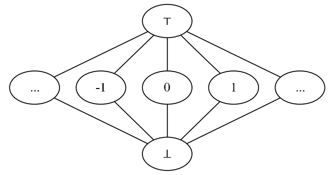



```
def sum(e1: Constant, e2: Constant): Constant =
  match (e1, e2) with {
    case (_, Bot)            => Bot
    case (Bot, _)            => Bot
    case (Cst(n1), Cst(n2)) => Cst(n1 + n2)
    case _                   => Top
  }
```

9

- sum is a monotone transfer function
    - Adding anything to Bot is Bot
    - Adding two constants creates a new constant
    - Everything else is Top

Constant Propagation

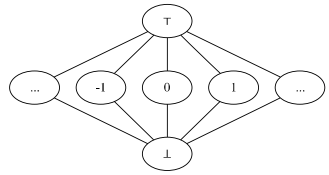



```
// analysis inputs
rel AsnStm(r: Str, c: Int)          // r = c
rel AddStm(r: Str, x: Str, y: Str)  // r = x + y

// 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).
```

10

- 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




```
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).
```

11

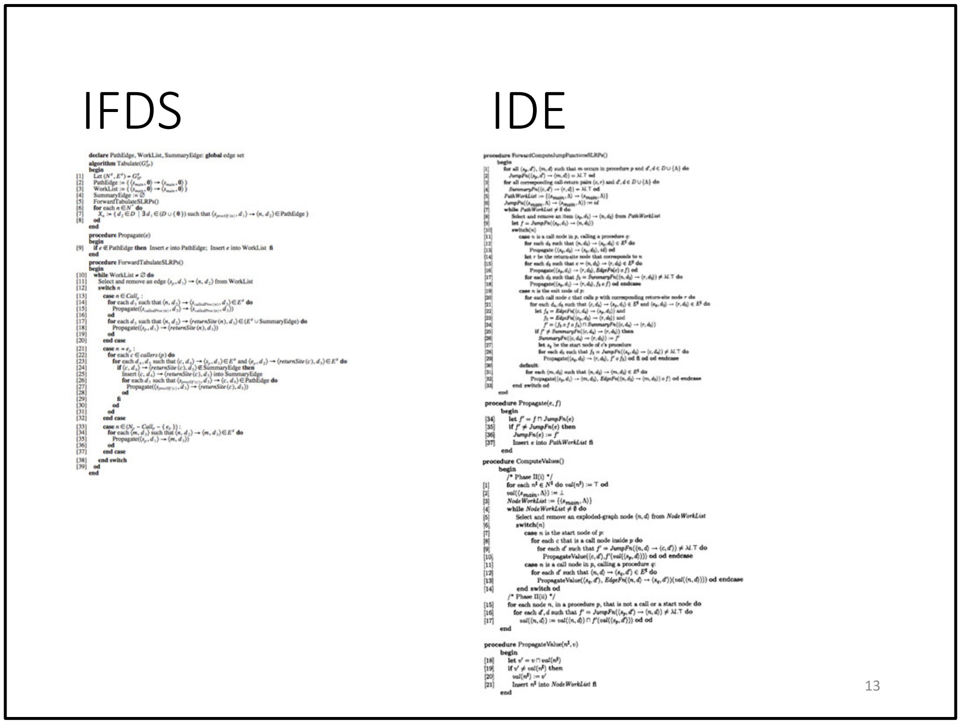
- Here's a small example of how Flix handles lattices.
  - We'll look at the first rule, and two input facts.
- Evaluating the rule, we infer that the local variable "x" has value 0 *and* 1.
  - But LocalVar is a lattice. We have two values for the same key.
  - We have to compress the values, using 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.

More Analyses in Flix

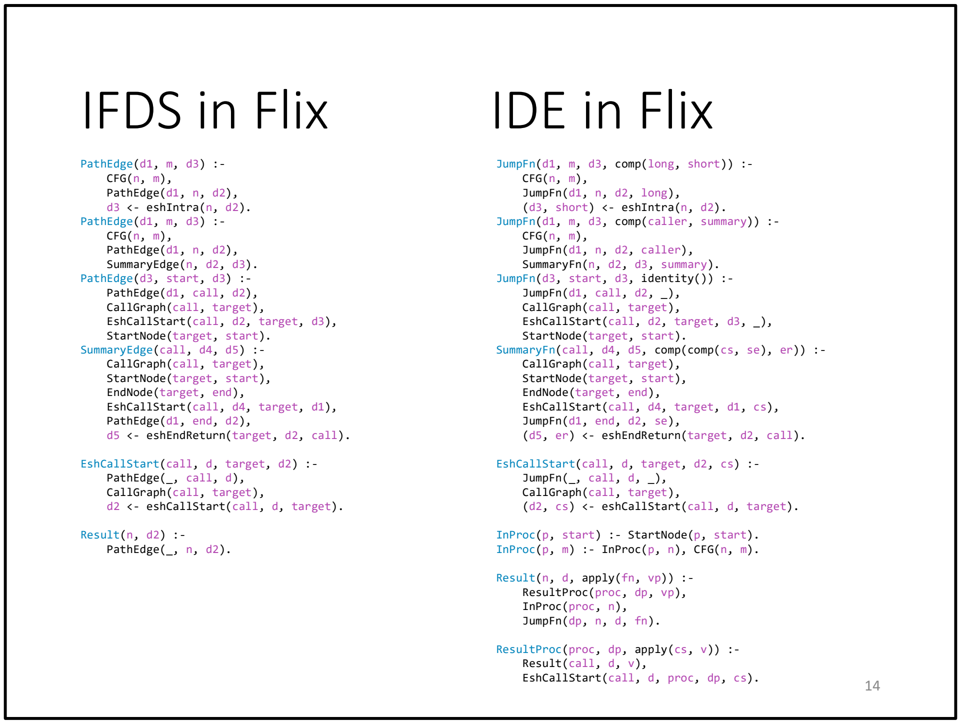
- Strong Update analysis
  - [Lhoták and Chung, POPL ‘11]
- IFDS algorithm
  - [Reps, Horwitz, and Sagiv, POPL ‘95]
- IDE algorithm
  - [Sagiv, Reps, and Horwitz, TCS ‘96]

12

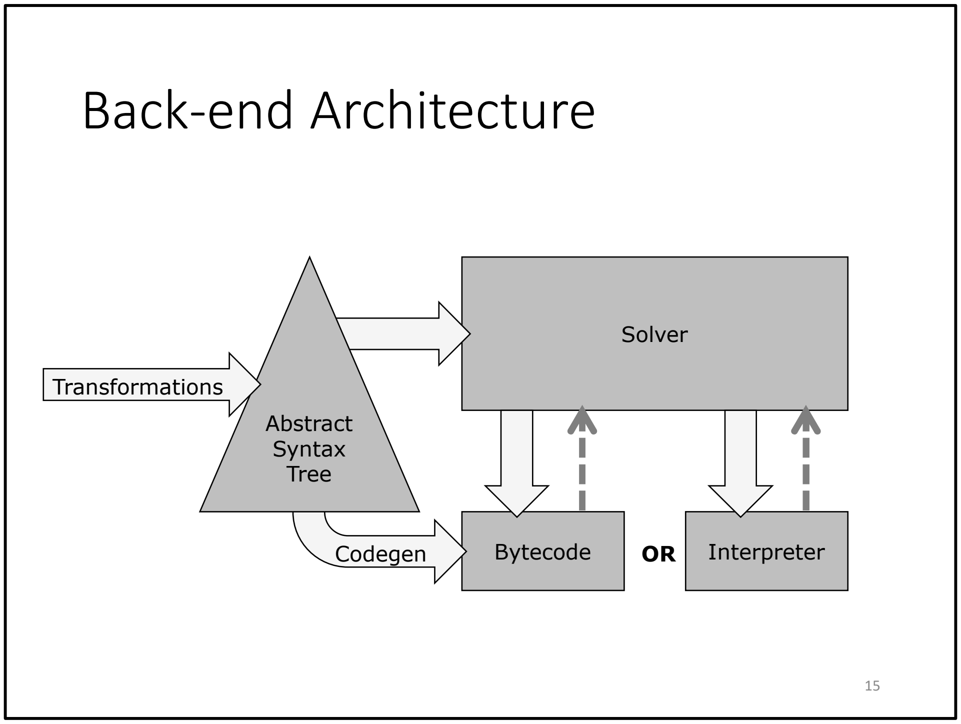
- Constant propagation is a bit of a “toy” analysis
- In the PLDI paper, we presented Flix implementations of three analyses
  - I’m not too familiar with these analyses, and only have a very basic understanding
- Strong Update analysis is a points-to analysis for C programs
  - Propagates singleton sets flow-sensitively, larger sets flow-insensitively
  - It’s possible to express in Datalog, but it uses a constant propagation lattice, so Datalog performance isn’t as good as Flix
- IFDS: Interprocedural Finite Distributive Subset
  - Framework for a specific class of problems
  - Transforms a dataflow problem into a graph reachability problem
  - Instantiate framework with a specific analysis by providing transfer functions
    - Can’t implement in Datalog, because of functions
- IDE: Interprocedural Distributive Environment
  - Generalization of IFDS



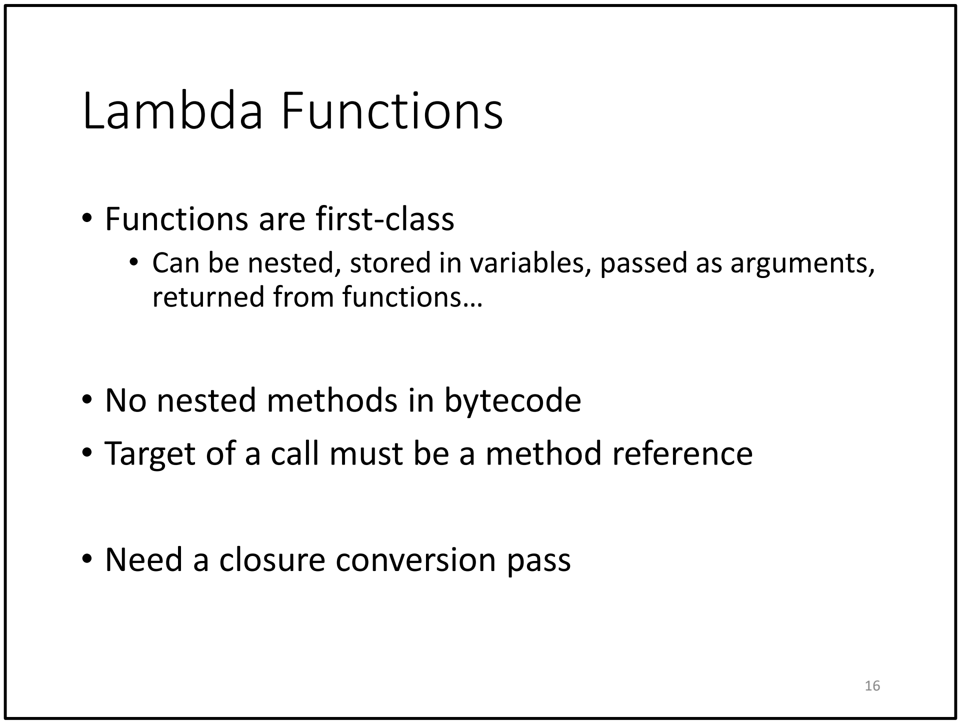
- The IFDS and IDE papers present the algorithm in pseudocode
    - 1 page for IFDS, 2 pages for IDE
- This is very difficult to understand
    - Even more difficult to implement
- It’s also not clear that IDE is an extension of IFDS



- With Flix, you can implement the algorithms declaratively and much more succinctly
- If you trust these implementations and squint a little, you can see the similarity
    - E.g. PathEdge corresponds to JumpFn
- Next slide begins implementation



- **10:00 (23:00 total) to get here.**
- 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 cover the code generator.



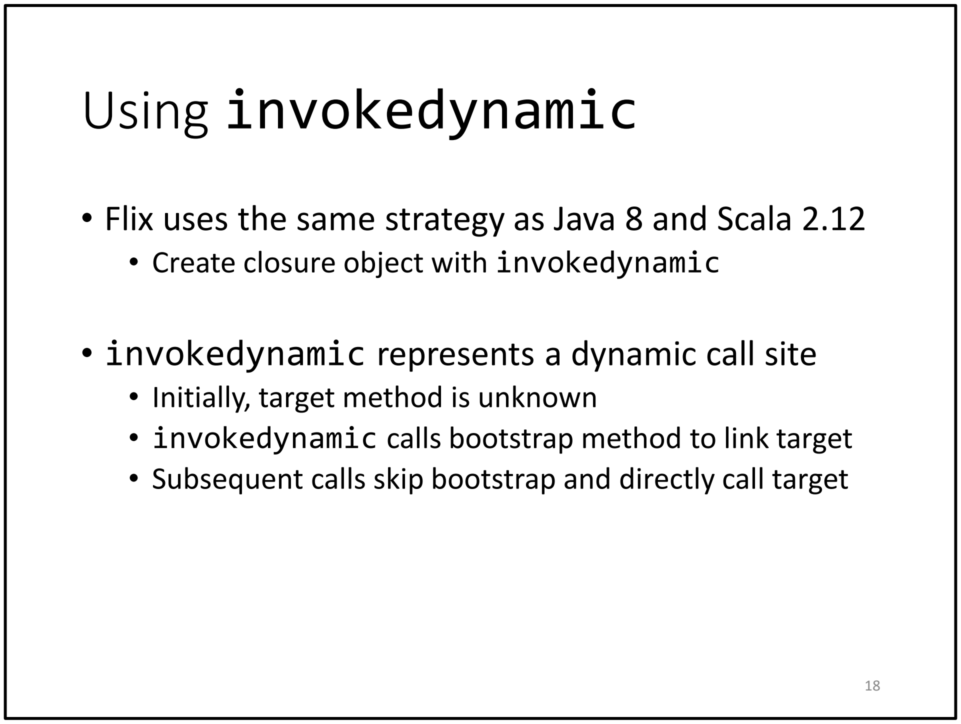
- In Flix, functions are first-class.
    - You can nest function definitions, store a function in a variable, pass it as an argument, and return from a function.
- This does not hold for bytecode.
    - All methods must be defined at the top-level. No nesting.
    - The target of a method call must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

Implementing Closures…?

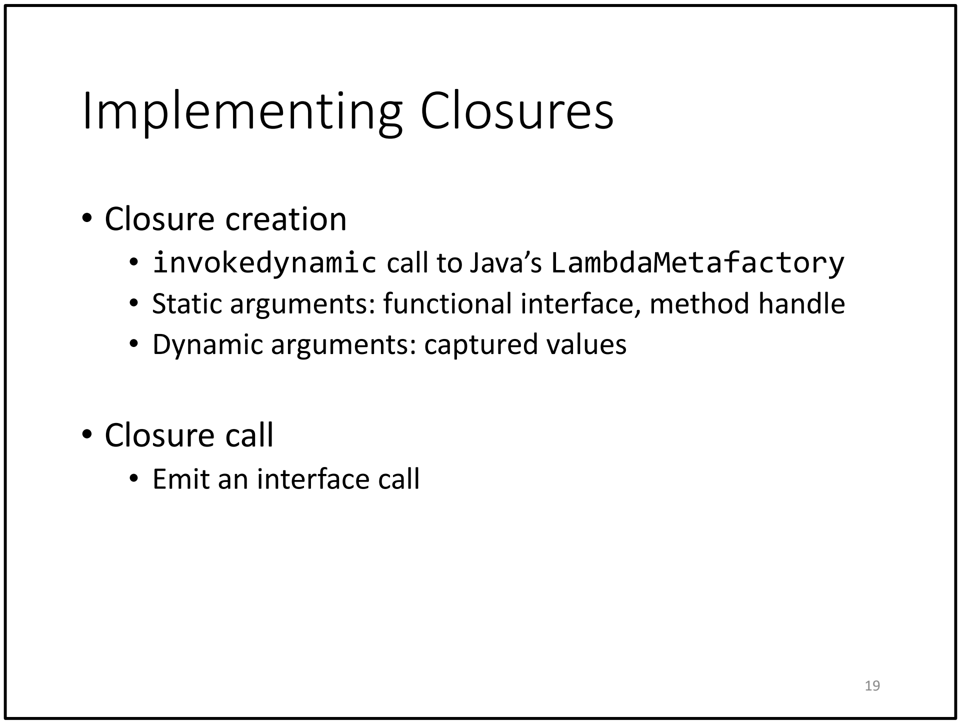
```
// Scala
val a = 10
val f = (x: Int, y: Int) => a + x + y
f(1, 2) // 13


// 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
```

17

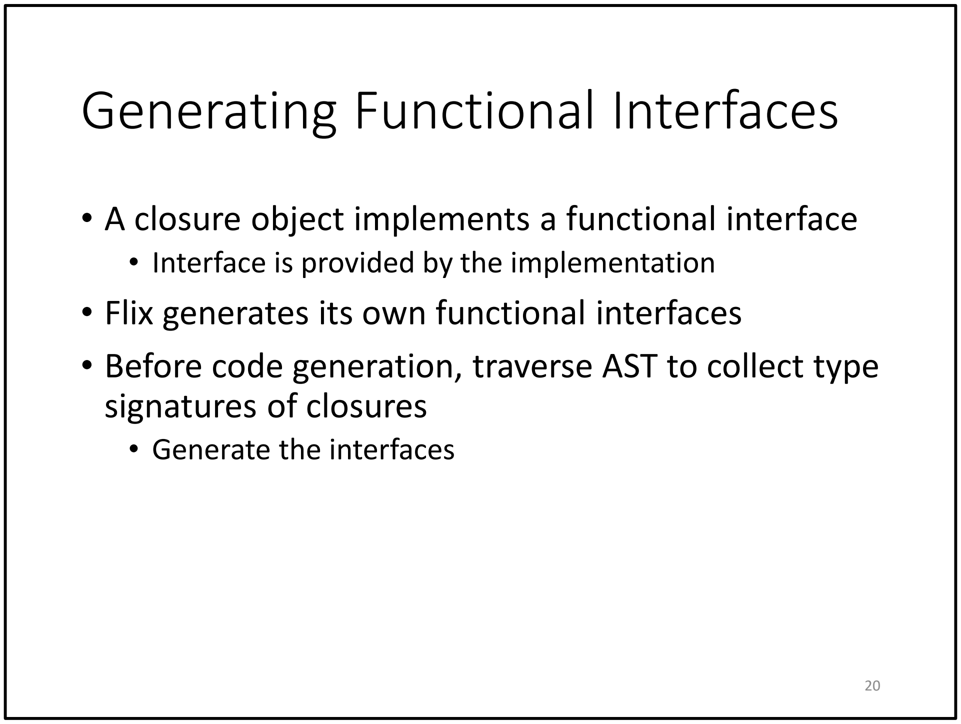
- **10:00 (33:00 total) to get here.**
- So, how do you actually implement closures in bytecode?
- In object-oriented languages, one way to implement closures is to use function objects.
    - C++, C#, and Scala 2.11 use this method.
- Every lambda function has an associated anonymous class.
    - The 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.



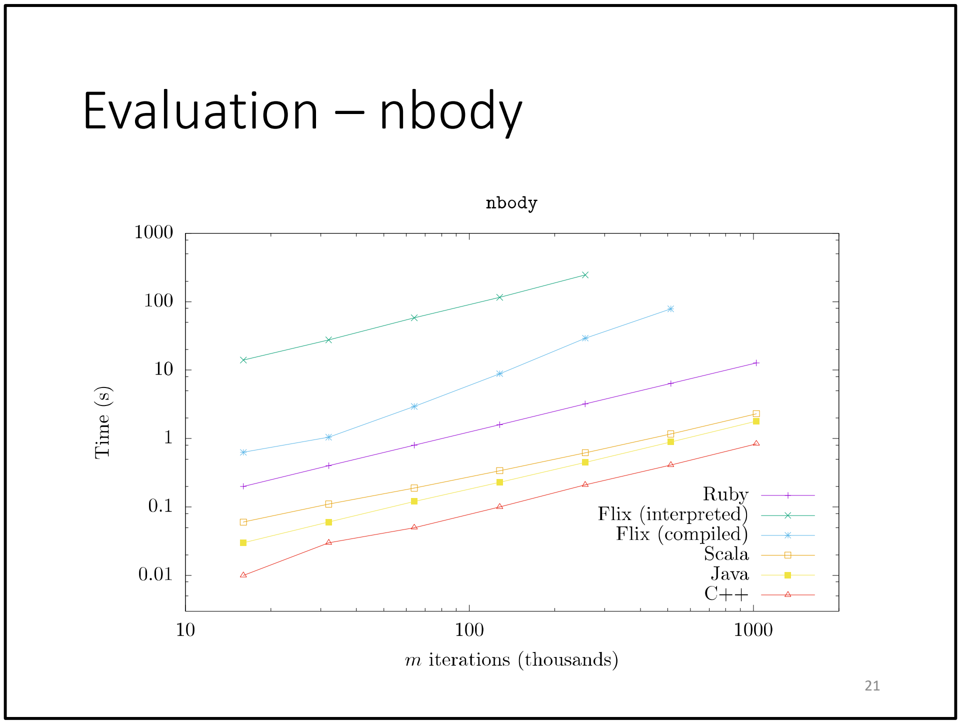
- An alternate approach, used by Java 8 and Scala 2.12, is invokedynamic.
  - Instead of the code generator statically creating the classes, invokedynamic will dynamically create the classes.
- Initially, the invokedynamic instruction is a dynamic call site, and the target of the call is unknown
  - 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"
  - Compared to existing methods, invokedynamic relaxes method calls – you don't need to provide the exact signature
- invokestatic – static method calls
- invokespecial – constructors, private methods, super calls
  - Methods are known statically and cannot be overridden
- invokevirtual – invoking method on a known object, vtable entry known statically, but not the target
- invokeinterface – invoking a method on an interface, vtable entry determined at runtime



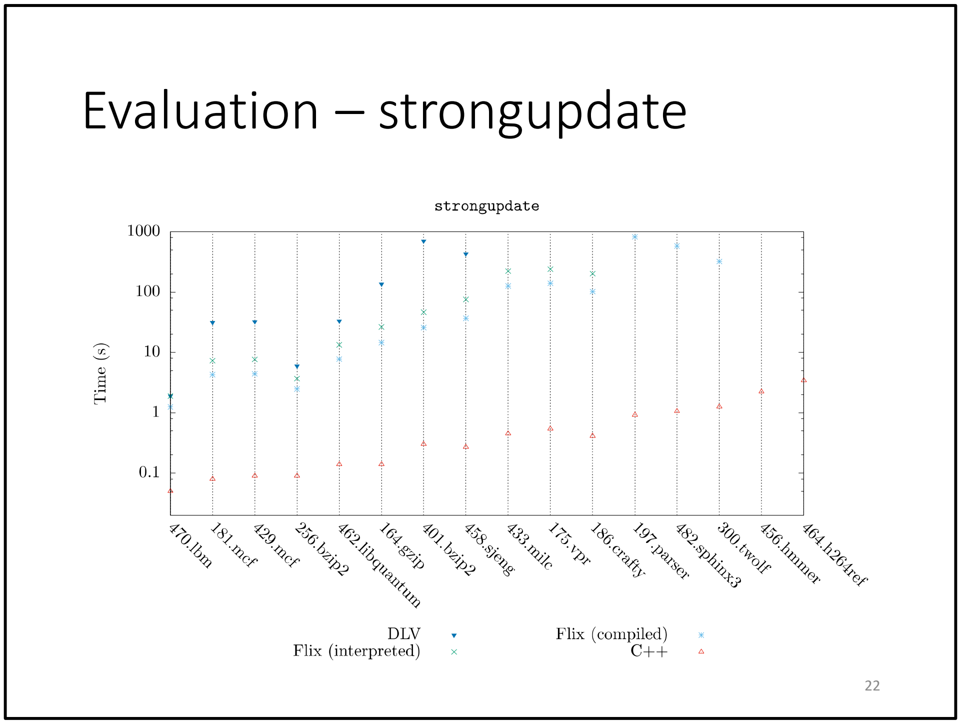
- To create a closure, code generator emits an invokedynamic call to LambdaMetafactory, which is defined in the Java standard library.
    - Static arguments represent the functional interface implemented by the closure, and a handle to the method implementing the function.
    - Dynamic arguments represent 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.



- 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.

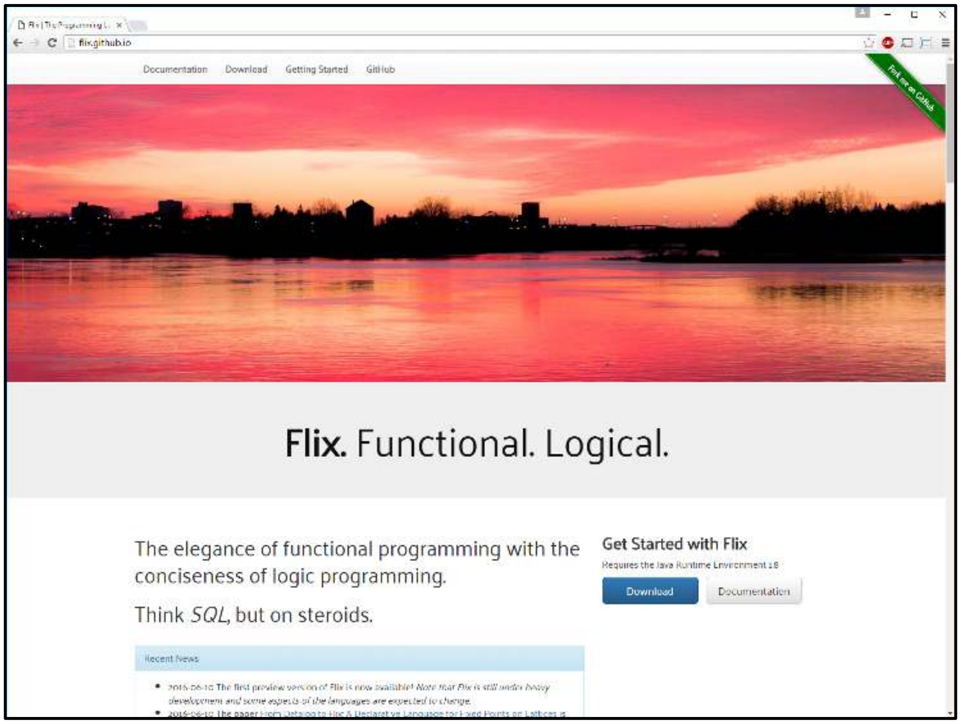


- **15:00 (38:00 total) to get here**.
- Nbody from Computer Language Benchmarks Game
- Both Flix implementations are the slowest.
  - But compiled Flix is 17x faster than interpreted Flix.
- nbody is the most complicated functional program implemented in Flix, and highlights many inefficiencies.
  - 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.

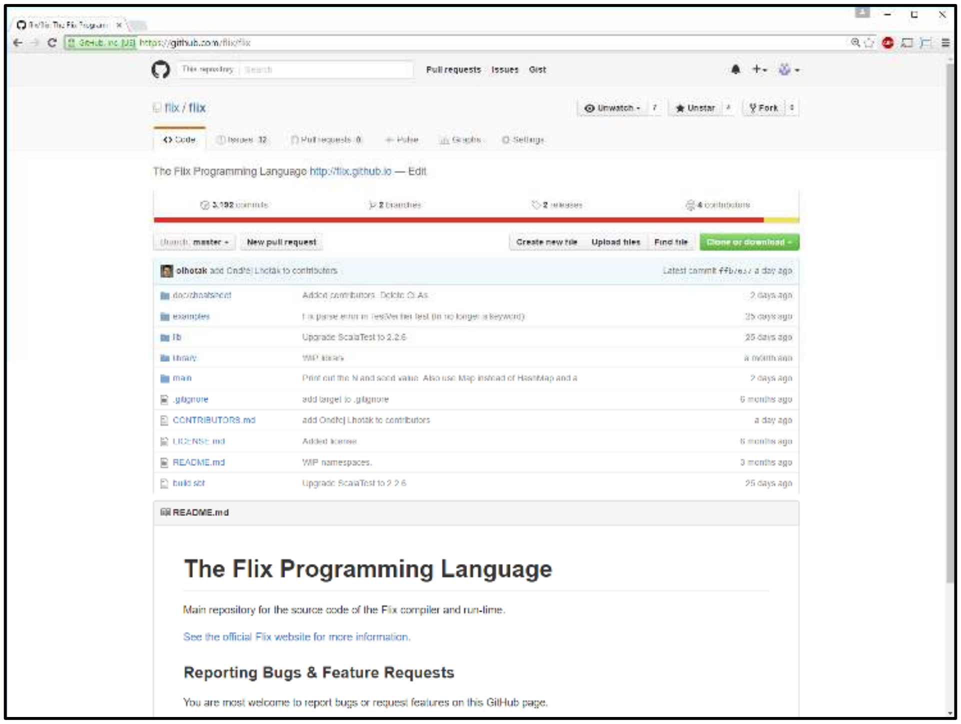


- DLV is the Datalog implementation, running on the DLV solver
- C++ is from the original Strong Update paper
- Uses SPEC integer benchmarks as inputs
- Differences are consistent.
    - Datalog slower than interpreted Flix, slower than compiled Flix, slower than C++.
- The analysis requires a constant 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.
- Compile logic language to JVM bytecode



- If you want to use Flix today or get more information, you can check out our website
    - Paper is linked there, and also some presentation slides



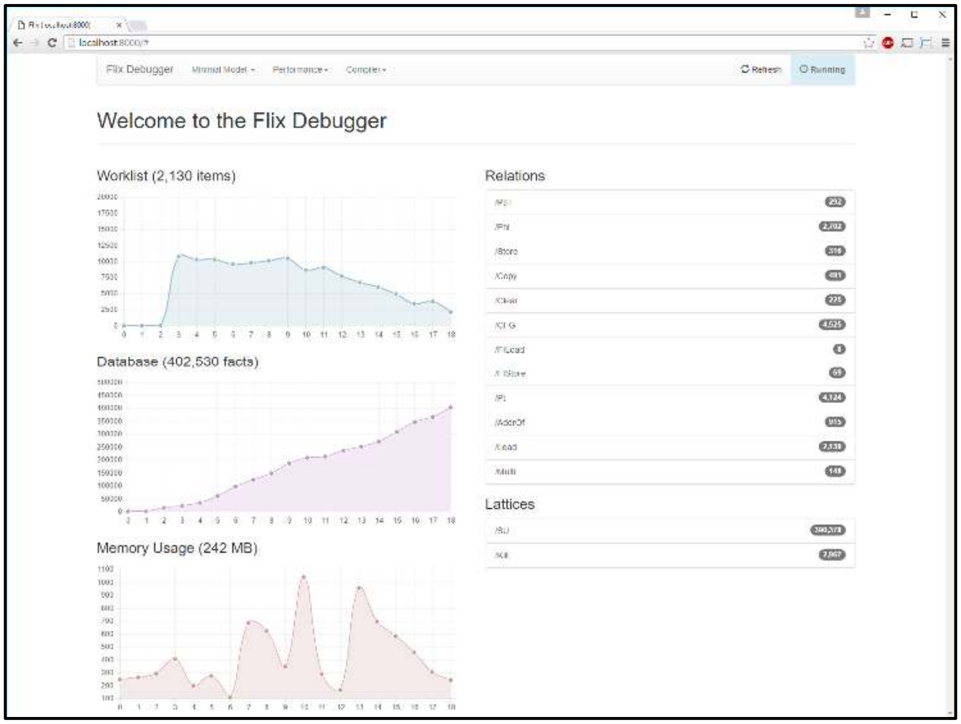
- We’re open-source and on GitHub
  - All you need is JDK 1.8
  - You can download Flix right now and try it out

```
→ java -jar flix.jar --verifier Sign.flix
-- VERIFIER ERROR ----------------------------------- Sign.flix

>> The function is not monotone.

Counter-example: x1$1402 -> Zer, x2$1406 -> Neg, y1$1404 -> Pos,
y2$1408 -> Pos

The function was defined here:
238|     def or(e1: Sign, e2: Sign): Sign = match (e1, e2) with {
             ^^
```

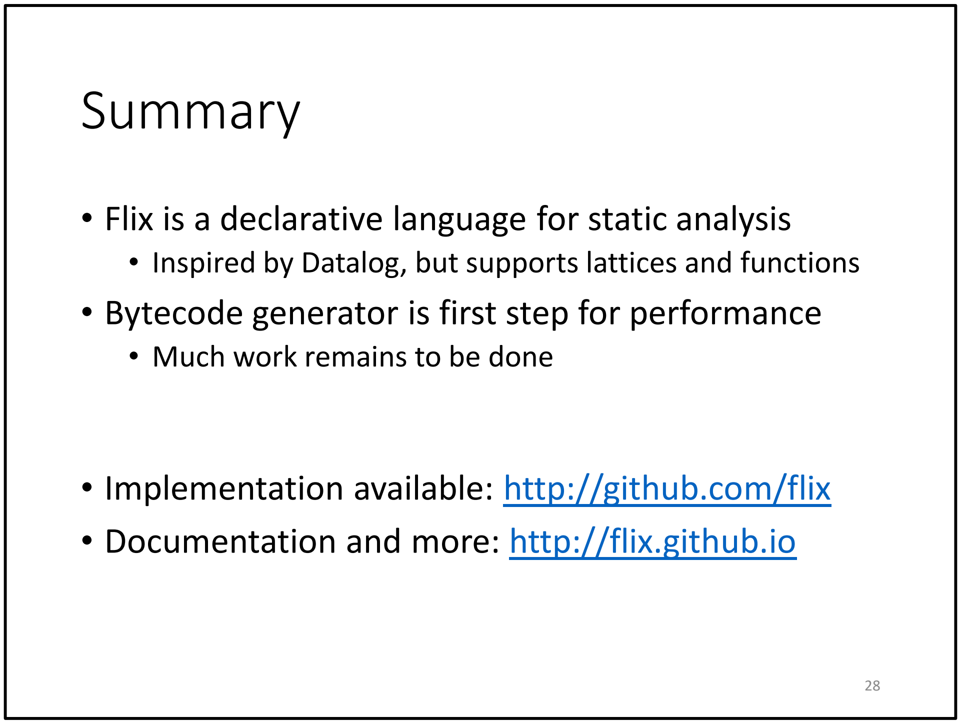
- Verifier to check correctness of your Flix program
    - Lattices need to actually be lattices with finite height (actually, ascending chain condition)
    - Functions need to be strict and monotone
    - Otherwise, Flix may not terminate, or worse, produce incorrect results
    - Use symbolic execution to ensure properties hold

```
→ java -jar flix.jar --delta out.flix delta-debugging.flix
Caught `ca.uwaterloo.flix.api.RuleException' with message:
    `The integrity rule defined at delta-debugging.flix:45:5 is violated.'
Delta Debugging Started. Trying to minimize 30 facts.

--- iteration:   1, current facts:    30, block size:    15 ---
    [block   1]    15 fact(s) retained (program ran successfully).
    [block   2]    15 fact(s) discarded.
--- Progress:   15 out of   30 facts (50.0%) ---

--- iteration:   2, current facts:    15, block size:     7 ---
    [block   1]     7 fact(s) retained (program ran successfully).
    [block   2]     7 fact(s) retained (program ran successfully).
    [block   3]     1 fact(s) discarded.
--- Progress:   14 out of   30 facts (46.7%) ---

--- iteration:   3, current facts:    14, block size:     3 ---
    [block   1]     3 fact(s) retained (program ran successfully).
    [block   2]     3 fact(s) retained (program ran successfully).
    [block   3]     2 fact(s) discarded.
    [block   4]     3 fact(s) discarded.
    [block   5]     3 fact(s) retained (program ran successfully).
--- Progress:    9 out of   30 facts (30.0%) ---

--- iteration:   4, current facts:     9, block size:     1 ---
    [block   1]     1 fact(s) retained (program ran successfully).
    [block   2]     1 fact(s) discarded.
    [block   3]     1 fact(s) discarded.
    [block   4]     1 fact(s) retained (program ran successfully).
    [block   5]     1 fact(s) discarded.
    [block   6]     1 fact(s) discarded.
    [block   7]     1 fact(s) retained (program ran successfully).
    [block   8]     1 fact(s) discarded.
    [block   9]     1 fact(s) discarded.
--- Progress:    3 out of   30 facts (10.0%) ---

    >>> Delta Debugging Complete! <<<
    >>> Output written to `out.flix'. <<<
```

- Flix has a delta debugging tool
    - Some large set of input facts causes an error
    - Prune the set to create a smaller set of facts that still triggers the error



- There's a visual debugger, which can help you pinpoint performance issues in your Flix program



- **7:00 (45:00 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.