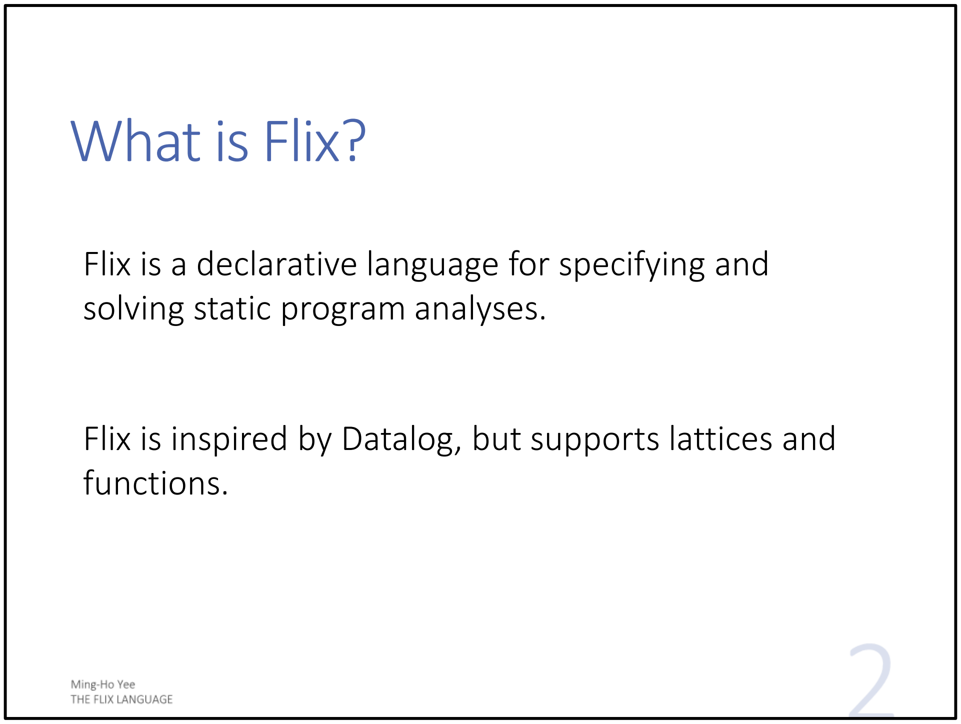
The Flix Language

Magnus Madsen, **Ming-Ho Yee**, Ondřej Lhoták
University of Waterloo

March 4, 2016

- Today I’m going to be talking about the Flix language.
- Flix is a project that Magnus, Ondřej, and I have been working on for some time.
- Our paper was (conditionally) accepted at PLDI.
    - Conditionally, because we have to work on revisions.
    - We’re not ready to share the draft yet, because we want to finish our revisions.

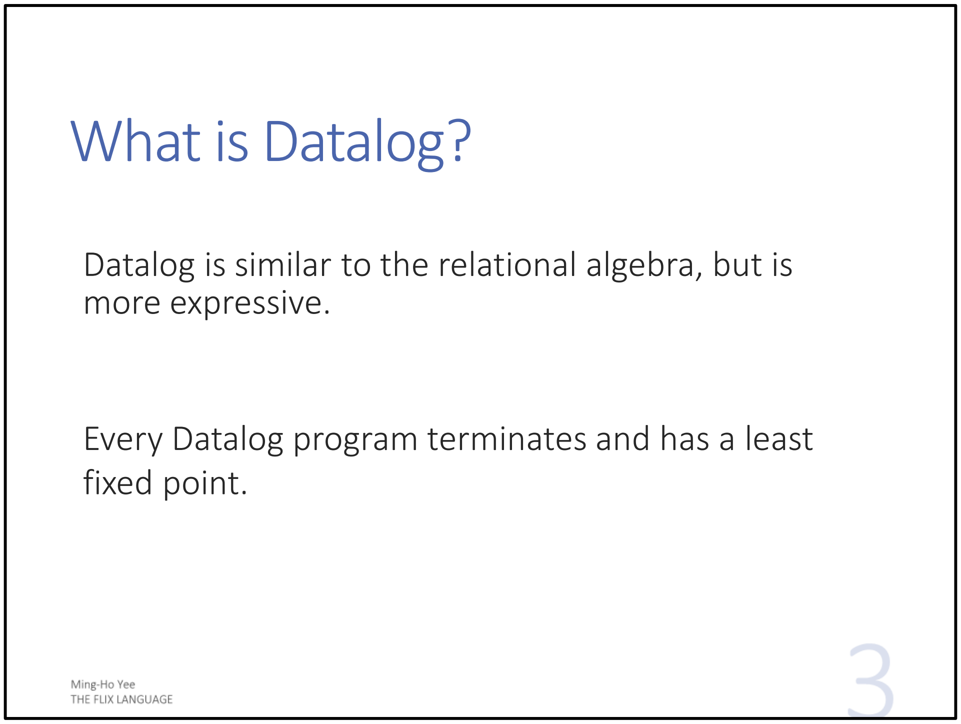


What is Flix?

Flix is a declarative language for specifying and solving static program analyses.

Flix is inspired by Datalog, but supports lattices and functions.

- Flix is a declarative language for specifying and solving fixed-point computations on lattices.
  - This is really specific, but the main use case is for writing static program analyses.
- We want the language to be declarative. This will make it much easier to write static analyses.
  - If you write your analyzer in C++, it can be very complicated and difficult to understand.
- Our main inspiration is Datalog, but Flix supports user-defined lattices and functions.

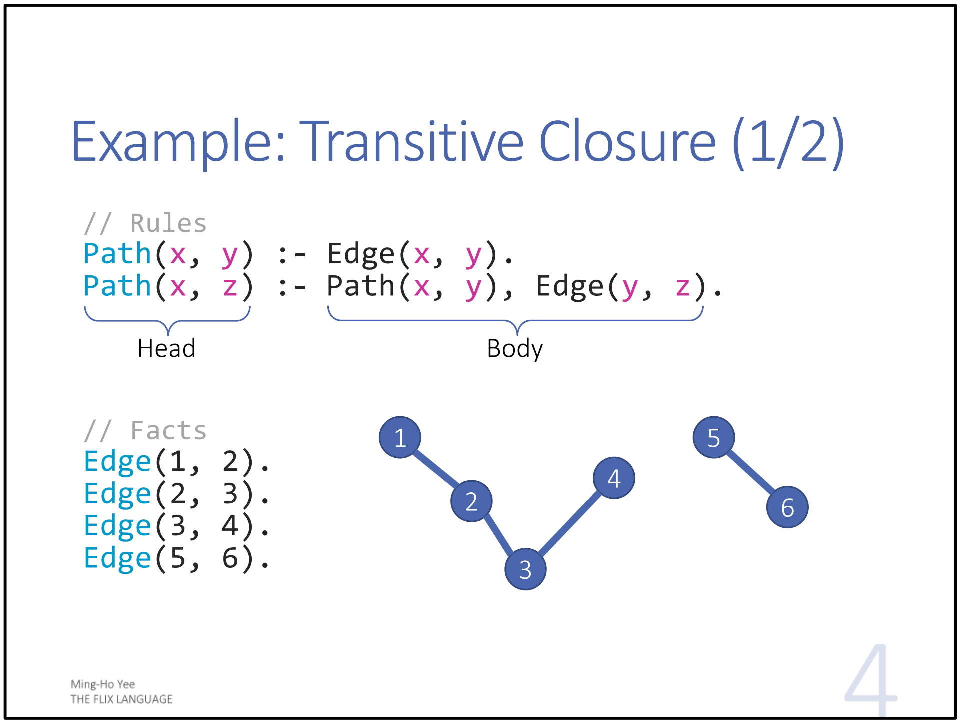


What is Datalog?

Datalog is similar to the relational algebra, but is more expressive.

Every Datalog program terminates and has a least fixed point.

- At a very high level, Datalog is similar to the relational algebra, but is more expressive.
    - Like SQL + recursion
- General idea: start with “database” of initial facts, and infer new facts with rules.
- Nice properties about Datalog that we want to ensure in Flix:
    - Semantics guarantee that every program terminates and has a least fixed point



Example: Transitive Closure (1/2)

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

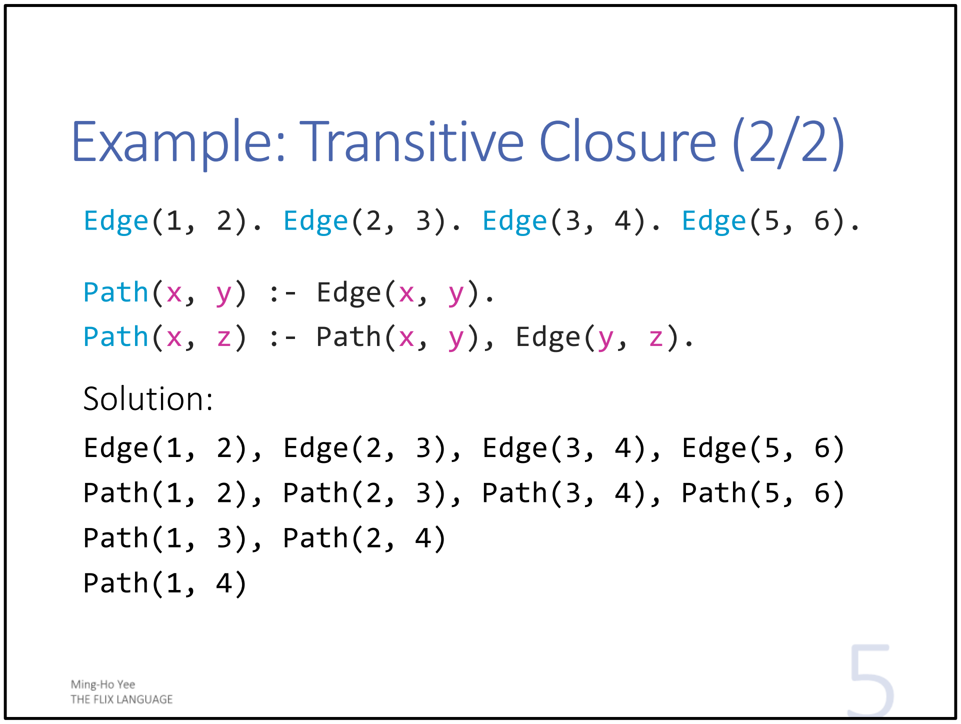
Head

Body

```
// Facts
Edge(1, 2).
Edge(2, 3).
Edge(3, 4).
Edge(5, 6).
```

Ming-Ho Yee
THE FLIX LANGUAGE

- Common example Datalog program
    - Computes the transitive closure (i.e. reachability) of a graph.
- Here, Path and Edge are relations. We start with some known edges, and want to compute all 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)*”
- Here we explicitly list out the initial facts. In this case, the graph has four edges.
- Order doesn’t matter, so I can list the facts after the rules.



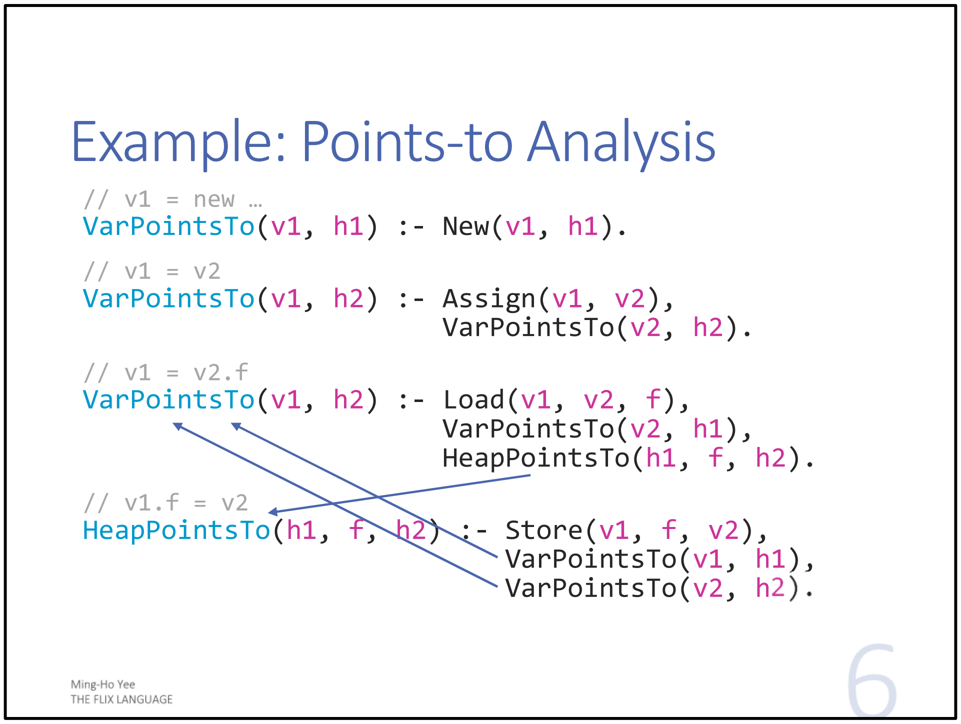
Example: Transitive Closure (2/2)

```
Edge(1, 2). Edge(2, 3). Edge(3, 4). Edge(5, 6).

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

Solution:

```
Edge(1, 2), Edge(2, 3), Edge(3, 4), Edge(5, 6)
Path(1, 2), Path(2, 3), Path(3, 4), Path(5, 6)
Path(1, 3), Path(2, 4)
Path(1, 4)
```

- The solution is the *minimal* set of facts that satisfies a Datalog program.
  - This includes the initial facts (e.g. edges).
- Solution:
  - Start with the initial facts (edges)
  - First rule is straightforward; edges are also paths
  - We look at the existing facts and combine paths and edges to create new paths
- In this example solution, if we removed or added anything, it would no longer be a solution.
  - Remove a fact and we don’t satisfy the program. Add a fact and it’s no longer minimal.

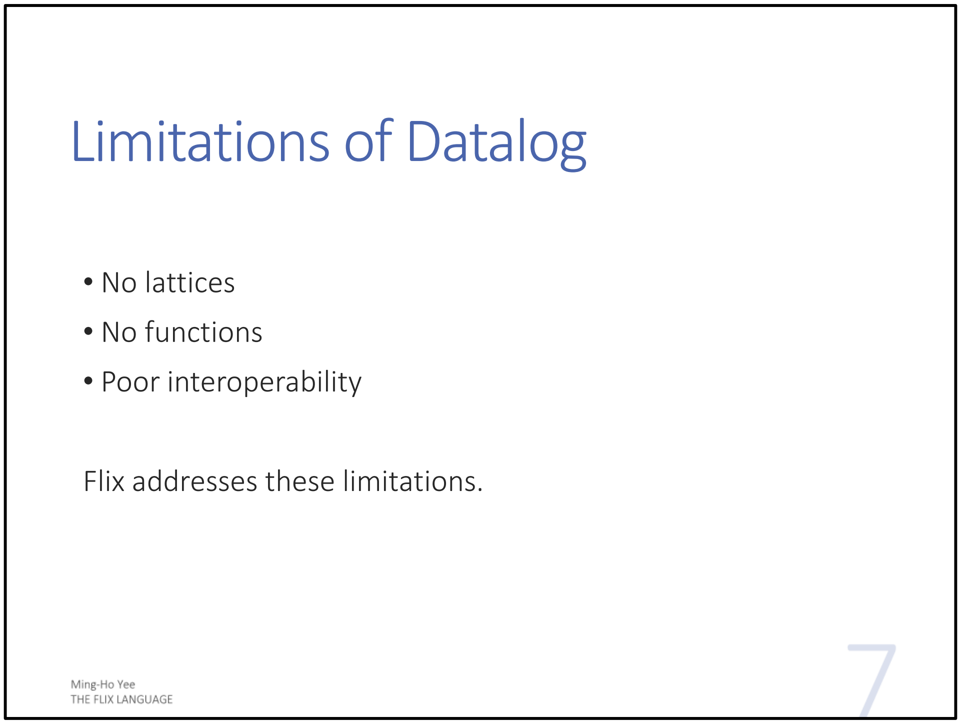


Example: Points-to Analysis

```
// v1 = new …
VarPointsTo(v1, h1) :- New(v1, h1).

// v1 = v2
VarPointsTo(v1, h2) :- Assign(v1, v2),
                       VarPointsTo(v2, h2).

// v1 = v2.f
VarPointsTo(v1, h2) :- Load(v1, v2, f),
                       VarPointsTo(v2, h1),
                       HeapPointsTo(h1, f, h2).

// v1.f = v2
HeapPointsTo(h1, f, h2) :- Store(v1, f, v2),
                           VarPointsTo(v1, h1),
                           VarPointsTo(v2, h2).
```

Ming-Ho Yee
THE FLIX LANGUAGE

- This is an example of a more realistic Datalog program: a points-to analysis.
- The interesting point about this example is the mutual recursion in the third and fourth rules.
    - Writing this out in Datalog is nice, but writing it in an imperative language is not so nice.

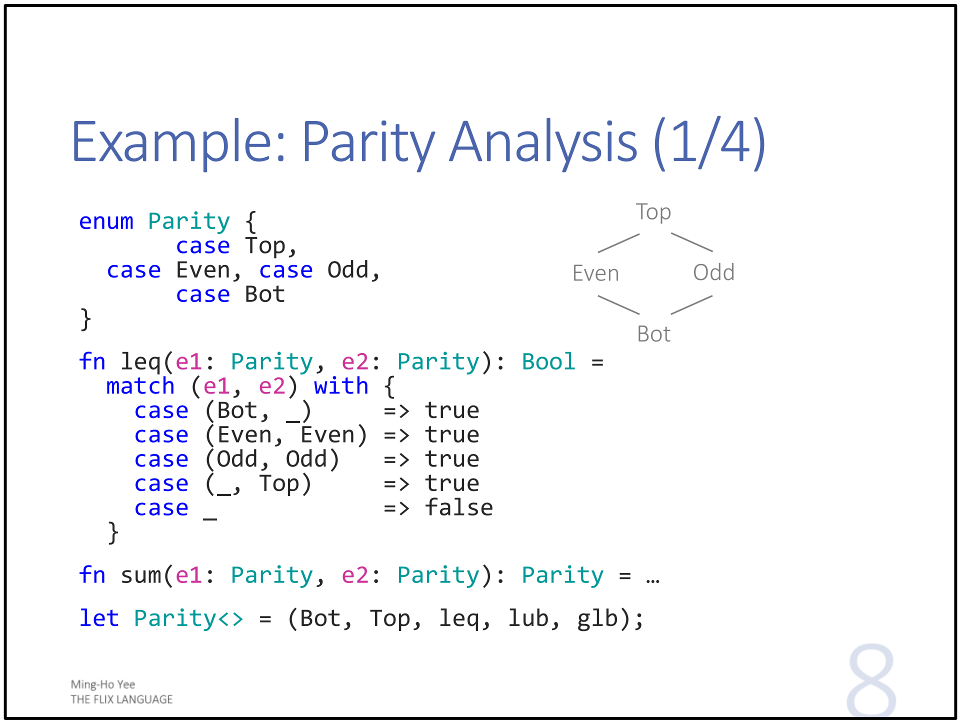


Limitations of Datalog

- No lattices
- No functions
- Poor interoperability

Flix addresses these limitations.

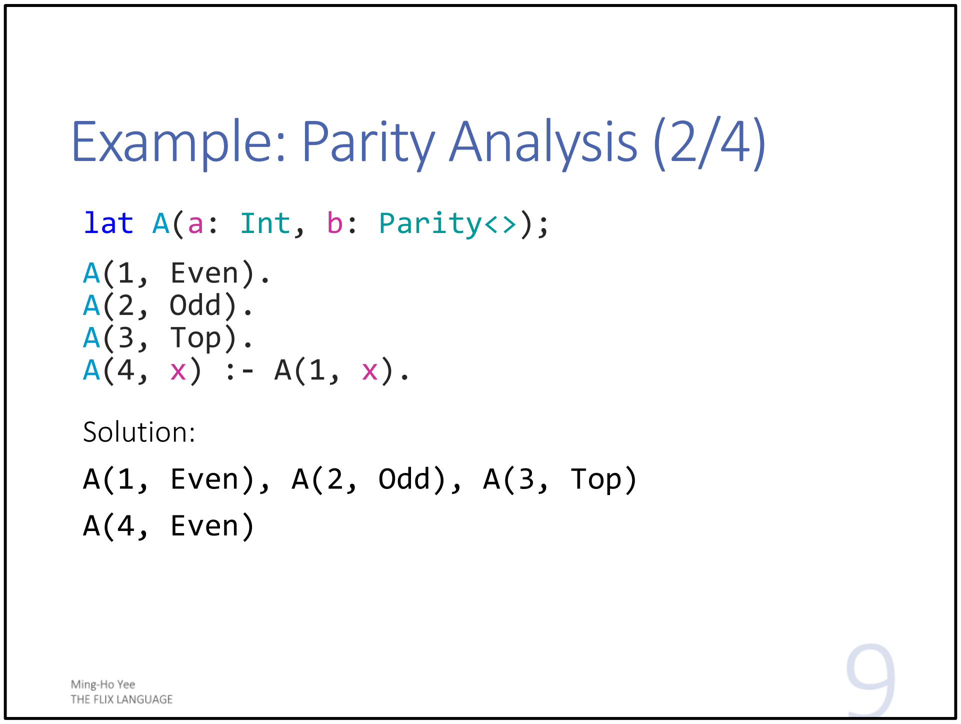
- The declarative nature of Datalog is a significant advantage.
- But Datalog has some limitations. It only allows constraints on relations. Thus:
    - No lattices
    - No functions
- It is possible to work around some of these (e.g. representing a lattice as a powerset, explicitly tabulating a function).
    - But this is slow, cumbersome, and sometimes not even possible (e.g. function with infinite domain, lattice with infinite domain (constant propagation)).
- An unrelated limitation (that won’t really be discussed today): poor interoperability with other tools
- The goal of Flix is to address these limitations.



Example: Parity Analysis (1/4)

```
enum Parity {
       case Top,
  case Even, case Odd,
       case Bot
}
fn leq(e1: Parity, e2: Parity): Bool =
  match (e1, e2) with {
    case (Bot, _)      => true
    case (Even, Even)  => true
    case (Odd, Odd)    => true
    case (_, Top)      => true
    case _             => false
  }
fn sum(e1: Parity, e2: Parity): Parity = …
let Parity<> = (Bot, Top, leq, lub, glb);
```

- Example: parity analysis implemented in Flix
    - Example is simplified; some details are omitted and syntax changed.
- The Flix language has two components:
    - A small, pure, functional language with Scala-like syntax
    - A logic language for expressing constraints with Datalog-like syntax
- The enum defines the elements of the parity lattice
- We then define the lattice operations (leq, lub, glb)
- We can also define other functions:
    - A function that sums two parity elements (E+E = E, etc.).
- Finally, we declare a lattice type (taking an enum, lattice operations, top, bot)



Example: Parity Analysis (2/4)

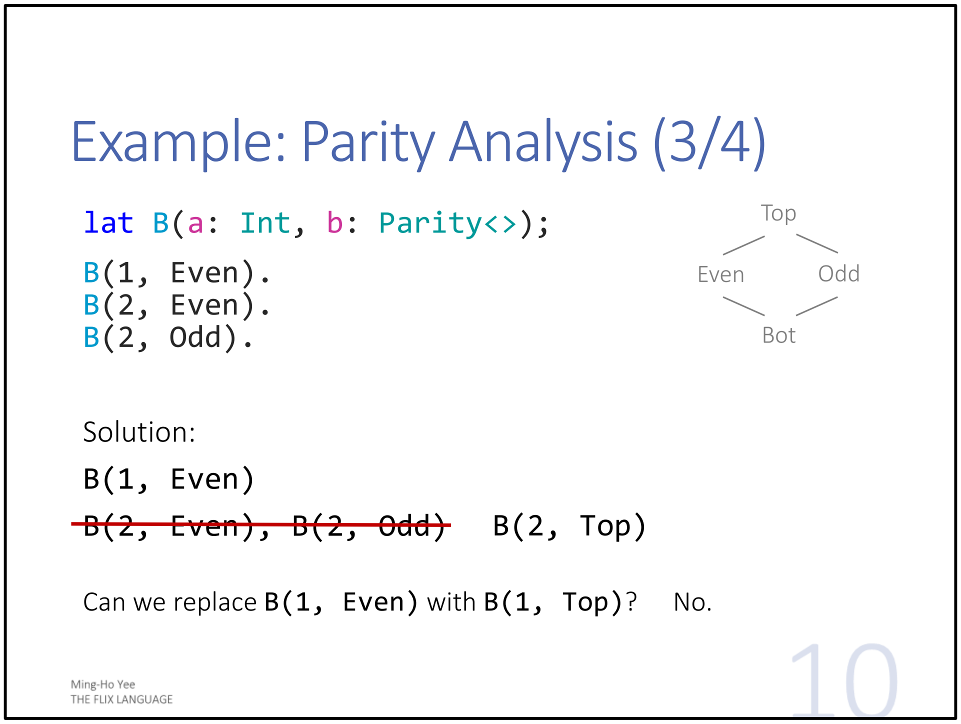
```
lat A(a: Int, b: Parity<>);
```

```
A(1, Even).
A(2, Odd).
A(3, Top).
A(4, x) :- A(1, x).
```

Solution:

```
A(1, Even), A(2, Odd), A(3, Top)
A(4, Even)
```

- Facts and rules are similar to Datalog.
  - Facts are explicitly listed in the Flix program.
  - But they come from somewhere else, e.g. run the source code through a phase that extracts this information.
- Here, A is a *lattice* and not a *relation* (as it would be in Datalog).
  - Intuitively, it is a *map lattice* from integers (“identifiers”) to parity elements.
  - Note that variable “b” is a “Parity<>”, which means we want to apply lattice semantics.
  - “A(1, Even)“ means “variable/expression/statement 1 has even parity.”
  - The rule says “If 1 has parity x, then 4 must have parity x.”
- As in Datalog, a solution is the minimal set of facts that satisfy the program.
  - We’ll have to tweak the definition of “minimal” to account for lattices, but first, we’ll look at simpler examples.
- This specific solution is straightforward and not very different from Datalog.
  - The main difference is that we have parity elements “Even,” “Odd,” “Top,” and “Bot” (not shown).
  - This is the minimal set of facts.
    - Remove a fact and we don’t satisfy the program. Add a fact and it’s not minimal.



Example: Parity Analysis (3/4)

```
lat B(a: Int, b: Parity<>);
B(1, Even).
B(2, Even).
B(2, Odd).
```

Top

Even Odd

Bot

Solution:

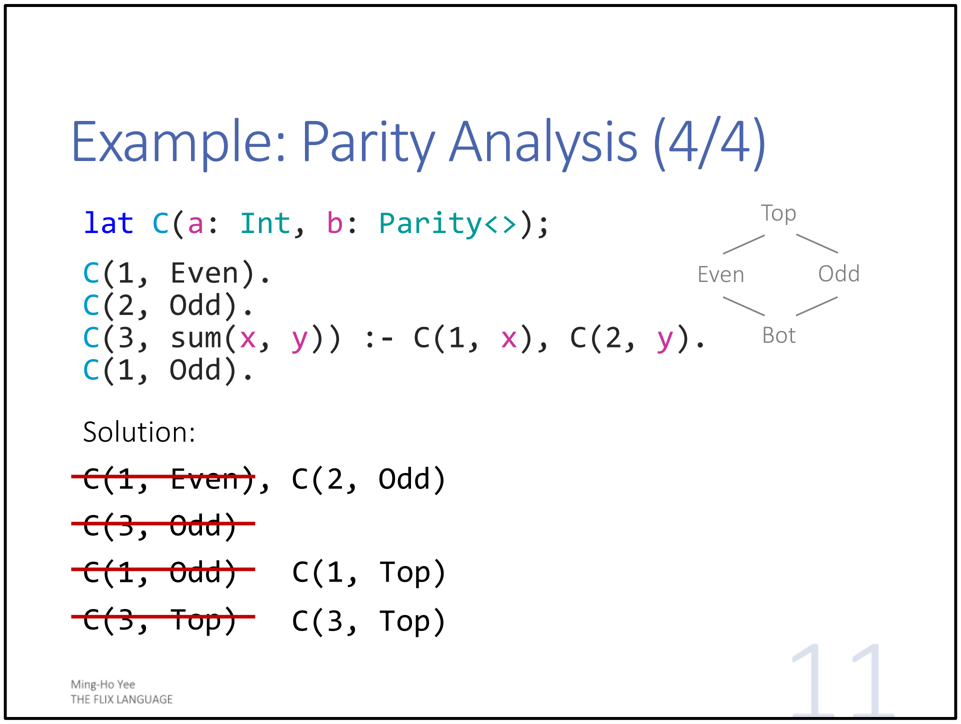
`B(1, Even)`

~~`B(2, Even), B(2, Odd)`~~ `B(2, Top)`

Can we replace `B(1, Even)` with `B(1, Top)`? No.

Ming-Ho Yee
THE FLIX LANGUAGE

- In this example, we'll see how lattices cause Flix to differ from Datalog.
  - Note how we're assigning two different parities to "2."
- With these three facts, we satisfy the program. But is this minimal?
  - That depends on the definition of "minimal."
- Flix's semantics say "no, this is not minimal." We want to actually use lattices here.
  - We need to "compress" the two facts.
    - In the declaration, we used "Parity<>" instead of "Parity" to say that we want to compress elements here.
    - We don't want to compress "B(1, Even)" with "B(2, Even)" because we wrote "Int" and not "Int<>".
  - Even and Odd are both elements of the parity lattice. So take the least upper bound to get Top.
  - "2 is Top" satisfies both "2 is Even" and "2 is Odd."
- Can we replace "B(1, Even)" with "B(1, Top)"? It satisfies the fact.
  - But it isn't minimal, because "B(1, Even)" is more precise, since Even ⊑ Top.



Example: Parity Analysis (4/4)

```
lat C(a: Int, b: Parity<>);

C(1, Even).
C(2, Odd).
C(3, sum(x, y)) :- C(1, x), C(2, y).
C(1, Odd).
```

Solution:

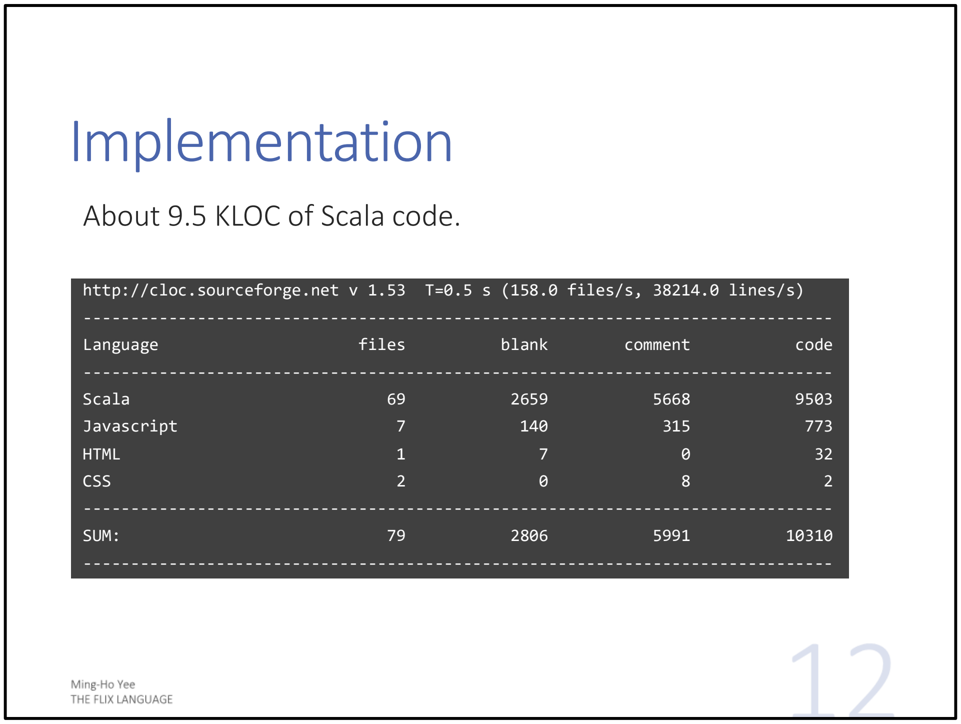
~~C(1, Even)~~, C(2, Odd)

~~C(3, Odd)~~

~~C(1, Odd)~~ C(1, Top)

~~C(3, Top)~~ C(3, Top)

- Here’s our final example.
- First step is easy: 1 is Even and 2 is Odd.
- To evaluate the rule, we have to call the sum function.
  - If 1 has parity x and 2 has parity y, then the parity of 3 must be sum(x, y).
  - In this case, we conclude that 3 is Odd.
- Now we see that 1 is also Odd. As before, we need to take the join of Even and Odd, to conclude that 1 is Top.
- This changes our facts, so we have to re-evaluate the rule, and conclude that 3 is Top.
- Finally, we take the join of Odd and Top to conclude that 3 is Top.
- Note that I chose this order of evaluation. Any other order will return the same answer.
- We have a model-theoretic semantics for Flix, and it’s in our paper
  - We’re working on a fixed-point semantics
  - We have an implementation (but not a proof of correctness)



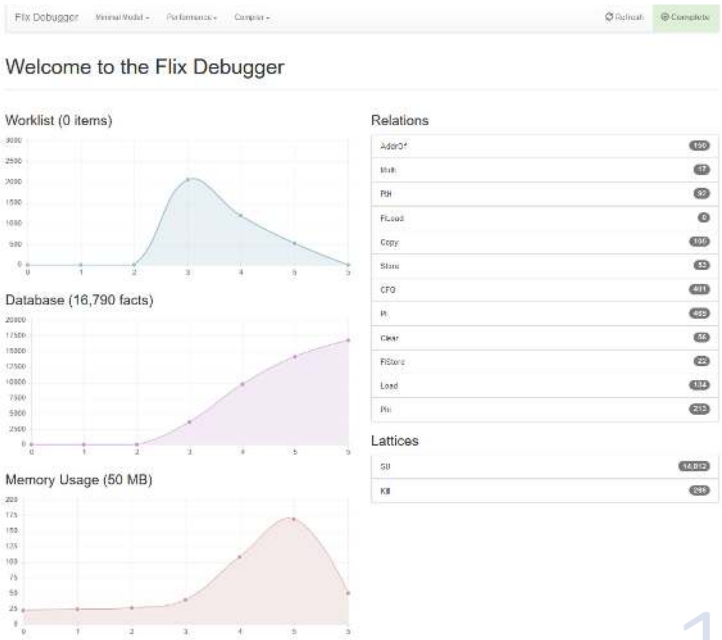
Implementation

About 9.5 KLOC of Scala code.

```
http://cloc.sourceforge.net v 1.53  T=0.5 s (158.0 files/s, 38214.0 lines/s)
-------------------------------------------------------------------------------
Language                     files          blank        comment           code
-------------------------------------------------------------------------------
Scala                           69           2659           5668           9503
Javascript                       7            140            315            773
HTML                             1              7              0             32
CSS                              2              0              8              2
-------------------------------------------------------------------------------
SUM:                            79           2806           5991          10310
-------------------------------------------------------------------------------
```

Ming-Ho Yee
THE FLIX LANGUAGE

- Our implementation language is Scala
    - We can take advantage of nice language features (immutability, pattern matching, Scala's standard library, etc.)
    - Being JVM-based is also nice, as it makes interop easier.
- Currently we have roughly 9.5 KLOC (and growing).
    - This number excludes blanks, comments, and tests.
    - The JS/HTML/CSS is our browser-based debugger.



Welcome to the Flix Debugger

Worklist (0 items)

Database (16,790 facts)

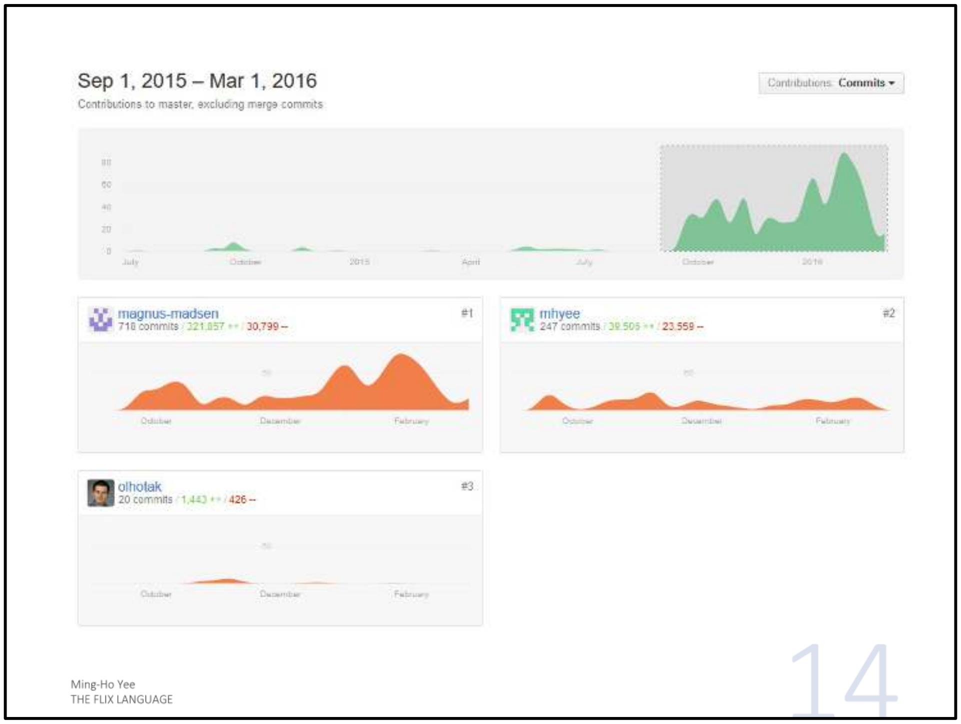
Memory Usage (50 MB)

Relations

AddrOf	150
Multi	17
PtH	92
FILoad	0
Copy	100
Store	53
CFG	401
Pt	465
Clear	56
FIStore	22
Load	134
Phi	213

Lattices

SU	14,012
Kill	286



Sep 1, 2015 – Mar 1, 2016

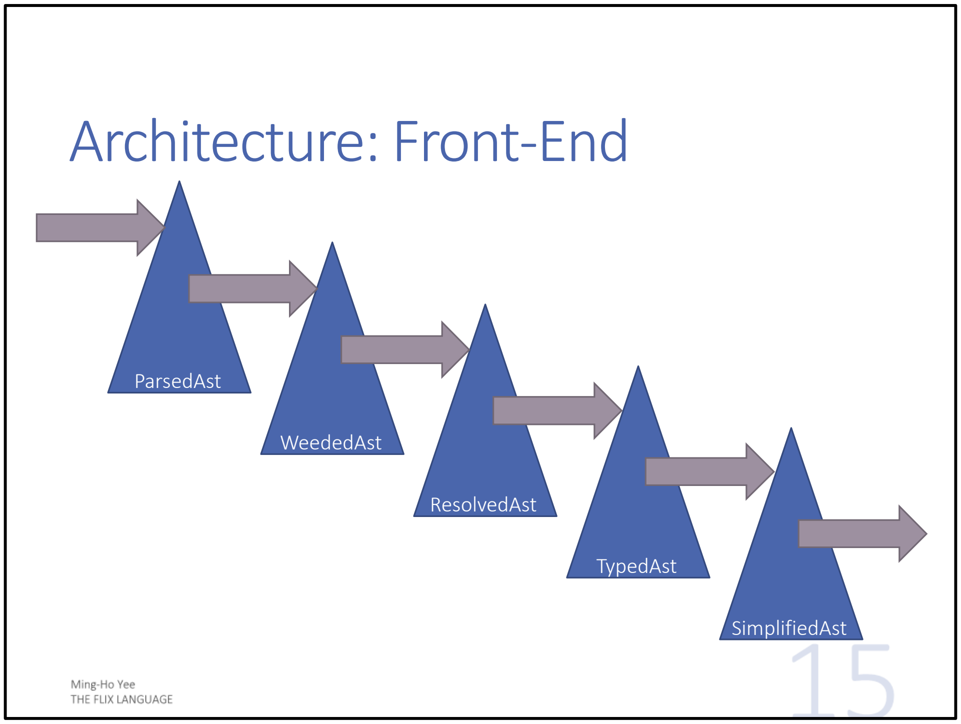
Contributions to master, excluding merge commits

magnus-madsen #1
718 commits / 321,857 ++ / 30,799 --

mhyee #2
247 commits / 39,505 ++ / 23,559 --

olhotak #3
20 commits / 1,443 ++ / 426 --

- The bulk of the work started in September 2015.
    - Everything before that is old, prototype code that’s been thrown away.
- Note that the commits and diffs are inflated, due to refactoring (moving files).



Architecture: Front-End

ParsedAst

WeededAst

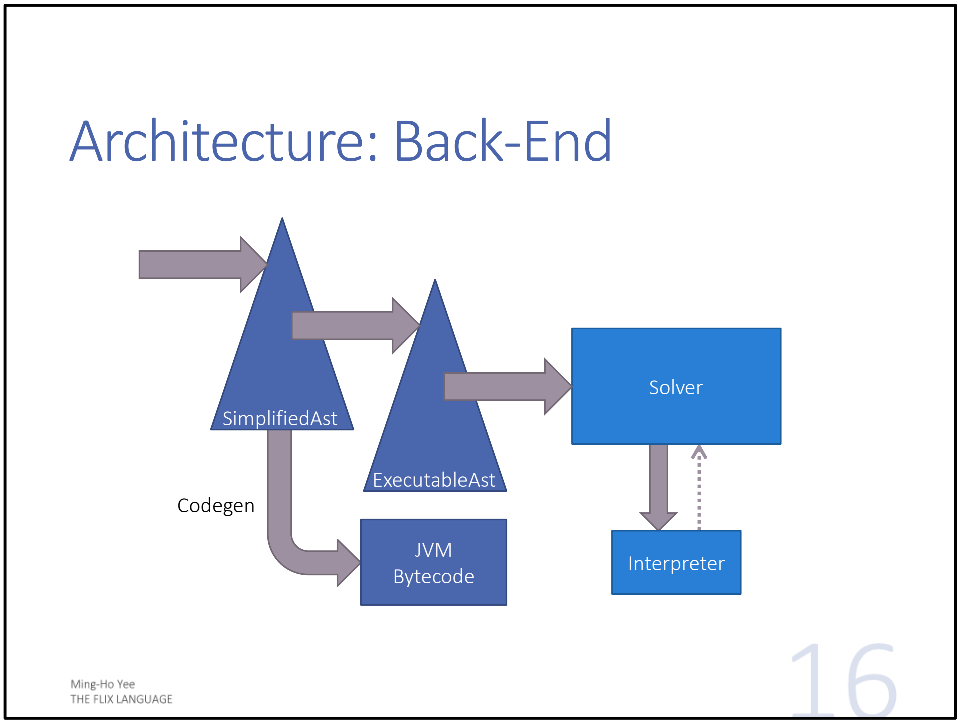
ResolvedAst

TypedAst

SimplifiedAst

Ming-Ho Yee
THE FLIX LANGUAGE

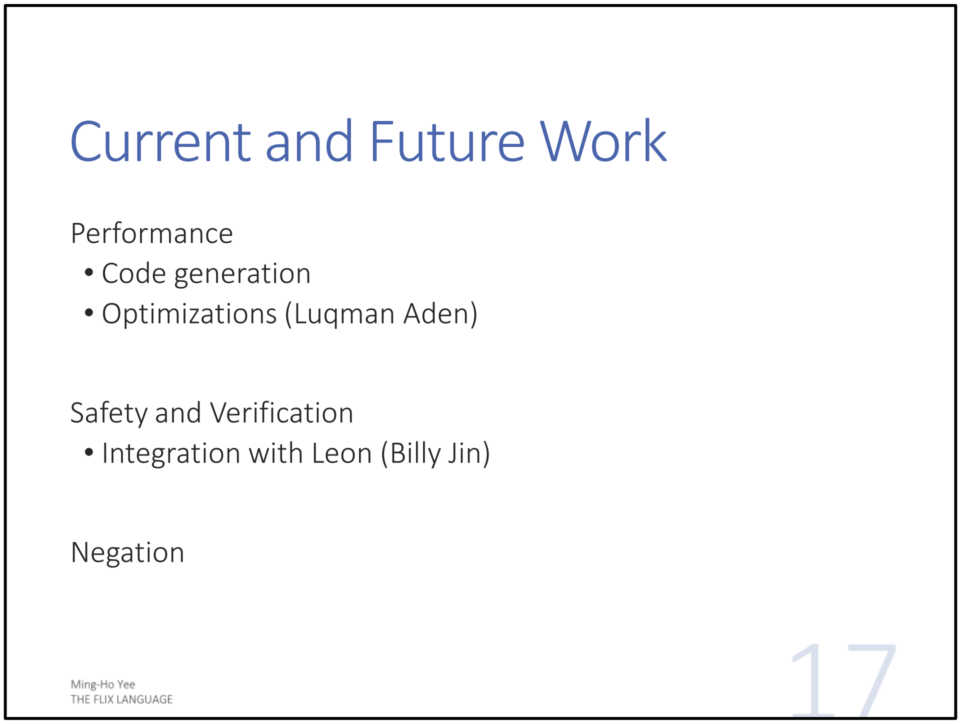
- How many ASTs do we have? At least five.
    - There’s one more AST, but it’s on the next slide because 1) it doesn’t fit here, and 2) it’s in the back-end.
- Because we want our ASTs to be immutable, the front-end generates a new AST after each phase.
    - We also don’t want one giant AST that we repeatedly process.
    - Standard phases: parsing, weeding, name resolution, and type-checking.
    - Final phase creates the SimplifiedAst, which is designed to make code generation easier
        - Currently we desguar pattern matches.
        - Later we’ll be doing more things, such as rewriting and optimizations.



Architecture: Back-End

Ming-Ho Yee
THE FLIX LANGUAGE

- The final phase creates the ExecutableAst, which is consumed by the solver and interpreter.
  - We copy lists over to arrays, to improve run-time performance.
  - The ExecutableAst also keeps track of performance data.
- The solver is where most of the execution takes place.
  - This is where the logic language (rules and constraints) is processed, and new facts are generated.
  - When a function needs to be evaluated, the solver calls the interpreter, which then returns the result.
    - Lattice operations or other user-defined functions (e.g. sum).
- There is also a code generator, that takes functions from the SimplifiedAst and generates JVM bytecode.
  - This is still in progress – eventually we want the solver to call bytecode functions instead of the interpreter.
  - The idea is to keep an interpreter for prototyping/debugging, and a code generator for performance.
- My main responsibility has been the interpreter and code generator.
  - Though sometimes other things come up.



Current and Future Work

Performance

- Code generation
- Optimizations (Luqman Aden)

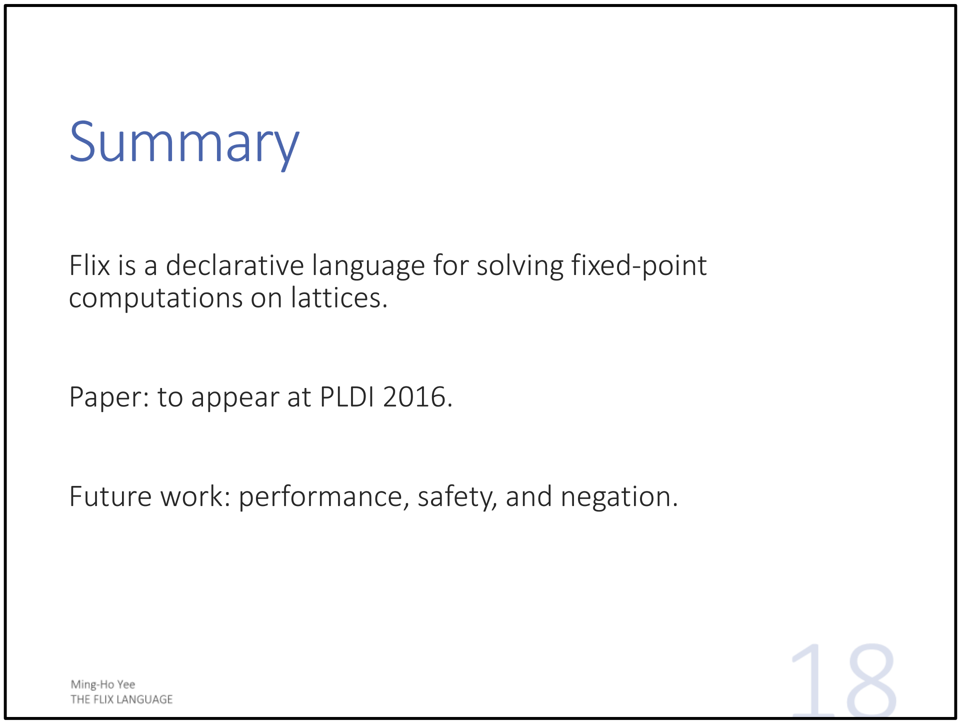
Safety and Verification

- Integration with Leon (Billy Jin)

Negation

Ming-Ho Yee
THE FLIX LANGUAGE

- There’s three main directions we’re working on or interested in.
- Performance will always be worked on
  - No one will use Flix if our performance is terrible
  - Approaches: code generation for the functional language (current WIP), code generation for the logic language, general compiler optimizations
  - Luqman Aden, an undergraduate, will be joining us in the spring term to work on optimizations
- Safety and Verification
  - Every Datalog program terminates and computes the correct answer
  - We want the same guarantee with Flix
    - Flix requires certain properties to hold, e.g. every lattice is actually a lattice
  - We want to automatically verify these properties
  - Billy Jin, an undergraduate, is currently working on integrating Flix with Leon
    - Leon is an automated system for verifying functional Scala programs
    - http://leon.epfl.ch/
- Negation
  - In contrast to the other two, this is more theoretical
  - Pure Datalog doesn’t support negation, but there are some Datalog extensions that do
  - How can we extend Flix to support negation?



Summary

Flix is a declarative language for solving fixed-point computations on lattices.

Paper: to appear at PLDI 2016.

Future work: performance, safety, and negation.