# Week 2

Magnus Madsen

Friday 14th March, 2025 at 15:00

# Week 2: Outline

**Tuesday**

**Lecture** (45min)

- Datalog programs as first-class values in a general-purpose language.
- A row polymorphic type system for Datalog program values.

**Exercises** (45min)

- Work on the assignment alone or together in small groups.

**Thursday**

**Lecture** (45min)

- Datalog program values and rho abstraction.
- Datalog extended with lattice semantics.
- Computing provenance information.

**Exercises** (45min)

- Work on the assignment alone or together in small groups.

# Quote of the Day

"A programming language is low level when its programs require attention to the irrelevant."

— Alan Perlis

# Pull Requests are Welcome

You can improve the course material!

- Exercises are in `src/weekX.md`
- Slides are in `slides/weekX.tex`

PRs can be submitted on GitHub:

`https://github.com/magnus-madsen/advprog/`

# Introduction to Flix

# The Flix Programming Language (1/2)

A **functional**, **imperative**, and **declarative** logic programming language.

- Developed at Aarhus University in collaboration with programming language researchers from Waterloo (Canada), Tübingen (Germany), and Copenhagen.
- My personal research project.

Free, open source, and ready for use:

`https://flix.dev/`

# The Flix Programming Language (2/2)

Flix is an **advanced** programming language with a **unique** combination of **powerful** programming language features:

- algebraic data types and pattern matching
- tuples and extensible records
- parametric polymorphism
- type classes (traits)
- higher-kinded and associated types
- type match and purity reflection
- a polymorphic effect system
- scoped mutable state
- structured concurrency
- channels and processes
- first-class Datalog programs
- local type inference
- full tail call elimination
- and more ...

# First-Class Datalog Programs

# Motivation (1/2)

Given the Datalog facts:

```
ParentOf("Pompey", "Strabo").
ParentOf("Gnaeus", "Pompey").
ParentOf("Pompeia", "Pompey").
ParentOf("Sextus", "Pompey").
```

# Motivation (1/2)

Given the Datalog facts:

```
ParentOf("Pompey", "Strabo").
ParentOf("Gnaeus", "Pompey").
ParentOf("Pompeia", "Pompey").
ParentOf("Sextus", "Pompey").
```

We can compute the ancestor of every person:

```
AncestorOf(x, y) :- ParentOf(x, y).
AncestorOf(x, z) :- AncestorOf(x, y), AncestorOf(y, z).
```

# Motivation (2/2)

Given the additional facts:

```
AdoptedBy("Augustus", "Caesar").
AdoptedBy("Tiberius", "Augustus").
```

We can extend the original program to include adoptions:

```
AncestorOf(x, y) :- AdoptedBy(x, y).
```

# Motivation (2/2)

Given the additional facts:

```
AdoptedBy("Augustus", "Caesar").
AdoptedBy("Tiberius", "Augustus").
```

We can extend the original program to include adoptions:

```
AncestorOf(x, y) :- AdoptedBy(x, y).
```

This example demonstrates the *elegance* of Datalog:

- We can extend the meaning of a program by adding new rules.
- i.e. we have extension by *addition*, not by *modification*.

# The Problem

But now we have ***two*** programs:

- one with biological parents, and
- one with biological parents and adoptions.

# The Problem

But now we have ***two*** programs:

- one with biological parents, and
- one with biological parents and adoptions.

How do we **maintain** and **develop** these programs?

# The Problem

But now we have ***two*** programs:

- one with biological parents, and
- one with biological parents and adoptions.

How do we **maintain** and **develop** these programs?

- With separate copies? $\Rightarrow$ **multiple maintenance problem?**

# The Problem

But now we have ***two*** programs:

- one with biological parents, and
- one with biological parents and adoptions.

How do we **maintain** and **develop** these programs?

- With separate copies? $\Rightarrow$ **multiple maintenance problem?**
- With textual generation? $\Rightarrow$ **correctness? expressive power?**

# The Problem

But now we have ***two*** programs:

- one with biological parents, and
- one with biological parents and adoptions.

How do we **maintain** and **develop** these programs?

- With separate copies? $\Rightarrow$ **multiple maintenance problem?**
- With textual generation? $\Rightarrow$ **correctness? expressive power?**
- With procedural macros? $\Rightarrow$ **correctness? expressive power?**

# The Problem

But now we have ***two*** programs:

- one with biological parents, and
- one with biological parents and adoptions.

How do we **maintain** and **develop** these programs?

- With separate copies? $\Rightarrow$ **multiple maintenance problem?**
- With textual generation? $\Rightarrow$ **correctness? expressive power?**
- With procedural macros? $\Rightarrow$ **correctness? expressive power?**

**Idea:** Datalog programs as a *first-class* values.

# Example (1/2)

We define a function which returns a Datalog program value:

```
def getAncestors(withAdoptions: Bool): #{ ... } =
    let p1 = #{
        AncestorOf(x, y) :- ParentOf(x, y).
        AncestorOf(x, z) :- AncestorOf(x, y), AncestorOf(y, z).
    };
    let p2 = #{
        AncestorOf(x, y) :- AdoptedBy(x, y).
    };
    if (withAdoptions) (p1 <+> p2) else p1
```

If `withAdoptions` is `true` we return the extended program with adoptions. Otherwise we return the original program with only biological parents.

# Example (2/2)

We can use the `getAncestors` as follows:

```
def main(): Unit \ IO =
    let db = #{
      ParentOf("Pompey", "Strabo").
      ParentOf("Gnaeus", "Pompey").
      ParentOf("Pompeia", "Pompey").
      ParentOf("Sextus", "Pompey").
      AdoptedBy("Augustus", "Caesar").
      AdoptedBy("Tiberius", "Augustus").
    };
    let r = query db, getAncestors(true)
            select x from AncestorOf("Tiberius", x);
    println(r)
```

which prints `Vector#{Augustus, Caesar}`.

# First-Class Datalog Programs

We propose the idea of **first-class Datalog programs**:

- A *Datalog program value* is a set of Datalog facts and rules.
- Datalog programs can be passed as arguments, stored in local variables, returned, and composed with other Datalog programs.

# First-Class Datalog Programs

We propose the idea of **first-class Datalog programs**:

- A *Datalog program value* is a set of Datalog facts and rules.
- Datalog programs can be passed as arguments, stored in local variables, returned, and composed with other Datalog programs.

We can **construct**, **compose**, and **query** Datalog programs.

- The solution to Datalog program value is its minimal model.
- The minimal model is itself a Datalog program value.

**Upshot:** We can create pipelines of Datalog programs.

# Fundamental Operations

## Datalog Literals

- A Datalog literal is written with bracket syntax `#{...}`.

# Fundamental Operations

## Datalog Literals

- A Datalog literal is written with bracket syntax `#{...}`.

## **Injection** — Getting facts into Datalog

- The `inject e1, ..., en into A1, ... An` expression returns a Datalog program where each tuple in the collection $e_i$ is associated with predicate symbol $A_i$.

# Fundamental Operations

## Datalog Literals

- A Datalog literal is written with bracket syntax `#{...}`.

## **Injection** — Getting facts into Datalog

- The `inject e1, ..., en into A1, ... An` expression returns a Datalog program where each tuple in the collection $e_i$ is associated with predicate symbol $A_i$.

## Composition

- The `e1 <+> e2` expression combines two Datalog programs $e_1$ and $e_2$.

# Fundamental Operations

## Datalog Literals

- A Datalog literal is written with bracket syntax `#{...}`.

## **Injection** — Getting facts into Datalog

- The `inject e1, ..., en into A1, ... An` expression returns a Datalog program where each tuple in the collection $e_i$ is associated with predicate symbol $A_i$.

## Composition

- The `e1 <+> e2` expression combines two Datalog programs $e_1$ and $e_2$.

## **Solving** — Getting facts out of Datalog

- The `query e1, ..., en select (x1, ..., xm) from A1, ..., A_o` expression computes the minimal model of the expressions $e_1, \cdots, e_n$ and then it selects the variables $x_1, \cdots, x_m$ from the relations $A_1, \cdots, A_o$. The result is a `Vector` of tuples.

# Datalog Literals (1/3)

A Datalog literal is written[1]:

```
#{ ... }
```

[1] The empty Datalog literal `#{ }` is a legal Datalog program value.

# Datalog Literals (1/3)

A Datalog literal is written[1]:

```
#{ ... }
```

A Datalog literal may contain facts:

```
#{ A(1). A(2). A(3). B(42). }
```

[1] The empty Datalog literal `#{ }` is a legal Datalog program value.

# Datalog Literals (1/3)

A Datalog literal is written[1]:

```
#{ ... }
```

A Datalog literal may contain facts:

```
#{ A(1). A(2). A(3). B(42). }
```

A Datalog literal may contain rules:

```
#{ A(x) :- B(x), C(x). }
```

[1]The empty Datalog literal `#{ }` is a legal Datalog program value.

# Datalog Literals (2/3)

A Datalog literal may contain both facts and rules:

```
#{ A(1).
   A(2).
   B(1).
   C(x) :- A(x), B(x). }
```

A Datalog program is *inert* until its minimal model is evaluated with `query`.

- i.e. in the above Datalog literal the fact `C(1)` is *not* automatically derived.

# Datalog Literals (3/3)

Datalog program values are first-class:

- We can store them in local variables.
- We can pass them as arguments to functions.
- We can return them from functions.
- We can store them inside data structures (e.g. in lists, maps).

# Datalog Literals (3/3)

Datalog program values are first-class:

- We can store them in local variables.
- We can pass them as arguments to functions.
- We can return them from functions.
- We can store them inside data structures (e.g. in lists, maps).

Datalog program values do *not* implement any traits.

- In particular they do *not* implement `Eq[t]` nor `Order[t]`.
- Hence, we can only manipulate them using `query`.

## Values as Terms

Primitive values are permitted as terms:

```
#{ A(1, 2, 3). };                  // OK
#{ A("Hello"). };                  // OK
```

# Values as Terms

Primitive values are permitted as terms:

```
#{ A(1, 2, 3). };              // OK
#{ A("Hello"). };              // OK
```

Compound values are also permitted as terms:

```
#{ A((1, 1), (2, 2)). };       // OK
#{ A(Set#{1, 2, 3}). };        // OK
```

Any type which implements `Eq[t]` and `Order[t]` can be used as a term.

# Values as Terms

Primitive values are permitted as terms:

```
#{ A(1, 2, 3). };              // OK
#{ A("Hello"). };              // OK
```

Compound values are also permitted as terms:

```
#{ A((1, 1), (2, 2)). };       // OK
#{ A(Set#{1, 2, 3}). };        // OK
```

Any type which implements `Eq[t]` and `Order[t]` can be used as a term.

**Question:** What types are then excluded?

# Lexical Scope (1/2)

Datalog literals integrate with lexical scope.

For example, we can capture variables from lexical scope:

```
def mkParentOf(c: String, p: String): #{ ... } =
  #{ ParentOf(c, p). }
```

Here *c* and *p* are Flix program variables, *not* Datalog variables.

# Lexical Scope (1/2)

Datalog literals integrate with lexical scope.

For example, we can capture variables from lexical scope:

```
def mkParentOf(c: String, p: String): #{ ... } =
  #{ ParentOf(c, p). }
```

Here *c* and *p* are Flix program variables, *not* Datalog variables.

We can use `mkParentOf` to write:

```
mkParentOf("Pompey", "Strabo") <+> mkParentOf("Sextus", "Pompey")
```

to construct a Datalog program with two `ParentOf` facts in it.

# Lexical Scope (2/2)

We can take this idea further and write a function to convert a list of pairs into a Datalog program value with `ParentOf` facts:

```
def mkParentOf(l: List[(String, String)]): #{ ... } =
    match l {
        case Nil           => #{}
        case (c, p) :: xs =>
            #{ ParentOf(c, p). } <+> mkParentOf(xs)
    }
```

# Lexical Scope (2/2)

We can take this idea further and write a function to convert a list of pairs into a Datalog program value with `ParentOf` facts:

```
def mkParentOf(l: List[(String, String)]): #{ ... } =
    match l {
        case Nil           => #{}
        case (c, p) :: xs =>
            #{ ParentOf(c, p). } <+> mkParentOf(xs)
    }
```

This works, but ...

**Problem:** Writing such functions for every data type can get tedious.

# Injecting Facts (1/4)

We have an **impedance mismatch** between functional programming and Datalog:

- Functional languages uses data structures: lists, sets, and maps.
- Datalog uses relations, i.e. sets of facts.

# Injecting Facts (1/4)

We have an **impedance mismatch** between functional programming and Datalog:

- Functional languages uses data structures: lists, sets, and maps.
- Datalog uses relations, i.e. sets of facts.

How can we reconcile the two?

- We need a mechanism to translate between data structures and relations.

We introduce the `inject` construct as mechanism to associate a collection with a predicate symbol and to translate it into a Datalog representation.

# Injecting Facts (2/4)

For example, we can translate a list of tuples:

```
let edges = (1, 2) :: (2, 3) :: (3, 3) :: Nil
```

into a Datalog relation, i.e. a set of facts, using `inject`:

```
inject edges into Edge
```

which evaluates to the Datalog program value:

```
#{ Edge(1, 2). Edge(2, 3). Edge(3, 3). }
```

# Injecting Facts (3/4)

We can use `inject` to translate multiple heterogeneous collections into relations.

For example,

```
let nodes = Set#{1, 2, 3, 4};
let edges = (1, 2) :: (2, 3) :: (3, 3) :: Nil
inject nodes, edges into Node, Edge
```

evaluates to the Datalog program value:

```
#{ Node(1). Node(2). Node(3). Node(4).
   Edge(1, 2). Edge(2, 3). Edge(3, 3). }
```

# Injecting Facts (4/4)

The general form of `inject` is:

```
inject exp_1, exp_2, ... exp_n into sym_1, sym_2, ..., sym_n
```

The `inject` construct works for any collection that implements `Foldable[t]`.

- E.g. `List[t]`, `Set[t]` and `Map[k, v]`, and many more...

# Injecting Facts (4/4)

The general form of `inject` is:

```
inject exp_1, exp_2, ... exp_n into sym_1, sym_2, ..., sym_n
```

The `inject` construct works for any collection that implements `Foldable[t]`.

- E.g. `List[t]`, `Set[t]` and `Map[k, v]`, and many more...

**Upshot:** `Foldable[t]` can be implemented for user-defined data types, hence `inject` builds upon an extensible foundation.

# Composition

We have already seen that we can compose Datalog programs with:

```
s1 <+> s2
```

which evaluates to the union of the constraints in both $s_1$ and $s_2$.

Composition is a well-behaved operation since the order of constraints in a Datalog program value is immaterial.

Composition is a low-level operation and we rarely use it directly.

# Querying the Minimal Model (1/3)

Given a Datalog program value:

```
let p = #{ A(1). A(2). B(x) :- A(x). }
```

We can compute its minimal model with `query` and extract all its `B` facts:

```
query p select x from B(x)
```

which evaluates to the the vector:

```
Vector#{ 1, 2 }
```

# Querying the Minimal Model (2/3)

Given two Datalog program values:

```
let db = #{ A(1). A(2).   }
let pr = #{ B(x) :- A(x). }
```

We can use query to compose them and compute their minimal model:

```
query db, pr select x from B(x)
```

which, as before, evaluates to:

```
Vector#{ 1, 2 }
```

# Querying the Minimal Model (3/3)

We can use `query` for more complex queries.

For example, given:

```
let p = #{ A(1). A(2). A(3), B(1, 2). }
```

We can write the more interesting query:

```
query p select (x, y + 1) from A(x), A(y), B(x, y) where x > 0
```

which evaluates to the vector:

```
Vector#{ (1, 3) }
```

# Inject and Query

We have seen how `inject` and `query` bridge the gap between Datalog and Flix:

- We can use `inject` to translate any data type, which implements the `Foldable` trait, into a set of Datalog facts, and
- We can use `query` to compute the minimal model of a collection of Datalog program values, and to extract tuples as an immutable `Vector`.

**Upshot:** We can easily transport data into and out of the Datalog world.

# Example I

What does the following program print?

```
def main(): Unit \ IO =
    let p1 = #{ Edge(1, 2). Edge(2, 3). };
    let p2 = #{
        Edge(y, x) :- Edge(x, y).
    };
    let p3 = #{
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
    };
    let result = query p1, p2, p3 select (a, b) from Edge(a, b);
    println(result)
```

# Example II: Trick Question

What does the following program print?

```
def main(): Unit \ IO =
    let x = #{ Leg("BLL", "LH", "FRA"). Leg("FRA", "LH", "YYZ").
               Leg("YYZ", "AC", "YVR"). Leg("YYZ", "AC", "SFO"). };
    let y = #{
        Route(x, a, y) :- Leg(x, a, y).
    };
    let z = #{
        Route(x, a, z) :- Route(x, a, y), Leg(y, a, z).
    };
    let result = query x, z select (src, dst) from Leg(src, dst);
    println(result)
```

# A Type System for First-class Datalog

# Why a Type System?

The Flix type system gives us three important properties:

- (**Safety**) Well-typed programs cannot go wrong.
- (**Synthesis**) Automatic resolution and derivation of code via traits.
- (**IDE Support**) Auto-complete, automatic refactoring, etc.

**Footnote:** Flix also has an *effect system* which enables enforcement of purity.

# What could possibly go wrong? (1/3)

Workers shovel raw blue asbestos tailings into drums at an asbestos shovelling competition at Wittenoom, in the Pilbara, WA, in 1962.

# What could possibly go wrong? (2/3)

We want to ensure that programmers do not confuse **term types**:

```
let p1 = #{ Edge(1, 2). };
let p2 = #{ Edge("Aarhus", "Copenhagen"). };
p1 <+> p2
```

# What could possibly go wrong? (2/3)

We want to ensure that programmers do not confuse **term types**:

```
let p1 = #{ Edge(1, 2). };
let p2 = #{ Edge("Aarhus", "Copenhagen"). };
p1 <+> p2
```

If we try to compile this program, Flix reports:

```
>> Unable to unify the types: 'Int32' and 'String'.

3 |      p1 <+> p2
         ^^^^^^^^^
         mismatched types.
```

# What could possibly go wrong? (3/3)

We also want to ensure that programmers do not confuse **predicate arity**:

```
let p1 = #{ Edge(1, 2). };
let p2 = #{ Edge(1, 2, 3). };
p1 <+> p2
```

# What could possibly go wrong? (3/3)

We also want to ensure that programmers do not confuse **predicate arity**:

```
let p1 = #{ Edge(1, 2). };
let p2 = #{ Edge(1, 2, 3). };
p1 <+> p2
```

If we try to compile this program, Flix reports:

```
>> Unable to unify the types: '(?, ?)' and '(Int32, ?, ?)'.

3 |     p1 <+> p2
        ^^^^^^^^^
        mismatched types.
```

# Polymorphic Type Systems

You are probably already familiar with two types of polymorphism:

- **Subtype polymorphism** — “inheritance”
- **Parametric polymorphism** — “generics”

# Polymorphic Type Systems

You are probably already familiar with two types of polymorphism:

- **Subtype polymorphism** — "inheritance"
- **Parametric polymorphism** — "generics"

Flix uses another kind of polymorphism to type Datalog programs:

- **Row polymorphism**

# Row Types

A row type is of the form:

$$\rho = \alpha \mid \epsilon \mid \{p(\tau_1, \cdots, \tau_n) \mid \rho\}$$

where $\tau$ is a collection of base types (e.g. `Bool`, `Int32`, `String`).

# Row Types

A row type is of the form:

$$\rho = \alpha \mid \epsilon \mid \{p(\tau_1, \cdots, \tau_n) \mid \rho\}$$

where $\tau$ is a collection of base types (e.g. `Bool`, `Int32`, `String`).

We consider rows equivalent up to associativity and commutativity.

**Note:** The Flix type system ensures that a predicate symbol $p$ can occur at most once in a row.

# Example (1/3)

The Datalog program:

```
#{ A(1, 2). B("Hello"). }
```

has the type:

$$\forall \alpha. \{A(\mathsf{Int32}, \mathsf{Int32}) \mid \{B(\mathsf{String}) \mid \alpha\}\}$$

# Example (1/3)

The Datalog program:

```
#{ A(1, 2). B("Hello"). }
```

has the type:

$$\forall\alpha.\,\{A(\mathsf{Int32}, \mathsf{Int32}) \mid \{B(\mathsf{String}) \mid \alpha\}\}$$

but it also has the *equivalent* type:

$$\forall\alpha.\,\{B(\mathsf{String}) \mid \{A(\mathsf{Int32}, \mathsf{Int32}) \mid \alpha\}\}$$

# Example (1/3)

The Datalog program:

```
#{ A(1, 2). B("Hello"). }
```

has the type:

$$\forall\alpha.\, \{A(\mathsf{Int32}, \mathsf{Int32}) \mid \{B(\mathsf{String}) \mid \alpha\}\}$$

but it also has the *equivalent* type:

$$\forall\alpha.\, \{B(\mathsf{String}) \mid \{A(\mathsf{Int32}, \mathsf{Int32}) \mid \alpha\}\}$$

and more interestingly it also has the *less general* type:

$$\forall\alpha.\, \{A(\mathsf{Int32}, \mathsf{Int32}) \mid \{B(\mathsf{String}) \mid \{C(\mathsf{Bool}) \mid \alpha\}\}\}$$

# Example (2/3)

The Datalog program:

```
#{ Path(x, y) :- Edge(x, y). }
```

has the type:

$$\forall a, b, \alpha. \{\text{Edge}(a, b) \mid \{\text{Path}(a, b) \mid \alpha\}\}$$

# Example (2/3)

The Datalog program:

```
#{ Path(x, y) :- Edge(x, y). }
```

has the type:

$$\forall a, b, \alpha. \{\mathrm{Edge}(a, b) \mid \{\mathrm{Path}(a, b) \mid \alpha\}\}$$

whereas the Datalog program:

```
#{ Path(x, z) :- Path(x, y), Edge(y, z) }
```

# Example (2/3)

The Datalog program:

```
#{ Path(x, y) :- Edge(x, y). }
```

has the type:

$$\forall a, b, \alpha. \{\text{Edge}(a, b) \mid \{\text{Path}(a, b) \mid \alpha\}\}$$

whereas the Datalog program:

```
#{ Path(x, z) :- Path(x, y), Edge(y, z) }
```

has the type:

$$\forall a, b, \alpha. \{\text{Edge}(\underline{b}, b) \mid \{\text{Path}(a, b) \mid \alpha\}\}$$

# Example (3/3)

The two Datalog programs

```
let p1 = #{ A(1). A(2). A(3). };
let p2 = #{ B(1). B(2). B(3). };
```

have the types:

$$\forall \alpha_1. \{A(\mathsf{Int32}) \mid \alpha_1\} \qquad \text{and} \qquad \forall \alpha_2. \{B(\mathsf{Int32}) \mid \alpha_2\}$$

# Example (3/3)

The two Datalog programs

```
let p1 = #{ A(1). A(2). A(3). };
let p2 = #{ B(1). B(2). B(3). };
```

have the types:

$$\forall \alpha_1. \{A(\mathsf{Int32}) \mid \alpha_1\} \qquad \text{and} \qquad \forall \alpha_2. \{B(\mathsf{Int32}) \mid \alpha_2\}$$

Hence the composition `p1 <+> p2` has the type:

$$\forall \alpha_3. \{A(\mathsf{Int32}) \mid \{B(\mathsf{Int32}) \mid \alpha_3\}\}$$

# Pitfall (1/2)

The following does not work:

```
def f(): #{ Edge(Int32, Int32) } = #{ Edge(1, 2). }
def g(): #{ Path(Int32, Int32) } = #{ Path(2, 3). }
def h(): #{ Edge(Int32, Int32), Path(Int32, Int32) } =
    f() <+> g()
```

# Pitfall (1/2)

The following does not work:

```
def f(): #{ Edge(Int32, Int32) } = #{ Edge(1, 2). }
def g(): #{ Path(Int32, Int32) } = #{ Path(2, 3). }
def h(): #{ Edge(Int32, Int32), Path(Int32, Int32) } =
    f() <+> g()
```

Specifically, the Flix compiler reports:

```
>> Missing predicate 'Edge' of type 'Relation(Int32, Int32)'.

4 |     f() <+> g()
        ^^^^^^^^^^^
        missing predicate.
```

# Pitfall (2/2)

We we should have done is to use **open rows**:

```
def f(): #{ Edge(Int32, Int32) | r } = #{ Edge(1, 2). }
def g(): #{ Path(Int32, Int32) | r } = #{ Path(2, 3). }
def h(): #{ Edge(Int32, Int32), Path(Int32, Int32) | r} =
    f() <+> g()
```

Here, because each row is open, we can build bigger rows.

# Summary

**Beyond Datalog**: Datalog programs as first-class values in Flix:

- Datalog programs are *values*. We can pass them around.
- Datalog literals may capture variables from the lexical scope.
- Use `inject` to translate data structures to Datalog facts.
- Use `query` to compute minimal models and to extract facts.
- A row polymorphic type system ensures safety.

**Upshot:** We can create modular and reusable families of Datalog programs.

# Week 2: Outline

**Tuesday**

**Lecture** (45min)

- Datalog programs as first-class values in a general-purpose language.
- A row polymorphic type system for Datalog program values.

**Exercises** (45min)

- Work on the assignment alone or together in small groups.

**Thursday**

**Lecture** (45min)

- Datalog program values and rho abstraction.
- Datalog extended with lattice semantics.
- Computing provenance information.

**Exercises** (45min)

- Work on the assignment alone or together in small groups.

# Quote of the Day

"Every program is a part of some other program and rarely fits."

— Alan Perlis

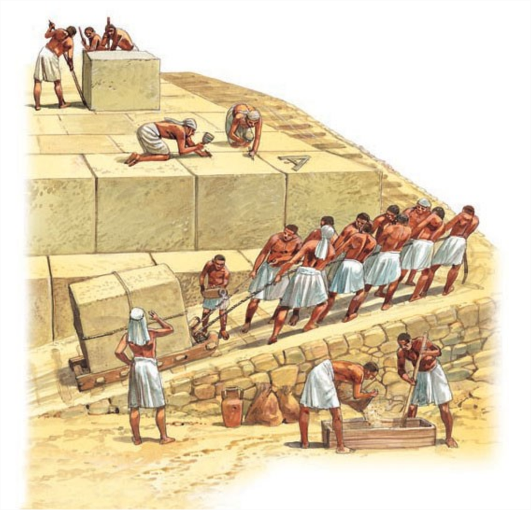
# Pull Requests are Welcome

You can improve the course material!

- Exercises are in `src/weekX.md`
- Slides are in `slides/weekX.tex`

PRs can be submitted on GitHub:

`https://github.com/magnus-madsen/advprog/`



# Rho Abstraction

# Motivation (1/2)

We have seen how Datalog programs can be typed with row types:

```
def reach(): #{ Edge(t, t), Path(t, t) | r} = #{
    Path(x, y) :- Edge(x, y).
    Path(x, z) :- Path(x, y), Edge(y, z).
}
```

# Motivation (2/2)

But such types can quickly become unwieldy:

```
def disconnected():
    #{ Edge(t, t), Path(t, t), Vertex(t), Disconnected(t, t) | r} = #{
        Vertex(x)  :- Edge(x, _).
        Vertex(y)  :- Edge(_, y).
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
        Disconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
    }
```

# Motivation (2/2)

But such types can quickly become unwieldy:

```
def disconnected():
    #{ Edge(t, t), Path(t, t), Vertex(t), Disconnected(t, t) | r} = #{
        Vertex(x)  :- Edge(x, _).
        Vertex(y)  :- Edge(_, y).
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
        Disconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
    }
```

**Observation:** The `Vertex` and `Path` relations are really internally implementations.

# Motivation (2/2)

But such types can quickly become unwieldy:

```
def disconnected():
    #{ Edge(t, t), Path(t, t), Vertex(t), Disconnected(t, t) | r} = #{
        Vertex(x)  :- Edge(x, _).
        Vertex(y)  :- Edge(_, y).
        Path(x, y) :- Edge(x, y).
        Path(x, z) :- Path(x, y), Edge(y, z).
        Disconnected(x, y) :- Vertex(x), Vertex(y), not Path(x, y).
    }
```

**Observation:** The `Vertex` and `Path` relations are really internally implementations.

**Idea:** What do we usually do with internal implementation details? ***We hide them.***

# Rho Abstraction

We introduce *rho abstraction* as a mechanism to *hide* predicate symbols.

- A rho abstraction is of the form `#(A, ...) -> e` where *e* must be a Datalog expression.

# Rho Abstraction

We introduce *rho abstraction* as a mechanism to *hide* predicate symbols.

- A rho abstraction is of the form `#(A, ...) -> e` where *e* must be a Datalog expression.
- A rho abstraction hides, by renaming, all predicate symbols *not* listed in the argument list.

# Rho Abstraction

We introduce *rho abstraction* as a mechanism to *hide* predicate symbols.

- A rho abstraction is of the form `#(A, ...) -> e` where *e* must be a Datalog expression.
- A rho abstraction hides, by renaming, all predicate symbols *not* listed in the argument list.
- The row type of a rho abstraction includes only those predicates in the argument list.

# Rho Abstraction

We introduce *rho abstraction* as a mechanism to *hide* predicate symbols.

- A rho abstraction is of the form `#(A, ...) -> e` where *e* must be a Datalog expression.
- A rho abstraction hides, by renaming, all predicate symbols *not* listed in the argument list.
- The row type of a rho abstraction includes only those predicates in the argument list.
- Evaluation of a rho abstraction renames all hidden predicate symbols with fresh names.

# Rho Abstraction

We introduce *rho abstraction* as a mechanism to *hide* predicate symbols.

- A rho abstraction is of the form `#(A, ...) -> e` where *e* must be a Datalog expression.
- A rho abstraction hides, by renaming, all predicate symbols *not* listed in the argument list.
- The row type of a rho abstraction includes only those predicates in the argument list.
- Evaluation of a rho abstraction renames all hidden predicate symbols with fresh names.

**Example:** `#(A, B) -> #{A(123). C("a").}` evaluates to `#{A(123). C17("a").}`. where `C17` is a fresh predicate symbol that has never been used before.

# Rho Abstraction: The Wrong Way

We may think that we can statically rename abstracted predicate symbols:

```
def disconnected(): #{ Edge(t, t), Disconnected(t, t) | r} =
    #(Edge, Disconnected) -> #{
        Vertex17(x) :- Edge(x, _).
        Vertex17(y) :- Edge(_, y).
        // ... omitted for brevity ...
    }
```

# Rho Abstraction: The Wrong Way

We may think that we can statically rename abstracted predicate symbols:

```
def disconnected(): #{ Edge(t, t), Disconnected(t, t) | r} =
    #(Edge, Disconnected) -> #{
        Vertex17(x) :- Edge(x, _).
        Vertex17(y) :- Edge(_, y).
        // ... omitted for brevity ...
    }
```

**But this does not work.** Why?

# Rho Abstraction: The Wrong Way

We may think that we can statically rename abstracted predicate symbols:

```
def disconnected(): #{ Edge(t, t), Disconnected(t, t) | r} =
    #(Edge, Disconnected) -> #{
        Vertex17(x) :- Edge(x, _).
        Vertex17(y) :- Edge(_, y).
        // ... omitted for brevity ...
    }
```

**But this does not work.** Why?

```
let p1 = #(A) -> (#{ Edge(123, 456). } <+> disconnected());
let p2 = #(A) -> (#{ Edge("a", "b"). } <+> disconnected());
query p1, p2 ...
```

**Oops.** Now the Datalog program contains the facts `Vertex17(123)` and `Vertex17("a")` – which is a type error! We must rename predicates at *runtime* to ensure fresh names!

# Rho Abstraction: The Right Way

Each evaluation of a rho abstraction introduces fresh names.

Hence, in the previous example, we get:

```
let p1 = #{ Vertex17(123). Vertex17(456). ... };
let p2 = #{ Vertex18("a"). Vertex18("b"). ... };
query p1, p2 ...
```

where there is no longer any type error between `Vertex17(123)` and `Vertex18("a")`.

# Rho Abstraction: The Right Way

Each evaluation of a rho abstraction introduces fresh names.

Hence, in the previous example, we get:

```
let p1 = #{ Vertex17(123). Vertex17(456). ... };
let p2 = #{ Vertex18("a"). Vertex18("b"). ... };
query p1, p2 ...
```

where there is no longer any type error between `Vertex17(123)` and `Vertex18("a")`.

**Upshot:** The abstracted predicate symbols have become truely local.

# Datalog and Lattice Semantics

# Motivation (1/4)

We know how to compute reachability in a graph:

```
def reach(origin: t, edges: List[(t, t)]): Vector[t] with Order[t] =
    let db = inject edges into Edge;
    let pr = #{
        Reach(origin).
        Reach(y) :- Reach(x), Edge(x, y).
    };
    query db, pr select x from Reach(x)
```

# Motivation (1/4)

We know how to compute reachability in a graph:

```
def reach(origin: t, edges: List[(t, t)]): Vector[t] with Order[t] =
    let db = inject edges into Edge;
    let pr = #{
        Reach(origin).
        Reach(y) :- Reach(x), Edge(x, y).
    };
    query db, pr select x from Reach(x)
```

But, what if we wanted to compute the **shortest distance** to every vertex from an origin, i.e. *single-source shortest distance* (SSSD)?

# Motivation (2/4)

We can use *lattice semantics* to solve this problem:

```
def sssd(origin: t, edges: List[(t, Int32, t)]): ... =
    let db = inject edges into Edge;
    let pr = #{
        Dist(origin; Down(0)).
        Dist(y; d1 + Down(d2)) :- Dist(x; d1), Edge(x, d2, y).
    };
    query db, pr select (x, d) from Dist(x; d) |> Vector.toMap

def main(): Unit \ IO =
    println(sssd("a", List#{("a", 2, "b"), ("b", 5, "c")}))
```

Prints `Map#{a => 0, b => 2, c => 7}`.

# Motivation (3/4)

A lot is going on, so let us break it down.

The fact: `Dist(origin; Down(0)).`

# Motivation (3/4)

A lot is going on, so let us break it down.

The fact: `Dist(origin; Down(0)).`

- Asserts that `Dist` is a (map) lattice, and *not* a relation (the semicolon).

# Motivation (3/4)

A lot is going on, so let us break it down.

The fact: `Dist(origin; Down(0)).`

- Asserts that `Dist` is a (map) lattice, and *not* a relation (the semicolon).
- Asserts that the distance to the origin is at most zero.

# Motivation (3/4)

A lot is going on, so let us break it down.

The fact: `Dist(origin; Down(0)).`

- Asserts that `Dist` is a (map) lattice, and *not* a relation (the semicolon).
- Asserts that the distance to the origin is at most zero.
- The `Down` data type, which wraps zero, *reverses* the order on `Int32`.

# Motivation (4/4)

The rule: `Dist(y; d1 + Down(d2)) :- Dist(x; d1), Edge(x, d2, y).`

[2]Technically, it asserts that the distance is *at least*, but since the lattice order is reversed, *at least* becomes *at most*.

# Motivation (4/4)

The rule: `Dist(y; d1 + Down(d2)) :- Dist(x; d1), Edge(x, d2, y).`

- Asserts that `Dist` is a (map) lattice, and *not* a relation (as before).

[2] Technically, it asserts that the distance is *at least*, but since the lattice order is reversed, *at least* becomes *at most*.

# Motivation (4/4)

The rule: `Dist(y; d1 + Down(d2)) :- Dist(x; d1), Edge(x, d2, y).`

- Asserts that `Dist` is a (map) lattice, and *not* a relation (as before).
- Asserts that the distance to `y` is *at most*[2] `d1 + Down(d2)` *if* the distance to `x` is `d1` and the distance on the edge from `x` to `y` is `d2`.

[2]Technically, it asserts that the distance is *at least*, but since the lattice order is reversed, *at least* becomes *at most*.

# Motivation (4/4)

The rule: `Dist(y; d1 + Down(d2)) :- Dist(x; d1), Edge(x, d2, y).`

- Asserts that `Dist` is a (map) lattice, and *not* a relation (as before).
- Asserts that the distance to `y` is *at most*[2] `d1 + Down(d2)` *if* the distance to `x` is `d1` and the distance on the edge from `x` to `y` is `d2`.

What if there are two paths leading to `y` but with different distances? In that case, we compute their *join* which, according the reversed lattice order, is the minimum of the two distances– exactly what we want.

[2]Technically, it asserts that the distance is *at least*, but since the lattice order is reversed, *at least* becomes *at most*.

# From Relations to Lattices

We have seen that Flix supports *constraints on relations*.

- But now also *constraints on lattices*.

We use the semicolon ; to indicate when we want lattice semantics.

# From Relations to Lattices

We have seen that Flix supports *constraints on relations*.

- But now also *constraints on lattices*.

We use the semicolon `;` to indicate when we want lattice semantics.

A lattice has the following components:

- Least and Greatest Elements (`LowerBound` and `UpperBound`).
- A partial order (`PartialOrder`).
- A least upper bound for any two elements (`JoinLattice`).
- A greatest lower bound for any two elements (`MeetLattice`).

which we define by implementing instances for the traits in parenthesis.

# Joins and Meets

Given the two facts:

```
A(1; Neg).
B(1; Pos).
```

The program:

```
P(x; l) :- A(x; l).
P(x; l) :- B(x; l).
Q(x; l) :- A(x; l), B(x; l).
```

# Joins and Meets

Given the two facts:

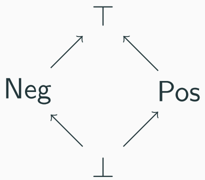
```
A(1; Neg).
B(1; Pos).
```

The program:

```
P(x; l) :- A(x; l).
P(x; l) :- B(x; l).
Q(x; l) :- A(x; l), B(x; l).
```

Evaluates to a minimal model with:

```
P(1; Top).
```



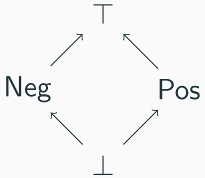
# Joins and Meets

Given the two facts:

```
A(1; Neg).
B(1; Pos).
```

The program:

```
P(x; l) :- A(x; l).
P(x; l) :- B(x; l).
Q(x; l) :- A(x; l), B(x; l).
```




Evaluates to a minimal model with:

```
P(1; Top).
```

**Warning:** Do not mistake , for ;. We must use ; when we want lattice semantics.

# The `Down` Lattice (1/2)

The `Down` data type is defined as:

```
pub enum Down[a] {
    case Down(a)
}
```

It defines instances for the traits `PartialOrder`, `LowerBound`, `UpperBound`, `JoinLattice`, and `MeetLattice` under the *reverse* order on the underlying type `a`.

# The Down Lattice (2/2)

For example, here are two instances:

```
instance PartialOrder[Down[a]] with PartialOrder[a] {
    pub def lessEqual(x: Down[a], y: Down[a]): Bool =
        match (x, y) {
            case (Down.Down(xx), Down.Down(yy)) =>
                yy `PartialOrder.lessEqual` xx
    }
}

instance JoinLattice[Down[a]] with MeetLattice[a] {
    pub def leastUpperBound(x: Down[a], y: Down[a]): Down[a] =
        match (x, y) {
            case (Down.Down(xx), Down.Down(yy)) =>
                Down.Down(xx `MeetLattice.greatestLowerBound` yy)
    }
}
```

# Relation and Lattice Semantics

We can combine relational and lattice semantics with a new form of stratification:

```
Degree("Kevin Bacon"; Down(0)).
Degree(x; n + Down(1)) :- Degree(y; n), StarsWith(y, x).
Layer(n; Set#{ x })     :- fix Degree(x; n).
Count(n, Set.size(s))   :- fix Layer(n; s)
```

# Relation and Lattice Semantics

We can combine relational and lattice semantics with a new form of stratification:

```
Degree("Kevin Bacon"; Down(0)).
Degree(x; n + Down(1)) :- Degree(y; n), StarsWith(y, x).
Layer(n; Set#{ x })     :- fix Degree(x; n).
Count(n, Set.size(s))  :- fix Layer(n; s)
```

This Datalog program computes how many actors are separated from Kevin Bacon by $1, 2, 3, \cdots$ degrees.

# Relation and Lattice Semantics

We can combine relational and lattice semantics with a new form of stratification:

```
Degree("Kevin Bacon"; Down(0)).
Degree(x; n + Down(1)) :- Degree(y; n), StarsWith(y, x).
Layer(n; Set#{ x })     :- fix Degree(x; n).
Count(n, Set.size(s))  :- fix Layer(n; s)
```

This Datalog program computes how many actors are separated from Kevin Bacon by $1, 2, 3, \cdots$ degrees.

Importantly, the use of `fix` enforces that `Degree` is computed before `Layer` which is computed before `Count`.

# Computing Provenance

# Motivation

We have seen that we can compute shortest distances with lattice semantics:

```
def sssd(origin: t, edges: List[(t, Int32, t)]): Map[t, Down[Int32]]
```

but what if we wanted to compute the *shortest path* itself?

# Shortest Paths: Attempt I

What if we try:

```
Reach(origin, Nil; Down(0)).
Reach(y, y :: p; d1 + Down(d2)) :- Reach(x, p; d1), Edge(x, d2, y).
```

**Question:** What does this compute?

## Shortest Paths: Attempt I

What if we try:

```
Reach(origin, Nil; Down(0)).
Reach(y, y :: p; d1 + Down(d2)) :- Reach(x, p; d1), Edge(x, d2, y).
```

**Question:** What does this compute?

**Oops:** What if there are cycles?

## Shortest Paths: Attempt II

We need a new idea (ignoring distances for the moment).

We define a lattice on *paths*:

- The bottom element is the set of all infinite paths.

# Shortest Paths: Attempt II

We need a new idea (ignoring distances for the moment).

We define a lattice on *paths*:

- The bottom element is the set of all infinite paths.
- The top element is the empty path.

# Shortest Paths: Attempt II

We need a new idea (ignoring distances for the moment).

We define a lattice on *paths*:

- The bottom element is the set of all infinite paths.
- The top element is the empty path.
- A path is smaller than another path if it is *longer*, i.e. as we move *up* the lattice, paths get shorter.

# Shortest Paths: Attempt II

We need a new idea (ignoring distances for the moment).

We define a lattice on *paths*:

- The bottom element is the set of all infinite paths.
- The top element is the empty path.
- A path is smaller than another path if it is *longer*, i.e. as we move *up* the lattice, paths get shorter.

```
Reach(origin; P(Nil)).
Reach(y; cons(y, p)) :- Reach(x; p), Edge(x, y).
```

where

```
enum P { case P(List[Int32]) }
```

# Shortest Paths: Attempt II

We define the `PartialOrder` and `JoinLattice` instances as:

```
instance PartialOrder[P] {
    pub def lessEqual(x: P, y: P): Bool =
        let (P(xs), P(ys)) = (x, y);
        List.length(xs) >= List.length(ys)
}
instance JoinLattice[P] {
    pub def leastUpperBound(x: P, y: P): P =
        let (P(xs), P(ys)) = (x, y);
        if (List.length(xs) <= List.length(ys)) x else y
}
```

# Shortest Paths: Attempt II

We define the `PartialOrder` and `JoinLattice` instances as:

```
instance PartialOrder[P] {
    pub def lessEqual(x: P, y: P): Bool =
        let (P(xs), P(ys)) = (x, y);
        List.length(xs) >= List.length(ys)
}
instance JoinLattice[P] {
    pub def leastUpperBound(x: P, y: P): P =
        let (P(xs), P(ys)) = (x, y);
        if (List.length(xs) <= List.length(ys)) x else y
}
```

**Question:** Do you see any problems here?

## Shortest Paths: Attempt II

We define the `PartialOrder` and `JoinLattice` instances as:

```
instance PartialOrder[P] {
    pub def lessEqual(x: P, y: P): Bool =
        let (P(xs), P(ys)) = (x, y);
        List.length(xs) >= List.length(ys)
}
instance JoinLattice[P] {
    pub def leastUpperBound(x: P, y: P): P =
        let (P(xs), P(ys)) = (x, y);
        if (List.length(xs) <= List.length(ys)) x else y
}
```

**Question:** Do you see any problems here?

**Answer:** Comparing the two paths or computing their join is stupidly expensive!

# Shortest Paths: Attempt III

**Idea:**

- We modify the lattice to track the path length *implicitly*.
- We introduce an explicit bottom element:

```
enum P {
    case P(Int32, List[Int32])
    case Bottom
}
```

# Shortest Paths: Attempt III

**Idea:**

- We modify the lattice to track the path length *implicitly*.
- We introduce an explicit bottom element:

```
enum P {
    case P(Int32, List[Int32])
    case Bottom
}
```

**Exercise:** Add instances for `PartialOrder`, `JoinLattice`, etc. for `P`.

# Example: Strongly Connected Components (1/2)

**Problem:** We are given an undirected graph and we want to compute the SCCs.

```
// `Reachable` is simply bi-directional reachability.
Reachable(n, n)   :- Node(n).
Reachable(n1, n2) :- Edge(n1, n2).
Reachable(n1, n2) :- Edge(n2, n1).
Reachable(n1, n2) :- Reachable(n1, m), Reachable(m, n2).
```

# Example: Strongly Connected Components (2/2)

```
// `ReachUp` contains nodes that can reach at least one other node
// with a higher value. This contains all nodes that are not the
// maximum node of their component.
ReachUp(n1) :- Reachable(n1, n2), if n1 < n2.

// `n` is in a component that is represented by `rep`.
// `rep` is the highest node of the component.
ComponentRep(n, rep) :- Reachable(n, rep), not ReachUp(rep).

// `Component(rep; c)` describes that the node `rep` is the
// representative of the component `c` which is a set of nodes.
Component(rep; Set#{n}) :- ComponentRep(n, rep).
```

## Summary

We have seen several extensions that enrich Datalog in Flix:

- Rho abstraction as a mechanism to hide predicate symbols.
- From *constraints on relations*, to *constraints on lattices*.
- How to compute provenance information with lattice semantics.