一、引入

前面聚焦的点在算法思想上，但终归，我们要将其落实到具体的计算机系统上。

Goal: select adults from a set of persons

- Imperative: how to do (~implementation) 命令式

```
Set<Person> selectAdults(Set<Person> persons) {
  Set<Person> result = new HashSet<>();
  for (Person person : persons)
    if (person.getAge() >= 18)
      result.add(person);
  return result;
}
```

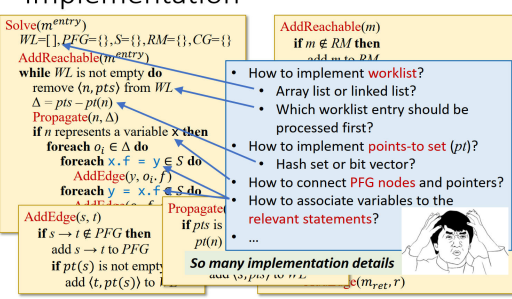
- Declarative: what to do (~specification) 声明式

```
SELECT * FROM Persons WHERE Age >= 18;
```

前者对程序设计的粒度更细，掌控的细节更多，但这同时也意味着烦琐；后者的层次更高，而把逻辑、脏活都被系统规范给封装好了。

对于PTA等静态分析算法，两种方法的工作量是完全不同的

Pointer Analysis, Imperative Implementation



Pointer Analysis, Declarative Implementation (via Datalog)

```
VarPointsTo(x, o) <-
  Reachable(m),
  New(x, o, m).

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

```
VarPointsTo(this, o),
Reachable(m),
CallGraph(l, m) <-
  VCall(l, x, k),
  VarPointsTo(x, o),
  Dispatch(o, k, m),
  ThisVar(m, this).

VarPointsTo(pi, o) <-
  CallGraph(l, m),
  Argument(l, i, ai),
  Parameter(m, i, pi),
  VarPointsTo(ai, o).

VarPointsTo(r, o) <-
  CallGraph(l, m),
  MethodReturn(m, ret),
  VarPointsTo(ret, o),
  CallReturn(l, r).
```

- Succinct
- Readable (logic-based specification)
- Easy to implement

二. Datalog 介绍

1. 基础介绍

- Datalog is a declarative logic programming language that is a subset of Prolog.
- It emerged as a database language (mid-1980s)*
- Now it has a variety of applications
  - Program analysis
  - Declarative networking
  - Big data
  - Cloud computing
  - ...

*David Maier, K. Tuncay Tekle, Michael Kifer, and David S. Warren, "*Datalog: Concepts, History, and Outlook*". Chapter, 2018.

Datalog = Data + Logic
(and, or, not)

- No side-effects 赋值，改值.
- No control flows
- No functions
- Not Turing-complete 非图灵完备

2. 谓词

用来描述若干项之间的联系，$R: D^n \to \{0,1\}$。由于论域 D 的离散性，其在某种程度下可以视为全组合论域 D^n 中所有变元组针对某个 agenda 的陈述集（有真有假）

Ps: 为真的陈述被称为 fact

3. 原子公式

谓词 + 项 构成原子公式 → $P(t_1, t_2, \cdots, t_n)$

在形式化中，将使用以下四类符号。

(1)常量符号：$a, b, c, \ldots, a_i, b_i, c_i, \ldots, i \geq 1$，当论域D给出时，它可以是D中的某个元素；

(2)变量符号：$x, y, z, \ldots, x_i, y_i, z_i, \ldots, i \geq 1$，当论域D给出时，它可以是D中的任何一个元素；

(3)函数符号：$f, g, h, \ldots, f_i, g_i, h_i, \ldots, i \geq 1$，当论域D给出时，n元函数符号$f(x_1, \ldots, x_n)$可以是$D^n$到$D$的任意一个映射；

(4)谓词符号：$P, Q, R, \ldots, P_i, Q_i, R_i, \ldots, i \geq 1$，当论域D给出时，n元谓词符号$P(x_1, \ldots, x_n)$可以是$D^n$到{1，0}的任意一个谓词。

定义1一阶逻辑中的项(item)，被递归定义如下：

(1)常量符号是项；

(2)变量符号是项；

(3)若$f(x_1, \ldots, x_n)$是n元函数符号，$t_1, \ldots, t_n$是项，则$f(t_1, \ldots, t_n)$是项；

(4)只有有限次地使用(1)、(2)、(3)所生成的符号串才是项。

例如a，b，x，y是项，$f(x, y) = x + y$，$g(x, y) = x \cdot y$是项，$f(a, g(x, y)) = a + x \cdot y$也是项。

定义2设$P(x_1, \ldots, x_n)$是n元谓词，$t_1, \ldots, t_n$是项，则称$P(t_1, \ldots, t_n)$为原子公式，或简称原子。[2]

① 使用算术谓词来描述项之间的关系，在Datalog中被称为arithmetic atom

- In addition to relational atoms, Datalog also has arithmetic atoms
  - E.g., age >= 18

② 使用其它特殊的谓词来描述项之间的关系，被称为 relational atom

- Age("Xiaoming",18) is true
- Age("Alan",23) is false

Age

person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

本质上都是 D^n 谓词 → {0, 1}，是 Datalog 中的基本单元

4. 逻辑推断

我们前面讲到的一些概念中，论域作为承载所有谓词的域，取决于需求和真实情况，谓词作为定义在其上的函数，其产生的不同事实分布描述了论域的一种关系性结构。推断实际上就是谓词在论域上的推导。

```
H <- B1,B2,…,Bn.
```

Head (consequent)
H is an atom

Body (antecedent)
Bi is a (possibly negated) atom
Each Bi is called a subgoal

```
H(X1,X2) <- B1(X1,X3),B2(X2,X4),…,Bn(Xm).
```

抽象地来看，其意味着当且仅当所有Bi为真时，H为真。但由于Bi与H都是原子公式，所以这种过程实际上是由具体的事实所描述构建的。结果是，我们可以得到一种全新的、对论域的结构性认识。

Datalog program = Facts + Rules

Age

person	age
Xiaoming	18
Xiaohong	23
Alan	16
Abao	31

+

```
Adult(person) <-
  Age(person,age),
  age >= 18.
```

→

一些新的facts

Adult

person
Xiaoming
Xiaohong
Abao

因此我们发现Datalog实际上就是在论域上，基于已有的谓词和Rules进行推导，得到一些新的谓词（以及其用在论域上的结果）

Conventionally, predicates in Datalog are divided into two kinds:

1. EDB (extensional database) → 初始输入的谓词
   - The predicates that are defined in a priori
   - Relations are immutable → 不可变
   - Can be seen as input relations
2. IDB (intensional database) → 根据rules推断出来的谓词
   - The predicates that are established only by rules
   - Relations are inferred by rules
   - Can be seen as output relations

```
H <- B1,B2,…,Bn.
```

- H can only be IDB
- Bi can be EDB or IDB

☆Datalog的Rules

(1) 与，

(2) 或

```
SportFan(person) <-
  Hobby(person,“jogging”);
  Hobby(person,“swimming”).
```

Ps：与的运算优先级要高于或

；分成两行去写

```
SportFan(person) <- Hobby(person,“jogging”).
SportFan(person) <- Hobby(person,“swimming”).
```

(3) 非 !

```
MakeupExamStd(student) <-
  Student(student),
  !PassedStd(student).
```

Where Student stores all students, and PassedStd stores the students who passed the exam.

(4) 递归 Recursion

也即IDB可由其自身直接/间接的推断得出。其使得Datalog超越了操作简单代数的查询语言(SQL),而能表达更加复杂的语义特性。

- For example, we can compute the reachability information (i.e., transitive closure) of a graph with recursive rules:

```
Reach(from, to) <-
  Edge(from, to).

Reach(from, to) <-
  Reach(from, node),
  Edge(node, to).
```

Where `Edge(a,b)` means that the graph has an edge from node a to node b, and `Reach(a,b)` means that b is reachable from a.

PS:①变量域约束

如果论域无穷 + 推断结构不闭合

⇓

可能需要枚举无穷个项来判断结果

```
A(x) <- B(y), x > y.
```

```
A(x) <- B(y), !C(x,y).
```

For both rules, infinite values of x can satisfy the rule, which makes A an ***infinite*** *relation*.

因此我们规定一个rule是safe的当且仅当其所涉及到的变量至少被一个non-negated relational atom所约束。

↑即使一个变量的论域是无穷的,施加一个non-negated的关系型原子公式后也变成有穷的了

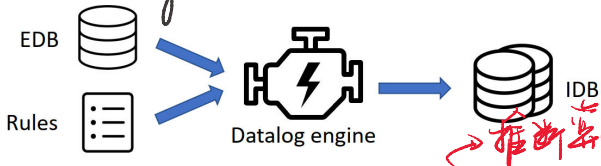
② 循环悖论

```
A(x) <- B(x), !A(x).
```

某个原子公式成立当且仅当其取反成立本身就很糟糕。

因此规则二就是避免对一个原子公式同时进行取反与递归。

5. Datalog 程序的执行



推断算法　初始谓词

- Datalog engine deduces facts by given rules and EDB predicates until no new facts can be deduced. Some modern Datalog engines
  LogicBlox, Soufflé, XSB, Datomic, Flora-2, …
- Monotonicity: Datalog is monotone as facts cannot be deleted
- Termination: A Datalog program always terminates as
  1) Datalog is monotone 单调
  2) Possible values of IDB predicates are finite (rule safety) 可得到的IDB谓词有穷？

其可终止性某种程度上就揭示了Datalog并不是一种图灵完备的语言

三. DataLog 的应用

1. 指针分析

首先确定初始的、已想等的谓词。

Kind	Statement
New	i: x = new T()
Assign	x = y
Store	x.f = y
Load	y = x.f

EDB

`New(x : V, o : O)`

`Assign(x : V, y : V)`

`Store(x : V, f : F, y : V)`

`Load(y : V, x : V, f : F)`

IDB

`VarPointsTo(v: V, o : O)`

e.g., fact `VarPointsTo`(x, o_i) represents $o_i \in pt(x)$

`FieldPointsTo(oi : O, f: F, oj : O)`

e.g., fact `FieldsPointsTo`(o_i, f, o_j) represents $o_j \in pt(o_i.f)$

Variables:	V
Fields:	F
Objects:	O

然后基于程序论域，得到 EDB 初始谓词的事实域。

An Example

```
1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;
```

Variables:	V
Fields:	F
Objects:	O

`New(x : V, o : O)`

New	
b	o_1
c	o_3

`Assign(x : V, y : V)`

Assign	
a	b
d	c

`Store(x : V, f : F, y : V)`

Store		
c	f	a
c	f	d

`Load(x : V, y : V, f : F)`

Load		
e	d	f

Datalog Rules for Pointer Analysis

Kind	Statement	Rule
New	`i: x = new T()`	$\frac{}{o_i \in pt(x)}$
Assign	`x = y`	$\frac{o_i \in pt(y)}{o_i \in pt(x)}$
Store	`x.f = y`	$\frac{o_i \in pt(x) \quad o_j \in pt(y)}{o_j \in pt(o_i.f)}$
Load	`y = x.f`	$\frac{o_i \in pt(x) \quad o_j \in pt(o_i.f)}{o_j \in pt(y)}$

```
VarPointsTo(x, o) <-
  New(x, o).

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

之后根据推断规则集，在EDB的事实域的基础上得到IDB的事实域。

An Example

```
1 b = new C();
2 a = b;
3 c = new C();
4 c.f = a;
5 d = c;
6 c.f = d;
7 e = d.f;
```

```
VarPointsTo(x, o) <-
  New(x, o).

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

New(x:V, o:O)

New	
b	o_1
c	o_3

Store(x:V, f:F, y:V)

Store		
c	f	a
c	f	d

Assign(x:V, y:V)

Assign	
a	b
d	c

Load(x:V, y:V, f:F)

Load		
e	d	f

VarPointsTo(v:V, o:O)

VarPointsTo	
b	o_1
c	o_3
a	o_1
d	o_3
e	o_1
e	o_3

FieldPointsTo

o_3	f	o_1
o_3	f	o_3

FieldPointsTo(oi:O, f:F, oj:O)

上面是单方法、不含调用的例子。下面我们来看看如何处理方法调用。

Kind	Statement	Rule
Call	l: r = x.k(a1,…,an)	$o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k})$ $o_u \in pt(aj), 1 \le j \le n$ $o_v \in pt(m_{ret})$ $\Rightarrow o_i \in pt(m_{this})$ $o_u \in pt(m_{pj}), 1 \le j \le n$ $o_v \in pt(r)$

EDB
- VCall(l:S, x:V, k:M)
- Dispatch(o:O, k:M, m:M)
- ThisVar(m:M, this:V)

IDB
- Reachable(m:M)
- CallGraph(l:S, m:M)

```
VarPointsTo(this, o),
Reachable(m),
CallGraph(l, m) <-
  VCall(l, x, k),
  VarPointsTo(x, o),
  Dispatch(o, k, m),
  ThisVar(m, this).
```

Statements (Labels):	S
Methods:	M

Kind	Statement	Rule
Call	l: r = x.k(a1,…,an)	$o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k})$ $\Rightarrow o_u \in pt(aj), 1 \le j \le n$ $o_v \in pt(m_{ret})$ $o_i \in pt(m_{this})$ $\Rightarrow o_u \in pt(m_{pj}), 1 \le j \le n$ $o_v \in pt(r)$

EDB
- Argument(l:S, i:N, ai:V)
- Parameter(m:M, i:N, pi:V)

```
VarPointsTo(pi, o) <-
  CallGraph(l, m),
  Argument(l, i, ai),
  Parameter(m, i, pi),
  VarPointsTo(ai, o).
```

Statements (Labels):	S
Methods:	M
Nature numbers (indexes)	N

Kind	Statement	Rule
Call	l: r = x.k(a1,…,an)	$o_i \in pt(x),\ m = \text{Dispatch}(o_i, \text{k})$ $o_u \in pt(aj), 1 \le j \le n$ $\Rightarrow o_v \in pt(m_{ret})$ $o_i \in pt(m_{this})$ $o_u \in pt(m_{pj}), 1 \le j \le n$ $\Rightarrow o_v \in pt(r)$

EDB
- MethodReturn(m:M, ret:V)
- CallReturn(l:S, r:V)

```
VarPointsTo(r, o) <-
  CallGraph(l, m),
  MethodReturn(m, ret),
  VarPointsTo(ret, o),
  CallReturn(l, r).
```

```
Reachable(m) <-
  EntryMethod(m).

VarPointsTo(x, o) <-
  Reachable(m),
  New(x, o, m).

VarPointsTo(x, o) <-
  Assign(x, y),
  VarPointsTo(y, o).

FieldPointsTo(oi, f, oj) <-
  Store(x, f, y),
  VarPointsTo(x, oi),
  VarPointsTo(y, oj).

VarPointsTo(y, oj) <-
  Load(y, x, f),
  VarPointsTo(x, oi),
  FieldPointsTo(oi, f, oj).
```

```
VarPointsTo(this, o),
Reachable(m),
CallGraph(l, m) <-
  VCall(l, x, k),
  VarPointsTo(x, o),
  Dispatch(o, k, m),
  ThisVar(m, this).

VarPointsTo(pi, o) <-
  CallGraph(l, m),
  Argument(l, i, ai),
  Parameter(m, i, pi),
  VarPointsTo(ai, o).

VarPointsTo(r, o) <-
  CallGraph(l, m),
  MethodReturn(m, ret),
  VarPointsTo(ret, o),
  CallReturn(l, r).
```

2、污点分析

- EDB predicates
  - `Source(m: M)` // source methods
  - `Sink(m: M, i: N)` // sink methods
  - `Taint(l: S, t: T)` // associates each call site to the tainted data from the call site
- IDB predicate
  - `TaintFlow(sr: S, sn: S, i: N)` // detected taint flows, e.g., `TaintFlow(`sr`,`sn`,`i`)` denotes that tainted data from source call sr may flow to i-th argument of sink call sn
- Handles sources (generates tainted data)

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{l \to m \in CG \quad m \in Sources}{t_l \in pt(r)}$

```
VarPointsTo(r, t) <-
  CallGraph(l, m),
  Source(m),
  CallReturn(l, r),
  Taint(l, t).
```

Call-site l 时在接收变量为r l时在污点为 t.

- Handles sinks (generates taint flow information)

Kind	Statement	Rule
Call	`l: r = x.k(a1,…,an)`	$\dfrac{l \to m \in CG \quad \langle m,i\rangle \in Sinks \quad t_j \in pt(ai)}{\langle j,l,i\rangle \in TaintFlows}$

```
TaintFlow(j, l, i) <-
  CallGraph(l, m),
  Sink(m, i),
  Argument(l, i, ai),
  VarPointsTo(ai, t),
  Taint(j, t).
```

3. Datalog的优缺点对比

- Pros
  - Succinct and readable 可读性
  - Easy to implement 易使用
  - Benefit from off-the-shelf optimized Datalog engines 可单独优化引擎模块
- Cons
  - Restricted expressiveness, i.e., it is impossible or inconvenient to express some logics → 表达能力受限.
  - Cannot fully control performance 能控制的粒度较粗.