CS 526
Advanced Compiler Construction

http://misailo.cs.Illinois.edu/courses/cs526
POINTER ANALYSIS

The slides adapted from Vikram Adve
Course

So far:
• Dataflow analysis (examples and theory)
• Dependency analysis
• SSA (sparse dataflow analysis)

Coming up next:
• Pointer analysis (generalize the dependence relationship)
• Interprocedural analysis (how to analyze function calls?)
• Abstract interpretation (generalize dataflow analysis)
• Fun topics (probabilistic, autotuning, VMs …)
POINTER ANALYSIS

The slides adapted from Vikram Adve
**Pointer Analysis**

Pointer and Alias Analysis are fundamental to reasoning about heap manipulating programs (pretty much all programs today).

- **Pointer Analysis:**
  - What objects does each pointer points to?
  - Also called points-to analysis

- **Alias Analysis:**
  - Can two pointers point to the same location?
  - Client of pointer analysis
Example

\[
X = 1 \\
P = &X \\
*P = 2 \\
return X
\]

// What is the value of X?
Aliases

Consider references r1 or r2,
• may be of the form “x” or “*p” “**p”, “(*p)->q->i”...
• We assume C notation for dereferencing pointers (*, ->)

Alias: r1 are r2 are aliased if the memory locations accessed by r1 and r2 overlap.

Alias Relation: A set of ordered pairs {(ri, rj)} denoting aliases that may hold at a particular point in a program.
• Sometimes called a may-alias relation.

May or Must: A kind of aliasing if it happens optionally or always
• May: e.g., depending on the control flow: if (b) { p = &q; }
• Must: determined that they always represents aliases
Aliases

We look at the language that extends the simple expressions with the additional pointer-like structures:

\[ p := &x \]
\[ p := q \]
\[ \ast p := q \]
\[ p := \ast q \]

Consider references r1 or r2,

• may be of the form “x” or “\ast p” “\ast \ast p”, “(*p)->q->i”…
• We assume C notation for dereferencing pointers (\ast, ->)
Example

\[ X = 1 \]
\[ P = \&X \]
\[ Q = P \]
\[ \ast P = 2 \]
Example

$X = 1$

$P = \&X$

$Q = P$

$*P = 2$

**Example**

**Aliasing pairs**

// (\*P, X)

// \{ (\*P, X), (\*Q, X) \}

**Alias:** r1 are r2 are aliased if the memory locations accessed by r1 and r2 overlap.
Points-To Facts

Points-to Pair: pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) may hold the address of one of the memory locations of \(r_2\).

• Also written: \(r_1 \rightarrow r_2\), means “\(r_1\) points to \(r_2\)”.

Points-to Set: \(\{(r_i, r_j)\}\) : A set of points-to pairs that may hold at a particular point in a program.

Points-To Graph: A directed graph where

• Nodes represents one or more memory objects;
• Each Edge \(p \rightarrow q\) means some object in the node \(p\) may hold a pointer to some object in the node \(q\).
Example

X = 1
P = &X
Q = P
*P = 2

Points-to Pair: pair (r1, r2) denoting that one of the memory locations of r1 An ordered may hold the address of one of the memory locations of r2.

Points-to pairs

// (P, X)
// { (P, X), (Q, X) }
**Example**

\[
\begin{align*}
X &= 1 \\
P &= \&X \\
Q &= P \\
R &= Q
\end{align*}
\]

**Points-to Pair**: pair \((r_1, r_2)\) denoting that one of the memory locations of \(r_1\) An ordered may hold the address of one of the memory locations of \(r_2\).

**Points-to pairs**

\[
\begin{align*}
// (P, X) \\
// \{ (P, X), (Q, X) \} \\
// \{ (P, X), (Q, X), (R, X) \}
\end{align*}
\]

"**Short notation**: vs the long one that would list all the aliases."
Challenges of Points-To Analysis

- **Pointers to pointers**, which can occur in many ways: take address of pointer; pointer to structure containing pointer; pass a pointer to a procedure by reference
- **Aggregate objects**: structures and arrays containing pointers
- **Recursive data structures** (lists, trees, graphs, etc.) closely related problem: anonymous heap locations
- **Control-flow**: analyzing different data paths
- **Interprocedural**: a location is often accessed from multiple functions; a common pattern (e.g., pass by reference)
- **Compile-time cost**
  - Number of variables, |V|, can be large
  - Number of alias pairs at a point can be $O(|V|^2)$
Common Simplifying Assumptions

**Aggregate objects**: arrays (and perhaps structures) containing pointers

**Simple solution**: Treat as a single big object!

- Commonplace for arrays.
- Not a good choice for structures?
  - *See Intel Paper (Ghiya, Lavery & Sehr, PLDI 2001)*
- Pointer arithmetic is only legal for traversing an array:
  
  \[
  q = p \pm i \quad \text{and} \quad q = \&p[i] \quad \text{are handled the same as} \quad q = p
  \]
Common Simplifying Assumptions

Recursive data structures (lists, trees, graphs, …)

Solution: Compute aliases, not “shape”

- Don’t prove something is a linked-list or a binary tree (leave that for \textit{shape analysis})
- \textbf{\textit{k-limiting}}: only track \( k \) or fewer levels of dereferencing
- Use simplified naming schemes for heap objects (later slide)
Common Simplifying Assumptions

**Control-flow:** analyzing different data paths blows up the analysis time/space

**Solution(?)**: Could ignore the issue and compute a single common result for any path!

**No consensus on this issue!** (Will discuss later)
Naming Schemes for Heap Objects

The Naming Problem: Example 1

```c
int main() {
    // Two distinct objects
    T* p = create(n);
    T* q = create(m);
}

T* create(int num) {
    // Many objects allocated here
    return new T(num);
}
```

Q. Should we try to distinguish the objects created in main()?
Naming Schemes for Heap Objects

The Naming Problem: Example 2

T* makelist(int len) {
    T* newObj = new T;    // Many distinct objects
                          // allocated here
    newObj->next = (--len == 0)? NULL :
                    makelist(len);
}

Q. Can we distinguish the objects created in makelist()?
Possible Naming Abstractions

\( H_0 \) : One name for the entire heap

\( H_T \) : One name per type \( T \) (for type-safe languages)

\( H_L \) : One name per heap allocation site \( L \) (line number)

\( H_C \) : One name per (acyclic) call path \( C \) ("cloning")

\( H_F \) : One name per immediate caller \( F \) or call-site ("one-level cloning")
**Flow-sensitivity of Analysis**

**Def.** A *flow-sensitive analysis* is one that computes a distinct result for each program point. A *flow-insensitive analysis* generally computes a single result for an entire procedure or an entire program.

**A flow-insensitive algorithm effectively ignores the order of statements!**

```c
int f(T q, T r){
    T* p;
    ...
    p = &q;
    ...
    p = &r;
}
```
Flow-sensitivity of Analysis

**Def.** A *flow-sensitive analysis* is one that computes a distinct result for each program point. A *flow-insensitive analysis* generally computes a single result for an entire procedure or an entire program.

A flow-insensitive algorithm effectively ignores the order of statements!

```c
int f(T q, T r){
    T* p;
    if (...)
        p = &q;
    else
        p = &r;
}
```

![Flow Sensitive Diagram](image)

![Flow Insensitive Diagram](image)
Flow-Sensitivity of Analysis

Pointer Analysis

• **Flow-sensitive**: At program point \( n \), compute alias pairs \(<a, b>\) that may hold at \( n \) for some path from program entry to \( n \).

• **Flow-insensitive**: Compute all alias pairs \(<a, b>\) such that \( a \) may be aliased to \( b \) at *some* point in a program (or function).

Important special cases

• Local scalar variables: SSA form gives flow-sensitivity

• Malloc or new: Allocates “fresh” memory, i.e., no aliases

• Read-only fields: e.g., array length
Realizable Paths

Definition: Realizable Path
A program path is realizable iff every procedure call on the path returns control to the point where it was called (or to a legal exception handler or program exit)

Whole-program Control Flow Graph?
Conceptually extend CFG to span whole program:
• split a call node in CFG into two nodes: CALL and RETURN
• add edge from CALL to ENTRY node of each callee
• add edge from EXIT node of each callee to RETURN
Problem: This produces many unrealizable paths

Focusing only on realizable paths requires context-sensitive analysis
Def. A context-sensitive interprocedural analysis computes results that may hold only for realizable paths through the program. Otherwise, the analysis is context-insensitive.

```c
T* identity(T* p) {
    return p;
}

void f1() {
    T* p1 = new T; // Object o1
    T* q1 = identity(p1);
}

void f2() {
    T* p2 = new T; // Object o2
    T* q2 = identity(p2);
}
```
Context-Sensitivity of Analysis

Pointer Analysis
Apply the definitions directly using points-to pairs \(<a, b>\).
But important variations exist:
• Heap cloning vs. no cloning: Cloning gives greater context-sensitivity
• Bottom-up vs. top-down: Does final result for a procedure include only “realizable” behavior from all contexts?
• Handling of recursive functions: Does analysis retain context-sensitivity within SCCs in the call graph?

Object Sensitivity: Context represents each allocation site. Typically offers quite precise context analysis

[Parameterized Object Sensitivity for Points-to and Side-Effect Analyses for Java; Milanova et al. ISSTA 2002]
Field-Sensitivity of Analysis

Def. A field-sensitive analysis is one that tracks distinct behavior for individual fields of a record type. Otherwise, it is field-insensitive.

```c
int f(T q, T r) {
    p.a = &q;
    p.b = &r;
}
```

Challenges

- Complexity: For certain analysis techniques, converts linear representation to worse (perhaps even exponential)
- Non-type-safe programs: May have to track behavior at every byte offset within the structure (not just each field)
Flow Insensitive Algorithms

3 popular algorithms
- Any address
- Andersen, 1994
- Steensgard, 1996

Acceptable precision in practice for compiler optimization, however perhaps insufficient for static analysis approaches for security, reliability, or bug finding
Any Address Analysis

- **Single points-to set**: contains all variables whose address is taken, passed by reference, etc.

- **Any pointer may point** to **any variable** in this set

- Simple, fast, linear-time algorithm

- Common choice for function pointers, and for global variables, e.g., for initial call graph
Example 1

```c
void main() {
    T *p, *q, *r;
    T t;

    o1: p = new T; // {p} -> {o1}
    q = &t; // {p,q} -> {o1,t}
    r = q; // {p,q,r} -> {o1,t}
}
```
Example 2 (Interprocedural)

T *p, *q, *r;

void main() {
    p = new T;
    g(&p);
    f();
    p = new T;
    .. = *p;
}

void f() {
    q = new T;
    g(&q);
    r = new T;
}

void g(T** fp) {
    T* local = new T;
    if (..)
        *fp = local;
    ..
}
Andersen’s Algorithm

- Generally the most precise flow- and context-insensitive algorithm
- Compute a single points-to graph for entire program
- Refinement by Burke: Separate points-to graph for each function
- Cost is $O(n^3)$ for program with n assignments
  - McAlister, On the complexity analysis of static analyses (SAS’99)
  - Sridharan and Fink, The Complexity of Andersen’s Analysis in Practice (SAS’09)
Andersen’s Algorithm: **Conceptual**

**Initialize:** Points-to graph with a separate node per variable

**Iterate until convergence:**
At each statement, evaluate the appropriate rule:

<table>
<thead>
<tr>
<th>Form</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>( p = &amp; x )</td>
<td>Add ( p \rightarrow x )</td>
</tr>
<tr>
<td>( p = q )</td>
<td>( \forall x : \text{if } q \rightarrow x, \text{add } p \rightarrow x )</td>
</tr>
<tr>
<td>( *p = q )</td>
<td>( \forall x, r: \text{if } q \rightarrow x \text{ and } p \rightarrow r, \text{add } r \rightarrow x )</td>
</tr>
<tr>
<td>( p = *q )</td>
<td>( \forall x, r: \text{if } q \rightarrow x \text{ and } x \rightarrow r, \text{add } p \rightarrow r )</td>
</tr>
</tbody>
</table>
Andersen's Algorithm: \textbf{Actual}

1. Build initial "inclusion constraint graph" and initial points-to sets
2. Iterate until converged:
   - Update constraint graph for new points-to pairs
   - Update the points-to sets according to new constraints

\textbf{Inclusion Constraint Graph}: Add constraint for pointer assignments (pts is points-to set):

<table>
<thead>
<tr>
<th>Name</th>
<th>Form</th>
<th>Constraint</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>Points-to pair</td>
<td>(p = &amp;x)</td>
<td>(p \supset {x})</td>
<td>pts(p) U= {x}</td>
</tr>
<tr>
<td>Direct constraint</td>
<td>(p = q)</td>
<td>(p \supset q)</td>
<td>pts(p) U= pts(q)</td>
</tr>
<tr>
<td>Indirect constraint</td>
<td>(*p = q)</td>
<td>(*p \supset q)</td>
<td>for (v \in pts(p)): pts(v) U= pts(q)</td>
</tr>
<tr>
<td>Indirect constraint</td>
<td>(p = *q)</td>
<td>(p \supset *q)</td>
<td>for (v \in pts(q)): pts(p) U= pts(v)</td>
</tr>
</tbody>
</table>
Example 1 Revisited

```c
void main() {
    T *p, *q, *r;
    T t;

    o1:p = new T;  // {p} -> {o1}
    q = &t;         // {p} -> {o1}, {q} -> {t}
    r = q;         // {r} -> {t}
}
```

<table>
<thead>
<tr>
<th>Form</th>
<th>Constraint</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = &amp;x</td>
<td>p ⊇ {x}</td>
<td>pts(p) U= {x}</td>
</tr>
<tr>
<td>p = q</td>
<td>p ⊇ q</td>
<td>pts(p) U= pts(q)</td>
</tr>
<tr>
<td>*p = q</td>
<td>*p ⊇ q</td>
<td>for v ∈ pts(p):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pts(v) U= pts(q)</td>
</tr>
<tr>
<td>p = *q</td>
<td>p ⊇ *q</td>
<td>for v ∈ pts(q):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pts(p) U= pts(v)</td>
</tr>
</tbody>
</table>
Andersen’s Algorithm: Cycles

Cycle in constraint graph:
\[ \text{pts}(p) \supseteq \text{pts}(q) \supseteq \text{pts}(r) \supseteq \text{pts}(p) \]
\[ \Rightarrow \text{pts}(p) = \text{pts}(q) = \text{pts}(r) = \text{pts}(p) \]
\[ \Rightarrow \text{No need to propagate points-to pairs around such cycles!} \]
Andersen’s Algorithm: Cycles

Cycle in constraint graph:
\[ \text{pts}(p) \supseteq \text{pts}(q) \supseteq \text{pts}(r) \supseteq \text{pts}(p) \]
\[ \Rightarrow \text{pts}(p) = \text{pts}(q) = \text{pts}(r) = \text{pts}(p) \]
\[ \Rightarrow \text{No need to propagate points-to pairs around such cycles!} \]

Offline cycle elimination:
• Find cycles due to direct pointer copies (direct constraints)
• Collapse each cycle into a single node, reduces size of constraint graph
• But many more cycles can be induced by indirect constraint edges: we need cycle elimination during transitive closure ("online")


Online cycle elimination:
• Fähndrich, Foster, Aiken and Su (PLDI ’98): Cycle elimination is essential for scalability.
• Heintze and Tardieu (PLDI 2001): "A million lines of code per second."
• Hardekopf and Lin (PLDI 2007)
Example 2

```c
void f(int i) {
    T *p, *q, *r;

    o1:p = new T;  // {p} -> {o1}
    o2:q = new T;  // {q} -> {o2}
    if (i>0)
        r = p;      // {r} -> {o1}
    else
        r = q;      // {r} -> {o2}
}
```

<table>
<thead>
<tr>
<th>Form</th>
<th>Constraint</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>p = &amp;x</td>
<td>p ⊇ {x}</td>
<td>pts(p) U= {x}</td>
</tr>
<tr>
<td>p = q</td>
<td>p ⊇ q</td>
<td>pts(p) U= pts(q)</td>
</tr>
<tr>
<td>*p = q</td>
<td>*p ⊇ q</td>
<td>for v ∈ pts(p):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pts(v) U= pts(q)</td>
</tr>
<tr>
<td>p = *q</td>
<td>p ⊇ *q</td>
<td>for v ∈ pts(q):</td>
</tr>
<tr>
<td></td>
<td></td>
<td>pts(p) U= pts(v)</td>
</tr>
</tbody>
</table>
Example 3

\[
p = &a; \quad \text{// } p \rightarrow \{a\} \quad \text{// } p \rightarrow \{a\}
\]
\[
s = &p; \quad \text{// } s \rightarrow \{p\} \quad \text{// } s \rightarrow \{p,q\}
\]
\[
r = *s; \quad \text{// } r \rightarrow \{a\} \quad \text{// } r \rightarrow \{a, b\}
\]
\[
q = &b; \quad \text{// } q \rightarrow \{b\} \quad \text{// } q \rightarrow \{t\}
\]
\[
s = &q; \quad \text{// } s \rightarrow \{p,q\} \quad \text{// } s \rightarrow \{p,q\}
\]

Done? 

Done?
Steensgaard’s Algorithm

Unification:
- Conceptually: restrict every node to only one outgoing edge (on the fly)
- If $p \rightarrow x$ and $p \rightarrow y$, merge $x$ and $y$ (“unify”)
- All objects “pointed to” by $p$ comprise a single equivalence class

\[
\begin{align*}
A & = \&B \\ 
B & = \&C \\ 
A & = \&D \\ 
D & = \&E 
\end{align*}
\]
Steensgard’s Algorithm

Unification: Conceptually: restrict every node to only one outgoing edge (on the fly)

- If p → x and p → y, merge x and y (“unify”)
- All objects “pointed to” by p form one equivalence class

Algorithm

1. For each statement, merge points-to sets:
   - p = q: Merge two equivalence classes (p’s and q’s targets)
     Less expensive than computing points-to iterations
     This may cause other nodes to collapse!

2. Use Tarjan’s “union-find” data structure to record equivalence classes
Steensgard’s Algorithm

“Union-find” aka Disjoint Set data structure:
• Partitions the set of elements into disjoint partitions
• Maintains the partition with every addition
• Operations:
  • Find(x): follows parent pointers from x until reaching root (i.e. finds the set containing x)
  • Union(x,y): 1) finds the roots of x,y; 2) merges the trees by connecting the root nodes. (i.e. merges the sets)
• Properties: addition and merge of sets in near constant time, i.e. $\alpha(n)$ – inverse Ackerman func. $\alpha(n) < 4$ even for large n.

Consequence for Steensgard’s analysis:
• Non-iterative algorithm, almost-linear running time: $O(n\alpha(n))$
• Like Anderson, single solution for entire program
Steensgard vs. Anderson

Consider assignment \( p = q \), i.e., only \( p \) is modified, not \( q \)

**Subset-based Algorithms** (Anderson’s algorithm is an example)
- Add a constraint: Targets of \( q \) must be subset of targets of \( p \)
- Graph of such constraints is also called “inclusion constraint graphs”
- Enforces unidirectional flow from \( q \) to \( p \)

**Unification-based Algorithms** (Steensgard is an example)
- Merge equivalence classes: targets of \( p \) and \( q \) must be identical
- Assumes bidirectional flow from \( q \) to \( p \) and vice-versa

**In-between solutions:**
- Unification-based Pointer Analysis with Directional Assignment, Das, PLDI 2000 – exploits the semantics of C; uses Andersen for top pointers, Steensgard elsewhere
Alias Analysis

- Alias analysis is a common client of pointer (points-to) analysis
  - **Pointer Analysis:** What objects does each pointer points to?
  - **Alias Analysis:** Can two pointers point to the same location?
- Once we have performed the pointer analysis, it is trivial to compute alias analysis (but not vice versa)

- Two pointers p and q may alias if $\text{points-to}(p) \cap \text{points-to}(q) \neq \emptyset$
Which Pointer Analysis To Use?
Hind & Pioli, ISSTA, Aug. 2000

Compared 5 algorithms (4 flow-insensitive, 1 flow-sensitive):
• Any address
• Steensgard
• Anderson
• Burke (like Anderson, but separate solution per procedure)
• Choi et al. (flow-sensitive)

Metrics
1. Precision: number of alias pairs
2. Precision of important optimizations: MOD/REF, REACH, LIVE, flow dependences, constant prop.
3. Efficiency: analysis time/memory, optimization time/memory

Benchmarks: 23 C programs, including some from SPEC benchmarks
Which Pointer Analysis To Use?

1. **Precision:** (Table 2)
   - Steensgard much better than Any-Address (6x on average)
   - Anderson/Burke significantly better than Steensgard (about 2x)
   - Choi negligibly better than Anderson/Burke

2. **MOD/REF precision:** (Table 2)
   - Steensgard much better than Any-Address (2.5x on average)
   - Anderson/Burke significantly better than Steensgard (15%)
   - Choi very slightly better than Anderson/Burke (1%)

3. **Analysis cost:** (Table 5)
   - Any-Address, Steensgard extremely fast
   - Anderson/Burke about 30x slower
   - Choi about 2.5x slower than Anderson/Burke

4. **Total cost of analysis + optimizations:** (Table 5)
   - Steensgard, Burke are 15% faster than Any-Address!
   - Anderson is as fast as Any-Address!
   - Choi only about 9% slower
## Analysis Scalability

<table>
<thead>
<tr>
<th></th>
<th>Equality-based</th>
<th>Subset-based</th>
<th>Flow-sensitive</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Context-insensitive</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>first paper on pointer analysis</td>
<td>1994: 5 KLOC</td>
<td>1993: 30 KLOC</td>
</tr>
<tr>
<td></td>
<td>first scalable pointer analysis</td>
<td>1998: 60 KLOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Heintze and Tardieu [11]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2001: 1 MLOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>Berndl et al. [2]</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>2003: 500 KLOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>first to use BDDs</td>
<td></td>
</tr>
<tr>
<td><strong>Context-sensitive</strong></td>
<td>Fähndrich et al. [8]</td>
<td>Whaley and Lam [35]</td>
<td></td>
</tr>
<tr>
<td></td>
<td>2000: 200K</td>
<td>2004: 600 KLOC</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>cloning-based BDDs</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Landi and Ryder [19]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1992: 3 KLOC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Wilson and Lam [37]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1995: 30 KLOC</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Whaley and Rinard [36]</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>1999: 80 KLOC</td>
</tr>
</tbody>
</table>

Derek Rayside, Points-To Analysis (Summary), 2005  

More recent: Flow-Sensitive Pointer Analysis for Millions of Lines of Code  
Hardekopf and Lin (CGO’11)
Advanced Techniques

• **Shape Analysis**: discovers and reasons about dynamically allocated data structures (e.g., lists, trees, heaps)

• **Escape Analysis**: computes which program locations can access a pointer (across function boundaries)

• **Datalog**: Declarative, constraint-based approach to specify analysis, offers pretty good scalability

  Pointer Analysis; Yannis Smaragdakis; George Balatsouras, Now Publishing, 2015
Datalog

Datalog: declarative language with Prolog-like notation

Elements: *atoms* of the form $p(X_1, X_2, \ldots X_n)$
- $p$ is a predicate
- $X_1, X_2, \ldots X_n$ are variables or constants

**Ground atoms**: predicate with only constant arguments
- Its value is either true or false

Rules: $H : B_1 \land B_2 \land \ldots \land B_n$
- $H$ is an *atom*, $B_1 \ldots B_n$ are *atoms* or *negations* of atoms
- $\land$ is “if” --- so $H$ is valid if all $B_1 \ldots B_n$ are valid

Datalog program is a collection of rules. The program is applied to a set of ground atoms. The result is the set of ground atoms inferred by applying the rules until fixpoint
Simple Datalog program (from Dragon book):

\[
\begin{align*}
\text{path}(X,Y) & : - \text{edge} (X,Y) \\
\text{path}(X,Y) & : - \text{path} (X,Z) \land \text{path} (Z,Y)
\end{align*}
\]

The meaning of the program: A single edge is a path; a path also exist if there is a path between the start point and some other point, and that other point and the end point.

Consider this example:

- True ground atoms: edge(1,2), edge(2,3), edge(3,4)
- Infer path(1,2), path(2,3), path(3,4) using rule #1
- Infer composite paths using successive application of rule #2
Flow-Insensitive Pointer Analysis

(Dragonbook) Compute:

- **Pts(V, H)** – the variable V can point to heap object H
- **Hpts(H, F, G)** – field F of heap object H points to heap object G

Rules constructed by traversing the program:

1. **Pts(V, H)** := “H: V = malloc”
   V points to heap loc H if it is allocated at H (say we use line number calling)

   V points to H if V points to W and W points to H

   In stmt V.F=W, field F of object H points to object G if ptr W points to G and ptr V points to H

   In stmt V=W.F,V points to H if W points to G and field F of G points to H
Context-Sensitive Pointer Analysis

First compute:

- **Pts(V, C, H)** – the variable V in context C can point to heap object H
- **Hpts(H, F, G)** – field F of heap object H points to heap object G
- **CSinvokes(S, C, M, D)** – the calls site S in context C calls the D context of M

Rules constructed by traversing the program:

5. Pts(V, D, H)  :-  CSinvokes(H, C, M, D) & formal(M,D,V) & actual(S,C,W) & pts(W,C,H)

If the call site S in context C calls method M of context D, then the formal parameters in method M of context D can point to the objects pointed to by the actual params in C