

The background of the slide is a solid brown color with a pattern of stylized, overlapping autumn leaves in various shades of brown and tan. The leaves are scattered across the entire frame, creating a textured, organic feel.

Practical Datalog

*Programming Language of the
Future?*

Introductions

David Zook

- Career: *Engineer -> Researcher*
- Education: *GA Tech PhD Candidate*
 - Programming Languages
- Employee: *LogicBlox, Inc.*

Introductions

David Zook

- Career: *Engineer -> Researcher*
- Education: *GA Tech PhD Candidate*
 - Programming Languages
- Employee: *LogicBlox, Inc.*
- Interests: *Improving Programming*
 - Mathematical methods for correctness
 - Visualization etc. for ease of use

Introductions

- Overview
 - LogicBlox Vision
 - “Standard” Datalog
 - Bottom-up Evaluation
 - Cool Extensions
 - Shameless Invitation

LogicBlox Vision

- An important calculation:
 - “Algorithms + Data Structures = Programs”
 - (Niklaus Wirth)

LogicBlox Vision

- An important calculation:
 - “Algorithms + Data Structures = Programs”
 - (Niklaus Wirth)
 - “Algorithm = Logic + Control”
 - (Robert Kowalski)

LogicBlox Vision

- An important calculation:
 - “Algorithms + Data Structures = Programs”
 - (Niklaus Wirth)
 - “Algorithm = Logic + Control”
 - (Robert Kowalski)
 - “Program = Logic + Control + Data Structures”
 - (Molham Aref)

LogicBlox Vision

- Keep orthogonal:
 - Description of the desired solution (“Logic”)
 - Solving/Search strategy (“Control”)
 - Storage of the information (“Data Structures”)

LogicBlox Vision

- Keep orthogonal:
 - Description of the desired solution (“Logic”)
 - Solving/Search strategy (“Control”)
 - Storage of the information (“Data Structures”)
- *What’s the language for the “Logic”?*

Datalog

- Datalog = Databases + Logic-Prog
 - “Deductive databases”

Datalog

- Datalog = Databases + Logic-Prog
 - “Deductive databases”
- Datalog = Prolog – Functions
 - Logical: *First-order Logic (FOL)*
 - Clausal: *Implied quantifiers*
 - Function-free: *Finite and “Structure-less”*
 - Terminating: *P_{TIME} Computation*
 - Bottom-Up: *Set-at-a-Time*

Datalog: Logical

- Concepts

- Proposition:

- *“A statement that affirms or denies something”*

- Predicate:

- *“That part of a proposition that is affirmed or denied about the subject(s)”*

- Example:

```
person("dave"). person("george"). person("suzy"). person("bob").
```

```
likes("dave", "suzy"). likes("bob", "suzy"). likes("suzy", "george").
```

Datalog: Function-free

- “Structure-less”
- Example:
 - Prolog functions

```
automobile(chassis(tire1,tire2,tire3,tire4), engine(crankshaft, cyl1, cyl2))
```

- “Flatten” the functions into relations

```
automobile(2, 3, 1).  
chassis(“tire1”, “tire2”, “tire3”, “tire4”, 2) .  
engine(“crankshaft”, “cyl1”, “cyl2”, 3).
```

Datalog: Clausal

- Clauses
 - Implied Quantifiers: “for-all”
 - Implication(“<-”): “if-then”
 - Conjunction of Atoms(“,”): “and”
 - (Independent variables: “there exists”)
- Extensional vs Intensional predicates
- Example:

```
friends(x,y) <- likes(x,y), likes(y,x).  
likedbysomeone(x) <- likes(x,y).
```

Datalog: Facts

- Fact Assertions: “this is always true”
 - Rule with no body: nice(“dave”) <- true.
- Example:

```
nice(“dave”) <-. //nuff said :-)
```

Datalog: Constraints

- Constraints: “do not allow this”
 - Clause with no head: $\text{false} \leftarrow \text{foo}(x,y)$.
 - For all x,y : foo is never true.
 - Equivalent to clause: $\neg(\text{foo}(x,y))$.
 - Use a clause: $\text{foo}(x,y) \rightarrow \text{bar}(x,y)$.
 - Whenever foo is true, so is bar .

■ $\leftarrow \text{loves}(x,y), \text{hates}(x,y)$.

$\text{knows}(x,y) \rightarrow \text{knows}(y,x)$. Or $\neg(\text{knows}(x,y), \neg\text{knows}(y,x))$.

$\text{father}(x1,y), \text{father}(x2,y) \rightarrow x1=x2$

Datalog: Recursion

- Recursion: Going beyond SQL
- Example: *Transitive Closure*

```
connected(x,y) <- knows(x,y).  
connected(x,y) <- knows(x,z), connected(z,y).  
                        or  
connected(x,y) <- connected(x,z), connected(z,y).
```

Datalog: Order

- Order: Captures PTIME
 - Ordered entities: First/next/previous preds
- Still: *Guaranteed termination*
- Example:

```
nums(i,v).
```

```
first(i), next(i,j) , last(i).
```

```
total(i,t) <- first(i), nums(i,v).
```

```
total(j,t) <- next(i,j), total(i,tp), nums(j,v), t=tp+v.
```

Execution (0)

- “Semi-Naïve” Algorithm: *Run to fixed-point*
- Example:

```
// Program
connected(x,y) <- knows(x,y).
connected(x,y) <- knows(x,z), connected(z,y).
```

```
// STEP 0
// Given facts for extensional preds.
knows = {"dave", "suzy"}, {"bob", "suzy"}, {"suzy", "george"}, {"george", "dave"}

// Empty set for intensional preds.
connected = {}
```

Execution (1)

```
// Program
connected(x,y) <- knows(x,y).
connected(x,y) <- knows(x,z), connected(z,y).
```

```
// Step 0
knows = {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
connected = {}
```

```
// Step 1
connected =
    {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
```

Execution (2)

```
// Program
connected(x,y) <- knows(x,y).
connected(x,y) <- knows(x,z), connected(z,y).
```

```
// Step 0
knows = {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
connected = {}
```

```
// Step 1
connected =
    {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
```

```
// Step 2
+connected = {"dave","george"}, {"bob","george"}, {"suzy","dave"}
```

Execution (3)

```
// Program
connected(x,y) <- knows(x,y).
connected(x,y) <- knows(x,z), connected(z,y).
```

```
// Step 0
knows = {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
connected = {}
```

```
// Step 1
connected =
    {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
```

```
// Step 2
+connected = { ("dave","george"), ("bob","george"), ("suzy","dave") }
```

```
// Step 3
+connected = { ("dave","dave"), ("bob","dave"), ("suzy","suzy") }
```

Execution (4)

```
// Program
connected(x,y) <- knows(x,y).
connected(x,y) <- knows(x,z), connected(z,y).
```

```
// Step 0
knows = {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
connected = {}
```

```
// Step 1
connected =
    {"dave","suzy"}, {"bob","suzy"}, {"suzy","george"}, {"george","dave"}
```

```
// Step 2
+connected = { ("dave","george"), ("bob","george"), ("suzy","dave") }
```

```
// Step 3
+connected = { ("dave","dave"), ("bob","dave"), ("suzy","suzy") }
```

```
// Step 4
+connected = { ("bob","bob") }
```

Extensions: Negation

- Negation (“!”): “not”
 - “Closed World Assumption” (CWA)
- Not-exists-not: “for-all”
- But: must be careful (stratification)
- Example:

```
unhappy(x) <- likes(x,y), !likes(y,x).
```

```
happy(x) <- !unlovedbyanyone(x).
```

```
unlovedbyanyone(x) <- person(y), !likes(x,y).
```

```
win(x) <- move(x,y), !win(y). //what if cyclic?
```

Extensions: Type-checking (1)

- Type-checking

- Type: *A unary predicate.*

- Native types: infinite.

- User-defined types: finite.

- Use clause constraints for type declarations.

- Example:

```
person(x) ->.    male(x) -> person(x).  
parent(x,y) -> person(x), person(y). father(x,y) -> male(x), person(y).  
gender(x,g) -> person(x), string(g).  
father(x,y) <- parent(x,y), gender(x,"male"). // type-correct?
```

Extensions: Type-checking (2)

- Type-checking

- Type: *A unary predicate.*

- Native types: infinite.

- User-defined types: finite.

- Use clause constraints for type declarations.

- Example:

```
person(x) ->.    male(x) -> person(x).  
parent(x,y) -> person(x), person(y). father(x,y) -> male(x), person(y).  
gender(x,g) -> person(x), string(g).  
father(x,y) <- parent(x,y), gender(x,"male"). // WRONG!  
father(x,y) <- parent(x,y), male(x). // CORRECT!
```

Extensions: Type-checking (3)

- Type-checking

- Type: *A unary predicate.*

- Native types: infinite.

- User-defined types: finite.

- Use clause constraints for type declarations.

- Example:

```
person(x) ->.    male(x) -> person(x).  
parent(x,y) -> person(x), person(y). father(x,y) -> male(x), person(y).  
gender(x,g) -> person(x), string(g).  
father(x,y) <- parent(x,y), gender(x,"male"). // CORRECT!  
father(x,y) <- parent(x,y), male(x). // CORRECT!  
male(x) <- gender(x,"male").
```

Extensions: Events

- Dynamic Logic
 - Transactions
 - Event/Condition/Action rules
 - Previous pred: $p@prev(x,y)$.
 - Delta preds: $+p(x,y)$ and $-p(x,y)$.

```
employee(e) ->.
salary(e,v) -> employee(e), float(v).
status(e,s) -> employee(e), string(s).
+salary(e,0.0) <- +status(e,"terminated").
-salary(e,_) <- -employee(e)..
```

Extensions: Meta-Programming

- Meta-programming
 - *Coming soon!*

Invitation

- *Come visit us:*
 - Internships
 - Research funding
 - Conversation
- E-mail: david.zook@logicblox.com
- Website: logicblox.com (but nothing there yet)