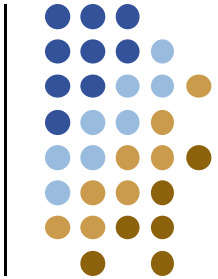
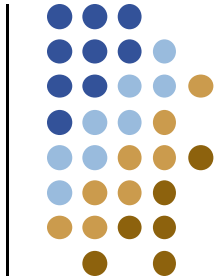
Datalog: Properties



- Limited logic programming
  - SQL with recursion
  - Prolog without complex terms (constructors)
- Captures PTIME complexity class
- Strictly declarative
  - as opposed to Prolog
    - conjunction commutative
    - rules commutative
  - increases algorithm space
    - enables different execution strategies, aggressive optimization

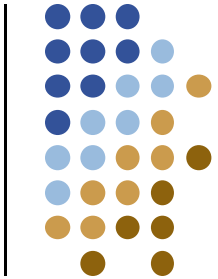
Less programming, more specification



From Algorithms To Specification

Declarative Pointer Analysis Specification

Example 1: Introducing Fields



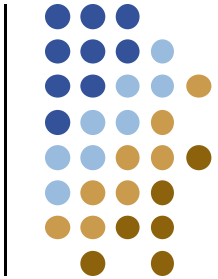
```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).

FieldPointsTo(?baseobj, ?field, ?obj) <-
  StoreField(?from, ?base, ?field),
  VarPointsTo(?base, ?baseobj),
  VarPointsTo(?from, ?obj).
```

field ?field
of object ?baseobj
may point to object ?obj

?base . ?field = ?from

?baseobj ?obj

Example 1: Introducing Fields

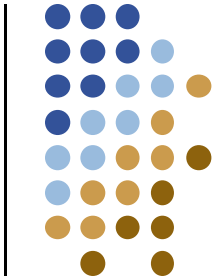


```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).

FieldPointsTo(?baseobj, ?field, ?obj) <-
  StoreField(?from, ?base, ?field),
  VarPointsTo(?base, ?baseobj),
  VarPointsTo(?from, ?obj).

VarPointsTo(?to, ?obj) <-
  LoadField(?to, ?base, ?field),
  FieldPointsTo(?baseobj, ?field, ?obj),
  VarPointsTo(?base, ?baseobj).
```

?baseobj

?to = ?base . ?field

Example 1: Introducing Fields



```
VarPointsTo(?var, ?obj) <-
  AssignObjectAllocation(?var, ?obj).

VarPointsTo(?to, ?obj) <-
  Assign(?from, ?to),
  VarPointsTo(?from, ?obj).

FieldPointsTo(?baseobj, ?field, ?obj) <-
  StoreField(?from, ?base, ?field),
  VarPointsTo(?base, ?baseobj),
  VarPointsTo(?from, ?obj).

VarPointsTo(?to, ?obj) <-
  LoadField(?base, ?field, ?to),
  VarPointsTo(?base, ?baseobj),
  FieldPointsTo(?baseobj, ?field, ?obj).
```

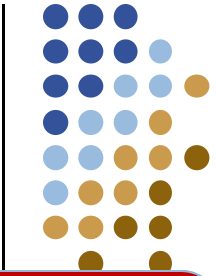
modularity: introducing new mutually recursive dependencies does not change existing rules

?baseobj

↑

?to = ?base . ?field

Example 2: Precise Exception Analysis



Method invocations: propagated exceptions

```
ThrowPointsTo(?caller, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  not exists ExceptionHandler[?objtype, ?invocation],
  Method[?invocation] = ?caller.
```

```
void f() {
  g();
}
```

Method invocations: caught exceptions

```
VarPointsTo(?param, ?obj) <-
  CallGraphEdge(?invocation, ?tomethod),
  ThrowPointsTo(?tomethod, ?obj),
  Type[?obj] = ?objtype,
  ExceptionHandler[?objtype, ?invocation] = ?handler,
  ExceptionHandler:FormalParam[?handler] = ?param.
```

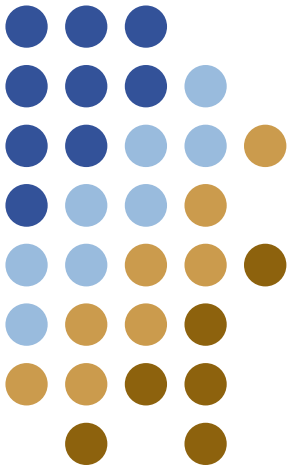
```
void f() {
  try {
    g();
  } catch(E e) {..}
}
```

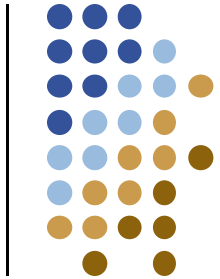
lots of recursion →
fully precise, modular, on-the-fly exception analysis logic

So, What Else Can We Do?

Flow-Sensitive Program Analysis Background

(Dataflow frameworks)

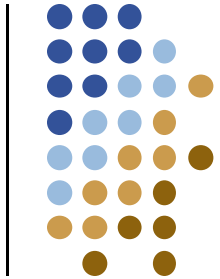




Use Datalog for Dataflow?

and more…

Basic blocks



- Basic block starts:
  - At a label
  - After a jump
- Basic block ends:
  - At a jump
  - Before a label
- **Note**: order still matters

```
label1:
jumpifnot p label2
```

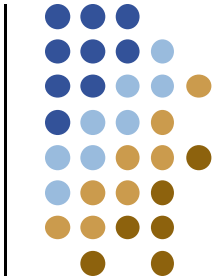
```
x = y + 1
y = 2 * z
jumpifnot c label3
```

```
x = y + z
```

```
label3:
z = 1
jump label1
```

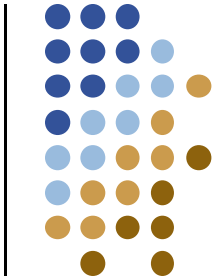
```
label2:
z = x
```

Can You Write A Basic-Blocks Recognition Algorithm?



- Define relations `BeginBasicBlock(?instr)`, `EndBasicBlock(?instr)` using relations
  - `MayPrecede(?i1, ?i2)`
  - `CountMayPrecede[?i] = ?count`
  - `MustFollow[?i1] = ?i2`
    - after instruction `?i1`, instruction `?i2` must follow
    - not the same as "before `?i2`, `?i1` must have executed"
  - `CountMayFollow[?i] = ?count`
    - all expressible from first, just given for convenience

(One) Solution

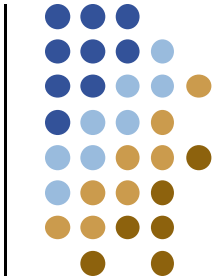


- Basic block starts:
  - At a label
  - After a jump
- Basic block ends:
  - At a jump
  - Before a label

```
BeginBasicBlock(?y) <-
  CountMayPrecede[?y] > 1.
BeginBasicBlock(?y) <-
  MayPrecede(?x,?y),
  CountMayFollow[?x] > 1.
EndBasicBlock(?y) <-
  CountMayFollow[?y] > 1.
EndBasicBlock(?y) <-
  MayPrecede(?y,?x),
  !MustFollow[?x] = ?y.
```

- `MayPrecede(?i1, ?i2)`
- `CountMayPrecede[?i] = ?count`
- `MustFollow[?i1] = ?i2`
  - after instruction `?i1`, instruction `?i2` must follow
- `CountMayFollow[?i] = ?count`

How About Live Ranges?

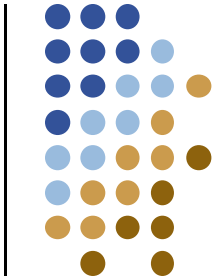


$$\mathbf{in}[l] = (\mathbf{out}[l] - \mathbf{def}[l]) \cup \mathbf{use}[l]$$
$$\mathbf{out}[l] = \mathbf{in}[succ(l)]$$
$$\mathbf{out}[B] = \bigcup_{B' \in succ(B)} \mathbf{in}[B']$$

- `LiveIn(?instr,?var) <-`
  `LiveOut(?instr,?var), !DefVar(?instr,?var).`
- `LiveIn(?instr,?var) <- UseVar(?instr,?var).`
- `LiveOut(?instr,?var) <-`
  `LiveIn(MustFollow[?instr], ?var).`
- `LiveOut(?instr,?var) <-`
  `LiveIn(?instr2,?var),`
  `MayFollow(?instr2,?instr).`

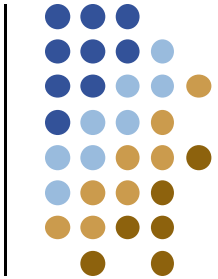
E’

Projects Ideas for This School (A)



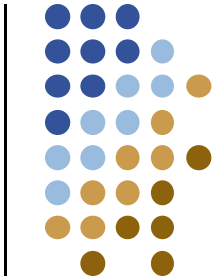
- Take the Datalog Educational System and implement a program analysis
  - http://des.sourceforge.net/
  - use your own imaginary input relations, as long as they are realistic
    - you can assume var-points-to information, etc.
  - test with some hand-written tables that correspond to small example programs
  - ideas for analyses: anything you can come up with or any analysis you find in a book (e.g., good programming practices)

Projects Ideas for This School (B)



- Create a front-end that reads in real programs, parses them and represents them as tables
- Express your tables in comma-separated-values format
  - ideas: Scheme, Javascript, any language you want for which you can derive a parser
- Challenge: represent all the program information meaningfully
  - use good relational data analysis practices (tables with few columns, minimal information overlap)
  - i.e., the issue will not be parsing, it will be database schema design

Projects Ideas for This School



- For an impressive result, combine (A) and (B)
  - produce a full analysis for a non-trivial language
  - this won't be easy because you can no longer assume whatever input you want from (A)
  - but you can still make it meaningful