Static Program Analysis
via Three-Valued Logic

Mooly Sagiv (Tel Aviv),
Thomas Reps (Madison),
Reinhard Wilhelm (Saarbrücken)
<table>
<thead>
<tr>
<th>University of Wisconsin</th>
<th>IBM Research</th>
</tr>
</thead>
<tbody>
<tr>
<td>F. DiMaio</td>
<td>J. Field</td>
</tr>
<tr>
<td>A. Loginov</td>
<td>D. Goyal</td>
</tr>
<tr>
<td>D. Gopan</td>
<td>H. Kolodner</td>
</tr>
<tr>
<td>A. Lal</td>
<td>G. Ramalingam</td>
</tr>
<tr>
<td>A. Mulhern</td>
<td>M. Rodeh</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Tel Aviv University</td>
<td>Clarkson</td>
</tr>
<tr>
<td>G. Arnold</td>
<td>W. Hesse</td>
</tr>
<tr>
<td>N. Rinetzky</td>
<td></td>
</tr>
<tr>
<td>N. Dor</td>
<td>Technical Univ. of Denmark</td>
</tr>
<tr>
<td>R. Shaham</td>
<td>H.R. Nielson</td>
</tr>
<tr>
<td>G. Erez</td>
<td>F. Nielson</td>
</tr>
<tr>
<td>A. Warshavsky</td>
<td></td>
</tr>
<tr>
<td>T. Lev-Ami</td>
<td>IRISA</td>
</tr>
<tr>
<td>E. Yahav</td>
<td>B. Jeannet</td>
</tr>
<tr>
<td>R. Manevich</td>
<td>Weizmann Institute</td>
</tr>
<tr>
<td>G. Yorsh</td>
<td>A. Pnueli</td>
</tr>
<tr>
<td>A. Rabinovich</td>
<td></td>
</tr>
<tr>
<td></td>
<td>MIT</td>
</tr>
<tr>
<td></td>
<td>V. Kuncak</td>
</tr>
<tr>
<td></td>
<td>M. Rinard</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Universität des Saarlandes</td>
<td></td>
</tr>
<tr>
<td>J. Bauer</td>
<td></td>
</tr>
<tr>
<td>J. Reineke</td>
<td></td>
</tr>
<tr>
<td>R. Biber</td>
<td></td>
</tr>
<tr>
<td>J. Herter</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>University of Massachusetts</td>
<td></td>
</tr>
<tr>
<td>N. Immerman</td>
<td></td>
</tr>
</tbody>
</table>
Complications through Pointers

Pointers

• introduce potential for aliasing,
  - which makes many semantics unsound and
  - dependence analysis in compilers difficult,
  - and gives surprising results at memory deallocation.

• Provides reference to anonymous objects - hard to talk about them!

• There may even be an unbounded number of anonymous objects!
• The administrator of the U.S.S. Yorktown’s Standard Monitoring Control System entered 0 into a data field for the Remote Data Base Manager program. That caused the database to overflow and crash all LAN consoles and miniature remote terminal units.
• The Yorktown was dead in the water for about two hours and 45 minutes.
Full Employment for Verification Experts

- A sailor on the U.S.S. Yorktown entered a 0 into a data field in a kitchen-inventory program. That caused the database to overflow and crash all LAN consoles and miniature remote terminal units.
- The Yorktown was dead in the water for about two hours and 45 minutes.
\[
x = 3; \\
y = 1/(x-3);
\]

need to track values other than 0

\[
x = 3; \\
p_x = &x; \\
y = 1/(p_x-3);
\]

need to track pointers

need to track heap-allocated storage

\[
x = 3; \\
p = (int*)malloc(sizeof int); \\
*p = x; \\
q = p; \\
y = 1/(q-3);
\]
Flow-Sensitive Points-To Analysis

\[ a = \&e \]
\[ b = a \]
\[ c = \&f \]
\[ *b = c \]
\[ d = *a \]

\[ p \quad q \quad p = \&q; \quad p \rightarrow q \]
\[ p \quad r_1 \rightarrow q \quad p = q; \quad p \leftarrow r_1 \rightarrow q \]
\[ p \quad s_1 \rightarrow r_1 \rightarrow q \quad p = *q; \quad p \leftarrow s_1 \rightarrow r_1 \rightarrow q \]
\[ p \quad r_2 \rightarrow s_1 \rightarrow r_1 \rightarrow q \quad p \leftarrow r_2 \rightarrow s_1 \rightarrow r_1 \rightarrow q \]
\[ *p = q; \quad p \leftarrow r_1 \rightarrow s_1 \rightarrow q \]

\[ p \quad r_2 \rightarrow s_2 \rightarrow q \quad *p = q; \quad p \leftarrow r_2 \rightarrow s_2 \rightarrow q \]
Flow-Insensitive Points-To Analysis
[Andersen 94, Shapiro & Horwitz 97]

\[ p = \&q; \quad p \rightarrow q \]

\[ p = q; \quad p \rightarrow q \]

\[ p = \ast q; \quad p \rightarrow q \]

\[ \ast p = q; \quad p \rightarrow q \]
Flow-Insensitive Points-To Analysis
What About Malloc?

• Each malloc site ⇒ malloc-site variable

```c
int *p, *q, *r;

a: p = (int*)malloc(sizeof(int));
b: q = (int*)malloc(sizeof(int));
r = p;
```

\[ p \rightarrow \text{malloc$a$} \]
\[ r \rightarrow \text{malloc$b$} \]

\[ q \rightarrow \text{malloc$b$} \]
What About Malloc?

- Each malloc site ⇒ malloc-site variable

```c
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

/* Create a List of length n */
List head;
List *p = &head;
for (int i = 0; i < n; i++) {
    *p = (List)malloc(sizeof(List*));
    p = &((*p)->next);
}
```
Malloc Sites as “Variables”

- **Concrete:** Assignments through pointers
- **Abstract:** *Accumulate edges* ("weak update")
- **Not very accurate**
Shape Analysis
[Jones and Muchnick 1981]

• Characterize *dynamically allocated* data
  - Identify may-alias relationships
  - x points to an acyclic list, cyclic list, tree, dag, ...
  - “disjointedness” properties
    • x and y point to structures that do not share cells
  - show that data-structure invariants hold

• Account for *destructive updates through pointers*
pointer analysis?
points-to analysis?
alias analysis?
shape analysis?

Dynamic storage allocation
Destructive updating through pointers
Applications: Software Tools

• Static detection of memory errors
  - dereferencing NULL pointers
  - dereferencing dangling pointers
  - memory leaks

• Static detection of logical errors
  - Is a data-structure invariant restored?
Applications: Code Optimization

• Parallelization
  - Operate in parallel on disjoint structures

• Software prefetching

• “Compile-time garbage collection”
  - Insert storage-reclamation operations

• Eliminate or move “checking code”
Why is Shape Analysis Difficult?

• Destructive updating through pointers
  - $p \rightarrow \text{next} = q$
  - Produces complicated aliasing relationships

• Dynamic storage allocation
  - No bound on the size of run-time data structures

• Data-structure invariants typically only hold at the beginning and end of operations
  - Want to verify that data-structure invariants are re-established
Outline

• Background on pointer analysis
• Informal introduction to shape analysis
• Shape analysis via 3-valued logic
• A lot more than just shape analysis!
• Extensions, applications
• Relationships with model checking
• Wrapup
Good Abstractions?
Good Abstractions?
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y; y = x; x = x->next;
        y->next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x \rightarrow next;
        y \rightarrow next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y; y = x; x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y; y = x; x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x -> next;
        y -> next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x→next;
        y→next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x->next;
        y->next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x->next;
        y->next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y; y = x; x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x->next;
        y->next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {  
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {  
    List y, t;  
    y = NULL;  
    while (x != NULL) {  
        t = y;  
        y = x;  
        x = x \rightarrow next;  
        y \rightarrow next = t;  
    }  
    return y;  
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        y = x
        x = x \rightarrow next;
        y \rightarrow next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x -> next;
        y -> next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x -> next;
        y -> next = t;
    }
    return y;
}
```c
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
```
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x \rightarrow next;
        y \rightarrow next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x->next;
        x = x->next;
        y->next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        y = x \rightarrow next;
        y \rightarrow next = t;
    }
    return y;
}
typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Example: In-Situ List Reversal

typedef struct list_cell {
    int val;
    struct list_cell *next;
} *List;

List reverse (List x) {
    List y, t;
    y = NULL;
    while (x != NULL) {
        t = y;
        y = x;
        x = x → next;
        y → next = t;
    }
    return y;
}
Idea for a List Abstraction

represents

\[
\begin{align*}
  t & \rightarrow \text{NULL} \\
  y & \rightarrow x \\
  x & \rightarrow \ldots
\end{align*}
\]
Properties of reverse(x)

- On entry, x points to an acyclic list
- On each iteration, x & y point to disjoint acyclic lists
- All the pointer dereferences are safe
- No memory leaks
- On exit, y points to an acyclic list
- On exit, x == NULL
- All cells reachable from y on exit were reachable from x on entry, and vice versa
- On exit, the order between neighbors in the y-list is opposite to their order in the x-list on entry
A ‘Yacc’ for Shape Analysis: TVLA

• Parametric framework
  - Some instantiations ⇒ known analyses
  - Other instantiations ⇒ new analyses

• Applications beyond shape analysis
  - Partial correctness of sorting algorithms
  - Safety of mobile code
  - Deadlock detection in multi-threaded programs
  - Partial correctness of mark-and-sweep gc alg.
  - Correct usage of Java iterators
A ‘Yacc’ for **Static Analysis**: TVLA

- **Parametric framework**
  - Some instantiations ⇒ known analyses
  - Other instantiations ⇒ new analyses

- **Applications beyond shape analysis**
  - Partial correctness of sorting algorithms
  - Safety of mobile code
  - Deadlock detection in multi-threaded programs
  - Partial correctness of mark-and-sweep gc alg.
  - Correct usage of Java iterators
Formalizing “...”

Informal:

Formal:

Summary node
The French Recipe for Program Verification

• Concrete operational semantics
  \[ \Sigma \Sigma \Sigma \rightarrow \Sigma \Sigma \Sigma \]
• Collecting semantics
  \[ \Sigma \Sigma \Sigma \rightarrow 2 \Sigma \Sigma \Sigma \]
• Abstract semantics
  \[ \alpha : 2\Sigma \rightarrow A, \ \gamma : A \rightarrow 2\Sigma, \ \alpha(\gamma(a)) = a, \ \gamma(\alpha(C)) \supseteq C \]
  \[ [st]# : A \rightarrow A \]
  - Upper approximation
    • Sound results
    • But may produce false alarms

Canonical Abstraction

A family of abstractions for use in logic
Using Relations to Represent Linked Lists

<table>
<thead>
<tr>
<th>Relation</th>
<th>Intended Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x(v)$</td>
<td>Does pointer variable $x$ point to cell $v$?</td>
</tr>
<tr>
<td>$y(v)$</td>
<td>Does pointer variable $y$ point to cell $v$?</td>
</tr>
<tr>
<td>$t(v)$</td>
<td>Does pointer variable $t$ point to cell $v$?</td>
</tr>
<tr>
<td>$n(v_1,v_2)$</td>
<td>Does the $n$ field of $v_1$ point to $v_2$?</td>
</tr>
</tbody>
</table>
Using Relations to Represent Linked Lists

\[
x \rightarrow y \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4
\]

<table>
<thead>
<tr>
<th>$n$</th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Variable</th>
<th>$x(u)$</th>
<th>$y(u)$</th>
<th>$t(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Formulas:
Queries for Observing Properties

Are $x$ and $y$ pointer aliases?

$\exists v: x(v) \land y(v)$
Are \( x \) and \( y \) Pointer Aliases?

\[ \exists v: x(v) \land y(v) = 1 \]

<table>
<thead>
<tr>
<th>( u )</th>
<th>( x(u) )</th>
<th>( y(u) )</th>
<th>( t(u) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>( u_1 )</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>( u_2 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( u_3 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>( u_4 )</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

\[
\begin{array}{cccc}
 n & u_1 & u_2 & u_3 & u_4 \\
 \hline 
 u_1 & 0 & 1 & 0 & 0 \\
 u_2 & 0 & 0 & 1 & 0 \\
 u_3 & 0 & 0 & 0 & 1 \\
 u_4 & 0 & 0 & 0 & 0 \\
\end{array}
\]
Predicate-Update Formulas for “y = x”

- $x'(v) = x(v)$
- $y'(v) = x(v)$
- $t'(v) = t(v)$
- $n'(v_1, v_2) = n(v_1, v_2)$
Predicate-Update Formulas for “$y = x$”

$y'(v) = x(v)$

$x \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4$

<table>
<thead>
<tr>
<th>$x(u)$</th>
<th>$y(u)$</th>
<th>$t(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$n$</th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Predicate-Update Formulas for “x = x → n”

• \( x'(v) = \exists v_1: x(v_1) \land n(v_1, v) \)
• \( y'(v) = y(v) \)
• \( t'(v) = t(v) \)
• \( n'(v_1, v_2) = n(v_1, v_2) \)
Predicate-Update Formulas for “x = x \rightarrow n”

\[ x'(v) = \exists v_1: x(v_1) \land n(v_1, v) \]

\[
x' = x \times (v) = \begin{cases} 
x(v_1) & \text{if } v_1 = u_1 \\
0 & \text{otherwise}
\end{cases}
\]

\[
y = y(v) = \begin{cases} 
y(v_1) & \text{if } v_1 = u_2 \\
0 & \text{otherwise}
\end{cases}
\]

\[
t = t(v) = n(v) = \begin{cases} 
t(v_1) & \text{if } v_1 = u_3 \\
0 & \text{otherwise}
\end{cases}
\]

<table>
<thead>
<tr>
<th>u_1</th>
<th>u_2</th>
<th>u_3</th>
<th>u_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>n</th>
<th>u_1</th>
<th>u_2</th>
<th>u_3</th>
<th>u_4</th>
</tr>
</thead>
<tbody>
<tr>
<td>u_1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>u_2</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>u_3</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>u_4</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Predicate-Update Formulas for “y → n = t”

- \( x'(v) = x(v) \)
- \( y'(v) = y(v) \)
- \( t'(v) = t(v) \)
- \( n'(v_1,v_2) = \neg y(v_1) \land n(v_1,v_2) \lor y(v_1) \land t(v_2) \)
Why is Shape Analysis Difficult?

• Destructive updating through pointers
  - $p \rightarrow \text{next} = q$
  - Produces complicated aliasing relationships
• Dynamic storage allocation
  - No bound on the size of run-time data structures
• Data-structure invariants typically only hold at the beginning and end of operations
  - Need to verify that data-structure invariants are re-established
### Two- vs. Three-Valued Logic

<table>
<thead>
<tr>
<th>Two-valued logic</th>
<th>Three-valued logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>{0,1}</td>
</tr>
<tr>
<td>1</td>
<td>{0} \subset {0,1}</td>
</tr>
<tr>
<td></td>
<td>{1} \subset {0,1}</td>
</tr>
</tbody>
</table>
# Two- vs. Three-Valued Logic

## Two-valued logic

<table>
<thead>
<tr>
<th>$\land$</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\lor$</th>
<th>1</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

## Three-valued logic

<table>
<thead>
<tr>
<th>$\land$</th>
<th>{1}</th>
<th>{0,1}</th>
<th>{0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>{1}</td>
<td>{0,1}</td>
<td>{0}</td>
</tr>
<tr>
<td>{0,1}</td>
<td>{1}</td>
<td>{0,1}</td>
<td>{0}</td>
</tr>
<tr>
<td>{0}</td>
<td>{0}</td>
<td>{0}</td>
<td>{0}</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$\lor$</th>
<th>{1}</th>
<th>{0,1}</th>
<th>{0}</th>
</tr>
</thead>
<tbody>
<tr>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
<td>{1}</td>
</tr>
<tr>
<td>{0,1}</td>
<td>{1}</td>
<td>{0,1}</td>
<td>{0,1}</td>
</tr>
<tr>
<td>{0}</td>
<td>{1}</td>
<td>{0,1}</td>
<td>{0}</td>
</tr>
</tbody>
</table>
# Two- vs. Three-Valued Logic

<table>
<thead>
<tr>
<th>Two-valued logic</th>
<th>Three-valued logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

- **Two-valued logic** has values 0 and 1.
- **Three-valued logic** includes the value 0, 1, and an additional value not specified in the table.
Two- vs. Three-Valued Logic

<table>
<thead>
<tr>
<th>Two-valued logic</th>
<th>Three-valued logic</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>( \frac{1}{2} )</td>
<td>( \frac{1}{2} )</td>
</tr>
</tbody>
</table>

Diagram:

```
0 \quad \frac{1}{2} \quad 1
0 \sqsubseteq \frac{1}{2}       1 \sqsubseteq \frac{1}{2}
```
### Boolean Connectives [Kleene]

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1/2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>∨</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>1/2</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1/2</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>∧</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1/2</td>
<td>0</td>
<td>1/2</td>
<td>1/2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1/2</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1/2</td>
<td>1/2</td>
<td>1/2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Canonical Abstraction

\[ x \rightarrow u_1 \rightarrow u_2 \rightarrow u_3 \rightarrow u_4 \rightarrow x \]

\[
\begin{array}{c|cc}
\text{x(u)} & \text{y(u)} \\
\hline
u_1 & 1 & 0 \\
u_2 & 0 & 0 \\
u_3 & 0 & 0 \\
u_4 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
\text{x(u)} & \text{y(u)} \\
\hline
u_1 & 1 & 0 \\
u_{234} & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|ccc}
\text{n} & \text{u}_1 & \text{u}_2 & \text{u}_3 & \text{u}_4 \\
\hline
\text{u}_1 & 0 & 1 & 0 & 0 \\
\text{u}_2 & 0 & 0 & 1 & 0 \\
\text{u}_3 & 0 & 0 & 0 & 1 \\
\text{u}_4 & 0 & 0 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
\text{n} & \text{u}_1 & \text{u}_{234} \\
\hline
\text{u}_1 & 0 & 1/2 \\
\text{u}_{234} & 0 & 1/2 \\
\end{array}
\]
Canonical Abstraction

\[
\begin{array}{c|cc}
 & x(u) & y(u) \\
\hline
u_1 & 1 & 0 \\
\hline
u_2 & 0 & 0 \\
u_3 & 0 & 0 \\
u_4 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
 & x(u) & y(u) \\
\hline
u_1 & 1 & 0 \\
\hline
u_{234} & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
 & u_1 & u_{234} \\
\hline
u_1 & 0 & 1/2 \\
\hline
u_{234} & 0 & 1/2 \\
\end{array}
\]
Canonical Abstraction

\[
\begin{array}{c|cc}
  x(u) & y(u) \\
\hline
  u_1 & 1 & 0 \\
  u_2 & 0 & 0 \\
  u_3 & 0 & 0 \\
  u_4 & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
  x(u) & y(u) \\
\hline
  u_1 & 1 & 0 \\
  u_{234} & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
  x(u) & y(u) \\
\hline
  u_1 & 1 & 0 \\
  u_{234} & 0 & 0 \\
\end{array}
\]

\[
\begin{array}{c|cc}
  x(u) & y(u) \\
\hline
  u_1 & 1 & 0 \\
  u_{234} & 0 & 1/2 \\
\end{array}
\]

\[
\begin{array}{c|c|c}
  = & u_1 & u_{234} \\
\hline
  u_1 & 1 & 0 \\
  u_{234} & 0 & 1/2 \\
\end{array}
\]
Canonical Abstraction

- Partition the individuals into equivalence classes based on the values of their unary predicates
- Collapse other predicates via $\sqcup$
Canonical Abstraction vs. Predicate Abstraction

\[ \exists v: x(v) \quad 1 \]

\[ \exists v: y(v) \quad 0 \]

\[ \exists v: x(v) \land \exists v: \neg x(v) \quad 1 \]

\[ \exists v_1, v_2, v: n(v_1, v) \land n(v_2, v) \land (v_1 \neq v_2) \quad 0 \]
Abstract **Value** Obtained via Canonical Abstraction

Abstract **Value** Obtained via Predicate Abstraction
Abstract Value Obtained via Canonical Abstraction

- The predicate-abstraction abstract domain is a **special case** of the canonical-abstraction abstract domain:
  - map everything to a single summary individual
  - retain only the nullary predicates
Property-Extraction Principle

- Questions about a family of two-valued stores can be answered conservatively by evaluating a formula in a three-valued store.
- Formula evaluates to 1
  - \( \Rightarrow \) formula holds in every store in the family
  - 😊
- Formula evaluates to 0
  - \( \Rightarrow \) formula does not hold in any store in the family
  - 😊
- Formula evaluates to 1/2
  - \( \Rightarrow \) formula may hold in some; not hold in others
  - 😒 😞
Are $x$ and $y$ Pointer Aliases?

Yes

$\exists v: x(v) \land y(v)$

$1 \land 1$

$1$
Is Cell $u$ Heap-Shared?

$\exists v_1, v_2: n(v_1, u) \land n(v_2, u) \land v_1 \neq v_2$

$1 \land 1 \land 1 \land 1 \land 1$
Is Cell u Heap-Shared?

\[ \exists v_1, v_2: n(v_1, u) \land n(v_2, u) \land v_1 \neq v_2 \]

\[ \frac{1}{2} \land \frac{1}{2} \land 1 \]

\[ \frac{1}{2} \]
The Embedding Theorem

\[ \exists \nu: x(\nu) \land y(\nu) \]

No

No

No

Maybe
The Embedding Theorem

• If a structure $B$ can be embedded in a structure $S$ by an onto function $f$, such that basic predicates are preserved, i.e.,
  \[ p^B(u_1, \ldots, u_k) \equiv p^S(f(u_1), \ldots, f(u_k)) \]

• Then every formula $\varphi$ is preserved:
  - If $[\varphi] = 1$ in $S$, then $[\varphi] = 1$ in $B$
  - If $[\varphi] = 0$ in $S$, then $[\varphi] = 0$ in $B$
  - If $[\varphi] = 1/2$ in $S$, then $[\varphi]$ could be 0 or 1 in $B$
Embedding
**Canonical Abstraction:**

An Embedding Whose Result is of **Bounded Size**

---

**Diagram:**

- **Input:** $x$
- **Nodes:** $u_1, u_2, u_3, u_4$ (arrows from $x$ to $u_1$, $u_1$ to $u_2$, $u_2$ to $u_3$, $u_3$ to $u_4$)
- **Output:** $x$ with $u_1$ and $u_{234}$

**Tables:**

1. **$x(u)$ and $y(u)$**

<table>
<thead>
<tr>
<th></th>
<th>$x(u)$</th>
<th>$y(u)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_4$</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

2. **$n$ and $u_1, u_2, u_3, u_4$**

<table>
<thead>
<tr>
<th></th>
<th>$u_1$</th>
<th>$u_2$</th>
<th>$u_3$</th>
<th>$u_4$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>$u_2$</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>$u_3$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>$u_4$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

3. **$n$ and $u_1, u_{234}$**

<table>
<thead>
<tr>
<th></th>
<th>$u_1$</th>
<th>$u_{234}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$u_1$</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>$u_{234}$</td>
<td>0</td>
<td>1/2</td>
</tr>
</tbody>
</table>
Predicate-Update Formulas for "y = x"

\[ y'(v) = x(v) \]

**Old:**

\[ x \quad u_1 \quad u \]

**New:**

\[ x \quad u_1 \quad u \]

<table>
<thead>
<tr>
<th></th>
<th>[ x(u) ]</th>
<th>[ y(u) ]</th>
<th>[ t(u) ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ u_1 ]</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>[ u ]</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>[ n ]</th>
<th>[ u_1 ]</th>
<th>[ u ]</th>
</tr>
</thead>
<tbody>
<tr>
<td>[ u_1 ]</td>
<td>0</td>
<td>1/2</td>
</tr>
<tr>
<td>[ u ]</td>
<td>0</td>
<td>1/2</td>
</tr>
</tbody>
</table>
Predicate-Update Formulas for “$x = x \rightarrow n$”

$$x'(v) = \exists v_1 : x(v_1) \land n(v_1, v)$$

**Old:**

$x$ \(\rightarrow\) \(u_1\)

$y$ \(\rightarrow\) \(u_1\)

$x \rightarrow y

$\begin{array}{c|c|c}
  x(u) & y(u) & t(u) \\
  \hline
  u_1 & 0 & 1 \\
  1/2 & 0 & 0 \\
  u & 1/2 & 0 \\
\end{array}$

**New:**

$y$ \(\rightarrow\) \(u_1\)

$x$ \(\rightarrow\) \(u_1\)

$\begin{array}{c|c|c}
  n & u_1 & u \\
  \hline
  u_1 & 0 & 1/2 \\
  u & 0 & 1/2 \\
\end{array}$
x != NULL

x = x -> next

y = x

y = x

x = x -> next

y -> next = t

return y
x != NULL

T = y

Y = x

X = X -> next

Y -> next = T

Return y
Naïve Transformer ($x = x \rightarrow n$)

$$x'(v) = \exists v_1: x(v_1) \land n(v_1, v)$$
Cyclic versus Acyclic Lists

1 → 31 → 71 → 91 → u → u_1 → x

x → 1 → 31 → 71 → 91 → u → u_1 → x
How Are We Doing?

- Conservative ☺
- Convenient ☺
- But not very precise ☹
  - Advancing a pointer down a list loses precision
  - Cannot distinguish an acyclic list from a cyclic list
The Instrumentation Principle

- Increase precision by storing the truth-value of some chosen formulas
Is Cell \( u \) Heap-Shared?

\[ \exists v_1, v_2: n(v_1, u) \land n(v_2, u) \land v_1 \neq v_2 \]
Example: Heap Sharing

\[ is(v) = \exists v_1, v_2: n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2 \]
Example: Heap Sharing

\[ is(v) = \exists v_1, v_2: n(v_1, v) \land n(v_2, v) \land v_1 \neq v_2 \]
Example: Cyclicity

c(v) = \exists v_1: n(v,v_1) \land n^*(v_1,v)
Is Cell \( u \) Heap-Shared?

\[ \exists v_1, v_2: n(v_1, u) \land n(v_2, u) \land v_1 \neq v_2 \]

\[ \frac{1}{2} \land \frac{1}{2} \land 1 \]

No!

\[ \frac{1}{2} \rightarrow \text{Maybe} \]
Formalizing “...”

Informal:

Formal:
Formalizing “...”

Informal:

\[ x \rightarrow \cdots \rightarrow y \rightarrow \cdots \]

Formal:

\[ x \rightarrow \cdots \rightarrow y \rightarrow \cdots \]
Formalizing “...”

Informal:

\[ x \rightarrow \cdots \]

\[ y \rightarrow \cdots \]

Formal:

reachable from variable \( x \)

reachable from variable \( y \)
Formalizing “. . .”

Informal:

Formal:
Updating Auxiliary Information

\[
\begin{align*}
\text{y} &= \text{NULL} \\
\{p[x], p[y]\} &\rightarrow \{r[x], r[y]\} \\
\{r[x], r[y]\} &\rightarrow \{p[t_1], r[x], r[y]\} \\
\{p[t_1], r[x], r[y]\} &\rightarrow \{r[t_1], r[x], r[y]\}
\end{align*}
\]

\[
\begin{align*}
\text{x} &\rightarrow \{p[x]\} \\
\{p[x]\} &\rightarrow \{r[x]\} \\
\{r[x]\} &\rightarrow \{p[t_1], r[x]\} \\
\{p[t_1], r[x]\} &\rightarrow \{r[t_1], r[x]\}
\end{align*}
\]
Automatic Generation of Update Formulas for Instrumentation Predicates

The Instrumentation Principle

- Increase precision by storing the truth-value of some chosen formulas

Where do we get the predicate-update formulas to update the extra predicates?
Automatic Generation of Update Formulas for Instrumentation Predicates

• Originally, user provided
  - Update formulas for core predicates: $\tau_{c,\text{st}}(v)$
  - Definitions of instrumentation predicates: $\psi_p(v)$, $\mu_{p,\text{st}}(v)$
  - Update formulas for instrumentation predicates $\frac{d\psi}{d\tau}$

Now: $\mu_{p,\text{st}}$ created from $\psi_p$ and the $\tau_{c,\text{st}}$. Consistently defined?
Useful Instrumentation Predicates

- doubly-linked(v)
- reachable-from-variable-x(v)
- acyclic-along-dimension-d(v)
- tree(v)
- dag(v)
- AVL trees:
  - balanced(v), left-heavy(v), right-heavy(v)
  - ... but not via height arithmetic

Need FO + TC
Materialization

Formal:

Informal:

Formal:
Naïve Transformer \( (x = x \to n) \)
Best Transformer \((x = x \rightarrow n)\)
“Focus”-Based Transformer \((x = x \rightarrow n)\)

- Evaluate update formulas
- Focus\((x \rightarrow n)\) "Partial \(\gamma\)"
- \(\alpha\)
Best Transformers via Decision Procedures

Outline

• Background on pointer analysis
• Informal introduction to shape analysis
• Shape analysis via 3-valued logic
• A lot more than just shape analysis!
• Extensions, applications
• Relationships with model checking
• Wrapup
Threads and Concurrency

A memory configuration:

- thread1 atStart
- thread2 atStart
- thread4 atStart
- lock1 isAcquired
- thread3 inCritical

Connections:
- csLock from thread1 to lock1
- csLock from thread2 to lock1
- csLock from thread4 to lock1
- heldBy from lock1 to thread3

Dotted box indicates unstarted threads.
Threads and Concurrency

An abstract memory configuration:

thread
atStart

lock1
isAcquired

heldBy

thread
inCritical

csLock

csLock
Threads and Concurrency

Static analysis using 3-valued logic provides a way to explore the (abstract) memory configurations that can arise.
Shape + Numeric Abstractions

\[ X \rightarrow u \rightarrow u' \rightarrow u'' \rightarrow u''' \]

Position in list

\[ (1,2,3,4) \]

\[ u_1 \rightarrow u_2 \rightarrow \{1\} \times [2,4] \]
Shape + Numeric Abstractions

y = x → next

\( x \rightarrow u_1 \rightarrow u_2 \rightarrow u_2.0 \rightarrow u_2.1 \rightarrow u_1 \)

\( u_1 \rightarrow (1,2) \)

\( u_2 \rightarrow (1,4) \)

\( u_2.0 \rightarrow (1,2,3) \)

\( u_2.1 \rightarrow (1,2,4) \)

\( \{1\} \times [2,4] \)

\( \{1\} \times \{2\} \times [3,4] \)
Example: Sortedness

\[ \text{inOrder}(v) = \forall v_1: n(v, v_1) \rightarrow (\langle v \rangle \leq \langle v_1 \rangle) \]

\[
\begin{array}{cccc}
1 & \rightarrow & 51 & \rightarrow & 71 & \rightarrow & 91 \\
\text{inOrder} = 1 & \text{inOrder} = 1 & \text{inOrder} = 1 & \text{inOrder} = 1 \\
\end{array}
\]

\[ \text{sorted} = \forall v: \text{inOrder}(v) \]

\[
\begin{array}{cccc}
x & \rightarrow & \square & \rightarrow & \square \\
\text{inOrder} = 1 & \text{inOrder} = 1 \\
\end{array}
\]

Yes
Example: Sortedness

\[ \text{inOrder}(v) = \forall v_1: n(v, v_1) \rightarrow (\langle v \rangle \leq \langle v_1 \rangle) \]

\[ x \rightarrow 1 \rightarrow 51 \rightarrow 45 \rightarrow 91 \]

\[ \text{inOrder} = 1 \quad \text{inOrder} = 0 \quad \text{inOrder} = 1 \quad \text{inOrder} = 1 \]

\[ \text{sorted} = \forall v: \text{inOrder}(v) \]

\[ \text{No} \]

\[ \text{inOrder} = 1 \quad \text{inOrder} = 0 \quad \text{inOrder} = 1 \]
TVLA System

• Input (FO+TC)
  - Concrete operational semantics
  - Definitions of instrumentation predicates
  - Abstraction predicates
  - Program (as transition system)
  - Definitions of error conditions

• Output
  - Warnings (possible errors)
  - The 3-valued structures at every node

• New version available soon
## Model Checking vs. TVLA

<table>
<thead>
<tr>
<th><strong>Model checking</strong></th>
<th><strong>TVLA</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>Determine properties of a transition system</td>
<td>Determine properties of a transition system</td>
</tr>
<tr>
<td>State-space exploration</td>
<td>State-space exploration</td>
</tr>
<tr>
<td>State labels: Propositions</td>
<td>State labels: 1&lt;sup&gt;st&lt;/sup&gt;-order structures</td>
</tr>
<tr>
<td>BDDs represent commonalities</td>
<td>3-valued structures represent commonalities</td>
</tr>
<tr>
<td>Properties checked: Formulas in temporal logic</td>
<td>Properties checked: Formulas in FO+TC</td>
</tr>
</tbody>
</table>
Clarke, Grumberg, Jha, Lu, Veith, CAV 00
Clarke, Grumberg, Jha, Lu, Veith, CAV 00

\[\text{CTL}^* \rightarrow \text{ACTL}^*\]

Sagiv, Reps, Wilhelm, POPL 99

\[\left[\text{FO + TC}\right]_2 \rightarrow \left[\text{FO + TC}\right]_3\]
Why is Shape Analysis Difficult?

• Destructive updating through pointers
  - \( p \rightarrow \text{next} = q \)
  - Produces complicated aliasing relationships
  - Track aliasing using 3-valued structures

• Dynamic storage allocation
  - No bound on the size of run-time data structures
  - Canonical abstraction \( \Rightarrow \) finite-sized 3-valued structures

• Data-structure invariants typically only hold at the beginning and end of operations
  - Need to verify that data-structure invariants are re-established
  - Query the 3-valued structures that arise at the exit
What to Take Away

- A ‘yacc’ for static analysis based on logic
- Broad scope of potential applicability
  - Not just linkage properties: *predicates are not restricted to be links!*
  - Discrete systems in which a relational (+ numeric) structure evolves
  - Transition: evolution of relational + numeric state
Canonical Abstraction

A family of abstractions for use in logic
Questions?