An Overview on Program Analysis

Mooly Sagiv


Tel Aviv University
640-6706

Textbook: Principles of Program Analysis
F. Nielson, H. Nielson, C.L. Hankin
Prerequisites

- Compiler construction course
Course Requirements

- Course Notes 15%
- Assignments 35%
- Exam 50%
Class Notes

- Prepare a document with (word, latex)
  - Original material covered in class
  - Explanations
  - Questions and answers
  - Extra examples
  - Self contained

- Send class notes by Sunday night to msagiv@tau

- Incorporate changes

- Available next class
Dynamic Program Analysis

- Automatically infer properties of the program while it is being executed

- Examples
  - Dynamic Array bound checking
    » Purify
    » Valgrind
  - Memory leaks
  - Likely invariants
    » Daikon
Static Analysis

- Automatic inference of static properties which hold on every execution leading to a program location
Example Static Analysis Problem

- Find variables with constant value at a given program location

Example program

```c
int p(int x){
    return x *x  ;
}

void main()
{
    int z;
    if (getc())
        z = p(6) + 8;
    else z = p(-7) -5;
    printf (z);
} 44
```
Recursive Program

```c
int x
void p(a) {
    read (c);
    if c > 0 {
        a = a -2;
        p(a);
        a = a + 2;
    }
    x = -2 * a + 5;
    print (x);
}
void main {
    p(7);
    print(x);
}
```
Memory Leakage

List reverse(Element *head)
{
    List rev, n;
    rev = NULL;
    while (head != NULL) {
        n = head -> next;
        head -> next = rev;
        head = n;
        rev = head;
    }
    return rev;
}
Memory Leakage

Element* reverse(Element *head)
{
    Element *rev, *n;
    rev = NULL;
    while (head != NULL) {
        n = head -> next;
        head -> next = rev;
        rev = head;
        head = n;
    }
    return rev; }

👍 No memory leaks
A Simple Example

```c
void foo(char *s )
{
    while ( *s != ' ' )
        s++;
    *s = 0;
}
```

Potential buffer overrun:
offset(s) \geq alloc(base(s))
A Simple Example

void foo(char *s) @require string(s)
{
    while ( *s != ' ' && *s != 0)
        s++;
    *s = 0;
}
int check_authentication(char *password) {
    int auth_flag = 0;
    char password_buffer[16];

    strcpy(password_buffer, password);
    if(strcmp(password_buffer, "brillig") == 0) auth_flag = 1;
    if(strcmp(password_buffer, "outgrabe") == 0) auth_flag = 1;
    return auth_flag;
}

int main(int argc, char *argv[]) {
    if(check_authentication(argv[1])) {
        printf("\n-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
        printf(" Access Granted.\n");
        printf("-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-=-\n");
    } else
        printf("\nAccess Denied.\n");
}
Example Static Analysis Problem

- Find variables which are *live* at a given program location
- Used before set on some execution paths from the current program point
A Simple Example

\begin{align*}
a &:= 0 \\
b &:= a + 1 \\
c &:= c + b \\
a &:= b + 2 \\
c &\geq N \\
c
\end{align*}
Compiler Scheme

source-program → Scanner → tokens → Parser → AST → Semantic Analysis → AST → Code Generator → IR → Static analysis → IR +information → Transformations
Other Example Program Analyses

- Reaching definitions
- Expressions that are ``available''
- Dead code
- Pointer variables never point into the same location
- Points in the program in which it is safe to free an object
- An invocation of virtual method whose address is unique
- Statements that can be executed in parallel
- An access to a variable which must be in cache
- Integer intervals
- The termination problem
The Program Termination Problem

- Determine if the program terminates on all possible inputs
Program Termination
Simple Examples

\[
z := 3; \quad \text{while } z > 0 \text{ do }
\]

\[
\text{while } z > 0 \text{ do }
\]

\[
\quad \text{if } (x == 1) z := z + 3; \quad \text{if } (x == 1) z := z - 1;
\]

\[
\quad \text{else } z := z + 1; \quad \text{else } z := z - 2;
\]
Program Termination
Complicated Example

while (x != 1) do {
    if (x % 2) == 0 {
        x := x / 2;
    } else {
        x := x * 3 + 1;
    }
}
Summary Program Termination

- Very hard in theory
- Most programs terminate for simple reasons
- But termination may involve proving intricate program invariants
- Tools exist
  - MSR Terminator
  - ARMC http://www.mpi-sws.org/~rybal/armc/
The Need for Static Analysis

◆ Compilers
  – Advanced computer architectures
  – High level programming languages
    (functional, OO, garbage collected, concurrent)

◆ Software Productivity Tools
  – Compile time debugging
    » Stronger type Checking for C
    » Array bound violations
    » Identify dangling pointers
    » Generate test cases
    » Generate certification proofs

◆ Program Understanding
Challenges in Static Analysis

- Non-trivial
- Correctness
- Precision
- Efficiency of the analysis
- Scaling
C Compilers

- The language was designed to reduce the need for optimizations and static analysis.
- The programmer has control over performance (order of evaluation, storage, registers).
- C compilers nowadays spend most of the compilation time in static analysis.
- Sometimes C compilers have to work harder!
Software Quality Tools

- Detecting hazards (lint)
  - Uninitialized variables
    ```c
    a = malloc();
    b = a;
    cfree(a);
    c = malloc();
    if (b == c)
      printf(“unexpected equality”);
    ```
- References outside array bounds
- Memory leaks (occurs even in Java!)
Foundation of Static Analysis

- Static analysis can be viewed as interpreting the program over an “abstract domain”
- Execute the program over larger set of execution paths
- Guarantee sound results
  - Every identified constant is indeed a constant
  - But not every constant is identified as such
Example Abstract Interpretation
Casting Out Nines

- Check soundness of arithmetic using 9 values
  0, 1, 2, 3, 4, 5, 6, 7, 8
- Whenever an intermediate result exceeds 8, replace by the sum of its
digits (recursively)
- Report an error if the values do not match
- Example query “123 * 457 + 76543 = 132654$?”
  - Left 123*457 + 76543= 6 * 7 + 7 =6 + 7 = 4
  - Right 3
  - Report an error

- Soundness
  
  \[(10a + b) \mod 9 = (a + b) \mod 9\]
  \[(a+b) \mod 9 = (a \mod 9) + (b \mod 9)\]
  \[(a*b) \mod 9 = (a \mod 9) * (b \mod 9)\]
Even/Odd Abstract Interpretation

- Determine if an integer variable is even or odd at a given program point
Example Program

while (x != 1) do { /* x=? */
    if (x % 2) == 0
        /* x=E */
        { x := x / 2; } /* x=? */
    else
        /* x=O */
        { x := x * 3 + 1; /* x=E */
         assert (x % 2 == 0); }
}
/* x=O*/
Abstract Interpretation

Concrete
Sets of stores

Abstract
Descriptors of sets of stores
Odd/Even Abstract Interpretation

All concrete states

- \{x: x \in \text{Even}\} \quad \{-2, 1, 5\}
- \{0, 2\}
- \{0\} \quad \{2\}
Odd/Even Abstract Interpretation

All concrete states

{x: x ∈ Even} {-2, 1, 5}

{0, 2}

{0} {2}

∅
Odd/Even Abstract Interpretation

All concrete states

\{ x : x \in \text{Even} \} \{ -2, 1, 5 \}

\{ 0, 2 \}

\emptyset

\gamma \alpha

\alpha

\oplus

E

O
Example Program

while (x != 1) do {
    if (x % 2) == 0 {
        x := x / 2;
    } else {
        x := x * 3 + 1;
        assert (x % 2 == 0);
    }
}

/* x=O */
/* x=E */
(Best) Abstract Transformer

Concrete Representation

Abstract Representation

Concretization

Operational Semantics

St

Concrete Representation

Abstract Representation

St

Abstract Semantics

Abstraction
Concrete and Abstract Interpretation

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>…</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>…</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>…</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>…</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>*</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>…</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>10</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>6</td>
<td>9</td>
<td>12</td>
<td>15</td>
</tr>
<tr>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
<td>…</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>’+’</th>
<th>?</th>
<th>O</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>O</td>
<td>?</td>
<td>E</td>
<td>O</td>
</tr>
<tr>
<td>E</td>
<td>?</td>
<td>O</td>
<td>E</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>’*’</th>
<th>?</th>
<th>O</th>
<th>E</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>E</td>
</tr>
<tr>
<td>O</td>
<td>?</td>
<td>O</td>
<td>E</td>
</tr>
<tr>
<td>E</td>
<td>E</td>
<td>E</td>
<td>E</td>
</tr>
</tbody>
</table>
## Runtime vs. Static Testing

<table>
<thead>
<tr>
<th></th>
<th>Runtime</th>
<th>Abstract</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Effectiveness</strong></td>
<td>Missed Errors</td>
<td>False alarms</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Locate rare errors</td>
</tr>
<tr>
<td><strong>Cost</strong></td>
<td>Proportional to program’s execution</td>
<td>Proportional to program’s size</td>
</tr>
<tr>
<td></td>
<td>No need to efficiently handle rare cases</td>
<td>Can handle limited classes of programs and still be useful</td>
</tr>
</tbody>
</table>
Abstract (Conservative) interpretation

Set of states \rightarrow \text{abstract representation} \rightarrow \text{abstract semantics} \rightarrow \text{statement } s \rightarrow \text{Set of states}

\text{concretization} \rightarrow \text{Set of states} \rightarrow \text{abstract representation} \rightarrow \text{Abstract semantics} \rightarrow \text{abstraction}
Example rule of signs

- Safely identify the sign of variables at every program location
- Abstract representation \{P, N, ?\}
- Abstract (conservative) semantics of *

<table>
<thead>
<tr>
<th>#</th>
<th>P</th>
<th>N</th>
<th>?</th>
</tr>
</thead>
<tbody>
<tr>
<td>P</td>
<td>P</td>
<td>N</td>
<td>?</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>P</td>
<td>?</td>
</tr>
<tr>
<td>?</td>
<td>?</td>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
Abstract (conservative) interpretation

Operational semantics

\[ x := x \times y \]

Abstract semantics

\[ x := x \times \# y \]

Concretization

\{..., <\text{176},-2>\,...\}\n
Abstraction

\{..., <\text{-88},-2>\,...\}\
Example rule of signs (cont)

- Safely identify the sign of variables at every program location
- Abstract representation \{P, N, ?\}
- \(\alpha(C) = \) if all elements in \(C\) are positive then return \(P\)
  else if all elements in \(C\) are negative then return \(N\)
  else return \(?\)
- \(\gamma(a) = \) if \(a==P\) then
  return \{0, 1, 2, \ldots\}
  else if \(a==N\)
  return \{-1, -2, -3, \ldots,\}
  else return \(Z\)
Example Constant Propagation

- Abstract representation set of integer values and an extra value “?” denoting variables not known to be constants
- Conservative interpretation of +

<table>
<thead>
<tr>
<th></th>
<th>?</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
Example Constant Propagation (Cont)

- Conservative interpretation of *

<table>
<thead>
<tr>
<th>*#</th>
<th>?</th>
<th>0</th>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
<td>0</td>
<td>?</td>
<td>?</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>?</td>
<td>0</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>?</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
</tbody>
</table>
Example Program

x = 5;
y = 7;
if (getc())
    y = x + 2;
z = x + y;
Example Program (2)

if (getc())
    x = 3; y = 2;
else
    x = 2; y = 3;

z = x + y;
Undecidability Issues

- It is undecidable if a program point is reachable in some execution
- Some static analysis problems are undecidable even if the program conditions are ignored
while (getc()) {
    if (getc()) x_1 = x_1 + 1;
    if (getc()) x_2 = x_2 + 1;
    ...
    if (getc()) x_n = x_n + 1;
}

y = truncate \left( \frac{1}{1 + p^2(x_1, x_2, ..., x_n)} \right)

/* Is y=0 here? */
Coping with undecidability

- Loop free programs
- Simple static properties
- Interactive solutions
- Conservative estimations
  - Every enabled transformation cannot change the meaning of the code but some transformations are not enabled
  - Non optimal code
  - Every potential error is caught but some “false alarms” may be issued
Analogies with Numerical Analysis

- Approximate the exact semantics
- More precision can be obtained at greater computational costs
Violation of soundness

- Loop invariant code motion
- Dead code elimination
- Overflow
  \((x+y)+z \neq (x + (y+z))\)
- Quality checking tools may decide to ignore certain kinds of errors
Abstract interpretation cannot be always homomorphic (rules of signs)

Operational semantics

Abstract semantics

<\text{-}8, 7> \rightarrow \text{x := x+y} \rightarrow <\text{-}1, 7>

<\text{N, P}> \rightarrow \text{abstraction} \rightarrow \text{x := x+#y} \rightarrow <? \text{P}> \rightarrow \text{abstraction} \rightarrow <\text{N, P}>
Local Soundness of Abstract Interpretation

Operational semantics

abstraction

statement

statement#

Abstract semantics

abstraction

⇒
Optimality Criteria

- Precise (with respect to a subset of the programs)
- Precise under the assumption that all paths are executable (statically exact)
- Relatively optimal with respect to the chosen abstract domain
- Good enough
## Relation to Program Verification

<table>
<thead>
<tr>
<th>Program Analysis</th>
<th>Program Verification</th>
</tr>
</thead>
<tbody>
<tr>
<td>✷ Fully automatic</td>
<td>✷ Requires specification and loop invariants</td>
</tr>
<tr>
<td>✷ Applicable to a programming language</td>
<td>✷ Program specific</td>
</tr>
<tr>
<td>✷ Can be very imprecise</td>
<td>✷ Relative complete</td>
</tr>
<tr>
<td>✷ May yield false alarms</td>
<td>✷ Provide counter examples</td>
</tr>
<tr>
<td></td>
<td>✷ Provide useful documentation</td>
</tr>
<tr>
<td></td>
<td>✷ Can be mechanized using theorem provers</td>
</tr>
</tbody>
</table>
Origins of Abstract Interpretation

- [Naur 1965] The Gier Algol compiler
  “A process which combines the operators and operands of the source text in the manner in which an actual evaluation would have to do it, but which operates on descriptions of the operands, not their value”

- [Reynolds 1969] Interesting analysis which includes infinite domains (context free grammars)

- [Syntzoff 1972] Well foundedness of programs and termination

- [Cousot and Cousot 1976,77,79] The general theory

- [Kamm and Ullman, Kildall 1977] Algorithmic foundations

- [Tarjan 1981] Reductions to semi-ring problems

- [Sharir and Pnueli 1981] Foundation of the interprocedural case

- [Allen, Kennedy, Cock, Jones, Muchnick and Schwartz]
Static Driver Verifier

Rules

Precise API Usage Rules (SLIC)

Environment model

Defects

100% path coverage

Driver’s Source Code in C
"Things like even software verification, this has been the Holy Grail of computer science for many decades but now in some very key areas, for example, driver verification we’re building tools that can do actual proof about the software and how it works in order to guarantee the reliability." Bill Gates, April 18, 2002. Keynote address at WinHec 2002
SLAM Dataflow

C program

SLIC rules

C2BP

Boolean Program

BEBOP

Abstract Trace

NEWTON

Concrete Program Trace
The ASTRÉE Static Analyzer

Patrick Cousot
Radhia Cousot
Jérôme Feret
Laurent Mauborgne
Antoine Miné
Xavier Rival

ENS France
Goals

- Prove absence of errors in safety critical C code
- ASTRÉE was able to prove completely automatically the absence of any RTE in the primary flight control software of the Airbus A340 fly-by-wire system
  - a program of 132,000 lines of C analyzed
A Simple Example

/* boolean.c */
typedef enum {FALSE = 0, TRUE = 1} BOOLEAN;

void main () {
    unsigned int x, y;
    BOOLEAN b;
    while (1) {
        ...
        b = (x == 0); /* b == 0 → x > 0 */

        if (!b)
            /* x > 0 */ { y = 1 / x; };
        ...
    }
}
Another Example

/* float-error.c */

void main ()
{
    float x, y, z, r;
    x = 1.0000000019e+38;
    y = x + 1.0e21;
    z = x - 1.0e21;
    r = y - z;
    printf("%f\n", r);
}

% gcc float-error.c % ./a.out 0.00000
Another Example

```c
void main()
{
    float x, y, z, r;
    scanf("%f", &x);
    if ((x < -1.0e38) || (x > 1.0e38)) return;
    /* -1.0e38 \leq x \leq 1.0e38 */
    y = x + 1.0e21;
    z = x - 1.0e21;
    r = y - z;
    /* r == 2.0 \times 10^{21} */
    printf("%f\n", r);
}
```
TVLA: A system for generating shape analyzers

Tal Lev-Ami
Alexey Loginov
Roman Manevich
Example: Concrete Interpretation

x = NULL

F T

T = malloc(..);

t→next = x;

x = t

return x
Example: Shape Analysis

x = NULL

F       T

<table>
<thead>
<tr>
<th>t = malloc(..);</th>
<th>t → next = x;</th>
<th>x = t</th>
<th>return x</th>
</tr>
</thead>
</table>

- F: 
  - t = malloc(..);
  - t → next = x;
  - x = t
  - return x

- T: 
  - empty
  - t → x
  - t → n → x
  - t → n → n → x
  - t → n → n → n → x

T/F
TVLA: A parametric system for Shape Analysis

- A research tool
- Parameters
  - Concrete Semantics
    - States
    - Interpretation Rules
  - Abstraction
  - Transformers
- Iteratively compute a probably sound solution
- Specialize the shape analysis algorithm to class of programs
typedef struct list_cell {
    int data;
    struct list_cell *n;
} *List;

List InsertSort(List x) {
    List r, pr, rn, l, pl; r = x; pr = NULL;
    while (r != NULL) {
        l = x; rn = r -> n; pl = NULL;
        while (l != r) {
            if (l -> data > r -> data) {
                pr -> n = rn; r -> n = l;
                if (pl = = NULL) x = r;
                else pl -> n = r;
                r = pr;
                break;
            }
            pl = l; l = l -> n;
        }
        pr = r; r = rn;
    }
    assert sorted[x,n];
    //assert perm[x, n, x, n];
    return x;
}