Program needs a representation for the analysis

- Sequences of characters
- Hard to work with (not even for machine)
- Convert to structured representation

Abstract syntax tree

An abstract syntax tree (AST) is a finite, labeled, directed tree, where the internal nodes are labeled by operators, and the leaf nodes represent the operands of the operators.
Abstract syntax tree

- ASTs are widely used in compilers (e.g., gcc) when parsing source code.
- ASTs are abstract
  - They don’t contain all information in the program
    - E.g., spacing, comments, brackets, parentheses
  - AST has many similar forms
    - e.g., for, while, repeat...until
  - ASTs are not good for binary code
- Need simpler representation for analysis (at least, for dataflow analysis)

Control Flow Graph

A directed graph where
- Each node represents a statement
- Edges represent control flow

CFG consists of
- A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next
- Edges represent potential flow of control between BBs

\[ \text{CFG} = \langle V, E, \text{Entry}, \text{Exit} \rangle \]
- \( V \) = Vertices, nodes (BBs)
- \( E = \text{Edges, potential flow of control} \)
  \( E \subseteq V \times V \)
- \( \text{Entry, Exit} \in V \), unique entry and exit
An Example of CFG

- BB - A maximal sequence of consecutive instructions such that inside the basic block an execution can only proceed from one instruction to the next.

```
1: sum=0
2: i=1
3: while (i<N) do
  4: i=i+1
  5: sum=sum+i
endwhile
6: print(sum)
```

Program Dependence Graph: Data Dependency

- S data depends on T if there exists a control flow path from T to S and a variable is defined at T and then used at S.

Program Dependence Graph: Dominator

- A block \(M\) dominates a block \(N\) if every path from the entry that reaches block \(N\) has to pass through block \(M\). By definition, every node dominates itself. The entry block dominates all blocks.

- A block \(M\) immediately dominates block \(N\) if \(M\) dominates \(N\), and there is no intervening block \(P\) such that \(M\) dominates \(P\) and \(P\) dominates \(N\). In other words, \(M\) is the last dominator on all paths from entry to \(N\). Not all blocks have immediate dominators (e.g. entry block).
Overview

Static Program Representations

Dynamic Program Representations

Summary

---

Dominator and I-Dominator Examples

1: `sum=0`  
2: `i=1`  
3: `while (i<N) do`  
4: `i=i+1`  
5: `sum=sum+i`  
6: `print(sum)`

DOM(6)={1,2,3,6} IDOM(6)=3

---

Program Dependence Graph: Control Dependence

A node (basic block) Y is control-dependent on another X iff X directly determines whether Y executes

- There exists a path from X to Y s.t. every node in the path other than X and Y is post-dominated by Y
- X is not strictly post-dominated by Y

CD(5)=3

---

Program Dependence Graph: Post Dominator

Post-Dominator

In the reverse direction, block M post-dominates block N if every path from N to the exit has to pass through block M.

- The exit block post-dominates all blocks.

Immediate Post-Dominator

It is said that a block M immediately post-dominates block N if M post-dominates N, and there is no intervening block P such that M post-dominates P and P post-dominates N. In other words, M is the last post-dominator on all paths from entry to N.

PDOM(5)={3,5,6} IPDOM(5)=3

---

Post-Dominator and I-Post-Dominator Examples

1: `sum=0`  
2: `i=1`  
3: `while (i<N) do`  
4: `i=i+1`  
5: `sum=sum+i`  
6: `print(sum)`

PDOM(5)={3,5,6} IPDOM(5)=3
Overview

Static Program Representations
- Abstract Syntax Tree
- Control Flow Graph
- Program Dependence Graph
- Points-to Graph
- Call Graph

Dynamic Program Representations
- Control Flow Trace, Address and Value Traces
- Dynamic Dependence Graph (Dynamic Slicing)

Summary

Points-to Graph

**Aliases**
Two expressions that denote the same memory location.

**Introduced by**
- pointers
- call-by-reference
- array indexing
- C unions

**Points-to Graph**

At a program point, compute a set of pairs of the form \( p \rightarrow x \), where \( p \) MAY/MUST points to \( x \).

```
m(p){
    \( r = \text{new C}(); \)
    \( p->f = r; \)
    \( t = \text{new C}(); \)
    \( \text{if (...)} \)
    \( q=p; \)
    \( r->f = t; \)
}
```
Points-to Graph

at a program point, compute a set of pairs of the form \( p \rightarrow x \), where \( p \) MAY/MUST points to \( x \).

\[
m(p)\{
    r = \text{new} \ C();
    p\rightarrow f = r;
    \quad \Rightarrow \quad t = \text{new} \ C();
    \quad \text{if (\ldots)}
    \quad q = p;
    \quad r\rightarrow f = t;
\}
\]

\( p\rightarrow f \rightarrow f \) and \( t \) are aliases

Outline

1. Overview
2. Static Program Representations
   - Abstract Syntax Tree
   - Control Flow Graph
   - Program Dependence Graph
   - Points-to Graph
   - Call Graph
3. Dynamic Program Representations
   - Control Flow Trace, Address and Value Traces
   - Dynamic Dependence Graph (Dynamic Slicing)
4. Summary
Overview

Static Program Representations
- Abstract Syntax Tree
- Control Flow Graph
- Program Dependence Graph
- Points-to Graph
- Call Graph

Dynamic Program Representations
- Control Flow Trace, Address and Value Traces
- Dynamic Dependence Graph (Dynamic Slicing)

Summary

Outline

1 Overview
2 Static Program Representations
3 Dynamic Program Representations
4 Summary

Control flow trace, address and value traces

N=2:
1: sum=0
2: i=1
3: while (i<N) do
4: i=i+1
5: sum=sum+i
6: print(sum)

<...xi,...> x is a program point, xi is an execution point
<...804805727,804805a29,...>
What is a slice?

```c
void main () {
    int I=0;
    int sum=0;
    while (I<N) {
        sum=add(sum,I);
        I=add(I,1);
    }
    print(sum);
    print(I);
}
```

**Slice of v at S is the set of statements involved in computing v's value at S.**

*Mark Weiser, 1982*

**Why is a static slice imprecise?**

- All possible program paths
  - S1: x=...
  - S2: x=...
  - LT: ...=x

- Use of Pointers - static alias analysis is very imprecise
  - S1: a=...
  - S2: b=...
  - LT: ...=p

- Use of function pointers - hard to know which function is called, conservative expectation results in imprecision

**Dynamic Slicing**

- Korel and Laski, 1988
- Dynamic slicing makes use of all information about a particular execution of a program and computes the slice based on an execution history (trace)
  - Trace consists control flow trace and memory reference trace
- A dynamic slice query is a triple
- Smaller, more precise, more helpful to the user
Dynamic Slicing Example

1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5: a = a+1
6: else
6: b = a*2
7: done
8: z = a+b
9: print(z)

For input N=2

1: b=0
2: a=2
3: for i= 1 to N do
4: if ((i++)%2==1) then
5: a = a+1
5: a = a+1
6: else
6: b = a*2
6: b = a*2
7: done
7: done
8: z = a+b
8: z = a+b
9: print(z)
9: print(z)

Issues about Dynamic Slicing

- Precision - perfect
- Running history - very big (GB)
- Algorithm to compute dynamic slice - slow and very high space requirement.

Comments

- Want to know more?
  - Frank Tip's survey paper (1995)
- Static slicing is very useful for static analysis
  - Code transformation, program understanding, etc.
  - Points-to analysis is the key challenge
- Dynamic slicing
  - Precise
    - Good for vulnerability identification.
  - Solution space is much larger.
  - There exist hybrid techniques.
How are dynamic slices computed?

- Execution traces
  - control flow trace – dynamic control dependences
  - memory reference trace – dynamic data dependences
- Construct a dynamic dependence graph
- Traverse dynamic dependence graph to compute slices

References

5. A brief survey of program slicing http://dl.acm.org/citation.cfm?id=1050865