



Compiler

Static Analysis: Overview

© Copyright 2005, Matthew B. Dwyer and Robby. The syllabus and lectures for this course are copyrighted materials and may not be used in other course settings outside University of Nebraska-Lincoln and Kansas State University in their current form or modified form without express written permission of one of the copyright holders. During this course, students are prohibited from selling notes to or being paid for taking notes by any person or commercial firm without the express written permission of one of the copyright holders.



Static Analysis: Overview

- *Static*

- ☐ performed at compile time, i.e., without executing the program

- *Analysis*

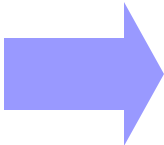
- ☐ Determination of interesting properties of the run-time behavior of the program

- The goal is to

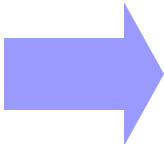
- ☐ enable optimizations
- ☐ can also be used for bug finding, testing, debugging,
...

Static Analysis: Motivation – Optimization

The optimization:

<code>x = 1;</code>		<code>y = 1;</code>
<code>y = x;</code>		

is unsound if `x` is used in a later instruction:

<code>x = 1;</code>		<code>y = 1;</code>
<code>y = x;</code>		
<code>...</code>		<code>...</code>
<code>z = x;</code>		<code>z = x;</code>



Static Analysis: Motivation — Checking for Unreachable Code

The following code are accepted by phases up to type checking:

```
int foo(int x) {  
    return x;  
    x = x + 1;  
    x = x - 1;  
}
```

```
void bar(boolean b) {  
    if (b) { return; }  
    else { return; }  
    return;  
}
```

We'd like to notify users about the above “ill-formed” code (similar to Eclipse IDE)!



Static Analysis: Motivation — Checking for Uninitialized Vars.

The following code are accepted by phases up to type checking:

```
void baz() {  
    int x;  
    int y;  
    y = x;  
}
```

```
int bazz(A a) {  
    int x;  
    if (a == null) { return 0; }  
    return a.x + x;  
}
```

We'd like to notify users about the above ill-formed code (similar to Eclipse IDE)!



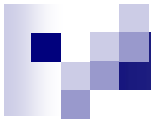
Static Analysis: Motivation – Checking for Null Dereference

The following code is well-formed:

```
void bazzz(A a, int x) {  
    if (a == null) {  
        a.x = 5;  
    }  
}
```

```
void bazzzz(A a, int x) {  
    A tmp = null;  
    if (x > 0) { tmp = a; }  
    tmp.x = 5;  
}
```

However, we'd like to notify users that the above code may raise exception (not done in Eclipse IDE)!



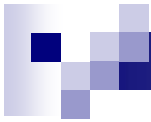
Interesting Program Properties

- are nearly all undecidable, so the analysis computes a *conservative* approximation:
 - if we say *yes*, then the property definitely holds;
 - if we say *no*, then the property may or may not hold;
 - only the *yes* answer will help us to perform the optimization;
 - a trivial analysis will say *no* always; so
 - the art is to say *yes* as often as possible.
- Properties need not be simply *yes* or *no*, in which case the notion of *approximation* is more subtle.



When is Analysis Performed?

- Static analysis may take place:
 - ☐ at the source code level;
 - ☐ at some intermediate level; or
 - ☐ at the machine code level.
- Static analysis may look at:
 - ☐ basic blocks only;
 - ☐ an entire function (intraprocedural); or
 - ☐ the whole program (interprocedural).
- In each case, we are maximally pessimistic at the boundaries.
- The precision and cost of an analysis rises as we include more information.



Static Analysis: Topics

- Control Flow Graph (CFG)
- Reaching Definition Analysis
- Monotonic Data Flow (MDF) Framework
 - Reaching Definition Analysis
 - Live Variable Analysis
 - a.k.a. Dead Variable Analysis
 - Null Dereference Analysis