

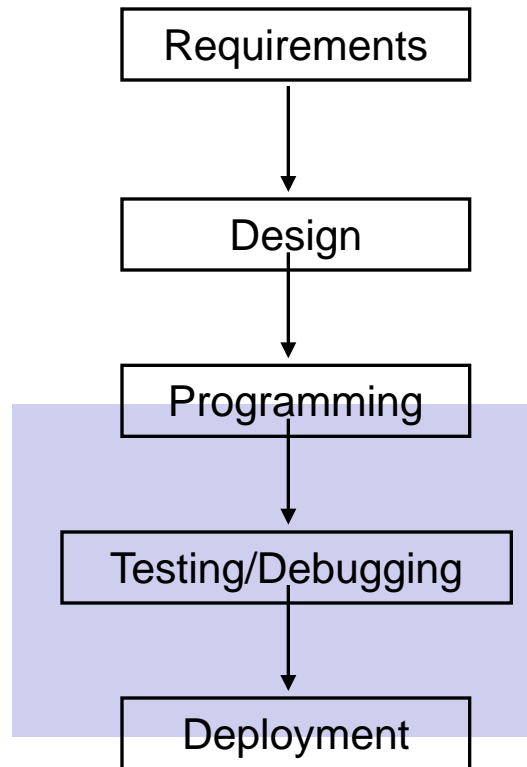
Analysis and Verification Techniques

Lecture 1: Introduction

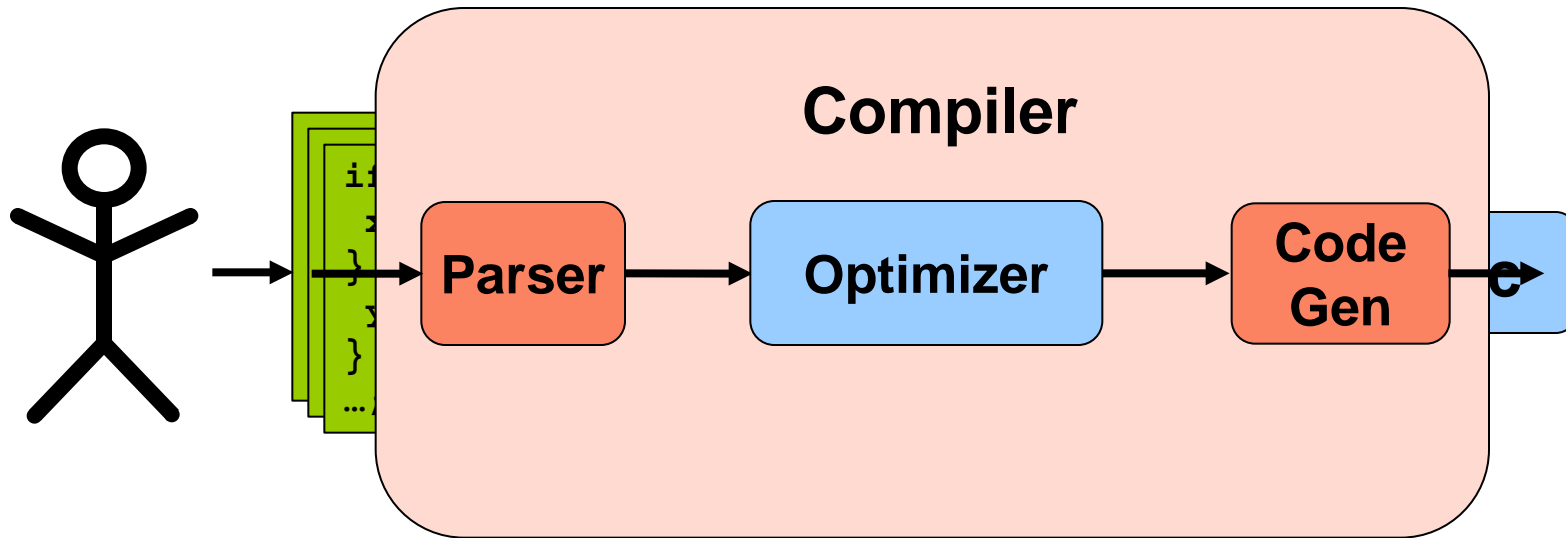
Static Program Analysis

- **Testing:** manually checking a property for some execution paths
- **Model checking:** automatically checking a property for all execution paths
- **Static analysis** consists of automatically discovering properties of a program that hold for all possible execution paths of the program

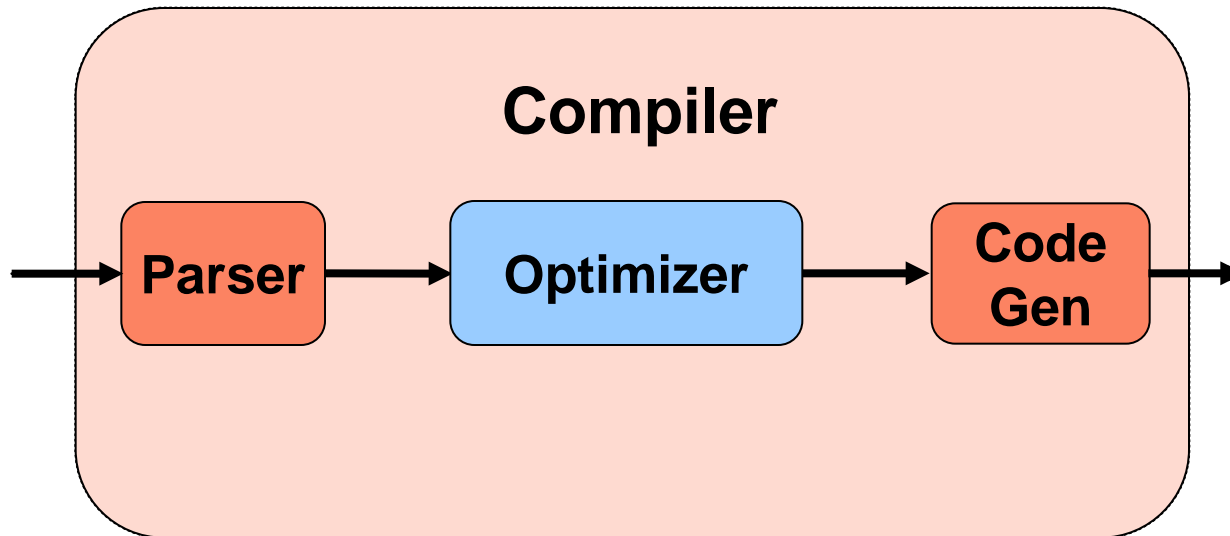
What is this course about?



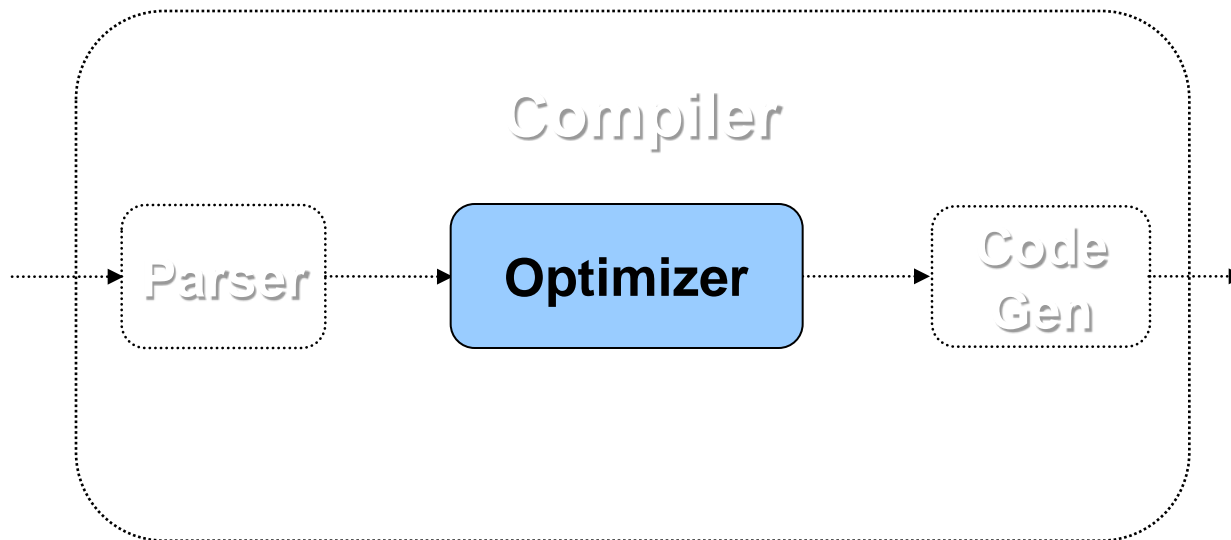
Let's look at a compiler



Let's look at a compiler

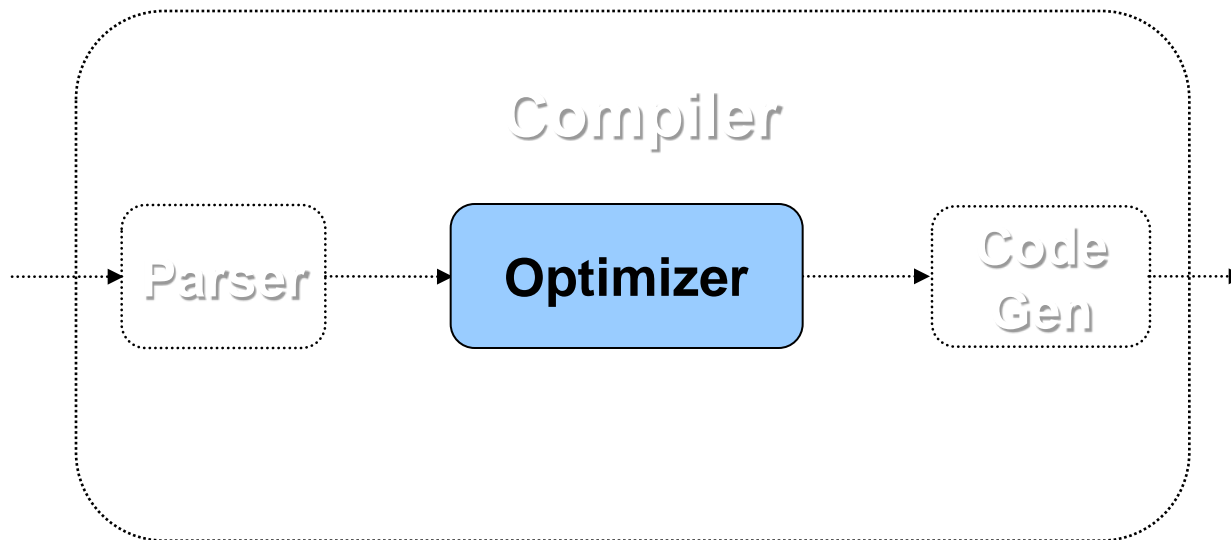


What does an optimizer do?



1. Analysis
2. Transformations

What does an optimizer do?



1. Compute information about a program
2. Use that information to perform program transformations
(with the goal of improving some metric, e.g. performance)

Example: Escape Analysis

- Consider these two C procedures:

```
typedef int A[10000];
```

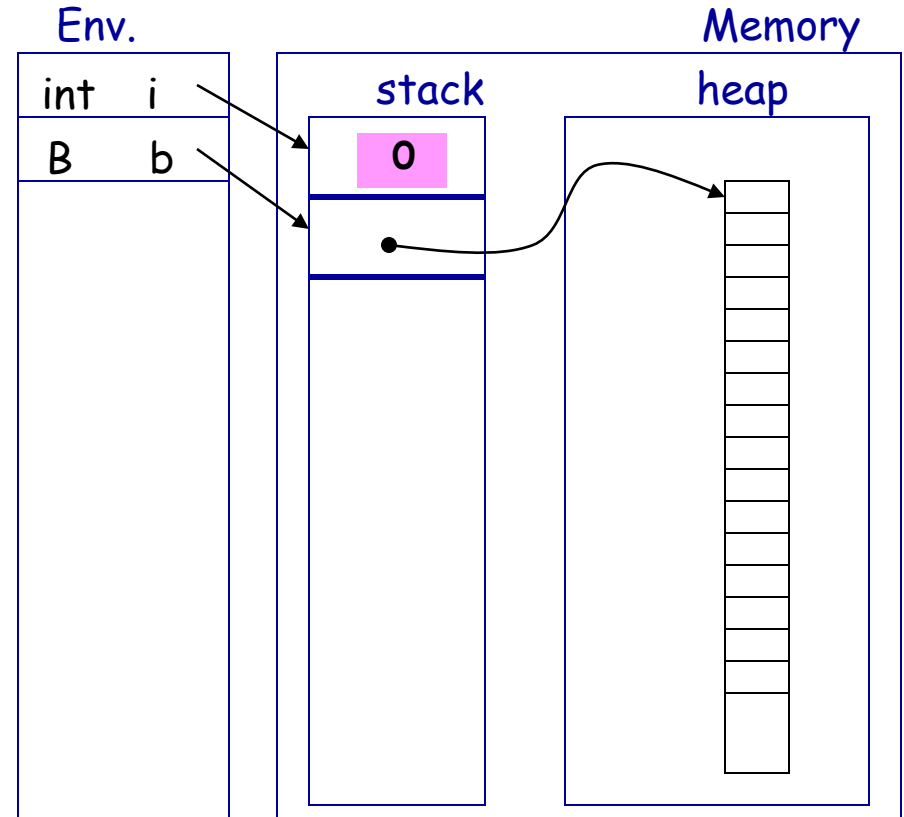
```
typedef A* B;
```

```
void execute1(){  
    int i;  
    B b;  
    b = (A*) malloc(sizeof(A));  
    for (i=0; i<10000; i++)  
        (*b)[i]=0;  
    free(b);  
}
```

```
void execute2(){  
    int i;  
    A a;  
    for (i=0; i<10000; i++)  
        a[i]=0;  
}
```



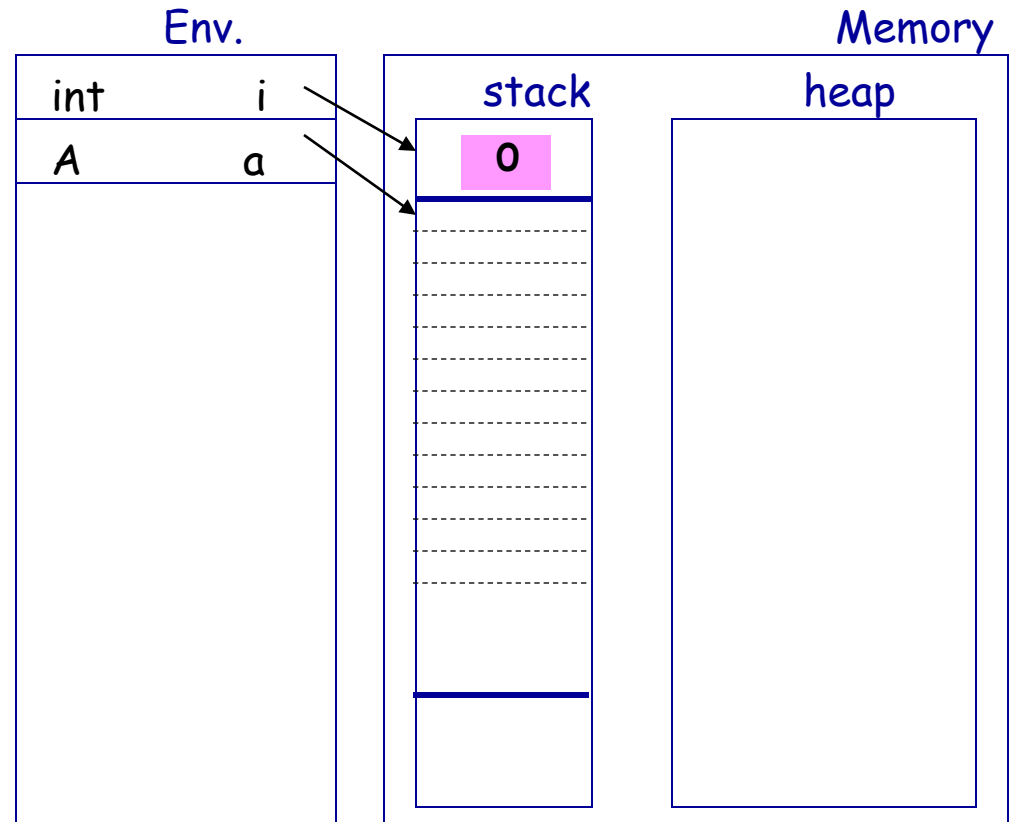
```
typedef int A[10000];  
typedef A* B;  
void execute1(){  
    int i;  
    B b;  
    b = (A*) malloc(sizeof(A));  
    for (i=0; i<10000; i++)  
        (*b)[i]=0;  
    free(b);  
}
```



- The array `b` is allocated in the heap, and then de-allocated

```
typedef int A[10000];

void execute2(){
    int i;
    A a;
    for (i=0; i<10000; i++)
        a[i]=0;
}
```



- The array **a** is in the stack

Memory allocation has a cost...

```
for (i=0; i<1.000.000; i++)  
    execute1();                // about 90 sec
```

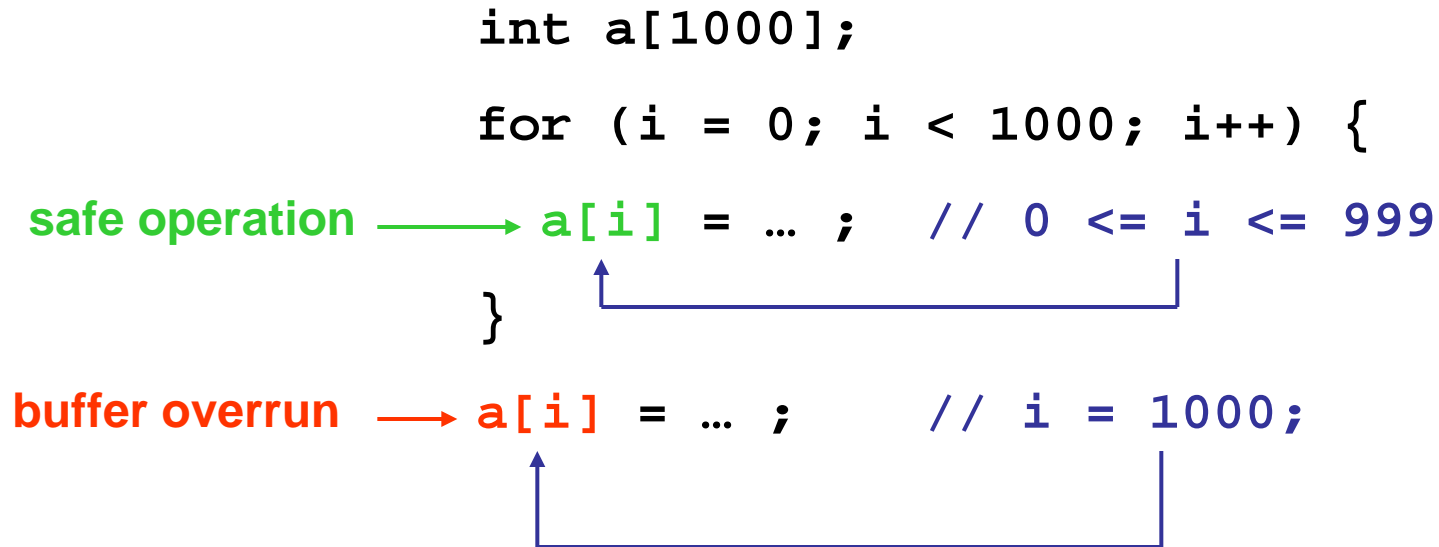
```
for (i=0; i<1.000.000; i++)  
    execute2();                // about 35 sec
```

- The second program is three times faster than the first one
- If we can predict that a dynamic variable does not escape out of the procedure in which it is allocated, we can use a static variable instead, getting a more efficient program

Program Verification

- Check that every operation of a program will never cause an error (division by zero, buffer overrun, deadlock, etc.)
- Example:

```
int a[1000];  
for (i = 0; i < 1000; i++) {  
    safe operation → a[i] = ... ;    // 0 ≤ i ≤ 999  
}  
buffer overrun → a[i] = ... ;    // i = 1000;
```



Why Is Software Verification Important?

- One of the most prominent challenges for IT.
 - Software bugs cost the U.S. economy about **\$59.5** billion each year (**0.6%** of the GDP) [NIST 02].
- Security is becoming a necessity.
 - The worldwide economic loss caused by all forms of overt attacks is **\$226** billion.
- Software defects make programming so painful.
- Stories
 - The Role of Software in Spacecraft Accidents (<http://sunnyday.mit.edu/papers/jsr.pdf>)
 - History's Worst Software Bugs (<http://www.wired.com/software/coolapps/news/2005/11/69355>)

Software bugs may cause big troubles...



On 4 June 1996, the maiden flight of the Ariane 5 launcher ended in a failure. Only about 40 seconds after initiation of the flight sequence, at an altitude of about 3700 m, the launcher veered off its flight path, broke up and exploded. »

The failure of the Ariane 5.01 was caused by the complete loss of guidance and attitude information 37 seconds after start of the main engine ignition sequence, 30 seconds after lift-off.

This loss of information was due to specification and design errors in the software of the inertial reference system.

Software bugs may cause big troubles...



A conversion from 64-bit floating point to 16 bit integer with a value larger than possible. The overflow caused a hardware trap



Remarks by Bill Gates

17th Annual ACM Conference on Object-Oriented
Programming, Seattle, Washington, November 8, 2002

“... When you look at a big commercial software company like Microsoft, there's actually as much testing that goes in as development.

We have as many testers as we have developers.

Testers basically test all the time, and developers basically are involved in the testing process about half the time...



Remarks by Bill Gates

17th Annual ACM Conference on Object-Oriented
Programming, Seattle, Washington, November 8, 2002

“... We've probably changed the industry we're in. We're not in the software industry; we're in the testing industry, and writing the software is the thing that keeps us busy doing all that testing.”

“...The test cases are unbelievably expensive; in fact, there's more lines of code in the test harness than there is in the program itself. Often that's a ratio of about three to one.”

The goal – reliable software systems

- Quality dilemma: quality / features / time
- More efficient methods for test and verification are always needed
 - No ‘silver-bullet’ when it comes to testing.
 - **Formal verification** is on the rise...

Testing / Formal Verification

A very crude dichotomy:

Testing
Correct with respect to the <i>set of test inputs</i> , and reference system
Easy to perform
Dynamic

In practice:

Many types of testing,

Testing / Formal Verification

A very crude dichotomy:

Testing	Formal Verification
Correct with respect to the <i>set of test inputs</i> , and reference system	Correct with respect to <i>all inputs</i> , with respect to a <i>formal specification</i>
Easy to perform	Decidability problems, Computational problems,
Dynamic	Static

In practice:

Many types of testing,
Many types of formal verification.

Our approach

Formal = based on rigorous **mathematical logic** concepts.

Semantics-based = based in a formal specification of the “**meaning**” of the program

Fundamental Limit: Undecidability

Rice Theorem

Any non-trivial semantic property of programs is undecidable.

Classical Example: Termination

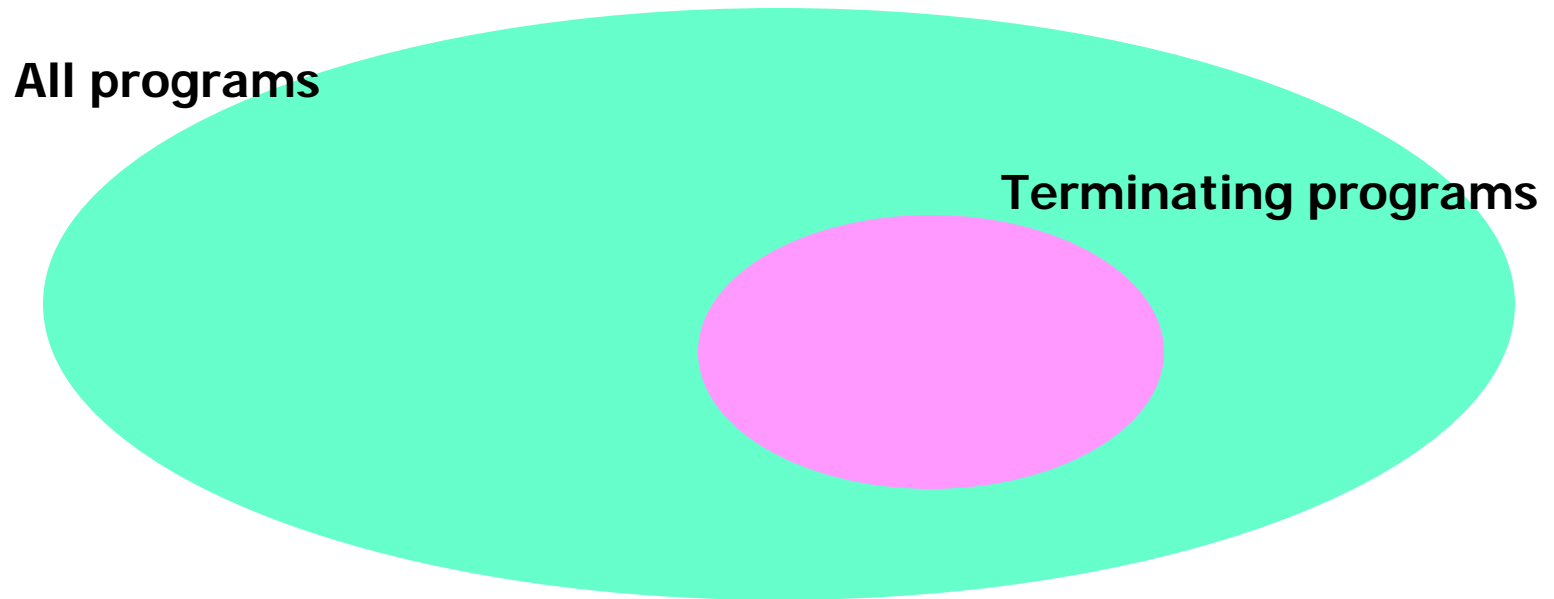
There exists no algorithm which can solve the halting problem: given a description of a program as input, decide whether the program terminates or loops forever.

Incompleteness of Program Analysis

- Discovering a sufficient set of properties for checking every operation of a program is an undecidable problem!
- **False positives:**
operations that are safe in reality but which cannot be decided safe or unsafe from the properties inferred by static analysis.

Example

- We all know that the halting problem is not decidable



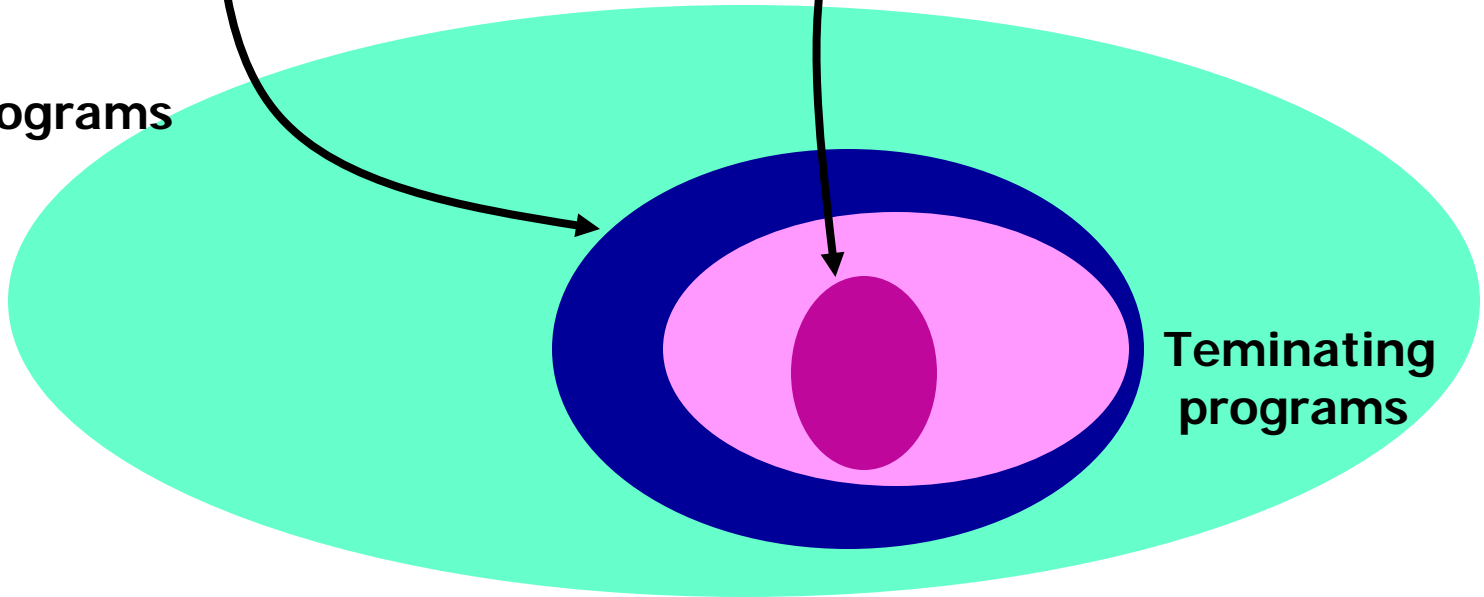
Under- and Over-approximation

Programs that *may* terminate

Program that *surely* terminate

All programs

Terminating programs



Soundness and Completeness

- A **sound** static analysis **overapproximates** the behaviors of the program.
A sound static analyzer is guaranteed to identify all violations of our property, but may also report some “false alarms”, or violations of the property that cannot actually occur.
- A **complete** static analysis **underapproximates** the behaviors of the program.
Any violation of our property reported by a complete static analyzer corresponds to an actual violation of the property, but there is no guarantee that all actual violations of \square will be reported
- Note that when a sound static analyzer reports no errors, our program is guaranteed not to violate \square the property. This is a powerful guarantee. As a result, most static analysis tools choose to be sound rather than complete.

Precision versus Efficiency

Precision: number of program operations that can be decided safe or unsafe by the analyzer.

- Precision and computational complexity are strongly related
- Tradeoff precision/efficiency: limit in the average precision and scalability of a given analyzer
- Greater precision and scalability is achieved through **specialization**

Specialization

- Tailoring the program analyzer algorithms for a specific class of programs (flight control commands, digital signal processing, etc.)
- Precision and scalability is guaranteed for this class of programs only
- Requires a lot of try-and-test to fine-tune the algorithms

What do these tools have in common?

- Bug finders
- Program verifiers
- Code refactoring tools
- Garbage collectors
- Runtime monitoring system
- And... optimizers

What do these tools have in common?

- Bug finders
- Program verifiers
- Code refactoring tools
- Garbage collectors
- Runtime monitoring system
- And... optimizers

They all analyze and transform programs

We will learn about the techniques underlying all these tools

Course goals

- Understand basic techniques for doing program analyses and verification
 - these techniques are the cornerstone of a variety of program analysis tools
 - they may come in handy, no matter what research you end up doing
- Get a feeling for what research is like in the area by reading research papers, and getting your feet wet in a small research project
 - useful if you don't have a research area picked
 - also useful if you have a research area picked: seeing what research is like in other fields will give you a broader perspective

Course topics

- Techniques for representing programs
- Techniques for analyzing and verifying programs
- Applications of these techniques

Course topics (more details)

- Representations
 - Abstract Syntax Tree
 - Control Flow Graph
 - Dataflow Graph
 - Static Single Assignment
 - Control Dependence Graph
 - Program Dependence Graph
 - Call Graph

Course topics (more details)

- AnalysisTechniques
 - Dataflow Analysis
 - Interprocedural analysis
 - Pointer analysis
 - Abstract interpretation
 - Model Checking
- Interaction between transformations and analyses
- Maintaining the program representation intact

Course topics (more details)

- Applications
 - Scalar optimizations
 - Loop optimizations
 - Object oriented optimizations
 - Program verification
 - Bug finding

Course pre-requisites

- No compilers background necessary
- Some familiarity with lattices
 - I will review what is necessary in class, but it helps if you know it already
- Some familiarity with program semantics
 - we will focus on semantics-based approaches
- A standard undergrad cs curriculum will most likely cover the above
 - Talk to me if you think you don't have the pre-requisites

Course evaluation

- Participation in class (20%)
- Regular homeworks (30%)
- Final homework (50%)

Homeworks

- Regular homeworks

Each week there will be an (individual) assignment. Each student has to send his original solution by an email to cortesi@unive.it with subject AVP surname

- Final homework

Each student will be assigned a research paper. This has to be deeply understood, by looking also at related works. A powerpoint presentation of about 15 minutes should be prepared to illustrate the work.

- The exam will consist in the discussion of the regular and final homeworks.

Administrative info

- Class web page:
 - <http://www.dsi.unive.it/~avp>
 - I will post lectures, project info, etc.

Questions?
