Static Program Analysis

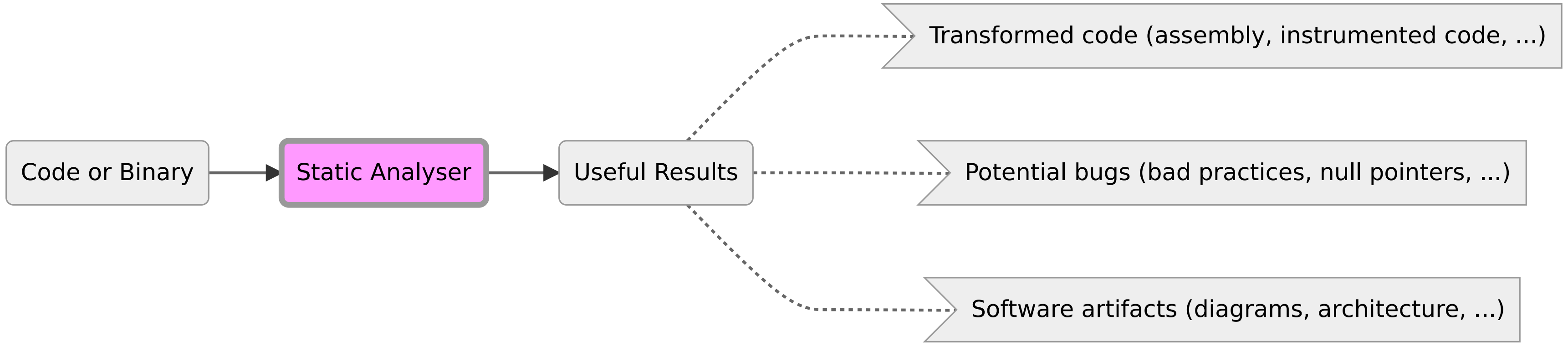
Jun Ma

majun@nju.edu.cn

Static Prgram Analysis: Overview

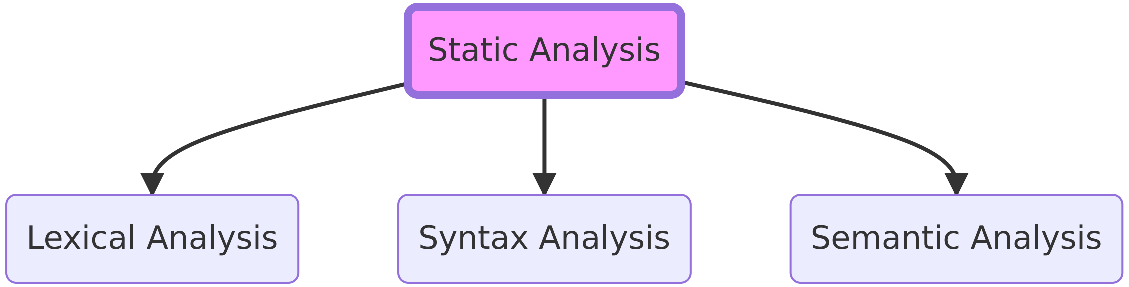
What is Static Analysis?

A program that takes programs as input and produces useful results.


Code or Binary
Static Analyser
Useful Results
Transformed code (assembly, instrumented code, ...)
Potential bugs (bad practices, null pointers, ...)
Software artifacts (diagrams, architecture, ...)

- Examples
    - Compilers (and optimization passes)
    - Static checkers (e.g., -Wall, lint, ...)
    - Useful results for SE practices

Categories of Static Program Analysis



Lexical Analysis

Treating program as a sequence of Symbols/Tokens

Example: Empirical Study on Variable Naming

What are the style, abbreviation, … of variable names?

- Are they correlated to bugs/code quality/…?

You can study this by treating code as a tokenized text stream

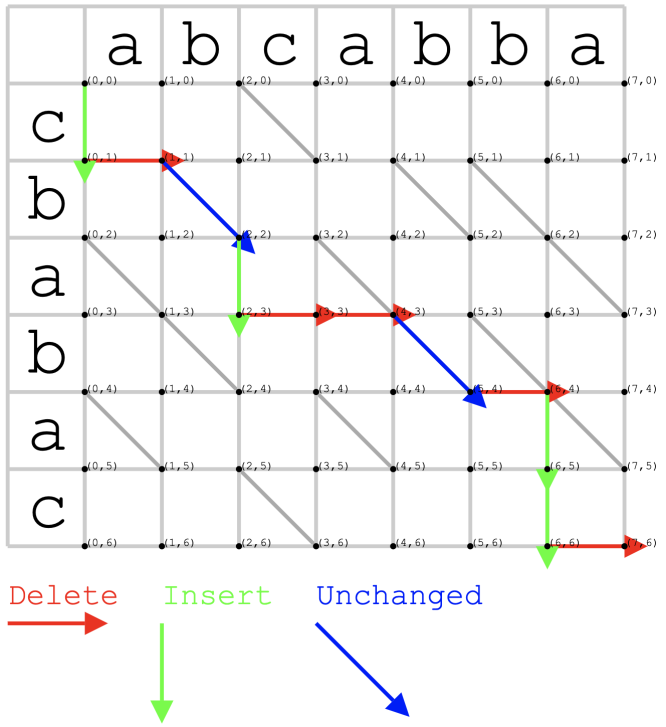
```
"(a + b) * 2" =>
  [ (SYM, '('), (ID, 'a'), (BIN_OP, '+'), (ID, 'b'), (SYM, ')'), (BIN_OP, '*'), (INT, '2') ]
```

We are interested in the IDs

Example: Differencing Files

How to define "diffs" between two file versions?

The Edit Distance Approximation



Myers, E.W. An $O(ND)$ difference algorithm and its variations. Algorithmica 1, 251–266 (1986).
https://doi.org/10.1007/BF01840446

Is Edit Distance a Good Idea?

Open Problem: How to produce even more **developer-friendly** diffs?

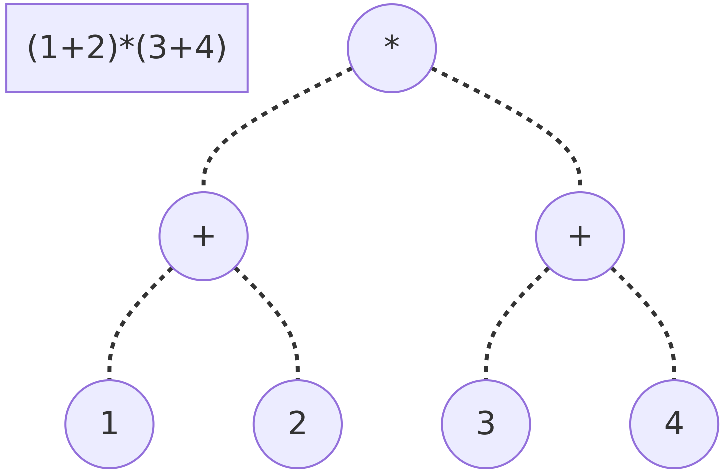
Minimizing edit distance is a good hack

- Lacks semantic explanations to what are changed
- Not work for adding indention, renaming variables, …
    - You can work out a paper on this!

Syntax Analysis on AST

Abstract Syntax Tree (AST)

AST: A tree representation of the abstract syntactic structure of source code



The syntax of a PL is defined by a **context-free** grammar

- The grammar expansion forms a tree
- Can also infer semantics information (e.g., variable type) on AST

Example: Clang AST Dump

Usage (linux):

```
$ clang -Xclang -ast-dump -fsyntax-only a.c
```

Example code:

```c
int f(int a, int b) { if (b == 0) return a; else { ... } }
```

The AST:

```
|-FunctionDecl used f 'int (int, int)'
| |-ParmVarDecl used a 'int'
| |-ParmVarDecl used b 'int'
| `-CompoundStmt
|   `-IfStmt
|     |-BinaryOperator 'int' '=='
|     | |-ImplicitCastExpr 'int'
|     | | `-DeclRefExpr 'int' lvalue ParmVar 'b' 'int'
|     | `-IntegerLiteral 'int' 0
|     |-ReturnStmt
|     | `-ImplicitCastExpr 'int'
|     |   `-DeclRefExpr 'int' lvalue ParmVar 'a' 'int'
|     `-...
```

Example: Python AST Dump

```python
import ast
print(ast.dump(ast.parse("(1+2)*(3+4)")))
```

Output:

```
Module(body=[Expr(value=BinOp(left=BinOp(left=Constant(value=1, kind=None), op=Add(), right=Constant(value=2, kinc
```

For a pretty-printed AST, try `astdump` (https://pypi.org/project/astdump/)

Application: Lint

Checks trivial syntatical errors (e.g., style violations)

- Unused variables
- Unreachable code
- Suspicious assignments (such as `if (a = b)`)
- Each line is at most 80 characters long.
- Variable names (function parameters) and data members are all lowercase.
- Data members of classes (but not structs) additionally have trailing underscores.
- Avoid using run-time type information (RTTI).
- …

Most of the rules can be checked by source-code scan or AST.

Splint

```
# apt install splint
# splint source.c
```

AST for Software Metrics

Halstead’s "software physics" (introduced in 1977)

- n_1/n_2: distinct operator/operand
- N_1/N_2: occurrences of operator/operand
- program length $N = N_1 + N_2$
- program volume $V = N\log_2(n_1 + n_2)$
- specification abstraction level $L = 2n_2/(n_1 \cdot N_2)$
- program effort $E = (n_1 + N_2 \cdot N \cdot \log_2(n_1 + n_2))/2n_2$
- ...

We can mine correlations between software metrics and quality/maintainability/...

AST for Clone Detection

> " "

Code clone is killing projects

- high-vote buggy code on Stackoverflow
- intellectual property (IP)
- ...

Typical work:

- Token-based detection: "CCFinder: A multilinguistic token-based code clone detection system for large scale source code" (TSE, 28(7), 2002)
- Tree-based detection: "Scalable detection of semantic clones" (ICSE'08)

AST for Code Transformation

Code formatting

- GNU indent, bcpp, Google Java format, …
- Formatting = traversal of AST (with style rules)

Transcompiler (transpiler), source-to-source compiler

- ES6/ES10/JSX → ES5 (for maximized compatibility)
- Earliest versions of C++ did not have a native compiler: C++ → C
- Emscripten: C/C++ programs (compiled with LLVM) → JavaScript/WebAssembly.

Many other applications in software engineering research

- Mutation testing, e.g., μJava
- Mutation space and GenProg for program repair
  - this paper is also recommended!

Limitations of AST-based Analyses

Lacks good understanding of program semantics

- E.g., checking that all paths return a value

Hard to do with meta-programming

```
#define FORALL(X) X(Tom) X(Jerry) X(Spike) X(Tyke)
#define PRINT(x) puts(#x);
// usage: FORALL(PRINT)
```

Semantics Analysis

Semantics Analysis

Tells you something about program’s execution

- whether `foo()` is reachable
- whether a pointer access is valid (not `NULL`, in bound, …)

Semantics analyses are useful to

- compilers (and optimizations)
- bug/security analysis
- program verification
- …

```c
char dest[SIZE];
strncpy(dest, src, SIZE);
int len = strlen(dest); // insecure!
```

A Vision

> In 2050, our compiler will reject a program if it cannot prove all assertions in the program.

Seemingly crazy today.

Hardness of Semantics Analysis

A general program analyzer gives you infinite computational power (and thus does not exist!)

Rice's Theorem:

> All non-trivial, semantic properties of programs are **undecidable**.

```python
def booooom(): ... # Reachable?

n = 6
while n := n + 2:
    sols = [ (i, n - i) for i in range(2, n / 2) \
        if is_prime(i) and is_prime(n - i) ]
    if not sols: # replace this with halting problem
        booooom()
    print(f'{n} = ' + ' = '.join(f'{x} + {y}' for x, y in sols))
```

Semantics Analyses

Suppose that

- Goldbach conjecture holds
- We have infinite memory

A practical static analyzer may report:

- booooom() is unreachable, print() is reachable (sound, complete)
- booooom(), print() may be reachable (sound, incomplete)
- print() is unreachable (unsound, usually problematic to compilers)

Notice:

> Sometimes the term "sound/complete" have reverse meaning in SE papers

```python
def booooom(): ... # Reachable?

n = 6
while n := n + 2:
    sols = [ (i, n - i) for i in range(2, n / 2) \
        if is_prime(i) and is_prime(n - i) ]
    if not sols: # replace this with halting problem
        booooom()
    print(f'{n} = ' + ' = '.join(f'{x} + {y}' for x, 
```

Mathmatical Logics

Soundness

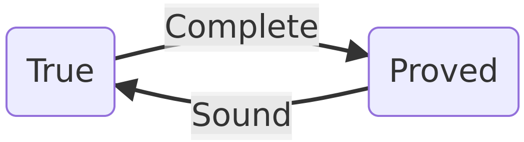
> $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ implies that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$

- *Anything can be proved is true*

Completeness

> $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ implies that $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$

- *Anything true can be proved*



$$\begin{tikzcd} \text{True} \arrow[r, bend left, "\text{Complete}"] & \text{Proved} \arrow[l, bend left, "\text{Sound}"] \end{tikzcd}$$

Program Analysis

Soundness

- Over-approximation:
  - result models ***all possible*** executions of the program
  - result will also model behaviors that ***do not actually occur*** in any program execution
  - Precision depends on how well avoiding such ***spurious*** results
- May produce *false alarms*

Completeness

- result may *miss* some possible executions of the program
- No false alarms

Comments on Soundness

Any sound reachability analysis should do:

- Reporting `foo()` unreachable → `foo()` is indeed unreachable
- Not missing any reachable function

A trivial sound reachability analysis:

> Everything may be reachable.

- **Useless**

Comments on Soundness (cont'd)

Extremely difficult to prove a function being unreachable!

- Control flow (Goldbach conjecture)
- Polymorphism/Dynamic dispatching (`lut[name]()`)
- Dynamic code (`eval('pri' + 'nt(1)')`)
- Reflection (Java)
- Foreign code (C: inline assembly, Java: JNI, ...)

Existing analyzers provide limited soundness

Claiming soundness in papers may mislead readers:

- Non-experts may erroneously conclude that the anaysis is sound and confidently rely on the results
- Experts find it is hard to interpret the analysis results (how sound, fast, precise is the analysis) without a clear explanation about how they treat those hard language features

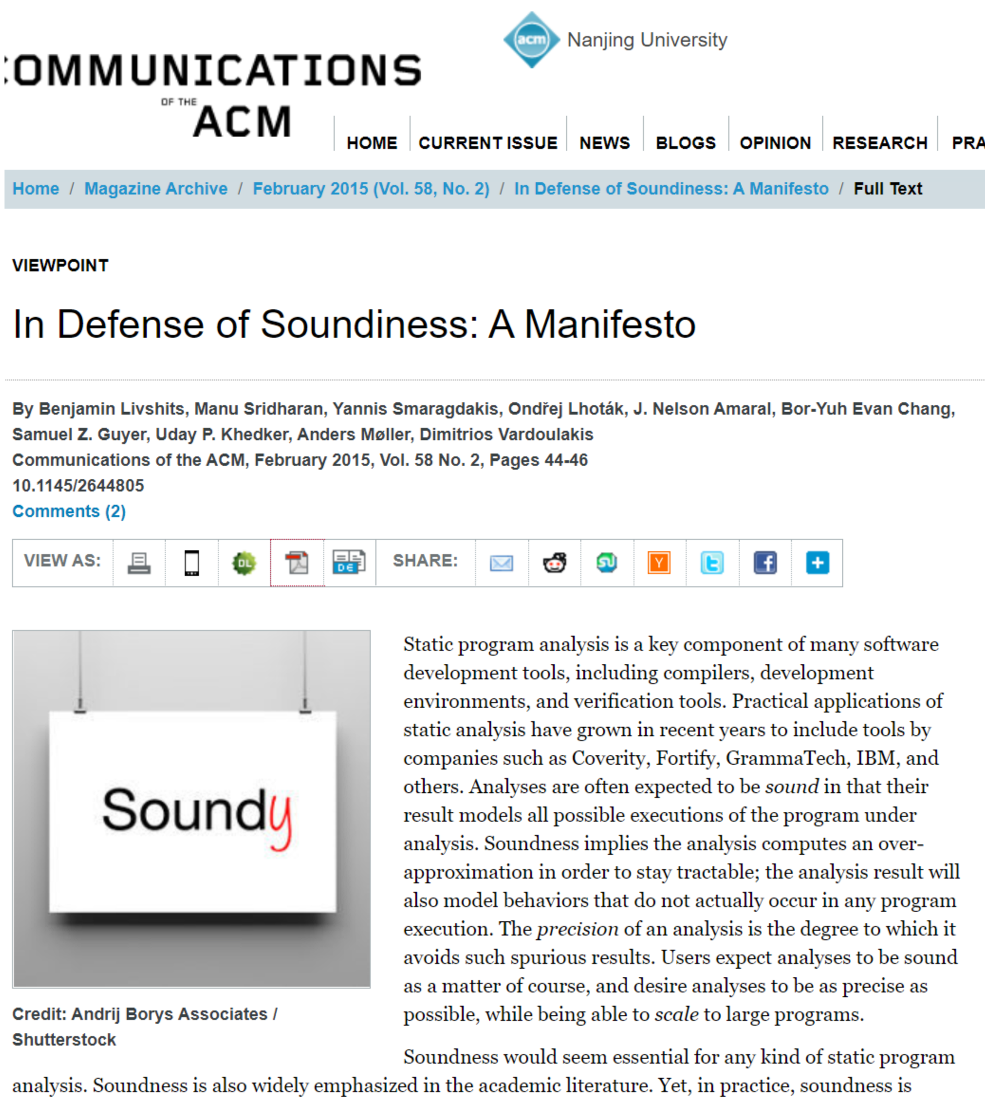
Soundiness

Truthiness

> A "truth" that sb. believes to be true intuitively, without any fact or evidence.

Soundiness

> A **soundy** analysis typically means that the analysis is mostly sound, with well-identified unsound treatments to hard/specific language features



OMMUNICATIONS OF THE ACM

Nanjing University

HOME | CURRENT ISSUE | NEWS | BLOGS | OPINION | RESEARCH | PRA

Home / Magazine Archive / February 2015 (Vol. 58, No. 2) / In Defense of Soundiness: A Manifesto / **Full Text**

VIEWPOINT

In Defense of Soundiness: A Manifesto

By Benjamin Livshits, Manu Sridharan, Yannis Smaragdakis, Ondřej Lhoták, J. Nelson Amaral, Bor-Yuh Evan Chang, Samuel Z. Guyer, Uday P. Khedker, Anders Møller, Dimitrios Vardoulakis

Communications of the ACM, February 2015, Vol. 58 No. 2, Pages 44-46
10.1145/2644805

Comments (2)

VIEW AS: SHARE:

Credit: Andrij Borys Associates / Shutterstock

Static program analysis is a key component of many software development tools, including compilers, development environments, and verification tools. Practical applications of static analysis have grown in recent years to include tools by companies such as Coverity, Fortify, GrammaTech, IBM, and others. Analyses are often expected to be *sound* in that their result models all possible executions of the program under analysis. Soundness implies the analysis computes an over-approximation in order to stay tractable; the analysis result will also model behaviors that do not actually occur in any program execution. The *precision* of an analysis is the degree to which it avoids such spurious results. Users expect analyses to be sound as a matter of course, and desire analyses to be as precise as possible, while being able to *scale* to large programs.

Soundness would seem essential for any kind of static program analysis. Soundness is also widely emphasized in the academic literature. Yet, in practice, soundness is

Soundness, Soundiness and Unsoundness

- A **Sound** analysis requires capturing all dynamic behaviors
- A **Soundy** analysis aims to capture all dynamic behaviors with certain hard language features unsoundly handled within reason
  - Most practical work in program analysis (PA)
- An **Unsound** analysis deliberately ignores certain behaviors in its design for better efficiency, precision or accessibility
  - Most practical work in SE

Implementing Static Analyses

Translate a program to an easier-to-handle representation

- Intermediate Representation (IR) for small-step semantics
  - E.g. The *jimple* 3-address code used by Soot for JAVA

```
public class Foo {                                                         ==>

  public static void main(String[] args) {
    Foo f = new Foo();
    int a = 7;
    int b = 14;
    int x = (f.bar(21) + a) * b;
  }

  public int bar(int n) { return n + 42; }
}
```

Implementing Static Analyses

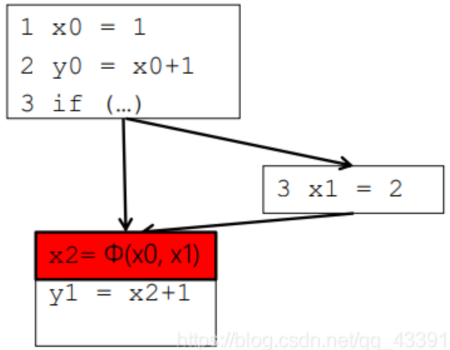
Translate a program to an easier-to-handle representation

- Intermediate Representation (IR) for small-step semantics
  - E.g. The *jimple* 3-address code used by Soot for JAVA
- Usually single static assignment (SSA) that each assignment creates a new variable

```
x = 1
y = x+1
x = 2
y = x +1
```

```
x = 1
y = x+1
x′= 2
y′= x′ +1
```

```
1 x = 1
2 y = x+1
3 if (…) x = 2
4 y = x +1
```



```
1 x0 = 1
2 y0 = x0+1
3 if (…)
```

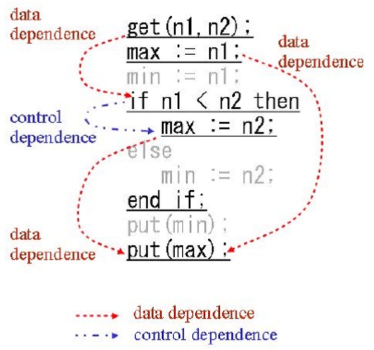
```
3 x1 = 2
```

```
x2= Φ(x0, x1)
y1 = x2+1
```

Examples

Program Slicing (ICSE’81)

- Highlights "dependent" code for a given program part



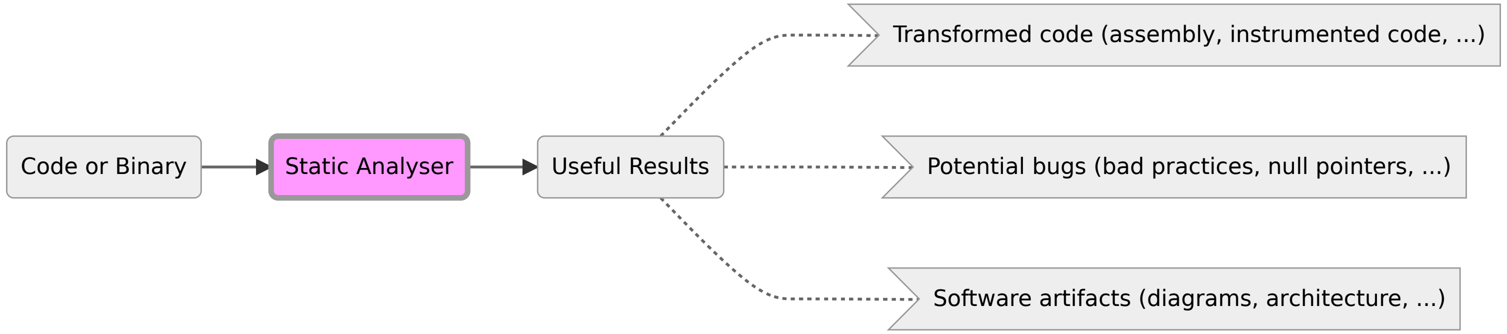
```
get(n1,n2);
max := n1;
min := n1;
if n1 < n2 then
   max := n2;
else
   min := n2;
end if;
put(min);
put(max);
```

Taint Analysis

- See input’s "reachable" part
- Useful in many security analyses
  - E.g., TaintDroid

Summary

Static Program Analysis


Code or Binary
Static Analyser
Useful Results
Transformed code (assembly, instrumented code, ...)
Potential bugs (bad practices, null pointers, ...)
Software artifacts (diagrams, architecture, ...)

Most SE research use existing analyses as **black-box** (use Clang/LLVM, Soot, ...)

- Don't expect too much
- Acknowledge the limitations of available tools (analysis is undecidable!)

Related courses at NJU

- Software analysis (by &)
- Formal semantics of programming languages (by)