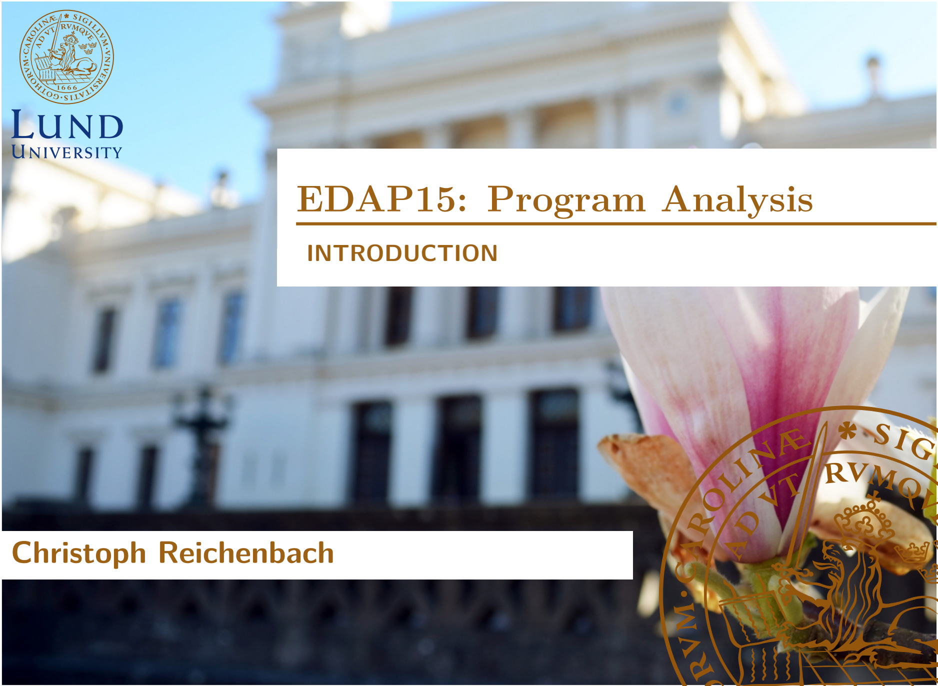
LUND
UNIVERSITY



EDAP15: Program Analysis

INTRODUCTION

Christoph Reichenbach

Welcome!

- **EDAP15: Program Analysis**
- **Instructor**: Christoph Reichenbach
  `christoph.reichenbach@cs.lth.se`
- **Teaching Assistant**: Idriss Riouak
  `idriss.riouak@cs.lth.se`
- **Course Homepage**:
  `http://cs.lth.se/EDAP15`

Course Format

- *Tentatively In-Person*
- Core material:
  - Lectures (**bring your questions!**)
  - Videos (Preparatory and others)
  - Homework
- Questions?
  - *Ask in class*
    - Ask-and-Upvote system (or just raise your hand!)
  - Online forum
  - Office hours
- Group Exercises
  - Labs in E:Gamma, Fri 08:00–12:00 (two time slots)
- Online Quizzes
- Written Exam

Topics

- Concepts and techniques for understanding programs
  - Analysing program structure
  - Analysing program behaviour
- Practical concerns in program analysis

Language focus: Teal, a teaching language

- Concepts generalise to other mainstream languages:
  - Imperative
  - Object-Oriented

Goals

- **Understand**:
  - What is program analysis (not) good for?
  - What are strenghts and limitations of given analyses?
  - How do analyses influence each other?
  - How do programming language features influence analyses?
  - What are some of the most important analyses?
- **Be able to**:
  - Implement typical program analyses
  - Critically assess typical program analyses

Resources

- **Course website** (`http://cs.lth.se/EDAP15`)
  - Links to everything listed here
  - List of expected skills
  - Slides
  - Announcements
- Textbooks
- **Moodle**
  - Slides
  - Quizzes
  - Videos
  - Forum
- **Course git** (GitLab)
  - Homework assignments

Textbooks

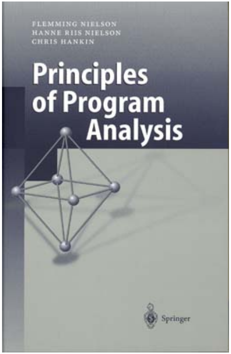
Static Program Analysis

Møller & Schwartzbach

- Optional
- PDF online from authors

Principles of Program Analysis

Nielson, Nielson & Hankin



- Optional
- 3 copies in the library
- Theory-driven

Week Overview

Mo	**Tu**	**We**	**Th**	**Fr**	
10:15 in E:2116 **Class**	**Videos and Quizzes**	10:15 in E:2116 **Class** **Homework** release	08:00 in E:Gamma first week only **Labs** **Videos and Quizzes**	08:00&10:00 in E:Gamma **Labs**	

Mo	**Tu**	**We**	**Th**	**Fr**	
		. . .			

Mo	**Tu**	**We**	**Th**	**Fr**	
		Homework deadline			

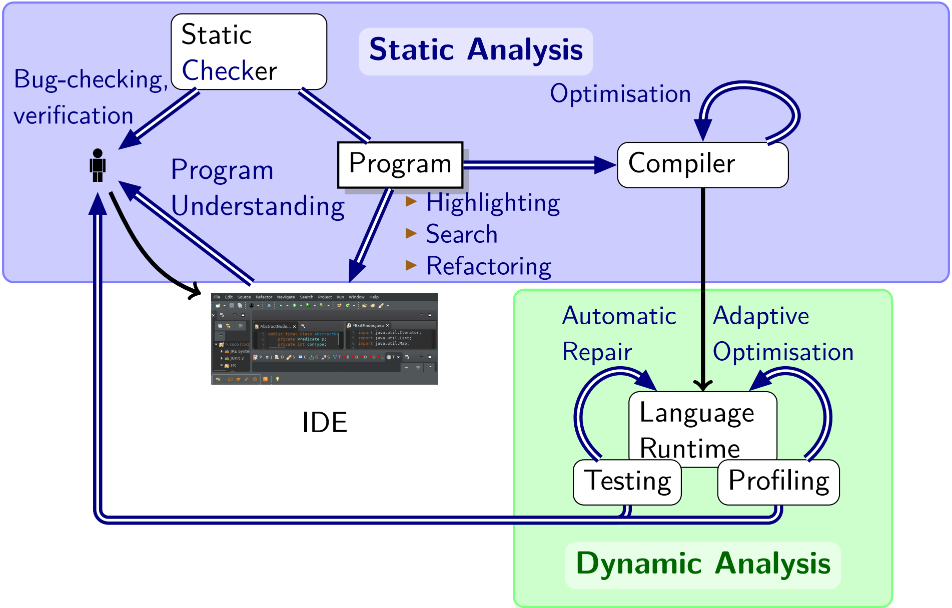
How to Pass

- **2020-11-02 18:00**: Form Groups of 2
  - Contact Idriss if you can't find a partner
- **2020-11-06 10:00**: Register group for lab slot
- **Warmup projects**:
  - **HW0**: optional (recommended!) lab, this Thursday/Friday
- **Homework projects**:
  - **Who**: Groups
  - **What**: Implement program analyses for Teal
  - **Start**: Homework up Wednesdays Weeks 2, 3, 4 & 6
  - **Grading**:
    - Submit solutions in course git
    - Explain solution to TA
  - **Deadline**:
    - **HW1–5**: Thursday 20:00, 13 days after homework is up
- **Final Exam** starting 2021-01-15
  - **Admission**: Passed all homework projects
  - **Format**: *written exam*

Structure

W44	Homework #0 (optional)	Wednesday: Groups formed
W45	Homework #1 start	Monday: Lab slot assignemnts
W46	Homework #2 start	
W47	Homework #3 start	Homework #1 due
W48		Homework #2 due
W49	Homework #4 start	Homework #3 due
W50		Homework #4 due

Uses of Program Analysis



Static Checker

Static Analysis

Bug-checking, verification

Optimisation

Program

Compiler

Program Understanding

- Highlighting
- Search
- Refactoring

Automatic Repair

Adaptive Optimisation

IDE

Language Runtime

Testing

Profiling

Dynamic Analysis

Categories of Program Analyses

	Manual / Interactive	**Automatic**
Static Analysis ▸ Examines *structure* ▸ Sees entire program (mostly. . .)	▸ Interactive Theorem Provers	▸ (Most) Type Checkers ▸ Static Checkers (FindBugs, SonarQube, . . .) ▸ Compiler Optimisers
Dynamic Analysis ▸ Examines *behaviour* ▸ Sees interactions program ↔ world	▸ Debuggers	▸ Unit Tests ▸ Benchmarks ▸ Profilers

Our Focus

Summary

- Program analyses are key components in *Software Tools*:
  - IDEs
  - Compilers
  - Bug and Vulnerability Checkers
  - Run-time systems

  ...
- Main Categories:
  - **Static Analysis**:
    Examine program structure
  - **Dynamic Analysis**:
    Examine program run-time behaviour
  - **Automatic Analysis**:
    "Black Box": Minimal user interaction
  - **Manual / Interactive Analysis**:
    User in the loop
    - Advanced manual analyses exploit automatic analysis

Examples of Program Analysis

Questions:

- 'Is the program well-formed?'

  ```
  gcc -c program.c
  javac Program.java
  ```

  At least for C, C++, Java; not so easy for JavaScript!
- 'Does my factorial function produce the right input in the range 0–5?'

Java

```java
@Test // Unit Test
public void testFactorial() {
  int[] expected = new int[] { 1, 1, 2, 6, 24, 120 };
  for (int i = 0; i < expected.length; i++) {
      assertEquals(expected[i], factorial(i));
} }
```

Let’s Analyse a Program!

- MISRA-C standard specifies:
  “*The library functions . . . ,* `gets`, *. . . shall not be used.*”
- Given some `program.c`:

```
user@host$ grep ’gets’ program.c     # string search
    gets(input_buffer);
    /* The code below gets the system configuration */
    int failed_gets_counter = 0;
user@host$
```

At least 2 of 3 resuls were wrong: “*False Positives*”

A First Challenge, Continued

```
user@host$ grep 'gets(' program.c
    gets(input_buffer);
user@host$
```

- More precise: no false positives!
- Will this catch *all* calls to gets?

C: program2.c

```c
#include <stdio.h>
void f(char* target_buffer) {
    char *(*dummy)(char*) = gets;
    dummy(target_buffer);
}
```

String search not smart enough: “*False Negative*”

A First Challenge, Continued Again

C: program2.c

```c
#include <stdio.h>
void f(char* target_buffer) {
    char *(*dummy)(char*) = gets;
    dummy(target_buffer);
}
```

```
user@host$ cc -c program.c -o program.o
user@host$ nm program.o
                # check symbol table in compiled program
0000000000000000 T f
                 U gets        ⟵ Aha!
                 U _GLOBAL_OFFSET_TABLE_
 user@host$
```

Using a more powerful analysis yielded better results

A First Challenge, Solved?

C: `program3.c`

```
#include<stdio.h>
#include<dlfcn.h>
int f(char* target_buffer) {
    void* handle = dlopen("/lib/x86_64-linux-gnu/libc.so.6",
                          RTLD_LAZY);
    void* sym = dlsym(handle, "gets");
    void(*p)(char*) = sym;
    p(target_buffer);
    return 0;
}
```

- Dynamic library loading: `gets` will not show up in symbol table

Fancier program ⟹ harder analysis

Analysis vs. Property-of-Interest

- Distinguish:
  - **Property** of interest: $P(\varphi)$
    - All lines in φ that reference `gets`
    - Does φ type-check?
    - Where does φ spend most execution time?
  - **Analysis** $\mathcal{A}(\varphi)$ that approximates $P(\varphi)$

$$P(\varphi) \approx \mathcal{A}(\varphi)$$

And How Good Is It?

- As we saw, program analyses may be incorrect
- We often describe them with *Information Retrieval* terminology:

	$\mathcal{A}(\varphi)$	*not* $\mathcal{A}(\varphi)$
$P(\varphi)$	**True Positive**	**False Negative**
not $P(\varphi)$	**False Positive**	**True Negative**

- How well does $\mathcal{A}$ approximate $\mathcal{P}$?
  - Assume $\mathcal{A}(\varphi)$ returns n reports
    $n = \#\text{True Positives} + \#\text{False Positives reports}$
  - Are the reports good?
    Precision $= \frac{\#\text{True Positives}}{n}$
  - Are the reports comprehensive?
    Recall $= \frac{\#\text{True Positives}}{\#\text{True Positives} + \#\text{False Negatives}}$
- #False Negatives (and thus **Recall**) is usually impossible to determine in program analysis

Summary

- Purpose of **Analysis** $\mathcal{A}$:
  - Compute **Property-of-interest** P
- Program Analysis is nontrivial
  - Programs can hide information that $\mathcal{A}$ wants
  - Analysis $\mathcal{A}$ can misunderstand parts of the program

Soundness and Completeness

Can we always build a $\mathcal{A}$ with $\mathcal{A}(\varphi) = P(\varphi)$?

- Connection to Mathematical Logic:
  - Assume P is boolean
  - $\mathcal{A}$ is **sound** (with respect to P) iff:

$$\mathcal{A}(\varphi) \implies P(\varphi) \qquad \text{(Perfect Precision)}$$

  - $\mathcal{A}$ is **complete** (with respect to P) iff:

$$\mathcal{A}(\varphi) \Longleftarrow P(\varphi) \qquad \text{(Perfect Recall)}$$

  - $\mathcal{A}(\varphi) = P(\varphi)$ iff $\mathcal{A}$ is both sound & complete

What if $P(\varphi)$ checks whether φ terminates?

The Bottom Line

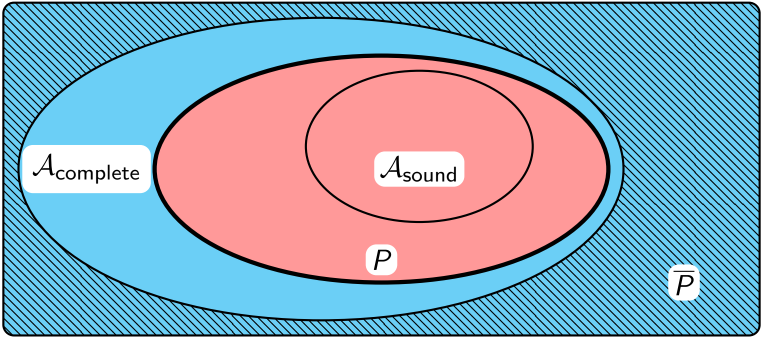
> "Everything interesting about the behaviour of programs is undecidable."
>
> — Anders Møller, paraphrasing H.G. Rice [1953]

We must choose:

- **Soundness**
- **Completeness**
- **Decidability**

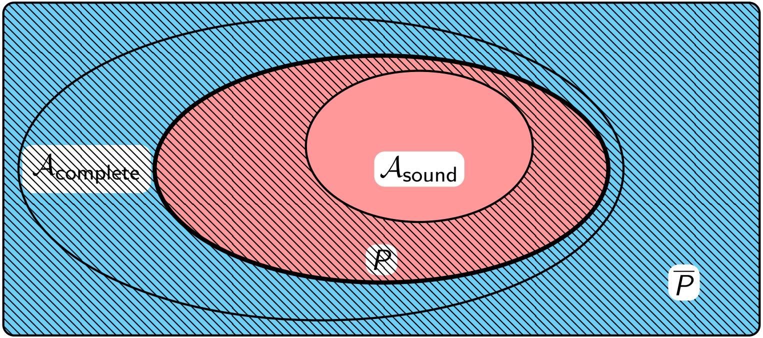
. . . pick any two.

Soundness and Completeness: Caveat



- *Beware*: “sound” and “complete” be confusing:
  - Example: $P(\varphi)$ is “φ has a bug”
  - If you now want to check $\overline{P}$, the *negation* of P:
    - $\overline{P}(\varphi)$ is “φ does not have a bug”
    - $\overline{\mathcal{A}_{\text{complete}}}$ (= run $\mathcal{A}_{\text{complete}}$ and invert output) is *sound wrt* $\overline{P}$

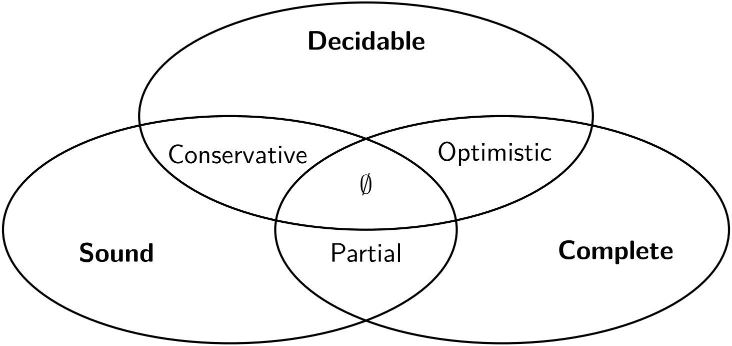
Soundness and Completeness: Caveat



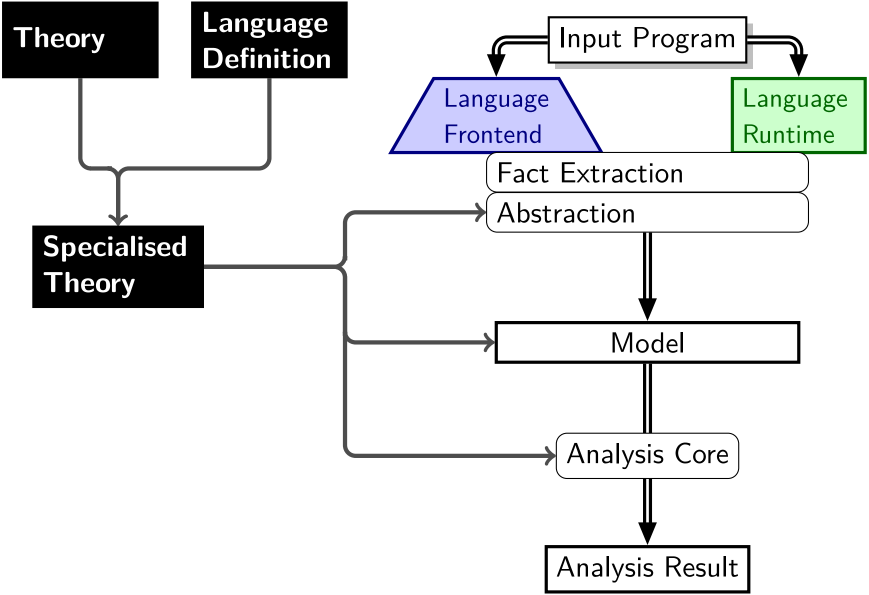
- *Beware*: “sound” and “complete” be confusing:
  - Example: $P(\varphi)$ is “φ has a bug”
  - If you now want to check $\overline{P}$, the *negation* of P:
    - $\overline{P}(\varphi)$ is “φ does not have a bug”
    - $\overline{\mathcal{A}_{\text{complete}}}$ ($=$ run $\mathcal{A}_{\text{complete}}$ and invert output) is *sound wrt* $\overline{P}$
    - $\overline{\mathcal{A}_{\text{sound}}}$ is *complete wrt* $\overline{P}$

Summary

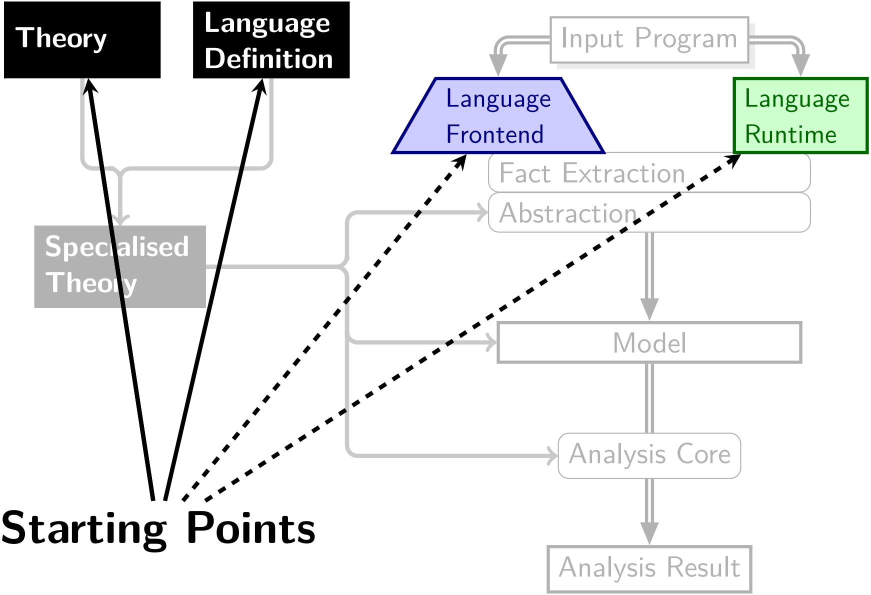
- Given property P and analysis $\mathcal{A}$:
  - $\mathcal{A}$ is **sound** if it triggers only on P
    $P =$ “program has bug”: $\mathcal{A}$ reports *only* bugs
  - $\mathcal{A}$ is **complete** if it always triggeres on P
    $P =$ “program has bug”: $\mathcal{A}$ reports *all* bugs
- If P is nontrivial (i.e., depend on behaviour):



Building a Program Analysis



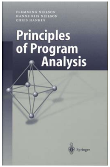
Building a Program Analysis



Starting Points

Gathering Our Tools

Theories

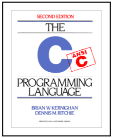


FLEMMING NIELSON
HANNE RIIS NIELSON
CHRIS HANKIN

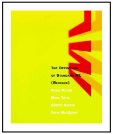
Principles of Program Analysis

Springer

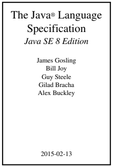
Language Definitions



SECOND EDITION
THE C PROGRAMMING LANGUAGE
ANSI C
BRIAN W. KERNIGHAN
DENNIS M. RITCHIE



The Definition of Standard ML (Revised)
Robin Milner
Mads Tofte
Robert Harper
David MacQueen



The Java® Language Specification
Java SE 8 Edition

James Gosling
Bill Joy
Guy Steele
Gilad Bracha
Alex Buckley

2015-02-13

Tools

Compilers

Analysis Frameworks

jastadd

PhASAR

Astrée

FindBugs™

Soot

Hardware

Language Definitions

- Define structure (syntax) and meaning (semantics) of language

Syntax example:

$$\begin{array}{rcl} e & ::= & \mathsf{zero} \\ & | & \mathsf{one} \\ & | & \langle e \rangle \mathsf{+} \langle e \rangle \\ & | & \langle e \rangle \mathsf{-} \langle e \rangle \\ & | & \mathsf{neg}\ \langle e \rangle \\ & | & \mathsf{(}\ \langle e \rangle\ \mathsf{)} \\ & | & \mathsf{log}\ \langle e \rangle \end{array}$$

- **Property of Interest**: Does a given program $\varphi \in e$ compute a number ≥ 0?

First, we must understand the language *semantics*

Language Definitions: Semantics

- Language Definitions also specify *Semantics*:
  - **Static Semantics**:
    - Connect parts of the program structure (variables, functions, classes, . . .)
    - Enforce restrictions (e.g., via *type checking*)
  - **Dynamic Semantics**:
    - Program run-time behaviour
- We will not explore formal semantics in depth, in this course
- Here: assume "obvious" semantics

Analysing Programs: A First Shot

Property of Interest: Does a given program $\varphi \in e$ compute a nonnegative number?

$$
\begin{array}{rcl}
e & ::= & \text{zero} \\
  & | & \text{one} \\
  & | & \langle e \rangle \texttt{+} \langle e \rangle \\
  & | & \langle e \rangle \text{-} \langle e \rangle \\
  & | & \text{neg } \langle e \rangle \\
  & | & (\ \langle e \rangle\ ) \\
  & | & \text{log } \langle e \rangle
\end{array}
$$

- How could we analyse this for a given program φ?
  - Just *run φ and check the result*
  - Works fine here!
  - Problematic once we add parameters or loops/recursion

Let's explore what we would do for a more complex language

Simplifying the Language

- Let's make it easier to analyse the language
- We don't need parentheses for the analysis
- $a\text{-}b = a\,\mathbf{+}\,\mathsf{neg}\ b$
  ⇒ Abstraction (we join similar problems into one)
- log is too difficult
  ⇒ *Restrict to sub-language (give up on some problems)*

$$\begin{array}{lcl} e & ::= & \mathsf{zero} \\ & \mid & \mathsf{one} \\ & \mid & \langle e \rangle \mathbf{+} \langle e \rangle \\ \\ & \mid & \mathsf{neg}\ \langle e \rangle \end{array}$$

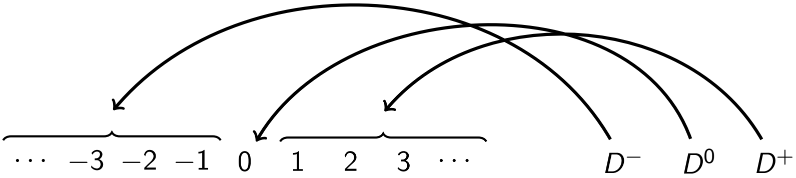
Simplification helps us get started, but restricting to a sublanguage can quickly render an analysis impractical

Finding a Good Theory

- Recall: $\mathcal{A}(\varphi)$ should check if φ computes result ≥ 0
- There are many theories for program analysis
- We pick *Abstract Interpretation* (Patrick & Radhia Cousot):
  - Map all values to a simpler *abstract domain*

- For example: classify programs into *abstract domain* containing:
  - D^0: Computes 0
  - D^+: Computes a positive value
  - D^-: Computes a negative value
- Notation: $\boxed{\varphi \leadsto^{D} a}$, where a is one of D^0, D^+, D^-

Correspondence: Concrete and Abstract



$$\overbrace{\cdots \quad -3 \quad -2 \quad -1}\quad 0 \quad \overbrace{1 \quad 2 \quad 3 \quad \cdots} \qquad D^{-} \quad D^{0} \quad D^{+}$$

Finding a Good Theory

- Recall: $\mathcal{A}(\varphi)$ should check if φ computes result ≥ 0
- There are many theories for program analysis
- We pick *Abstract Interpretation* (Patrick & Radhia Cousot):
  - Map all values to a simpler *abstract domain*
  - Map all operations so they respect the abstraction
- For example: classify programs into *abstract domain* containing:
  - D^0: Computes 0
  - D^+: Computes a positive value
  - D^-: Computes a negative value
- Notation: $\boxed{\varphi \leadsto^{D} a}$, where a is one of D^0, D^+, D^-

Semantics

$$
\begin{array}{lcl}
e & ::= & \mathsf{zero} \\
 & | & \mathsf{one} \\
 & | & \langle e \rangle \mathsf{+} \langle e \rangle \\
 & | & \mathsf{neg}\ \langle e \rangle
\end{array}
$$

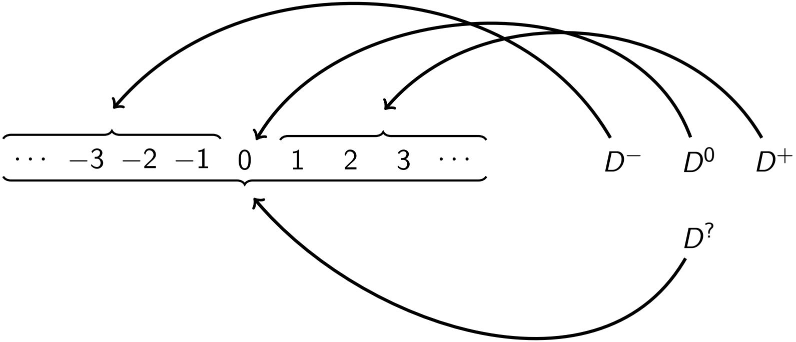
$$
\begin{array}{lcl}
\ominus D^0 & = & D^0 \\
\ominus D^+ & = & D^- \\
\ominus D^- & = & D^+ \\
\ominus D^? & = & D^?
\end{array}
$$

$$
a_1 \oplus a_2 = \left\{ \begin{array}{c||c|c|c|}
 & D^+ & D^0 & D^- \\
\hline\hline
D^+ & D^+ & D^+ & D^? \\
D^0 & D^+ & D^0 & D^- \\
D^- & D^? & D^- & D^-
\end{array} \right.
\qquad
D^? \oplus a = D^? = a \oplus D^?
$$

$$
\mathsf{zero} \rightsquigarrow^D D^0 \qquad \mathsf{one} \rightsquigarrow^D D^+
$$

$$
\frac{\text{if } x \rightsquigarrow^D a \text{ then}}{\mathsf{neg}\ x \rightsquigarrow^D \ominus a}
\qquad
\frac{\text{if } x \rightsquigarrow^D a_1 \text{ and } y \rightsquigarrow^D a_2 \text{ then}}{x \mathbin{\mathsf{+}} y \rightsquigarrow^D a_1 \oplus a_2}
$$

Correspondence: Concrete and Abstract



$\cdots\quad -3\quad -2\quad -1\quad 0\quad 1\quad 2\quad 3\quad \cdots$

D^- D^0 D^+

$D^?$

Also:

- $\ominus$ *"is compatible with"* neg
- $\oplus$ *"is compatible with"* $+$

Abstract Interpretation **explores these ideas in great detail**

Summary

- *Semantics* derive from *syntax*
  - **Static Semantics**: Compile-time behaviour
  - **Dynamic Semantics**: Run-time behaviour
- *Static program analysis approximates dynamic semantics, statically*
- **Abstract Interpretation**: Theory for program analysis
  - Map program semantics into *abstract domain*
  - Map operations to *compatible* operations on abstract domain
  - Challenge: remain precise yet decidable
  - Foundation to other static analysis theories

Outlook

- **Remember**:
  - Join Moodle
  - Check for Videos and Quizzes tomorrow
  - Form groups by Wednesday, 18:00!
- Our initial focus will be on *static program analysis*:
  - Type Analysis
  - Data Flow Analysis
  - Heap Analysis
- Next Lecture: Wednesday, same place:
  - Type-Based Analysis

`http://cs.lth.se/EDAP15`