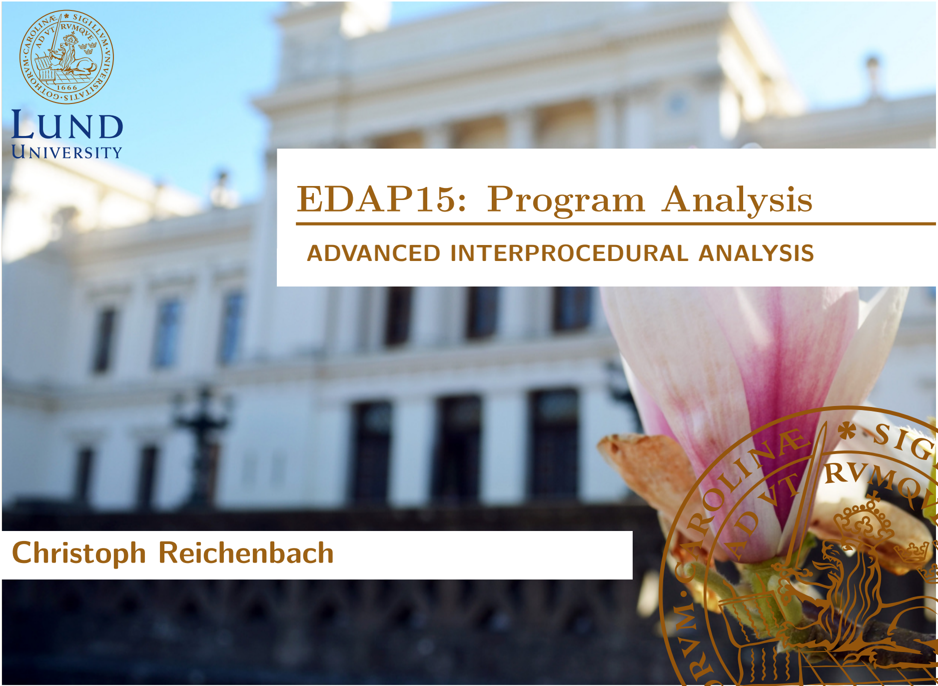


EDAP15: Program Analysis

ADVANCED INTERPROCEDURAL ANALYSIS

Christoph Reichenbach

Welcome back!

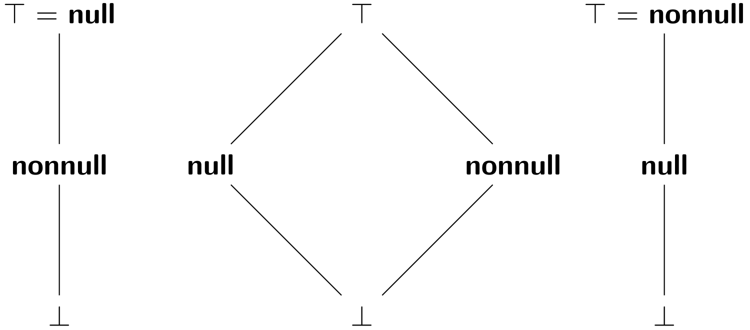
- Lab 3: More automated tests up
- **Guest Lecture**: First half of Thursday:
  - Patrik Åberg & Magnus Templing, Ericsson: Code Instrumentation with DMCE

Lattice Design

✓ Lattices that represent sets of values

- Simplification?
- $\perp$ vs $\top$?
- Lattices for properties that are not values
  - Side effects?
  - Liveness / Dead Assignment?
  - Available Expressions?

Simplifying Lattices (1/2)



May-Null | Null/Nonnull lattice | Must-Null

Can only go up in lattice: “May” = “Must Not” as top

Simplifying Lattices (2/2)

$$
\begin{tikzcd}
\top = \textbf{nonnull} \arrow[dd, dash] \\
\\
\bot = \textbf{null}
\end{tikzcd}
\qquad\qquad
\begin{tikzcd}
\top = \textbf{nonnull} \arrow[d, dash] \\
\textbf{null} \arrow[d, dash] \\
\bot
\end{tikzcd}
$$

Must-Null (Binary) Must-Null

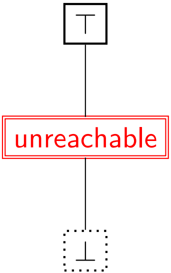
$\boxed{\textbf{No practical difference between } \bot \textbf{ and ``middle'' state}}$

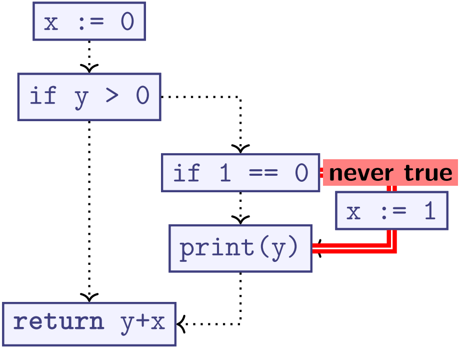
Lattices for Non-Value Properties

Unreachable Path Elimination

- "Which CFG edges can never be taken?"
- Usually depends on constant propagation / folding
- *Forward* analysis

Lattice:





No need to distinguish between unreachable and ⊥

Summary

- Lattice design is a bit of an art
- Can often simplify lattice structure
- Depending on analysis client: “Bottom”/“Top” may have specific meaning

Lecture Overview

Foundations	Static Analysis		Dynamic Analysis
	Properties	Control Flow	
01 Foundations	03 04 Types		12 Instrumentation
02 Constructing Program Analyses in JastAdd	05 06 07 Data Flow	05 Intraprocedural	13 Analysis
	08 09 Memory	10 Interprocedural	
		11 **Indirect**	

14 Review

Challenges Towards OO Support

- $(+)$ Flow-sensitivity
- $(+)$ Points-to information
- Dynamic Dispatch
- Advanced features:
  - Pointer arithmetic
  - Dynamic Class Loading
  - “Native Calls” (into C/assembly/Syscalls)
  - Reflection

Interprocedural Analysis in Java

Java

```
public static void main(String[] args) {
    Object obj = MyClass.getObj();
    System.err.println(obj.toString());
}
```

Subroutine call

- Analogous to Teal-0 calls
- ... need to know `MyClass`

Method call

- **Dynamic Dispatch**
- Exact subroutine depends on *dynamic type* of `obj`

Challenges

- **Other modules**:
  - Must have access to analysable representation of module
  - *Not always available*
- **Dynamic Dispatch**:

```
obj.toString()
```

  - Which `toString` method are we calling?
  - Worst case assumption: *any* class (`Integer.toString()`, `HashSet.toString()`, . . .)
  - Can we do better?

The Call Graph

```c
int main(int argc,
         char *argv) {
    if (argc > 1) {
        f(argv[0]);
    }
    g();
    return 0;
}
```

```c
void f(char *s) {
    for (char *p = s; *p; p++) {
        *p = up(*p);
    }
    puts(s);
}
```

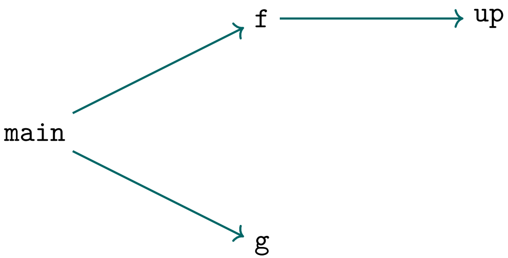
```c
char up(char c) {
    if (c >= 'a' && c <= 'z') {
        return c - ('a' - 'A');
    }
    return c;
}
```

```c
void g(void) {
    puts("Hello, World!");
}
```

Example in **C**
(No dynamic dispatch yet. . .)

The Call Graph

- $G_{\mathsf{call}} = \langle P, E_{\mathsf{call}} \rangle$
- Connects procedures from P via call edges from E_{call}
- 'Which procedure can call which other procedure?'
- Often refined to:
  'Which *call site* can call which procedure?'
- Used by program analysis to find procedure call targets



Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = {new A(), new B()};
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

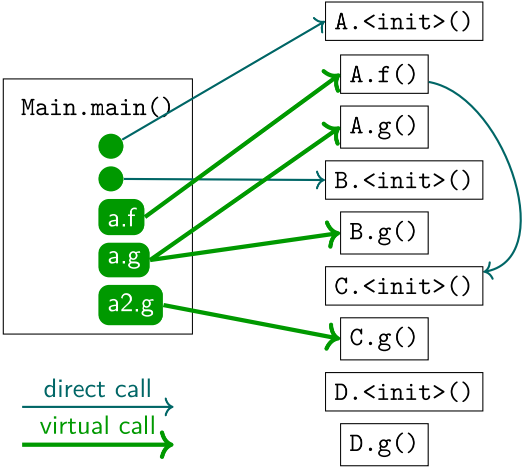
```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Dynamic Dispatch: Call Graph

Challenge: Computing the precise call graph:



Summary

- **Call Graphs** capture which procedure calls which other procedure
- For program analysis, further specialised to map:

$$\text{Callsite} \rightarrow \text{Procedure}$$

- Direct calls: straightforward
- Virtual calls (dynamic dispatch):
  - Multiple targets possible for call
  - No fully sound/precise solution in general

Finding Calls and Targets

```java
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

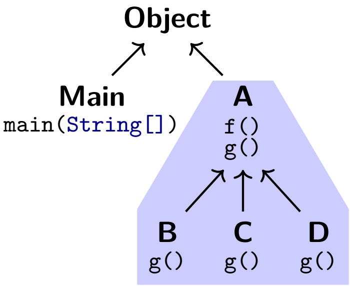
```java
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```java
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```java
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

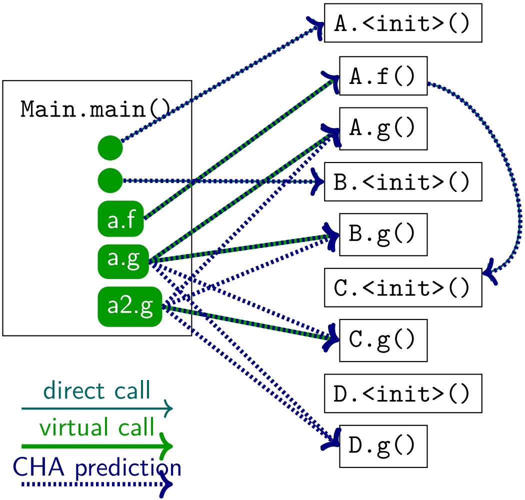
```java
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Class Hierarchy Analysis



- Use declared type to determine possible targets
- Must consider all possible subtypes
- In our example: assume a.f can call any of: `A.f()`, `B.f()`, `C.f()`, `D.f()`

Class Hierarchy Analysis: Example



Summary

- **Call Hierarchy Analysis** resolves virtual calls $a.f()$ by:
  - Examining static types T of receivers $(a : T)$
  - Finding all subtypes $S <: T$
  - Creating call edges to all $S.f$, if $S.f$ exists
- **Sound**
  - Assuming strongly and statically typed language with subtyping
  - Assuming whole-program knowledge (no dynamic classloading)
- Not very **precise**
  - Java: `((Object) obj).toString()`:
    Will use *all* `toString()` methods *anywhere*

Rapid Type Analysis

- Intuition:
  - Only consider reachable code
  - Ignore unused classes
  - Ignore classes instantiated only by unused code

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

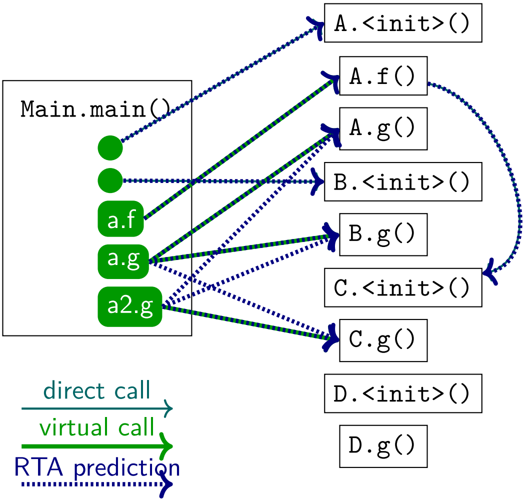
```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g() { return "D"; }
}
```

```
class C extends A {
  @Override
  public String
  g() { return "C"; }
}
```

```
class B extends A {
  @Override
  public String
  g() { return "B"; }
}
```

Rapid Type Analysis: Example



Rapid Type Analysis Algorithm Sketch

Procedure RTA(mainproc, $<:$):
begin
  $\textsc{Worklist} := \{\text{mainproc}\}$
  $\textsc{VirtualCalls} := \emptyset$
  $\textsc{LiveClasses} := \emptyset$
  while $s \in$ mainproc **do**
    foreach call $c \in s$ **do**
      if c is direct call to p **then**
        addToWorklist(p)
        registerCallEdge($c \rightarrow p$)
      else if c = v.m() **and** $v : T$ **then begin**
        $\textsc{VirtualCalls} := \textsc{VirtualCalls} \cup \{c\}$
        foreach $S <: T$ **do**
          addToWorklist(S.m)
          registerCallEdge($c \rightarrow$ S.m)
        done
      end else if c = **new** $C()$ **and** $C \notin \textsc{LiveClasses}$ **then begin**
        $\textsc{LiveClasses} := \textsc{LiveClasses} \cup \{C\}$
        foreach $v.m() \in \textsc{VirtualCalls}$ with $v : T$ and $C <: T$ **do**
          addToWorklist(C.m)
          registerCallEdge($c \rightarrow$ C.m)
        done
      end
done done end

Summary

- **Rapid Type Analysis** resolves virtual calls $a.f()$ as follows:
  - Find all classes that can be instantiated in reachable code
  - Expand reachable code:
    - For direct calls to p, add p as reachable
    - For all virtual calls to $v.m()$ with $v : T$:
      $\Rightarrow$ Add $S.m()$ as reachable
  - Iterate until we reach a fixpoint
- **Sound**
  - Assuming strongly and statically typed language with subtyping
- More **precise** than Class Hierarchy Analysis

Finding Calls and Targets

```
class Main {
  public void
  main(String[] args) {
    A[] as = { new A(), new B() };
    for (A a: as) {
      A a2 = a.f();
      print(a.g());
      print(a2.g());
    }
  }
}
```

```
class A {
  public A
  f() { return new C(); }

  public String
  g() { return "A"; }
}
```

```
class D extends A {
  @Override
  public String
  g()
}
```

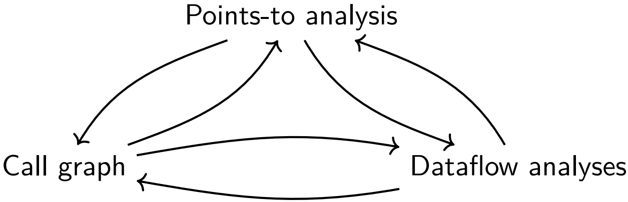
```
class C extends A {

  public String
```

```
class B extends A {
  @Override
  public String
  eturn "B"; }
```

Use **points-to analysis**?

But what call graph should the points-to analysis use?

Dependencies



- Mutual dependencies across program analyses

Analysis Composition

How do we handle mutual dependencies?

Analysis Composition: Example

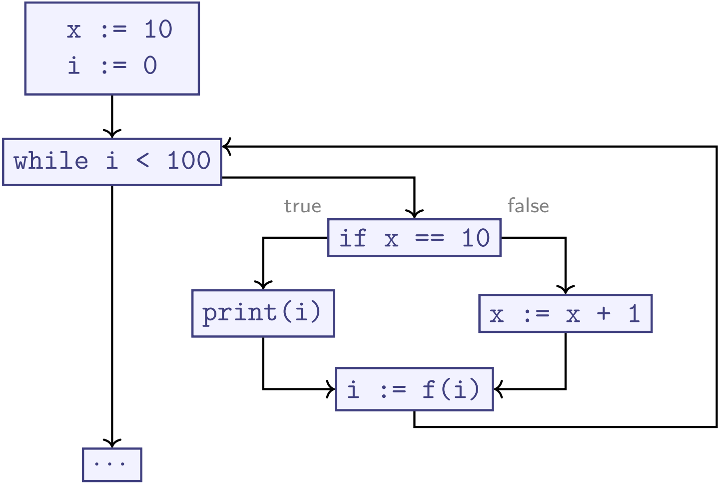
Teal

```
var x := 10;
var i := 0;
while i < 100 {
  if x == 10 {
    print(i);
  } else {
    x := x + 1;
  }
  i := f(i);
}
```

Always true

Adapted from Sorin Lerner, David Grove, Craig Chambers: "Composing Dataflow Analyses and Transformations", ACM SIGPLAN Conference on Principles of Programming langauges (POPL 2002)
Partly attributed to Mark N. Wegman and F. Kenneth Zadeck: "Constant Propagation with Conditional Branches", TOPLAS vol. 13(2), April 1991, 181–210

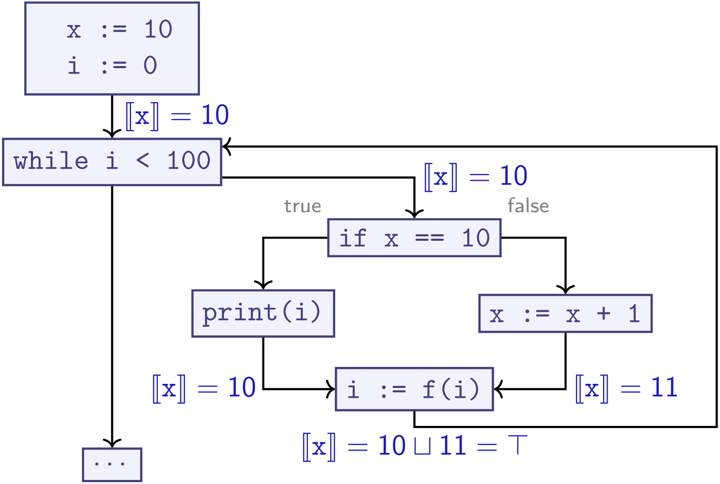
Analysis Composition: Loose (1/2)



- **First**: Unreachable Path Elimination
- **Second**: Constant Propagation / Constant Folding

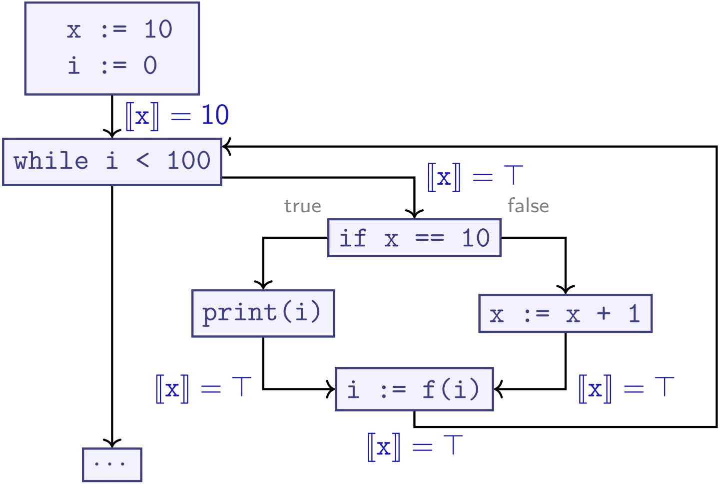
Unreachable Path Elimination can't evaluate any conditionals here

Analysis Composition: Loose (2/2)



- **First**: Constant Propagation / Constant Folding
- **Second**: Dead Path Elimination

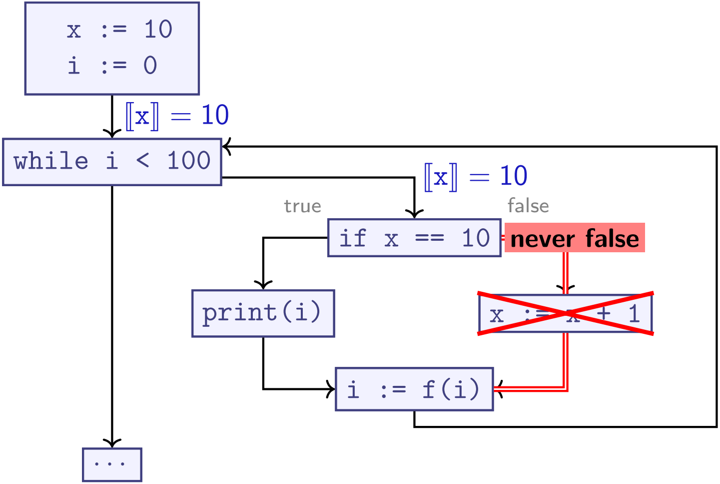
Analysis Composition: Loose (2/2)



- **First**: Constant Propagation / Constant Folding
- **Second**: Dead Path Elimination

With $[\![x]\!] = \top$, Dead Path Elimination can't proceed

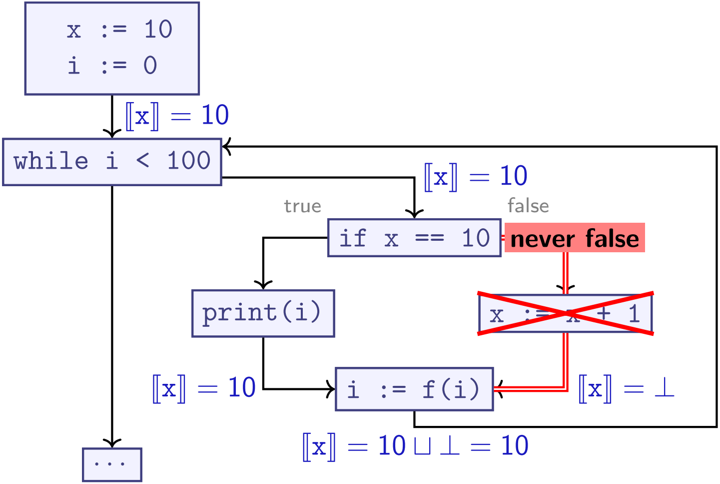
Analysis Composition: Tight



"Tight Composition": Run analyses *together*:

Constant Propagation / Folding & Dead Path Elimination

Analysis Composition: Tight



"Tight Composition": Run analyses *together*:
Constant Propagation / Folding & Dead Path Elimination

Executing at the same time gives correct result!

Loose Composition

Loose Composition: **Split analyses into multiple passes**

- Each pass finishes before next pass starts
- Standard approach in compilers

Tight Composition

Tight Composition: **Analyses depend on each other's intermediate results**

- Analyses run "together"
- *Not widely supported*
- Systemic support:
  - Reference Attribute Grammars (JastAdd etc.) with circular attributes
  - Logic programming (Datalog, Prolog)
  - Term Rewriting (Vortex/Cyclone/)
- **Challenges**:
  - Traditional worklist algorithms:
    - Complex manual engineering needed
  - Declarative approaches (JastAdd, Logic Programming):
    - Must guarantee **Monotonicity**

Summary

- Mutual dependencies between program analyses are common
- Two approaches:
  - **Loose composition**:
    - One analysis after the other
    - May need to run analyses multiple times
    - Strictly less powerful than tight composition
  - **Tight composition**:
    - Analyses can use each other's intermediate results
    - Difficult to engineer for worklist algorithms
    - Easier with declarative approaches (attribute grammars, logic programming, term rewriting)
    - **Caveat**: Lattices must be "aligned": monotone updates in one lattice must not require nonmonotone updates in another!

Analysing Realistic Programs

Challenges:

- **Semantics**:
  - Language semantics may be imprecisely defined (e.g., custom or domain-specific languages)
  - Certain **language features** intrinsically hard to analyse
- Non-Semantic Properties:
  - Property of interest may not be part of semantics
  - Examples: execution time, energy usage

Reflection

Java

```java
Class<?> cl = Class.forName(string);
Object obj = cl.getConstructor().newInstance();
System.out.println(obj.toString());
```

- Instantiates object by string name
- Similar features to call method by name
- **Challenge**:
  - `obj` may have *any* type $\Rightarrow$ imprecision
  - Sound call graph construction very conservative
- **Approaches**
  - Dataflow: what strings flow into `string`?
    - Common: code draws from finite set or uses string prefix/suffix (e.g., ("com.x.plugins." + ...))
    - `Class.forName`: class only from some point in package hierarchy
  - Dynamic analysis

Dynamic Loading

C

```
handle = dlopen("module.so", RTLD_LAZY);
op = (int (*)(int)) dlsym(handle, "my_fn");
```

- Dynamic library and class loading:
  - Add new code to program that was not visible at analysis time
- **Challenge**:
  - Can't analyse what we can't see
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *all that it can reach*
  - With dynamic support and static annotation:
  - Allow only loading of signed/trusted code
    - signature must guarantee properties we care about
    - annotation provides properties to static analysis
  - *Proof-carrying code*
    - Code comes with proof that we can check at run-time

Native Code

Java

```java
class A {
  public native Object op(Object arg);
}
```

- High-level language invokes code written in low-level language
  - Usually C or C++
  - May use nontrivial interface to talk to high-level language
- **Challenge**:
  - High-level language analyses don't understand low-level language
- **Approaches**:
  - Conservative approximation
    - Tricky: External code may modify *anything*
  - Manually model known native operations (e.g., Doop)
  - Multi-language analysis (e.g., Graal)

‘eval’ and dynamic code generation

Python

```
eval(raw_input())
```

- Execute a string as if it were part of the program
- **Challenge**:
  - Cannot predict contents of string in general
- **Approaches**:
  - Conservative approximation
    - Tricky: code may modify *anything*
  - Dynamically re-run static analysis
  - Special-case handling (cf. reflection)

Summary

- Static program analysis faces significant challenges:
  - **Decidability** requires lack of precision or soundness for most of the interesting analyses
  - **Reflection** allows calling methods / creating objects given by arbitrary string
  - **Dynamic module loading** allows running code that the analysis couldn't inspect ahead of time
  - **Native code** allows running code written in a different language
  - **Dynamic code generation** and `eval` allow building arbitrary programs and executing them
  - No universal solution
  - Can try to 'outlaw' or restrict problematic features, depending on goal of analysis
  - Can combine with dynamic analyses

Soundiness

- Can't analyse language feature?

$\Rightarrow$ We get $\top$ if we want soundness

$\Rightarrow$ Potentially many false positives

$\Rightarrow$ Tool may be useless

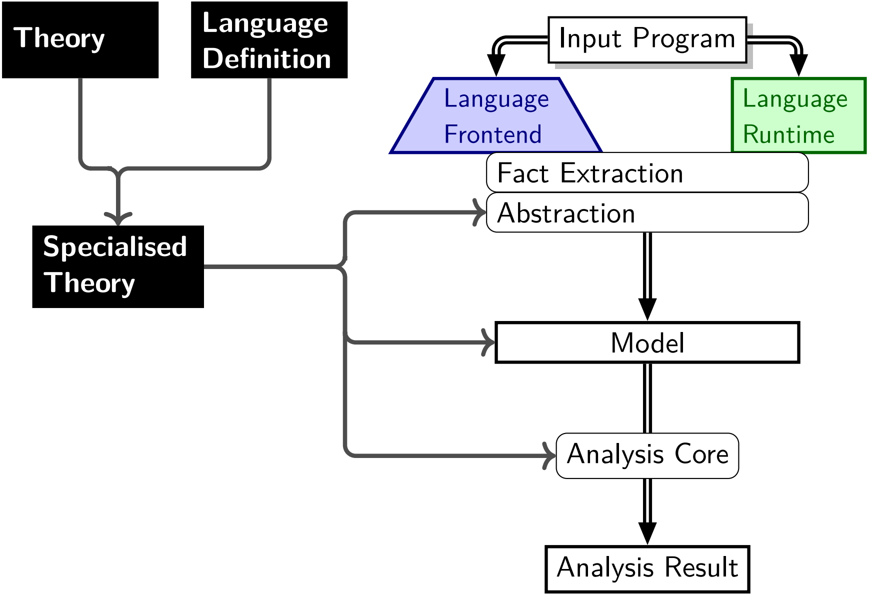
  - Google SWE practice: Bug checkers with $> 5\%$ false positives disabled automatically

- Soundness may not be *useful*
- Alternative proposal from research community: **Soundiness**
  - *Be explicit* about unsupported language features
  - Example: "Sound unless the code uses features X, Y, Z"

Soundiness: "capture all dynamic behaviour *within reason*"

B. Livshits, M. Sridharan, Y. Smaragdakis et al.: "In defense of Soundiness: A Manifesto", Communications of the ACM, 2015

Building a Program Analysis



Lecture Overview

Foundations

- 01 Foundations
- 02 Constructing Program Analyses in JastAdd

Static Analysis

Properties

- 03 04 Types
- 05 06 07 Data Flow
- 08 09 Memory

Control Flow

- 05 Intraprocedural
- 10 Interprocedural
- 11 **Indirect**

Dynamic Analysis

- 12 Instrumentation
- 13 Analysis

14 Review

Outlook

- Next lecture: Partly **Guest Lecture**
  - Patrik Åberg & Magnus Templing, Ericsson: Code Instrumentation with DMCE

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