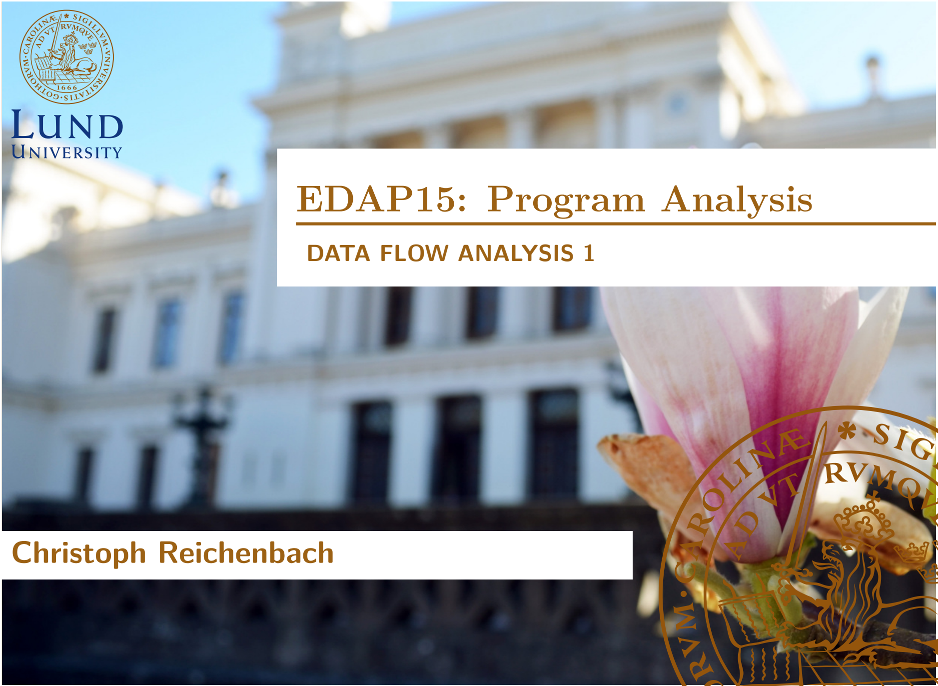


Lund University

EDAP15: Program Analysis

DATA FLOW ANALYSIS 1

Christoph Reichenbach

Announcements

- Registered for lab slots?
- Exercises start on Friday
- Homework 1 out on **Thursday**, deadline extended by 1 day

A New Analysis Challenge

Teal

```
var x := [0, 0];
print(x);     // A
if z {
  x[0] := 2; // B
  x := null;
}
x[0] := 1;    // C
```

- Analyse: Can there be a *failure* at B or C?
- Must distinguish between x at A vs. x at B and C
- Need to model flow of information: **Flow-Sensitive Analysis**
- Type analysis is *not Flow-Sensitive* (normally)

Need analysis that can represent *data flow* through program

Evaluation Order

Teal-0

```
fun p(a) = { print(a); return 1; }
p(p(0) + p(1));
```

Teal-0 with explicit order

```
var tmp1 := p(0);
var tmp2 := p(1);
var tmp3 := tmp1 + tmp2;
var tmp4 := p(tmp3);
```

Java or C or C++

```
// Many challenging constructions:
a[i++] = b[i > 10 ? i-- : i++] + c[f(i++, --i)];
```

Every analysis must remember the evaluation order rules!

Eliminating Nested Expressions

- No nested expressions

$\Rightarrow$ Evaluation order is explicit

$\Rightarrow$ Fewer patterns to analyse

- All intermediate results have a name

$\Rightarrow$ Easier to ‘blame’ subexpressions for errors

  - Names might be represented pointers in the implementation

- We still have nested statements

Multiple Paths

Teal

```
v := new array[int](1);
if condition {
  v := null;
} else {
  print(v);
}
v[0] := 1;
```

Teal

```
v := new array[int](1);
while condition {
  v := null;
}
v[0] := 1;
```

Need to reason about the order of execution of *statements*, too

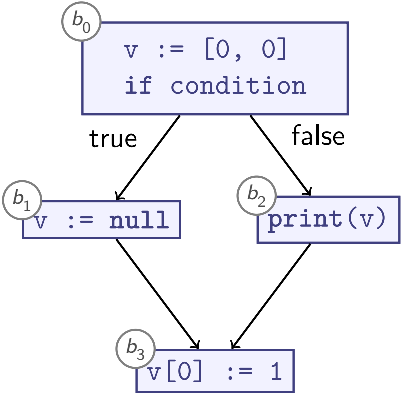
Summary

- Understanding variable updates requires **Flow-Sensitive Analysis**
- Type analysis is *not* flow sensitive
- "Flow" is complicated, influenced by:
  - Expression evaluation order
  - Short-circuit evaluation
  - Statement execution order
- Best analysed with special intermediate representation:
  - Flatten nested expressions
  - Introduce temporary variables as needed
  - . . . do something about statement execution? (up next!)

Control-Flow Graphs (CFGs)

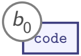
Teal

```
var v := [0, 0];
if condition {
  v := null;
} else {
  print(v);
}
v[0] := 1;
```

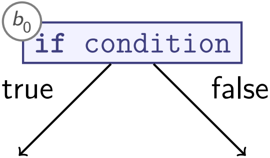


Control Flow Graphs encode statement execution order

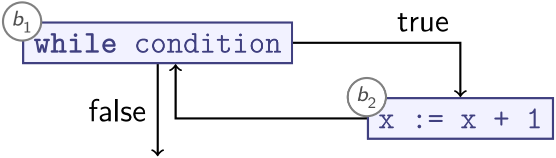
Control-Flow-Graphs

- Encode statement order by *nodes* 

and edges ⟶
- *Multiple* outgoing edges (branches): Add label:

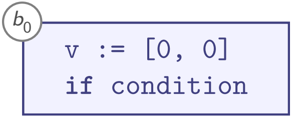


- Uniform representation for control statements:

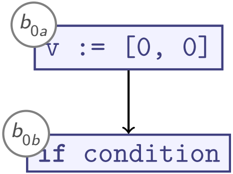


Basic Blocks

Can group statements into **Basic Blocks** or keep them separate:



Basic Block



- A **Basic Block** is a sequence of statements
- Last statement is *always* return, branch, or jump
- Other statements are *never always* return, branch, or jump
- Usually faster to process

Summary

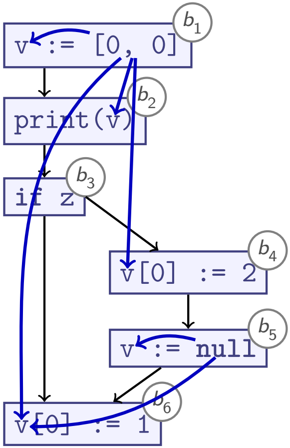
- Different **Intermediate Representations** (IRs) to pick
- Usually eliminate nested expressions
  - Make evaluation order explicit
- **Control-Flow Graph** (CFG):
  - Represent control flow as **Blocks** and **Control-Flow Edges**
  - Edges represent control flow, **labelled** to identify conditionals
  - Blocks can be single statements or **Basic Blocks**
    - Basic blocks are sequences of statements without branches

Control Flow

Understanding data flow requires understanding control flow:

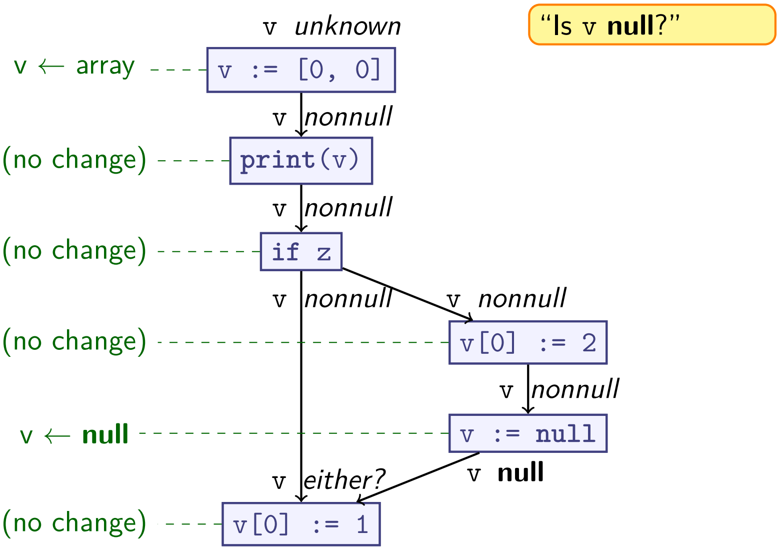
Teal

```
var v := [0, 0];
print(v);
if z {
  v[0] := 2;
  v := null;
}
v[0] := 1;
```



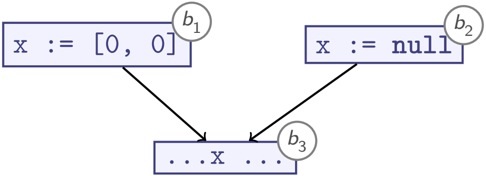
⟶ Control flow
⟶ Data flow

Intuition behind Data Flow Analysis



Knowledge about data "flows" through CFG

What does “either?” mean?



- Should analysis report `x` as **null** or as **nonnull**?
  - New category: **either**
  - “Can I safely dereference without a check?”
    $\Rightarrow$ better assume **null**
  - “is this guaranteed to be null?”
    $\Rightarrow$ better assume **nonnull**
- We might not need extra **either** category, depending on why we are analysing

“May” vs “Must” Analysis

- “**May**” analysis: we *cannot rule out* property
  - “either?” becomes **true**
  - Avoids *False Negatives*
- “**Must**” analysis: we *can guarantee* property
  - “either?” becomes **false**
  - Avoids *False Positives*

Another Analysis

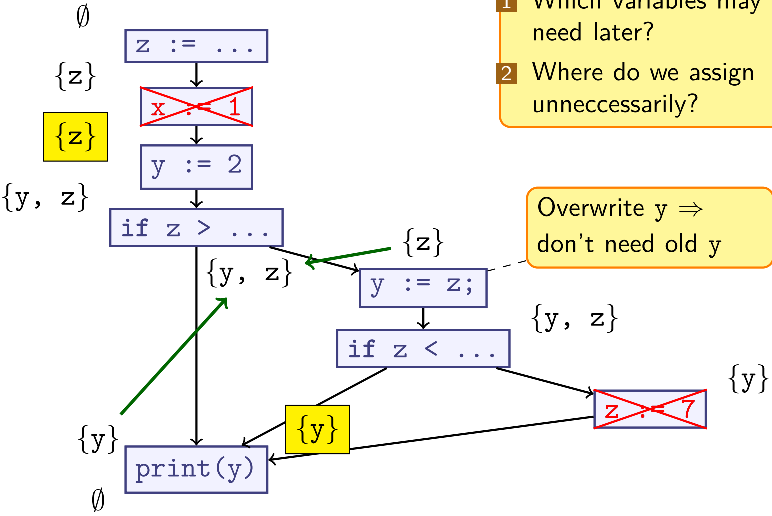
Teal

```
z := ...
x := 1;
y := 2;
if z > ... {
  y := z
  if z < ... {
    z := 7
  }
}
print(y);
```

- Which assignments are unnecessary?

$\Rightarrow$ Possible oversights / bugs
(*Live Variables Analysis*)

Unnecessary Assignments: Intuition



1. Which variables may we need later?
2. Where do we assign unneccessarily?

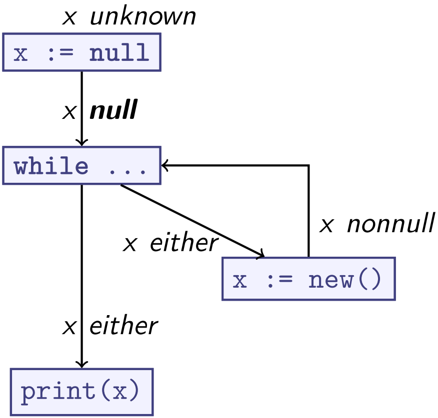
Overwrite y ⇒ don't need old y

Analysis effective: found useless assignments to z and x

Observations

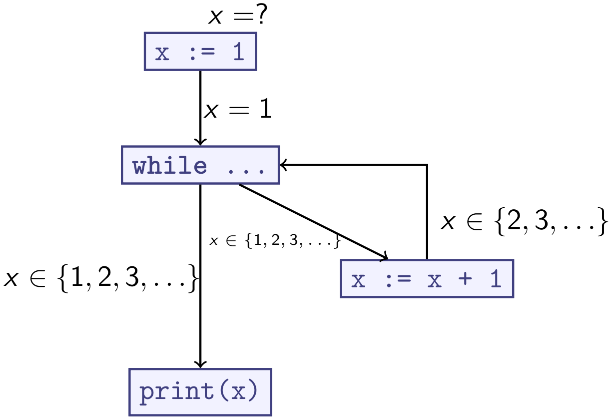
1. Data Flow analysis can be run *forward* or *backward*
2. May have to *join* results from multiple sources
3. Some analyses may need multiple “passes” (steps)

What about Loops? (1/2)



- Analysis: *Null Pointer Dereference*
- Stop when we're not learning anything new any more
- Works fine

What about Loops? (2/2)



- Analysis: *Reaching Definitions*

We need to bound repetitions!

Summary: Data-Flow Analysis (Introduction)

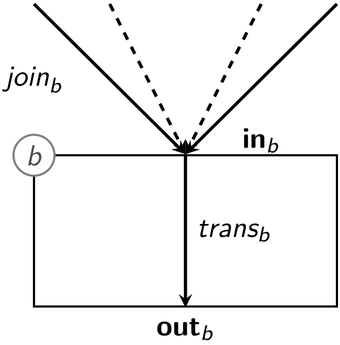
- Data flow depends on *control flow*
- Data flow analysis examines how variables or other program state change across control-flow edges
- May have to join multiple results
- When joining "yes" and "no", must decide:
  - "**May**" analysis: optimistically report what is possible
  - "**Must**" analysis: conservatively report what is guaranteed
  - Alternative: introduce value for "don't know"
- Can run *forward* or *backward* relative to control flow edges
- Handling loops is nontrivial

Engineering Data Flow Algorithms

1. General Algorithm
   - Keep updating until nothing changes
2. Termination
   - Assumption: Operate on Control Flow Graph
   - Theory: Ensure termination
3. (Correctness)

Data Flow Analysis on CFGs

- $\mathbf{in}_b$: knowledge at entrance of basic block b
- $\mathbf{out}_b$: knowledge at exit of basic block b
- $join_b$: combines all $\mathbf{out}_{b_i}$ for all basic blocks b_i that flow into b "**Join Function**"
- $trans_b$: updates $\mathbf{out}_b$ from $\mathbf{in}_b$ "**Transfer Function**"



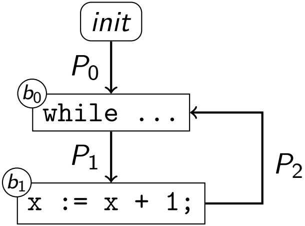
Characterising Data Flow Analyses

Characteristics:

- *Forward* or *backward* analysis
- L: Abstract Domain (the 'analysis domain')
- $trans_b : L \rightarrow L$
- $join_b : L \times L \rightarrow L$

Require properties of L, $trans_b$, $join_b$ to ensure termination

Limiting Iteration



- Does the following ever stop changing:

$$\mathbf{in}_{b_0} = join_{b_0}(P_0, P_2)$$

- Intuition: we keep generalising information
  - *Growth limit*: bound amount of generalisation
  - Make sure $join_b$, $trans_b$ never throw information away

Eventually, either nothing changes or we hit growth limit

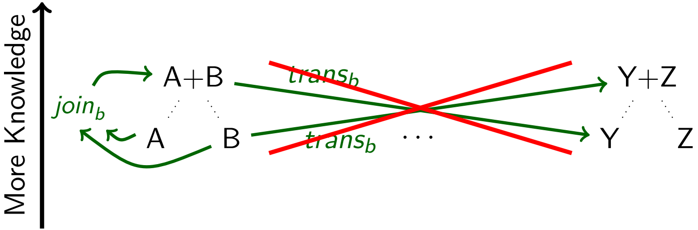
Ordering Knowledge

$$B \sqsupseteq A$$

- B describes at least as much knowledge as A
- Either:
  - $A = B$ (i.e., $A \sqsupseteq B \sqsupseteq A$), or
  - B has strictly more knowledge than A

Intuition: Knowing Less, Knowing More

Structure of L:

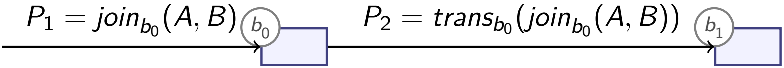


- $join_b$ must not lose knowledge
  - $join_b(A, B) \sqsupseteq A$
  - $join_b(A, B) \sqsupseteq B$
- $trans_b$ must be *monotonic* over amount of knowledge:

$$x \sqsupseteq y \implies trans_b(x) \sqsupseteq trans_b(y)$$

- Introduce bound: $\top$ means 'too much information'

Aggregating Knowledge

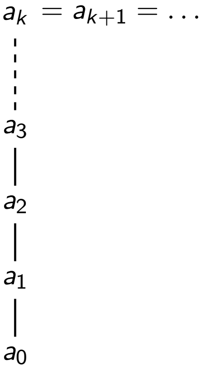


- Interplay between $trans_b$ and $join_b$ helps preserve knowledge
- $join_b(A, B) \sqsupseteq A$:
  As we add knowledge, P_1 either
  - Stays the same
  - Increases knowledge
- Monotonicity of $trans_b$: If P_1 goes up, then P_2 either
  - Stays the same
  - Increases knowledge

$\Rightarrow$ At each node, we either stay equal or grow

Now we must only set a growth limit...

Ascending Chains



- A (possibly infinite) sequence $a_0, a_1, a_2, \ldots$ is an *ascending chain* iff:

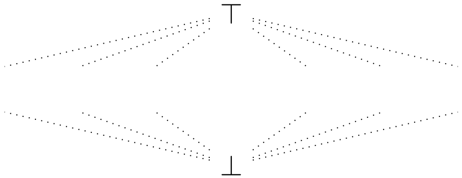
$$a_i \sqsubseteq a_{i+1} \text{ (for all } i \geq 0)$$

- *Ascending Chain Condition*:
  - For *every* ascending chain $a_0, a_1, a_2, \ldots$ in abstract domain L:
  - there exists $k \geq 0$ such that:

$$a_k = a_{k+n} \text{ for any } n \geq 0$$

ACC is formalisation of growth limit

Top and Bottom



- *Convention*: We introduce two distinguished elements:
  - **Top**: $\top$: $A \sqsubseteq \top$ for all A
  - **Bottom**: $\bot$: $\bot \sqsubseteq A$ for all A
- Since $join_b(A, B) \sqsupseteq A$ and $join_b(A, B) \sqsupseteq B$:
  - $join_b(\top, A) = \top = join_b(A, \top)$
  - $join_b(\bot, A) \sqsupseteq A \sqsupseteq \bot$
    - In practice, it's safe and simple to set: $join_b(\bot, A) = A = join_b(A, \bot)$
- Intuition:
  - $\top$: means 'contradictory / too much information'
  - $\bot$: means 'no information known yet'

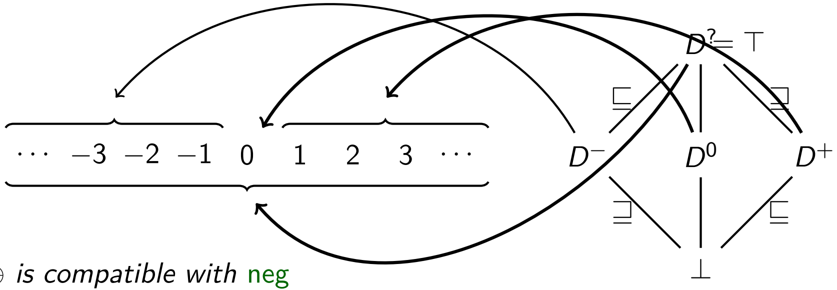
Summary

- Designing a *Forward* or *backward* analysis:
- Pick **Abstract Domain** L
  - Must be **partially ordered** with $(\sqsupseteq) \subseteq L \times L$:
    $A \sqsupseteq B$ iff A 'knows' at least as much as B
  - Unique top element $\top$
  - Unique bottom element $\bot$
- $trans_b : L \to L$
  - Must be *monotonic*:

$$x \sqsupseteq y \implies trans_b(x) \sqsupseteq trans_b(y)$$

- $join_b : L \times L \to L$ must produce an *upper bound* for its parameters:
  - $join_b(A, B) \sqsupseteq A$
  - $join_b(A, B) \sqsupseteq B$
- Satisfy **Ascending Chain Condition** to ensure termination
  - Easiest solution: make L finite

Abstract Domains Revisited



$\ominus$ *is compatible with* neg

$$
\begin{aligned}
\ominus \bot &= \bot \\
\ominus D^0 &= D^0 \\
\ominus D^+ &= D^- \\
\ominus D^- &= D^+ \\
\ominus D^? &= D^?
\end{aligned}
$$

$\ominus$ is monotonic (and $\oplus$ extended with $\bot$ is, too)

Summary

- We can extend $\{D^+, D^-, D^0, D^?\}$ by adding $\perp$

$$L_D = \{D^+, D^-, D^0, D^?, \perp\}$$

- $\perp$ representing "not known" – not needed for our example analysis from Lecture 1, but would be needed if we had variables / control flow in that language
- L_D is finite, so the DCC holds trivially
- Our *Transfer Functions* $\ominus, \oplus$ are monotonic