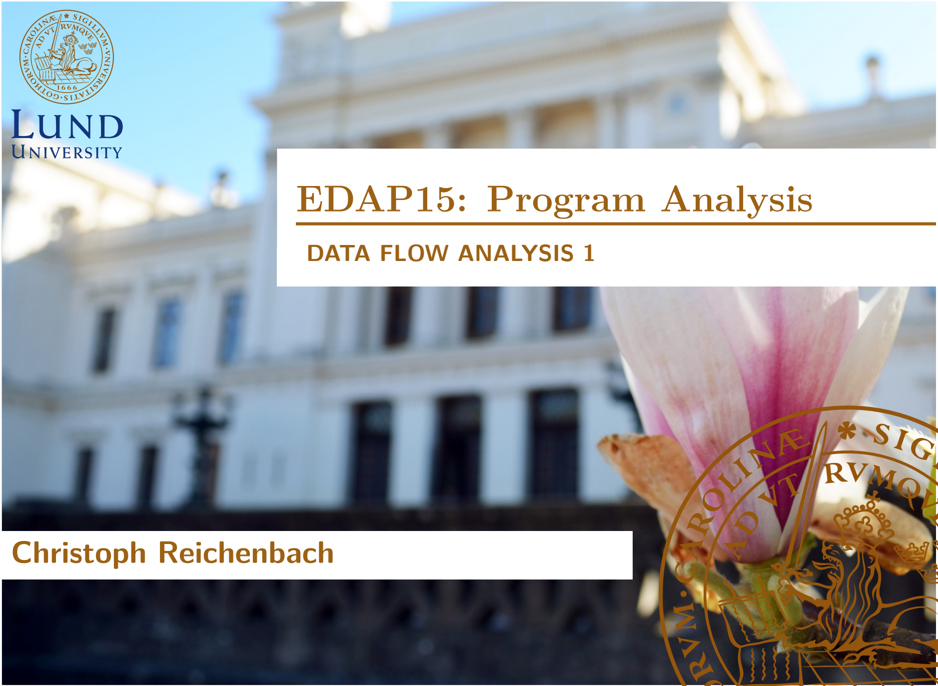


LUND UNIVERSITY

EDAP15: Program Analysis

DATA FLOW ANALYSIS 1

Christoph Reichenbach

Welcome back!

Some Administrativa:

- Labs:
- Ideally: done with exercise 0, working on exercise 1
- No new exercise this week
- Optional polls on how much time exercises took in Moodle

Questions?

Getting More out of Type Inference (1/4)

- Recall our typing rules from last lecture:

$$\frac{}{\texttt{true} : \textsc{Bool}} \qquad \frac{}{\texttt{false} : \textsc{Bool}}$$

$$\frac{e_1 : \textsc{Bool} \quad e_2 : \tau \quad e_3 : \tau}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}$$

- Could we make them more precise so we can e.g. tell that:
  - `if false then false else true` evaluates to `true`
  - `if true then false else false` evaluates to `false`

Getting More out of Type Inference (2/4)

Replacing $\textsc{Bool}$ by *more precise types* $\textsc{True}$ and $\textsc{False}$:

$$\frac{}{\texttt{true} : \textsc{True}} \qquad \frac{}{\texttt{false} : \textsc{False}}$$

$$\frac{e_1 : \textsc{True} \quad e_2 : \tau}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}\ (\textit{if-true})$$

$$\frac{e_1 : \textsc{False} \quad e_3 : \tau}{\texttt{if } e_1 \texttt{ then } e_2 \texttt{ else } e_3 : \tau}\ (\textit{if-false})$$

We can now infer:

$$\frac{\dfrac{}{\texttt{false} : \textsc{False}} \quad \dfrac{}{\texttt{true} : \textsc{True}}}{\texttt{if false then false else true} : \textsc{True}}\ (\textit{if-false})$$

Getting More out of Type Inference (3/4)

- Consider:

```
fun(x) = if x then false else true
```

- Can't know x in general $\Rightarrow$ we must allow both:
  - x : TRUE
  - x : FALSE

$\Longrightarrow$ We don't have a principal type any more

- How would this work for `int`?

Getting More out of Type Inference (4/4)

- Let's try it:
  - Using types: 0, 1, 2, ...

```
if x then 0 else 1   :   0
if x then 0 else 1   :   1
```

- No principal type, but two typings
- In larger programs:

Python

```python
print( (1 if a[0] else 0)
     + (2 if a[1] else 0)
     + (4 if a[2] else 0)
      ...
     + (2**999 if a[999] else 0) )
```

- 2^{1000} possible types!

Exponential analysis cost $\Longrightarrow$ too slow to analyse real programs!

Towards Abstract Interpretation

- Consider the following language:

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

- **Property of Interest**:
  Does a given program $\varphi \in E$ compute a number ≥ 0?
- We will use a different theoretical framework now:
  Abstract Interpratation
- Similar in many ways, but more suitable for the "subtyping"-like behaviour we just saw

Abstract Domains

- *Abstract Interpretation*:
  - Map all values to a simpler *abstract domain*

- Example: set *abstract domain* $\mathcal{D}$:

$$
\mathcal{D} = \{ \\
\quad D^0, \quad \text{Program computes 0} \\
\quad D^+, \quad \text{Program computes a positive value} \\
\quad D^- \quad \text{Program computes a negative value} \\
\}
$$

Patrick Cousot & Radhia Cousot, "Abstract Interpretation", published in Principles of Programming Languages, 1977

Correspondence: Concrete and Abstract

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

Abstract Domains

- *Abstract Interpretation*:
  - Map all values to a simpler *abstract domain*

- Example: set *abstract domain* $\mathcal{D}$:

$$\mathcal{D} = \{ \quad D^0, \ \text{Program computes 0} \quad D^+, \ \text{Program computes a positive value} \quad D^- \ \text{Program computes a negative value} \quad \}$$

- Notation: $\boxed{[\![\varphi]\!]^{\mathcal{D}} = a}$, where $a \in \mathcal{D}$

Patrick Cousot & Radhia Cousot, "Abstract Interpretation", published in Principles of Programming Languages, 1977

Abstract Interpretation

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

$$
\begin{array}{rcl}
E & ::= & \mathsf{zero} \\
 & \mid & \mathsf{one} \\
 & \mid & \langle E \rangle + \langle E \rangle \\
 & \mid & \mathsf{neg}\ \langle E \rangle
\end{array}
$$

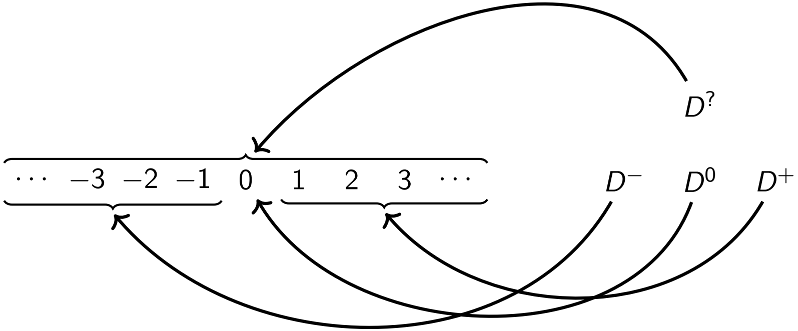
$$
a_1 \oplus a_2 = \left\{ \begin{array}{c||c|c|c|}
 & D^+ & D^0 & D^- \\
\hline\hline
D^+ & D^+ & D^+ & D^? \\
D^0 & D^+ & D^0 & D^- \\
D^- & D^? & D^- & D^-
\end{array} \right.
$$

$$D^? \oplus a = D^? = a \oplus D^?$$

$$[\![\mathsf{zero}]\!]^{\mathcal{D}} = D^0 \qquad [\![\mathsf{one}]\!]^{\mathcal{D}} = D^+$$

$$\frac{[\![x]\!]^{\mathcal{D}} = a}{[\![\mathsf{neg}\ x]\!]^{\mathcal{D}} = \ominus a} \qquad \frac{[\![x]\!]^{\mathcal{D}} = a_1 \quad [\![y]\!]^{\mathcal{D}} = a_2}{[\![x + y]\!]^{\mathcal{D}} = a_1 \oplus a_2}$$

Correspondence: Concrete and Abstract



Abstract Domains

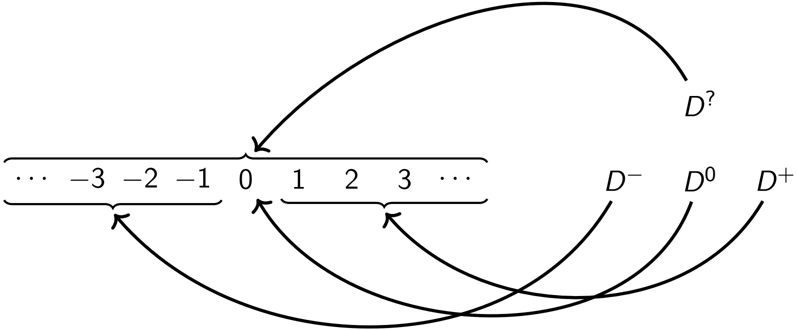
- *Abstract Interpretation*:
  - Map all values to a simpler *abstract domain*
  - For each operation, build corresponding *abstract operation*
- Example: set *abstract domain* $\mathcal{D}$:

$$\mathcal{D} = \{ \\ \begin{array}{ll} D^0, & \text{Program computes 0} \\ D^+, & \text{Program computes a positive value} \\ D^-, & \text{Program computes a negative value} \\ D^? & \text{Program computes any value} \end{array} \\ \}$$

- Notation: $\boxed{[\![\varphi]\!]^{\mathcal{D}} = a}$, where $a \in \mathcal{D}$

Patrick Cousot & Radhia Cousot, "Abstract Interpretation", published in Principles of Programming Languages, 1977

Correspondence: Concrete and Abstract



Also:

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

Will return later to examine connections between elements in $\mathcal{D}$

Summary

- **Abstract Interpretation** maps
  concrete values to "abstract value" in **Abstract Domain**
  - Mapping may describe interpretation of values, expressions, statements, whole programs:

$$[\![-]\!]^{\mathcal{D}} : \mathit{Program} \to \mathcal{D}$$

  - Maps operations to *compatible* operations on abstract domain
  - Many design options for abstract domain:
    - Can trade off between precision and efficiency
  - Theoretical foundation/generalisation of other analysis theories
- $[\![-]\!]^{\mathcal{D}}$ must be a *function*
  - Example: type inference with principal types
  - Non-example: type inference without principal types

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

Teal

Teal-0	Imperative and Procedural
Teal-1	Minor extensions to Teal-0

- Small enough for homework exercises
- Big enough to exhibit real challenges
- "Nonsensical" operations in Teal trigger dynamic *failure*s:
  - `null` dereference:

    Teal

    ```
    var a := null;
    print(a[0]);
    ```

  - Array-out-of-bounds access:

    Teal

    ```
    var a := [7]; // Array with one element, 7
    print(a[-1]);
    ```

Imperative Code Rears its Head

Teal

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

- Analyse: Can there be a *failure* at `B` or `C`?
- Could we use type variables and unification? **No**:
  - Can't unify: $\llbracket x \rrbracket = D^0$ vs. $\llbracket x \rrbracket = D^-$
  - $\llbracket x \rrbracket$ should be different at different lines
- Must distinguish between `x` at `A` vs. `x` at `B` and `C`
- Need to model program flow: **Flow-Sensitive Analysis**
  - Type inference is *not Flow-Sensitive*

Evaluation Order

Teal-0

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

Teal-0 with explicit order

```
fun main() = {
    var tmp1   := p(0);
    var tmp2   := p(1);
    var tmp3   := tmp1 + tmp2;
    var tmp4   := p(tmp3);
}
```

- Evaluation order specified in language definition

Every analysis must remember the evaluation order rules!

Evaluation Order: Other Languages

- Complex subexpressions / evaluation order:

Java / C / C++

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

- Beware: exact evaluation order is *undefined* in C and C++!
- Short-Circuit Evaluation:

Java (similar in C / C++)

```
int[] a2 = some_array;
bool v = (a == null)
         || ((a2 = a)[0] == 0);
```

- The assignment `a2 = a` is executed while computing `v`
  ... but *only* if `a == null` is **not true**!

Violates most coding styles, but allowed by language!

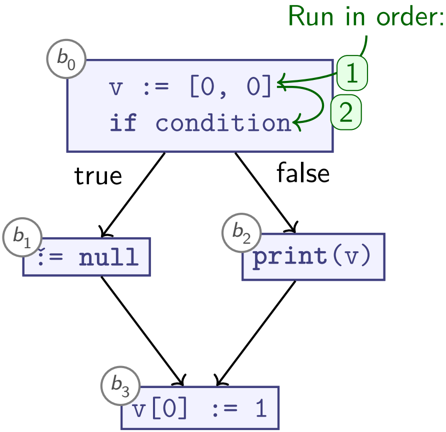
Summary

- Understanding differences before/after variable updates requires **Flow-Sensitive Analysis**
- Type inference is *not* flow sensitive
- "Flow" is complicated, influenced by:
  - Expression evaluation order
  - Short-circuit evaluation
  - Statement execution order

Control-Flow Graphs (CFGs)

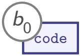
Teal

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

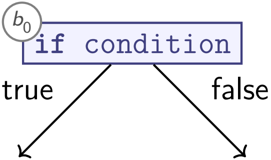


Control Flow Graphs encode statement execution order

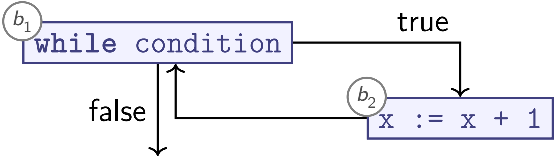
Control-Flow-Graphs

- Encode statement order by *nodes*  and edges →

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



- Uniform representation for control statements:



Summary

Control-Flow Graph (CFG):

- Motivation:
  - Universal representation of control flow
  - Computed once before running analyses
  - Flow-sensitive analyses can utilise CFG
- Idea:
  - Represent control flow as **Blocks** and **Control-Flow Edges**
  - Edges represent control flow, **labelled** to identify conditionals

Control Flow

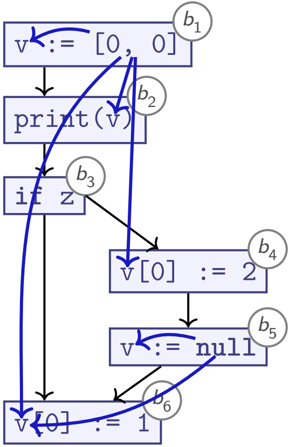
Understanding data flow requires understanding control flow:

Teal

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

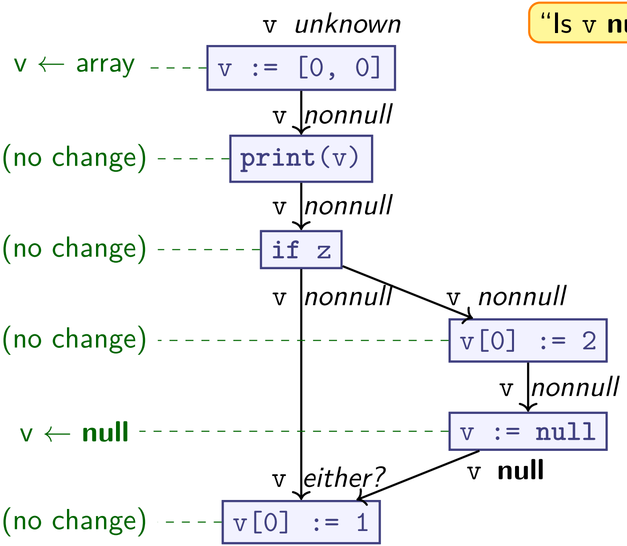
→ Control flow

→ Data flow



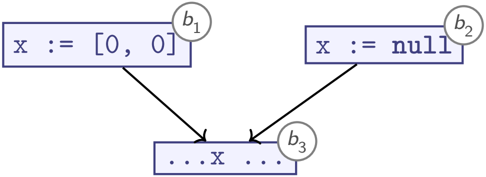
Intuition behind Data Flow Analysis

“Is `v` **null**?”



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 what properties we are looking for

“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

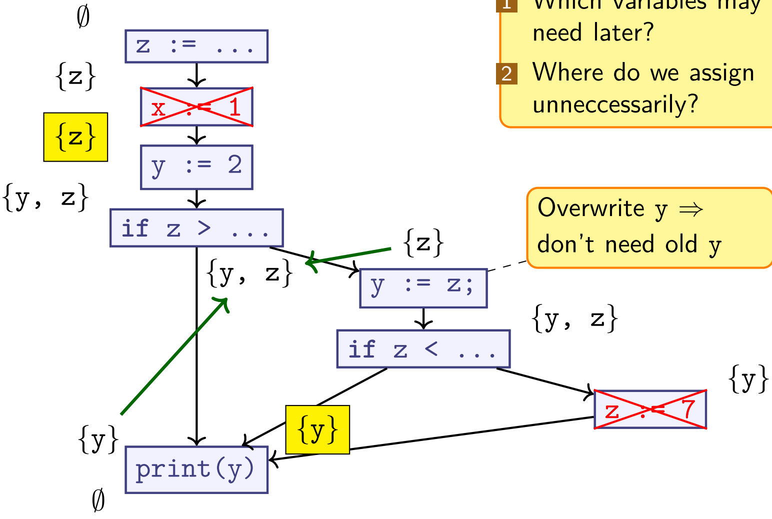
Teal

```
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?

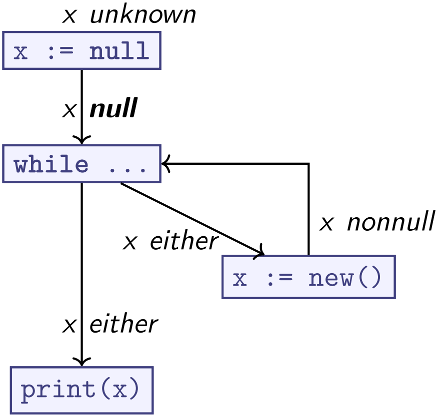
Overwrite y ⇒ don't need old y

Analysis effective: found useless assignments to z and x

Observations

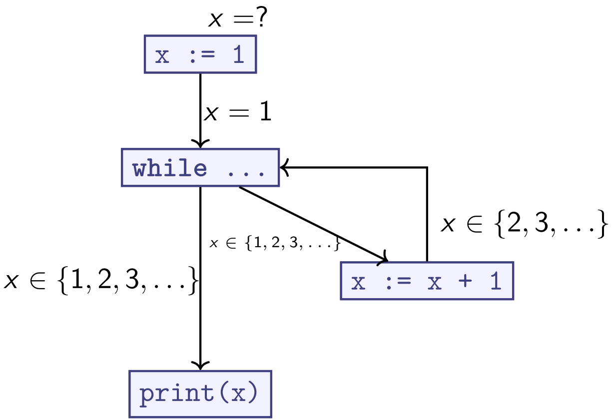
1. Data Flow analysis can be run *forward* or *backward*
2. May have to *join* results from multiple sources

What about Loops? (1/2)



- Analysis: *Null Pointer Dereference*
- May need to analyse each node/edge more than once
- Stop when we're not learning anything new any more

What about Loops? (2/2)



- Two analyses in one:
  - *Constant Propagation*: compute constant 'reaching values'
  - with *Constant Folding*: performs arithmetic on constants

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

Summary: Some Analyses

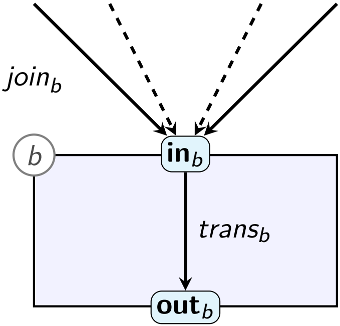
1. **Constant Propagation + Constant Folding**:
   - *What values might our variables contain?*
   - Forward analysis
   - Most common as a *Must* analysis:
     - ‘v has constant value c’, or
     - ‘v might not have constant value’
   - We will also use it as *May* analysis
2. **Live Variables**
   - *Which variables might still be read later in the program?*
   - Backward analysis
   - *May* analysis
3. **Unnecessary Assignments** (also “Dead Assignments”):
   - Refinement of *Live Variables* analysis
   - Flags assignments on variables that are not live

Engineering Data Flow Algorithms

1. General Algorithm
   - Keep updating until nothing changes
   - JastAdd: *Circular Attributes*
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**"



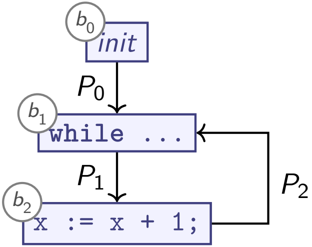
Characterising Data Flow Analyses

Characteristics:

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

Require properties of L, $\boldsymbol{trans_b}$, $\boldsymbol{join_b}$ to ensure termination

Limiting Iteration



- Does the following ever stop changing:

$$
\begin{aligned}
\mathbf{in}_{b_1} &= join_{b_1}(P_0, P_2) \\
\mathbf{in}_{b_2} &= trans_{b_1}(\mathbf{in}_{b_1}) \\
P_2 &= trans_{b_2}(\mathbf{in}_{b_2})
\end{aligned}
$$

- 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

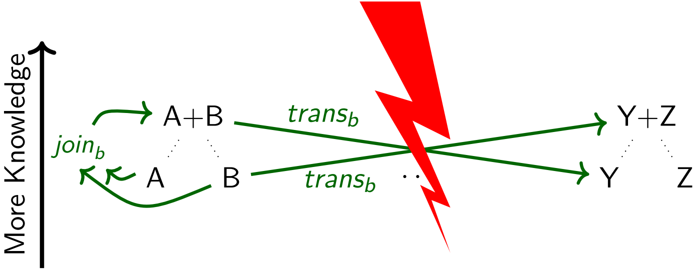
$$A \sqsubseteq B$$

$$\begin{tikzcd} B \arrow[d, no head] \\ A \end{tikzcd}$$

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

Intuition: Knowing Less, Knowing More

Structure of L:

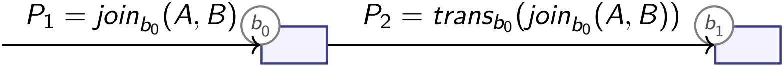


- *join*$_b$ must not lose knowledge
  - $A \sqsubseteq join_b(A, B)$
  - $B \sqsubseteq join_b(A, B)$
- *trans*$_b$ must be *monotonic* over amount of knowledge:

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

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

Aggregating Knowledge

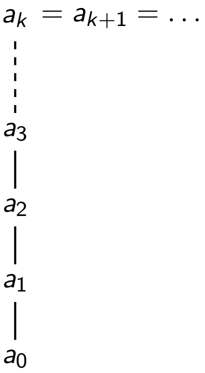


- Interplay between $trans_b$ and $join_b$ helps preserve knowledge
- $A \sqsubseteq join_b(A, B)$:
  As we add knowledge, P_1 either:
  - Stays the same
  - Increases knowledge
- Monotonicity of $trans_b$: If P_1 goes up, then P_2 either:
  - Stays the same
  - Increases knowledge

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

Now we must only prevent *infinite* growth...

Ascending Chains



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

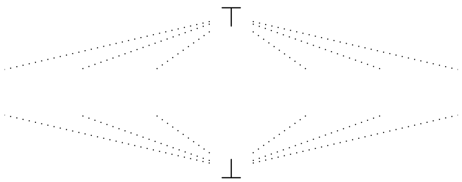
$$a_i \sqsubseteq a_{i+1} \text{ (for all } i \geq 0)$$

- *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:

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

$\boxed{\textbf{ACC is formalisation of growth limit}}$

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 $A \sqsubseteq join_b(A, B)$ and $B \sqsubseteq join_b(A, B)$:
  - $join_b(\top, A) = \top = join_b(A, \top)$
  - $\bot \sqsubseteq A \sqsubseteq join_b(\bot, A)$
    - In practice, it is 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’

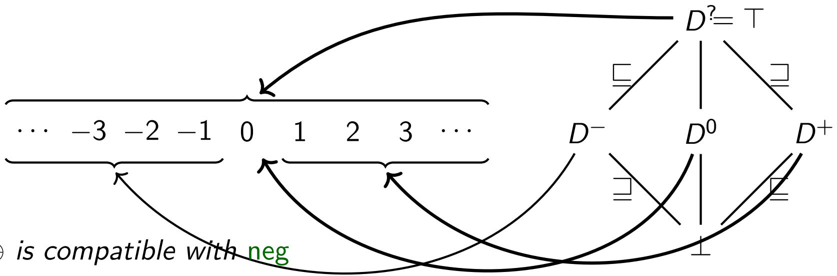
Summary

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

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

- $join_b : L \times L \to L$ must produce an *upper bound* for its parameters:
  - $A \sqsubseteq join_b(A, B)$
  - $B \sqsubseteq join_b(A, B)$
- 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 $\bot$

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

- $\bot$ representing "not known" – not needed for our example analysis ($[\![-]\!]^{\mathcal{D}}$), 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
  - Concretely, $\oplus$ is "pointwise monotonic", meaning: if $d \in L_D$ is constant, then
    - $x \mapsto d \oplus x$ is monotonic
    - $x \mapsto x \oplus d$ is monotonic

Outlook

- We will continue on Dataflow Analysis

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