



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

Questions?

Getting More out of Type Inference (1/4)

- ▶ Recall our typing rules from last lecture:

$$\frac{}{\text{true} : \text{BOOL}} \quad \frac{}{\text{false} : \text{BOOL}}$$
$$\frac{e_1 : \text{BOOL} \quad e_2 : \tau \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau}$$

- ▶ 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 `BOOL` by *more precise types* `TRUE` and `FALSE`:

$$\frac{}{\text{true} : \text{TRUE}} \quad \frac{}{\text{false} : \text{FALSE}}$$
$$\frac{e_1 : \text{TRUE} \quad e_2 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ (if-true)}$$
$$\frac{e_1 : \text{FALSE} \quad e_3 : \tau}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 : \tau} \text{ (if-false)}$$

We can now infer:

$$\frac{\frac{}{\text{false} : \text{FALSE}} \quad \frac{}{\text{true} : \text{TRUE}}}{\text{if false then false else true} : \text{TRUE}} \text{ (if-false)}$$

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

\Rightarrow 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

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

- ▶ 2^n possible types!

Exponential analysis cost \implies too slow to analyse real programs!

Towards Abstract Interpretation

- ▶ Consider the following language:

$$\begin{array}{l} E ::= \text{zero} \\ | \text{one} \\ | \langle E \rangle + \langle E \rangle \\ | \text{neg } \langle E \rangle \end{array}$$

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

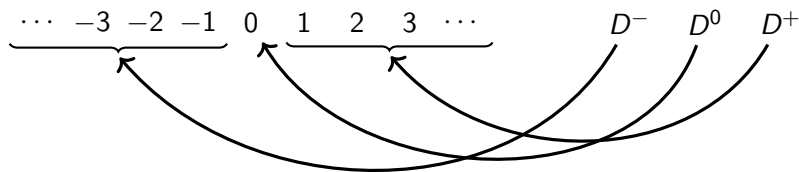
Abstract Domains

- ▶ *Abstract Interpretation*:
 - ▶ Map all values to a simpler *abstract domain*
- ▶ Example: set *abstract domain* \mathcal{D} :

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

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

Correspondence: Concrete and Abstract



Abstract Domains

- ▶ *Abstract Interpretation*:
 - ▶ Map all values to a simpler *abstract domain*
- ▶ Example: set *abstract domain* \mathcal{D} :

$$\mathcal{D} = \left\{ \begin{array}{l} D^0, \text{ Program computes } 0 \\ D^+, \text{ Program computes a positive value} \\ D^-, \text{ Program computes a negative value} \end{array} \right\}$$

- ▶ 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

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

E	::=	zero
		one
		$\langle E \rangle + \langle E \rangle$
		neg $\langle E \rangle$

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

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

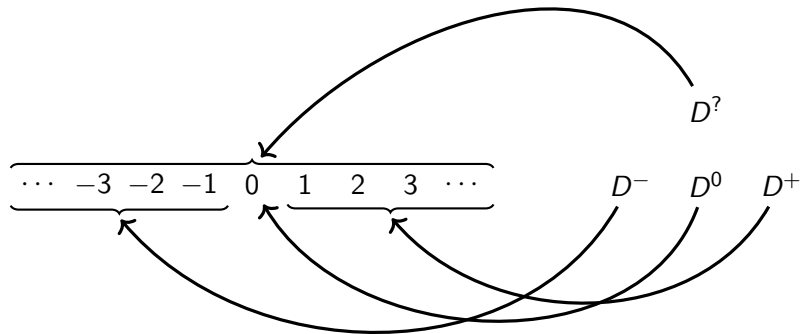
$$\llbracket \text{zero} \rrbracket^{\mathcal{D}} = D^0$$

$$\llbracket \text{one} \rrbracket^{\mathcal{D}} = D^+$$

$$\frac{\llbracket x \rrbracket^{\mathcal{D}} = a}{\llbracket \text{neg } x \rrbracket^{\mathcal{D}} = \ominus a}$$

$$\frac{\llbracket x \rrbracket^{\mathcal{D}} = a_1 \quad \llbracket y \rrbracket^{\mathcal{D}} = a_2}{\llbracket x + y \rrbracket^{\mathcal{D}} = a_1 \oplus a_2}$$

Correspondence: Concrete and Abstract



Abstract Domains

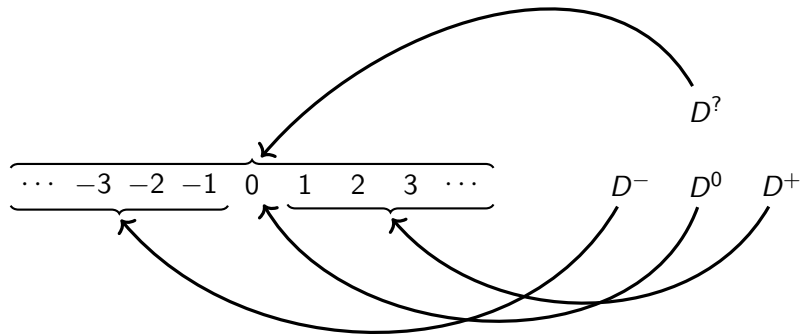
- ▶ *Abstract Interpretation*:
 - ▶ Map all values to a simpler *abstract domain*
 - ▶ For each operation, build corresponding *abstract operation*
- ▶ Example: set *abstract domain* \mathcal{D} :

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

- ▶ Notation: $\boxed{\llbracket \varphi \rrbracket^{\mathcal{D}} = a}$, 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:

$$\llbracket - \rrbracket^{\mathcal{D}} : Program \rightarrow \mathcal{D}$$

- ▶ 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
- ▶ $\llbracket - \rrbracket^{\mathcal{D}}$ must be a *function*
 - ▶ Example: type inference with principal types
 - ▶ Non-example: type inference without principal types

Lecture Overview

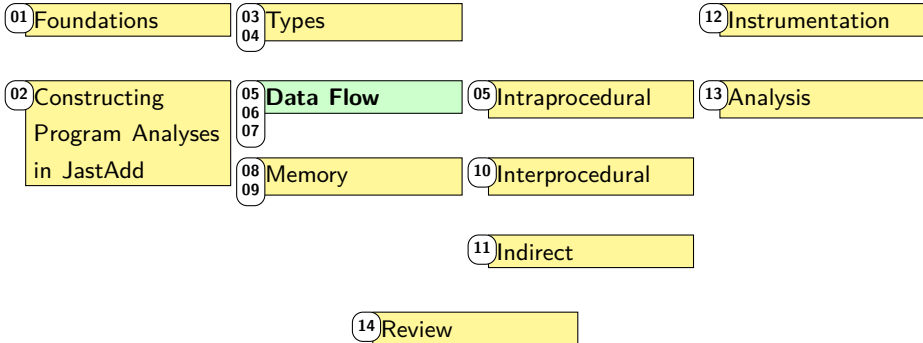
Foundations

Static Analysis

Dynamic
Analysis

Properties

Control Flow



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 *failures*:
 - ▶ 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; // D0
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!

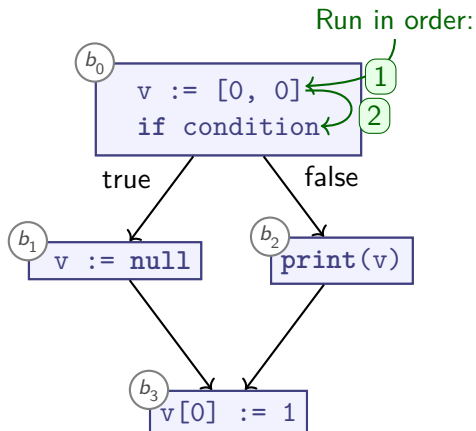
Summary

- ▶ 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

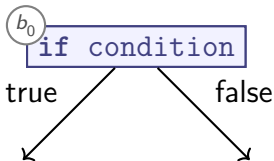
```
var v := [0, 0];  
if condition {  
  v := null;  
} else {  
  print(v);  
}  
v[0] := 1;
```



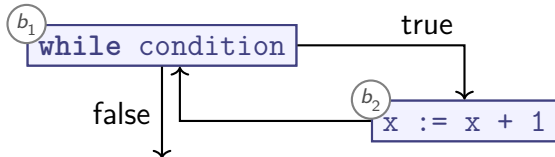
Control Flow Graphs encode statement execution order

Control-Flow-Graphs

- ▶ Encode statement order by *nodes*  and edges \rightarrow
- ▶ *Multiple* outgoing edges (branches): Add label:



- ▶ Uniform representation for control statements:



Summary

Control-Flow Graph (CFG):

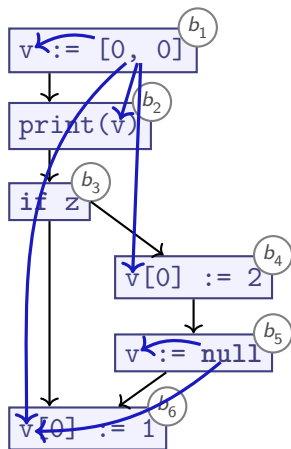
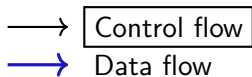
- ▶ Motivation:
 - ▶ Universal representation of control flow
 - ▶ Computed once before running analyses
 - ▶ Flow-sensitive analyses can utilise CFG
- ▶ Idea:
 - ▶ Represent control flow as **Blocks** and **Control-Flow Edges**
 - ▶ Edges represent control flow, **labelled** to identify conditionals

Control Flow

Understanding **data flow** requires understanding control flow:

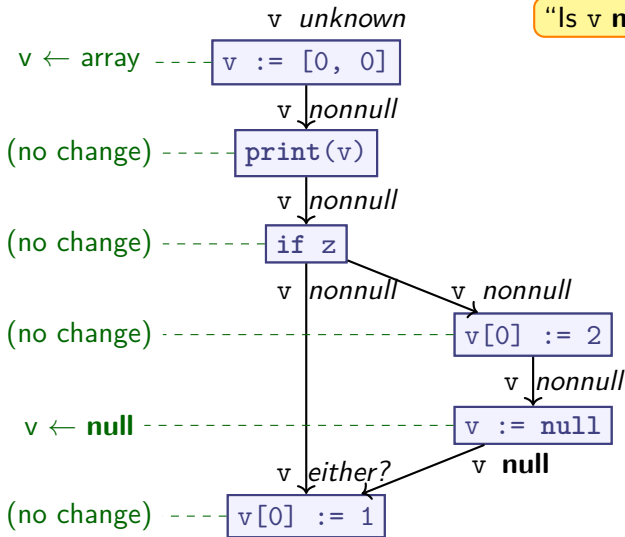
Teal

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



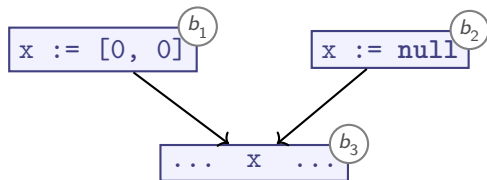
Intuition behind Data Flow Analysis

“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?”
 - ⇒ better assume **null**
 - ▶ “Is this guaranteed to be null?”
 - ⇒ better assume **nonnull**
- ▶ We might not need the 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*

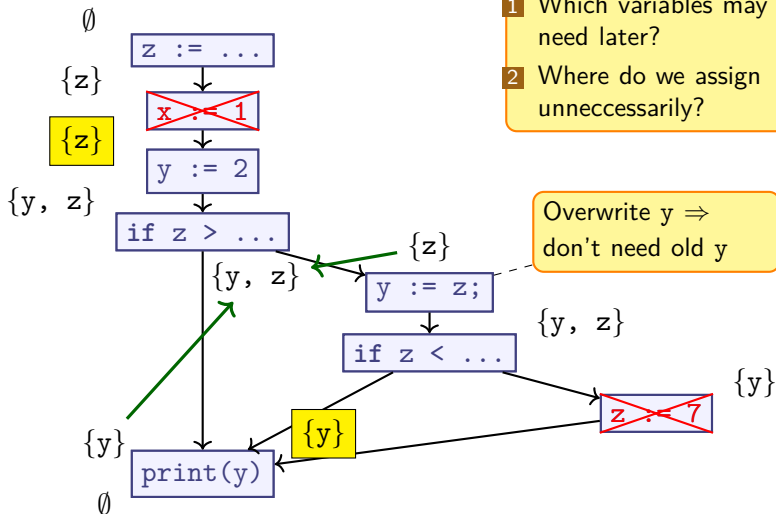
Another Analysis

Teal

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

- ▶ Which assignments are unnecessary?
- ⇒ Possible oversights / bugs
(*Live Variables Analysis*)

Unnecessary Assignments: Intuition

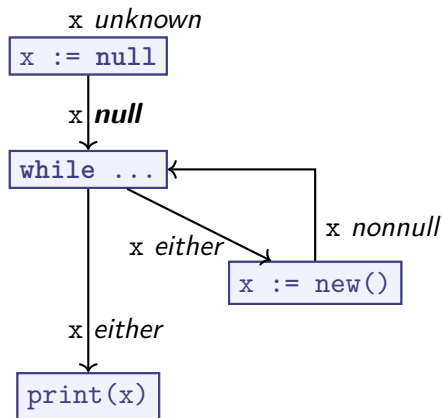


Analysis effective: found useless assignments to z and x

Observations

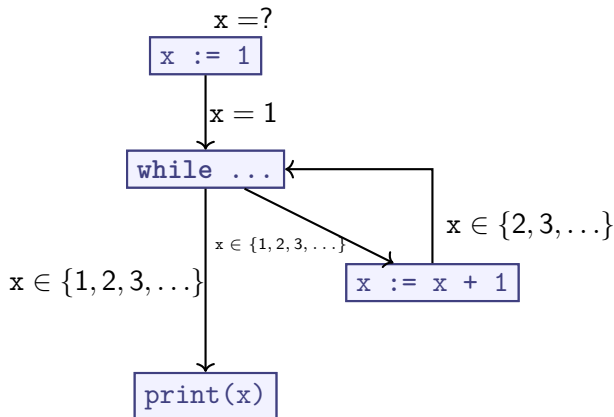
- 1 Data Flow analysis can be run *forward* or *backward*
- 2 May have to *join* results from multiple sources

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:
 - ▶ Either '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

4 Nullness analysis

- ▶ Forward analysis
- ▶ Both *May* and *Must* approaches are useful

Engineering Data Flow Algorithms

1 General Algorithm

- ▶ Keep updating until nothing changes
- ▶ JastAdd: *Circular Attributes*

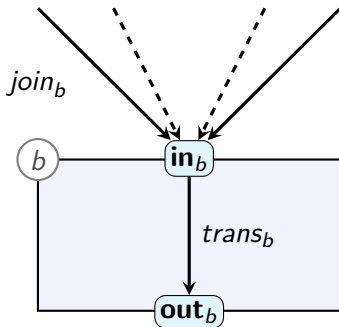
2 Termination

- ▶ Assumption: Operate on Control Flow Graph
- ▶ Theory: Ensure termination

3 (Correctness)

Data Flow Analysis on CFGs

- ▶ \mathbf{in}_b : knowledge at entrance of node b
- ▶ \mathbf{out}_b : knowledge at exit of node b
- ▶ \mathbf{join}_b : combines all \mathbf{out}_{b_i} for all nodes b_i that flow into b
“**Join Function**”
- ▶ \mathbf{trans}_b : updates \mathbf{out}_b from \mathbf{in}_b
“**Transfer Function**”



- ▶ For *backward analysis*:
 - ▶ flip edge directions
 - ▶ \mathbf{trans}_b : updates \mathbf{in}_b from \mathbf{out}_b

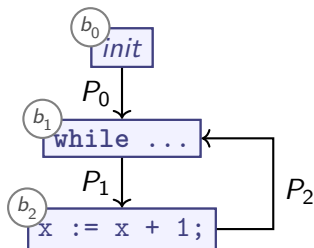
Characterising Data Flow Analyses

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:

$$\mathbf{in}_{b_1} = \mathit{join}_{b_1}(P_0, P_2)$$

$$\mathbf{in}_{b_2} = \mathit{trans}_{b_1}(\mathbf{in}_{b_1})$$

$$P_2 = \mathit{trans}_{b_2}(\mathbf{in}_{b_2})$$

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

Eventually, either nothing changes or we hit growth limit

Ordering Knowledge

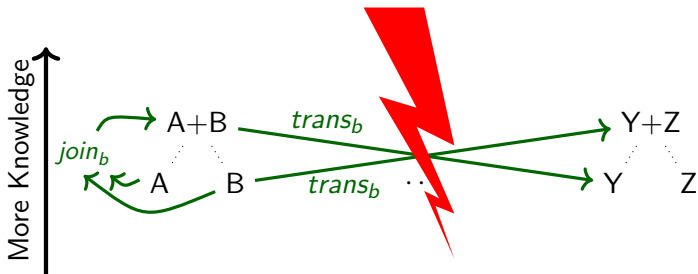
$$A \sqsubseteq B$$



- ▶ 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 :

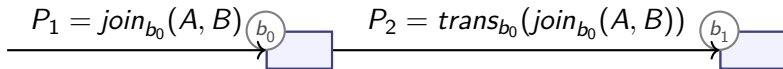


- ▶ $join_b$ must not lose knowledge
 - ▶ $A \sqsubseteq join_b(A, B)$
 - ▶ $B \sqsubseteq join_b(A, B)$
- ▶ $trans_b$ must be *monotonic* over amount of knowledge:

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

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

Aggregating Knowledge



- ▶ Interplay between trans_b and join_b helps preserve knowledge
 - ▶ $A \sqsubseteq \text{join}_b(A, B)$:
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
- ⇒ 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, \dots is an *ascending chain* iff:

$$a_k = a_{k+1} = \dots$$

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

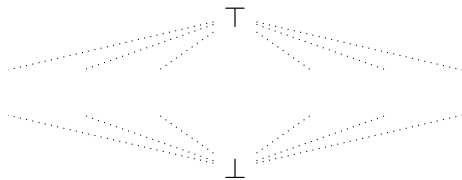
- ▶ *Ascending Chain Condition*:
 - ▶ For every ascending chain a_0, a_1, a_2, \dots in abstract domain L :
 - ▶ there exists $k \geq 0$ such that:

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

a_3
|
 a_2
|
 a_1
|
 a_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:** $\perp: \perp \sqsubseteq A$ for all A
- ▶ Since $A \sqsubseteq \text{join}_b(A, B)$ and $B \sqsubseteq \text{join}_b(A, B)$:
 - ▶ $\text{join}_b(\top, A) = \top = \text{join}_b(A, \top)$
 - ▶ $\perp \sqsubseteq A \sqsubseteq \text{join}_b(\perp, A)$
 - ▶ In practice, it is safe and simple to set:
 $\text{join}_b(\perp, A) = A = \text{join}_b(A, \perp)$
- ▶ *Intuition:*
 - ▶ \top : means 'contradictory / too much information'
 - ▶ \perp : 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 \perp

- ▶ $trans_b : L \rightarrow L$

- ▶ Must be *monotonic*:

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

- ▶ $join_b : L \times L \rightarrow L$ must produce an *upper bound* for its parameters:

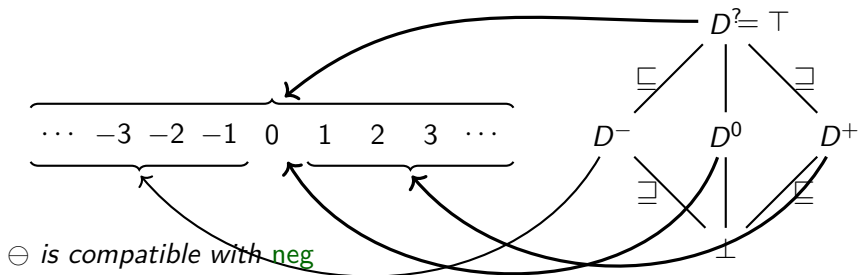
- ▶ $A \sqsubseteq join_b(A, B)$

- ▶ $B \sqsubseteq join_b(A, B)$

- ▶ Satisfy **Ascending Chain Condition** to ensure termination

- ▶ Easiest solution: make L finite

Abstract Domains Revisited



$$\ominus \perp = \perp$$

$$\ominus D^0 = D^0$$

$$\ominus D^+ = D^-$$

$$\ominus D^- = D^+$$

$$\ominus D^? = D^?$$

\ominus is monotonic (and \oplus extended with \perp 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 ($\llbracket - \rrbracket^D$), but would be needed if we had variables / control flow in that language
- ▶ L_D is finite, so the ACC 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

<https://cs.lth.se/EDAP15>