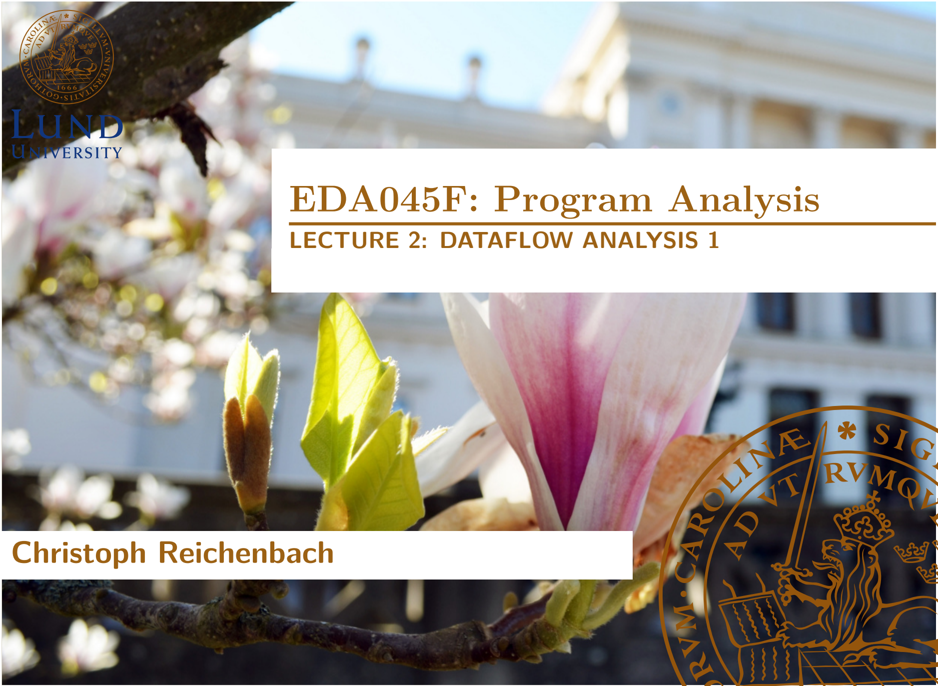


EDA045F: Program Analysis

LECTURE 2: DATAFLOW ANALYSIS 1

Christoph Reichenbach

In the last lecture...

- Uses of Program Analysis
- Static vs. Dynamic Program Analysis
- Soundness, Precision, Termination
- Abstraction and Simplification for Analysis
- Program Execution Pipeline
- Intermediate Representation

Announcements

- Moodle available
- Homework #1 on home page after class
  - Groups formation in break!
- Needed: Student representative

Intermediate Representations

```
...
0:    iload_0
1:    ifle 9
4:    iconst_1
5:    istore_1
6:    goto 11
9:    iconst_0
10:   istore_1
11:   iload_1
12:   ireturn
...
```

- Simplify analysis
  - Fewer cases to consider
  - Reduce risk of bugs in analyses
- (Simplify code generation)
- (Simplify code transformation)

⇒ We will need code transformation for dynamic analysis

A Buggy Example

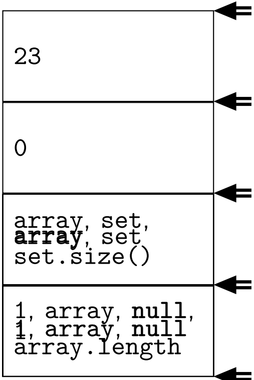
Java

```java
int[] array = new int[]{23};
Set<Integer> set = null;
print(array.length, set.size());
// create nonempty set
Set<Integer> set = new HashSet<Integer>(...);
```

Analysis: Connect dereference to `null` pointer

Example: Our program in Java bytecode

```
⇒   0    iconst_1
⇒   1    newarray          int
⇒   3    dup
⇒   4    iconst_0
⇒   5    bipush            23
⇒   7    iastore
⇒   8    astore_1
⇒   9    aconst_null
⇒   10   astore_2
⇒   11   aload_1
⇒   12   arraylength
⇒   13   aload_2
⇒   14   invokeinterface   java.util.Set.size()
⇒   19   invokestatic      print(int, int)
```



Stack

Local variables: | 1: array | 2: set/null |

The stack is not convenient for program analysis

Summary

- **Stack**: Cumbersome for connecting
  - Meaning of stack slot depends on position in the program
- **Local Variables**: Helpful for connecting
  - Meaning is associated with variable in original program
- **Dealing with intermediate results?**
  - No clear solution yet for dealing with e.g.:
    `((a > 0) ? null : array).length`

Simplifying Analysis with Simpler IRs

- Goal:
  - Make analyses easier to build
  - Make analyses less error-prone
- Start with ASTs
- Refine:
  - **Simpler statements**
    'Dummy names' for intermediate results
  - **Representing control flow**
  - **Breaking up multiple uses of the same name**

A Tiny Language

$$
\begin{array}{lcl}
\textit{name} & ::= & \textit{id} \\
 & | & \langle \textit{name} \rangle\ .\ \textit{id} \\
 & & \\
\textit{expr} & ::= & \textit{num} \\
 & | & \langle \textit{expr} \rangle + \langle \textit{expr} \rangle \\
 & | & \text{null} \\
 & | & \text{print}\ \langle \textit{expr} \rangle \\
 & | & \text{new()} \\
 & | & \langle \textit{name} \rangle
\end{array}
\qquad
\begin{array}{lcl}
\textit{stmt} & ::= & \langle \textit{name} \rangle = \langle \textit{expr} \rangle \\
 & | & \{\ \langle \textit{stmt} \rangle \star\ \} \\
 & | & \text{if}\ \langle \textit{expr} \rangle\ \langle \textit{stmt} \rangle\ \text{else}\ \langle \textit{stmt} \rangle \\
 & | & \text{while}\ \langle \textit{expr} \rangle\ \langle \textit{stmt} \rangle \\
 & | & \text{skip} \\
 & | & \text{return}\ \langle \textit{expr} \rangle
\end{array}
$$

Evaluation Order

ATL

```
v = print((print 1) + (print 2))
```

ATL with explicit order

```
tmp1 = print 1
tmp2 = print 2
tmp3 = tmp1 + tmp2
v    = print(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!

A Tiny Language: Simplified

$$
\begin{array}{lcl}
\mathit{name} & ::= & \mathit{id} \\
 & | & \mathit{id}\ .\ \mathit{id} \\
 & & \\
\mathit{val} & ::= & \langle \mathit{name} \rangle \\
 & | & \mathit{num} \\
 & & \\
\mathit{expr} & ::= & \langle \mathit{val} \rangle \\
 & | & \langle \mathit{val} \rangle + \langle \mathit{val} \rangle \\
 & | & \mathsf{null} \\
 & | & \mathsf{print}\ \langle \mathit{val} \rangle \\
 & | & \mathsf{new()}
\end{array}
$$

$$
\begin{array}{lcl}
\mathit{stmt} & ::= & \langle \mathit{name} \rangle = \langle \mathit{expr} \rangle \\
 & | & \{\ \langle \mathit{stmt} \rangle \star\ \} \\
 & | & \mathsf{if}\ \langle \mathit{val} \rangle\ \langle \mathit{stmt} \rangle\ \mathsf{else}\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{while}\ \langle \mathit{val} \rangle\ \langle \mathit{stmt} \rangle \\
 & | & \mathsf{skip} \\
 & | & \mathsf{return}\ \langle \mathit{val} \rangle
\end{array}
$$

Eliminating Nesting

- 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 just pointers in the implementation

- We still have nested statements
- Not all IRs de-nest as aggressively as this

Multiple Paths

ATL

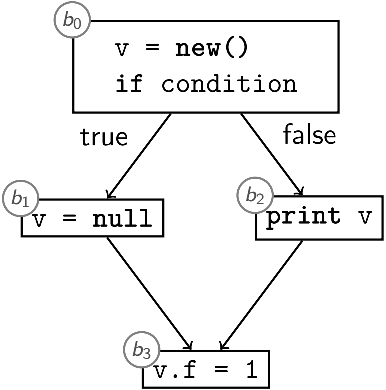
```
v = new()
if condition {
  v = null
} else {
  print v
}
v.f = 1
```

ATL

```
v = new()
while condition {
  v = null
}
v.f = 1
```

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

Control-Flow Graphs



Construct graph to show flow of control through program

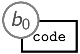
Making Flow Explicit

name	::=	*id*
	|	*id* . *id*
val	::=	⟨*name*⟩
	|	*num*
expr	::=	⟨*val*⟩
	|	⟨*val*⟩+⟨*val*⟩
	|	null
	|	print ⟨*val*⟩
	|	new()

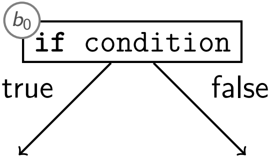
stmt	::=	⟨*name*⟩ = ⟨*expr*⟩
	|	skip
	|	return ⟨*val*⟩
$\boxed{\rightarrow}$	::=	⟨*stmt*⟩ $\star$ $\boxed{\rightarrow}$
	|	end
	|	⟨*stmt*⟩ $\star$ if ⟨*val*⟩ $\boxed{\rightarrow}$ else $\boxed{\rightarrow}$

For intuition only: $\boxed{\rightarrow}$ *is not a 'real' nonterminal*

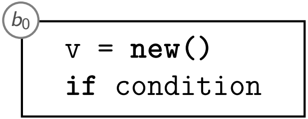
Control-Flow-Graphs

- Replace statement nesting by *nodes*  and edges →

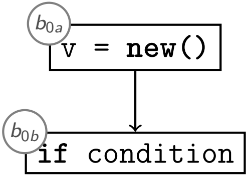
- *Multiple* outgoing edges: Label condition:



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

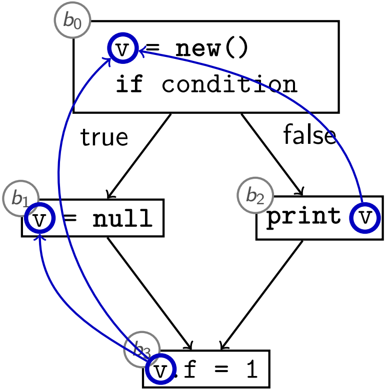


Basic Block



- Uniform representation for different control statements

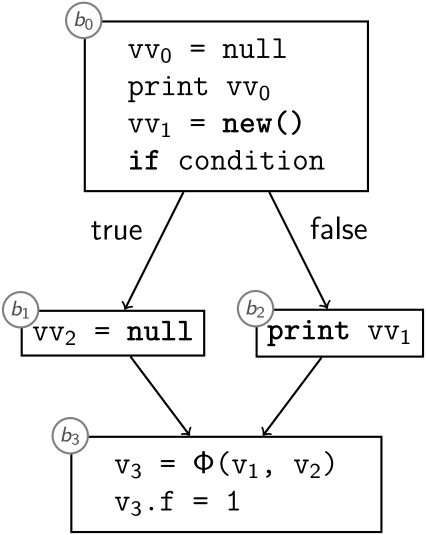
Use-Def Chains



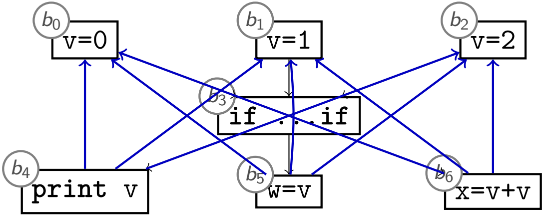
Use-Def chain: Map one *use* to all *definitions*
Def-Use chain: Map one *definition* to all *uses* (not shown here)

Alternative: Static Single Assignments

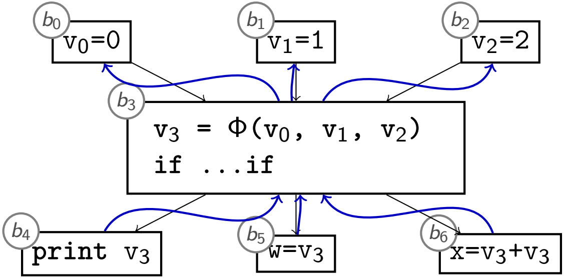
Idea: unique names for every assignment



Static Single Assignments Simplifies Def-Use/Use-Def Chains



without SSA



with SSA

Static Single Assignment Form

- From a static perspective:
  - Each variable is set exactly once in the program
  - Each name stands for exactly one computation
- Can connect definitions and uses without complex graphs
- Φ (Phi) functions merge points
  - *Minimal SSA* eliminates unnecessary Φ functions
- Similar representations:
  - Continuation-Passing Style IR (CPS)
  - A-Normal Form (ANF)
- Simpler Def-Use / Use-Def chains

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
- IRs try to expose and link:
  - **Definitions** of (= writes to) a variable
  - **Uses** of (= reads from) a variable
- **Use-Def Chain**: Links uses to all reaching definitions
- **Def-Use Chain**: Links definitions to all reachable uses
- **Static Single Assignment** (SSA) form:
  - Each variable has exactly one definition
  - Use Φ (Phi) expressions to merge variables across control-flow edges

Basic Formal Notation

- Tuples:
  - Notation:

$$\begin{array}{rl} \langle a \rangle & \\ \langle a, b \rangle & \text{(pair)} \\ \langle a, c, d \rangle & \text{(triple)} \end{array}$$

  - Fixed-length (unlike list)
  - Group items, analogous to (read-only) record/object
- Sets:

$$\begin{array}{rl} \emptyset = \{\} & \text{(the empty set)} \\ \{1\} & \text{(\textit{singleton} set containing precisely the number 1)} \\ \{2, 3\} & \text{(Set with two elements)} \\ \mathbb{Z} & \text{(The (infinite) set of integers)} \\ \mathbb{R} & \text{(The (infinite) set of real numbers)} \end{array}$$

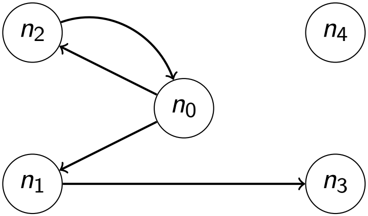
Basic operations on sets

$x \in S$	Is x contaìnd in S?	True: $1 \in \{1\}$ and $1 \in \mathbb{Z}$ False: $2 \in \{1\}$ or $\pi \in \mathbb{R}$
$x \notin S$	Is x NOT containd in S?	
$A \cup B$	Set union	$\{1\} \cup \{2\} = \{1, 2\}$ $\{1, 3\} \cup \{2, 3\} = \{1, 2, 3\}$
$A \cap B$	Set intersection	$\{1\} \cap \{2\} = \emptyset$ $\{1, 3\} \cap \{2, 3\} = \{3\}$
$A \subseteq B$	Subset relationship	True: $\emptyset \subseteq \{1\}$ and $\mathbb{Z} \subseteq \mathbb{R}$ False: $\{2\} \subseteq \{1\}$
$A \times B$	Product set	$\{1, 2\} \times \{3, 4\}$ $= \{\langle 1, 3\rangle, \langle 1, 4\rangle, \langle 2, 3\rangle, \langle 2, 4\rangle\}$

Graphs

A (directed) graph $\mathcal{G}$ is a tuple $\mathcal{G} = \langle \mathcal{N}, \mathcal{E} \rangle$, where:

- $\mathcal{N}$ is the set of *nodes* of $\mathcal{G}$
- $\mathcal{E} \subseteq \mathcal{N} \times \mathcal{N}$ is the set of *edges* of $\mathcal{G}$
- Often: Add function $f : \mathcal{E} \to X$ to *label* edges



$$\mathcal{N} = \{n_0, n_1, n_2, n_3, n_4\}$$

$$\mathcal{E} = \{\langle n_0, n_1 \rangle, \langle n_0, n_2 \rangle, \langle n_1, n_3 \rangle, \langle n_2, n_0 \rangle\}$$

Summary

- **Tuples** group a fixed number of items
- **Sets** represent a (possibly infinite) number of unique elements
  - Widely used in program analysis
- **(Directed) Graphs** represent *nodes* and *edges* between them
  - Optional *labels* on edges possible
  - Used e.g. for control-flow graphs

Dataflow Analysis: Example

ATL

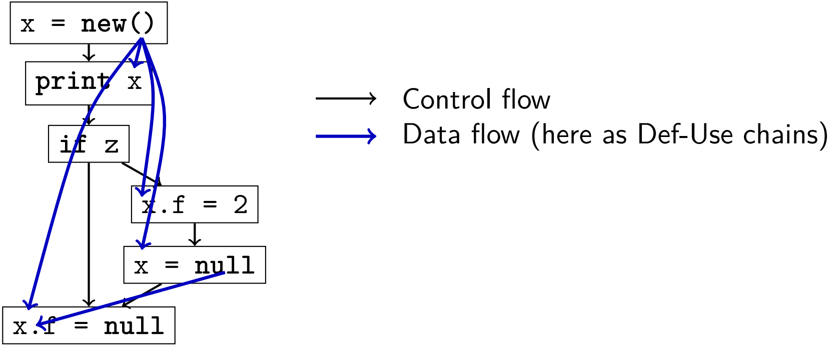
```
x = new()
print x   // A
if z {
  x.f = 2 // B
  x = null
} else skip
x.f = 1   // C
```

- Analyse: Will there be an error at `B` or `C`?
- Must distinguish between `x` at `A` vs. `x` at `B` and `C`
- Need to model flow of information Suitable IRs:
  - Control-Flow Graph (CFG)
  - Static Single-Assignment Form (SSA)

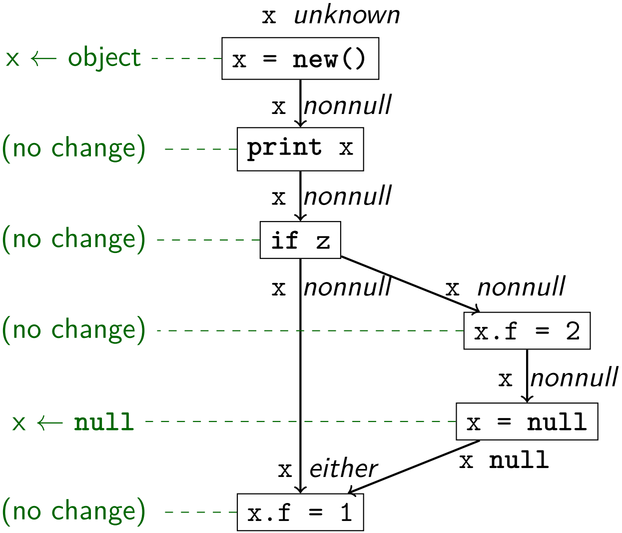
Need analysis that can represent *data flow* through program

Control Flow

Understanding data flow requires understanding control flow:



Basic Ideas of Data Flow Analysis



Another Analysis

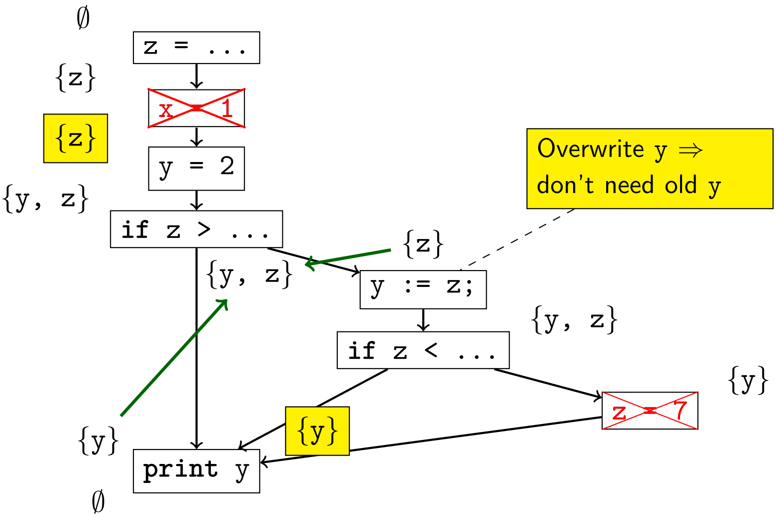
ATL

```
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*)

Control Flow

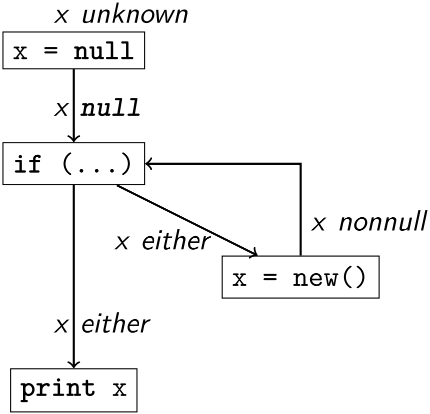


Analysis effective: found useless assignments to z and x

Observations

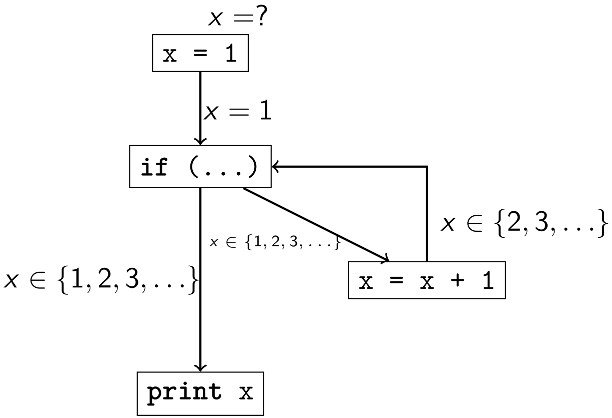
1. Data Flow analysis can be run *forward* or *backward*
2. May have to *join* results from multiple sources

What about Loops? (1/2)



- Analysis: *Null Pointer Dereference*
- 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)

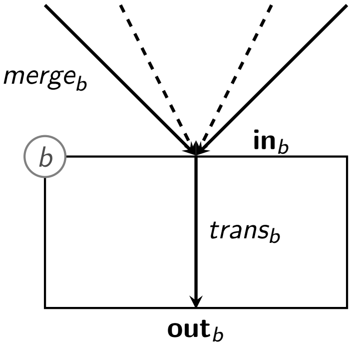
- Some important program analyses are *flow sensitive*: must consider how execution order affects variables
- Data flow depends on *control flow*
- Data flow analysis examines how variables change across control-flow edges
- May have to join multiple results
- Can run *forward* or *backward* wrt program control flow
- Handling loops is nontrivial

Engineering Data Flow Algorithms

1. Termination
   - Assumption: Operate on Control Flow Graph
   - Theory: Ensure termination
2. (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
- $merge_b$: merges all $\mathbf{out}_{b_i}$ for all basic blocks b_i that flow into b
- $trans_b$: updates $\mathbf{out}_b$ from $\mathbf{in}_b$



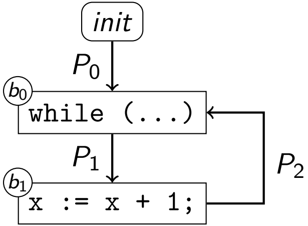
Characterising Data Flow Analyses

Characteristics:

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

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

Limiting Iteration



- Does the following ever stop changing:

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

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

Eventually, either nothing changes or we hit growth limit

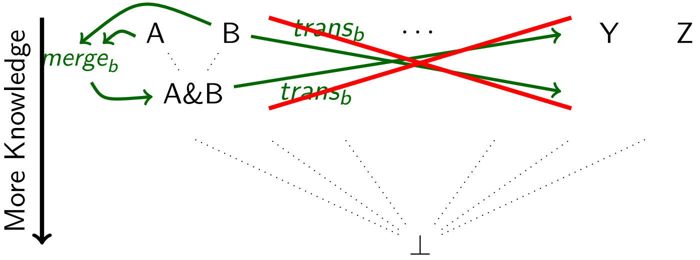
Ordering Knowledge

$$A \sqsubseteq B$$

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

Intuition: Knowing Less, Knowing More

Structure of L:

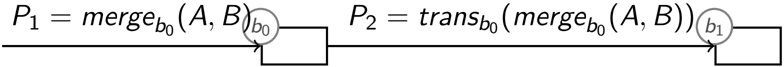


- $merge_b$ must not lose knowledge
  - $merge_b(A, B) \sqsubseteq A$
  - $merge_b(A, B) \sqsubseteq B$
- $trans_b$ must be *monotonic* over amount of knowledge:

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

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

Aggregating Knowledge



- Interplay between $trans_b$ and $merge_b$ helps preserve knowledge
- $merge_b(A, B) \sqsubseteq A$:
  As we add knowledge, P_1 either
  - Stays equal
  - 'Descends'
- Monotonicity of $trans_b$: If P_1 descends, then P_2 either
  - Stays equal
  - 'Descends'

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

Now we must only set a growth limit. . .

Descending Chains

a_0

a_1

a_2

a_3

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

- A (possibly infinite) sequence $a_0, a_1, a_2, \dots$ is a *descending chain* iff:

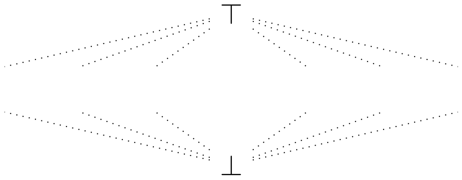
$$a_{i+1} \sqsubseteq a_i \text{ (for all } i \geq 0)$$

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

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

DCC 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 $merge_b(A, B) \sqsubseteq A$ and $merge_b(A, B) \sqsubseteq B$:
  - $merge_b(\bot, A) = \bot = merge_b(A, \bot)$
  - $merge_b(\top, A) \sqsubseteq A \sqsupseteq merge_b(A, \top)$
    - In practice, it's safe and simple to set: $merge_b(\top, A) = A = merge_b(A, \top)$
- Intuition:
  - $\top$: means 'no information known yet'
  - $\bot$: means 'contradictory / too much information'

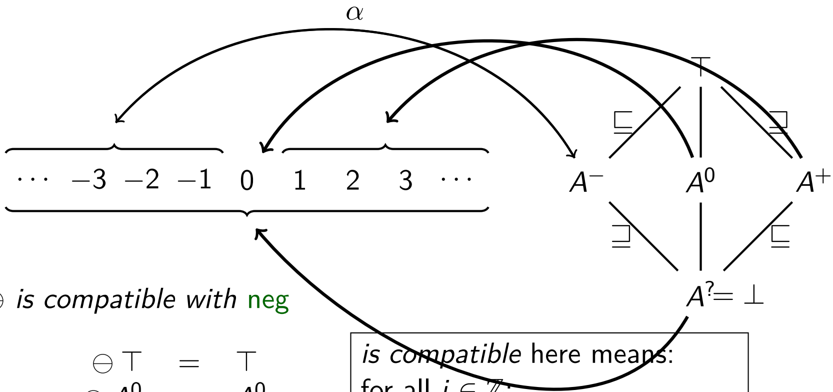
Summary

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

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

- $merge_b : L \times L \to L$ must produce a *lower bound* for its parameters:
  - $merge_b(A, B) \sqsubseteq A$
  - $merge_b(A, B) \sqsubseteq B$
- Satisfy **Descending Chain Condition** to ensure termination
  - Easiest solution: make L finite

Abstract Domains Revisited



$\ominus$ *is compatible with* neg

$$
\begin{aligned}
\ominus\, \top &= \top \\
\ominus\, A^0 &= A^0 \\
\ominus\, A^+ &= A^- \\
\ominus\, A^- &= A^+ \\
\ominus\, A^? &= A^?
\end{aligned}
$$

is compatible here means:
for all $i \in \mathbb{Z}$:

$$\ominus(\alpha(i)) \sqsubseteq \alpha(\mathsf{neg}(i))$$

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

Summary

- We could extend $\{A^+, A^-, A^0, A^?\}$ to an Abstract Domain by adding $\top$
$$L_A = \{A^+, A^-, A^0, A^?, \top\}$$
- L_A is finite, so the DCC holds trivially
- All our abstract operations are monotonic
- Making the abstraction function $\alpha : \mathbb{Z} \to L_A$ explicit allows us to check that our abstract operations are *compatible*:
$$\ominus(\alpha(i)) \sqsubseteq \alpha(\mathsf{neg}(i))$$
(cf. 'induced operation' in Abstract Interpretation)

Soot IRs

- Exercise #1 uses Soot, which offers four IRs:
  - **Jimple**: Soot's main CFG-based IR
  - **Shimple**: Jimple converted to SSA form
  - **Grimp**: Jimple with nested expressions
    Intended for decompiling/pretty-printing
  - **Baf**: Enhanced Java bytecode
    Intended for bytecode generation

Example Program with Bug

Java

```
int[] array = new int[]{23};
Set<Integer> set = null;
print(array.length, set.size());
// create nonempty set
Set<Integer> set = new HashSet<Integer>(...);
```

Soot's Jimple IR

```
l0      := @this
$r0     = newarray (int)[1]
$r0[0]  = 23
l2      = null
$i0     = lengthof $r0
$i1     = interfaceinvoke l2.<java.util.Set:  int size()>()
          staticinvoke <T2:  void print(int,int)>($i0, $i1)
```

Order of Side Effects

Java

```
int[] one = new int[1];
int[] two = new int[2];
int counter = 0;
one[counter++] = two[counter++]++;
return one;
```

Jimple

```
one      = newarray (int)[1]
two      = newarray (int)[2]
counter  = 0 + 1
$i0      = counter
$i1      = two[$i0]
$i2      = $i1 + 1
two[$i0] = $i2
one[0]   = $i1
           return one
```

Jimple IR

$$
\begin{array}{lcl}
Block & ::= & \langle Stmt\rangle \star \langle Trap\rangle\star \\
Stmt & ::= & \text{nop} \\
 & | & \langle v_r\rangle := \langle v\rangle \\
 & | & \langle v_r\rangle = \langle v_r\rangle \\
 & | & \langle Invoke\rangle \\
 & | & \text{goto } \langle i\rangle \\
 & | & \text{if } \langle v\rangle \text{ goto } \langle i\rangle \\
 & | & \text{return } \langle v\rangle \\
 & | & \text{return-void} \\
 & | & \text{entermonitor } \langle v\rangle \\
 & | & \text{exitmonitor } \langle v\rangle \\
 & | & \text{tableswitch } \ldots \\
 & | & \text{lookupswitch } \ldots \\
 & | & \text{breakpoint} \\
 & | & \text{ret} \\
 & | & \text{throw } \langle v\rangle \\
Trap & | & \text{catch } ty \text{ from i to } i_1 \text{ with } i_h
\end{array}
$$

$$
\begin{array}{lcl}
v & ::= & var \mid \langle v_c\rangle \mid \langle v_r\rangle \mid \langle v_e\rangle \\
v_c & ::= & int \mid long \mid float \mid double \\
 & | & string \mid \text{null} \\
 & | & \langle method\rangle \mid ty \\
v_e & ::= & \langle Invoke\rangle \\
 & | & \text{new } ty \\
 & | & \text{newarray } ty[int] \\
 & | & \text{nemultiwarray } ty([int])\star \\
 & | & v+v \\
 & | & v\text{-}v \\
 & & \ldots \\
v_r & ::= & \langle v_r\rangle[\langle v_r\rangle] \\
 & | & \text{@this} \\
 & | & \text{@parameter } i \\
 & | & \text{@caughtexception} \\
 & | & \langle v_r\rangle.\text{id} \\
 & | & \langle ty\rangle.\text{id} \\
Invoke & ::= & \ldots
\end{array}
$$

Homework

1. Find all `main` methods
2. Find all calls to deprecated methods
3. Simplified Array Out-Of-Bounds checking: find uses of negative array indices
4. Live Variables Analysis: Find useless assignments
5. Make your analysis reusable

To be continued...

Next week:

- Lattice theory
- Understanding our precision
- Procedure calls