Requirements

- Identifies the what of the system, not the actual implementation
- Functional requirements:
  - What is the software supposed to do
    - ↳ I/O relationships, sequence of execution, validity check
  - Interaction with people / APIs / Hardware
    - ↳ Screen size, destination of outputs, sources
- Non-functional requirements:
  - Performance (speed, response time)
  - Quality (correctness, security, availability)
  - Constraints (Environment, standards)
- System design: split into different components
  - Software architecture as a composition of sub systems
  - components and connectors
- Detailed Design: specify that components
  - Implementation strategy
  - Data structures, Algorithms & subclasses
    - ↳ Is null accepted
      - What can that method return?
      - Will the implementation be thread safe
- Initialization
  - Lazy: delay creation of an object, calculation of a value or something other expensive until the first time it's needed

- If object is accessed, it is checked if member is null or not
- A lot of comparisons
- Faster startup speed

- Immediate: Do the things when object is created

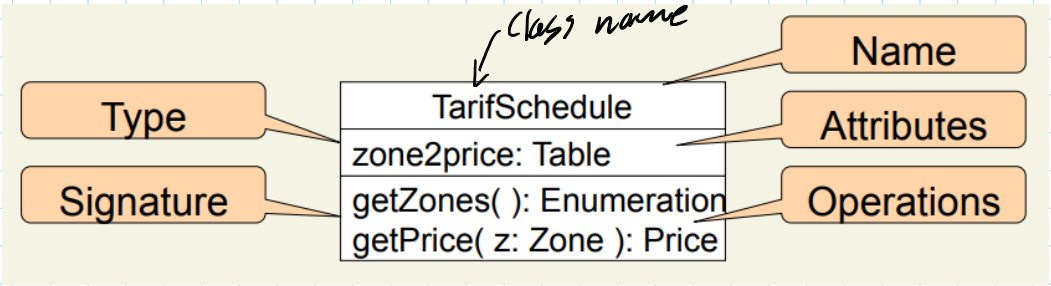
- Copy on write: Example shared lists
  - Multiple callers ask for same resource
  - Give all callers same pointer
  - When a caller tries to modify its "copy" of resource, at which point a true private copy is created (=copied) to prevent the changes becoming visible to everyone else.
  - Example shared lists:
    - Reference counting: global object
    - shared boolean: if one copies the list shared = True, if an element is changed, copy list and shared = false on copied object

- Documentation
  - For Implementers:
    - How does the code work
    - Includes hidden methods
  - For clients:
    - How to use the code/interface
    - constructors / Methods / Supertypes
    - Clients need to know how to call a method correctly and how it affects the state
    - No internal information

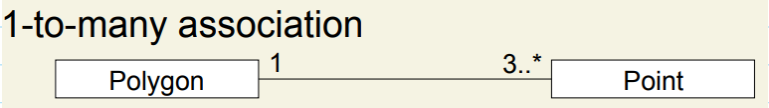
Informal Modelling

UML (=Unified modeling language)

- Static model: describes structure of a system
- Dynamic model: describes behavior of a system



- A link between two classes represent (=Associations)
  - Ability to send messages
  - Object A has an attribute whose value is B
  - Object A creates object B
  - Object A receives a message with object B as value
- Multiplicity of associations
  - Exact numbers
  - Arbitrary numbers: * (=zero or more)
  - Range: 1...3, 1...*



- Visibility: Person → Company

  Person knows about company

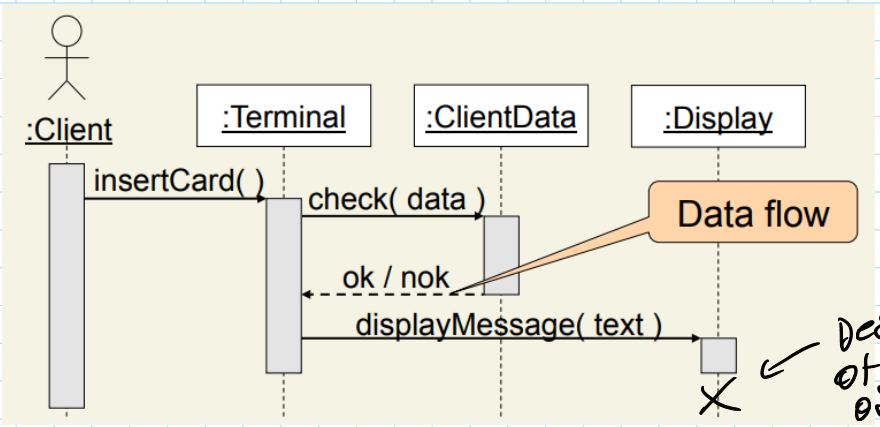
- Composition: · "Has-a" relationship

  · Circle ◆→ Center

- Generalization: · "Is-a" relationship
  - The child inherits the attributes and methods of the parent

child → Rectangle ⟶ Polygon ← Parent

- Sequence Diagram
  - Describes collaboration between objects

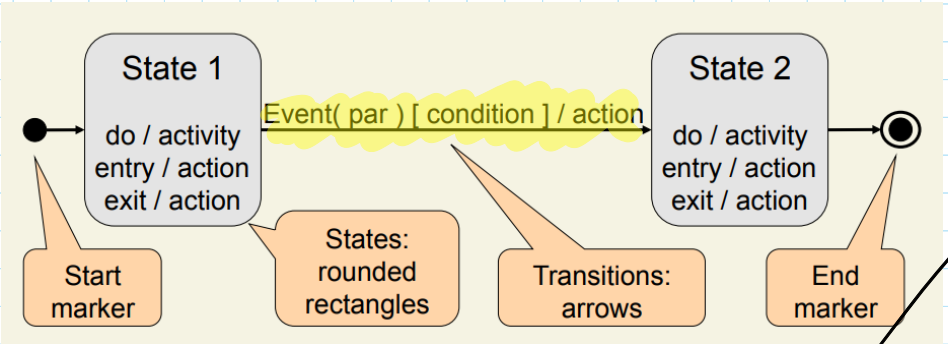


creation

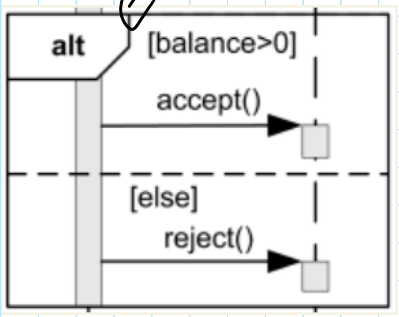
:Terminal create Account

Destruction of the object

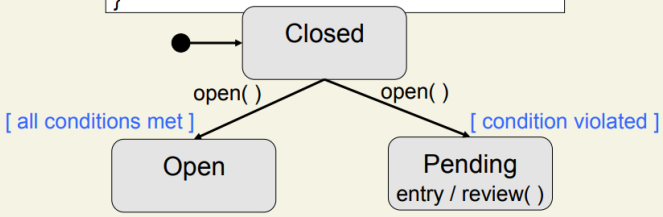
- State Diagram
  - Describes a lifetime of a single object



Alternative




```
abstract class Account {
  boolean open;
  abstract boolean allConditions( … );
  void open( … ) {
    if( allConditions( … ) )    open = true;
    else                        throw …;
  }
}
```

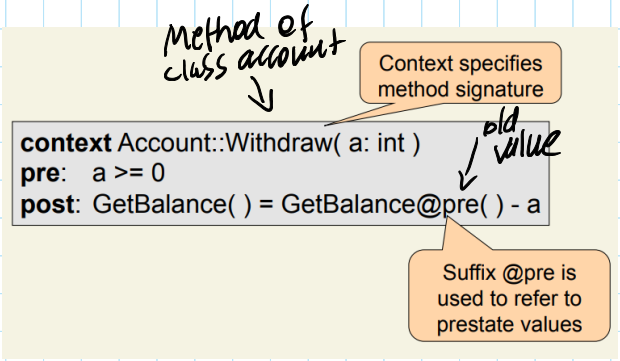


- Object Constraint Language (=OCL)

```
context SavingsAccount inv:
self.balance >= 0
```

```
context CheckingAccount inv:
self.owner.age >= 18
```

- Invariants
- SavingsAccount is a class



- Pre- and Postcondition of a method

Formal Modeling

- Class: sig FSObject {...}
- Abstract class: abstract sig FSObject {...}
- Subclass: sig file extends FSObject {...}
- Class cardinalities:
  - none (zero)
  - one (one) e.g. one sig Root
  - lone (zero/one)
  - some (one/two/three)
  - set (zero/one/two/...)
- Set operations:
  - + (Union)
  - & (Intersection)
  - - (Difference)
  - in (Subset)
  - = (Equality)
  - # (Cardinality)
  - none (Empty set)
- Fields: Declares attributes and relations
  - sig Customer {
    isPremium: Bool,
    address: one Address
    ↳ Refers to sig Address
  - sig University {
    enrollment: Student set -> one Program
    }

- Relation operations:
  - -> (Cross product)
  - . (relational join)
  - ~ (Transposition / Inverse)
  - ^ (Transitive closure)
  - * (reflexive transitive closure)
  - <: (Domain restriction)
  - :> (Range restriction
  - ++ (override)

- Quantification:
  - all x:e | F
  - some x:e | F
  - no x:e | F
  - lone x:e | F
  - one x:e | F

  all x,y:e | f
  all disj x,y:e | f

- Predicates:
  - Return true/false
  - pred isPremium [c:customer] {
    c.isPremium = true
    }

- Functions:
  - Return instances of a predicate
  - fun getAllOrders [c:Chef] : set order {
    c.Order
    }

    Returns orders

- Facts:
  - Constraints to model
  - fact pizza {
    all p: Pizza | one o: Order | p in o.pizza
    } // A pizza can only be assigned to one order

- Scopes: · run isPrevious for 5
  structure of given size
- Dynamic Behavior: · Choose a sig Counter as global state
  · Add ordering util/ordering[Counter]
  · Add initialisation of the first global state pred init[c: counter]
  · Add all transition predicates:

```
pred increase[c, c': counter, i: Int] {
    c'.max = c.max &&
    c'.count = c.count + i
}
```

  · Add execution fact:

```
fact execution {
    init[first] &&
    all c: Counter - last |
    (some i: Int | increase[c, c.next, i]) ||
    (other predicates)
}
```

  first element
  last element
- To add a file: c.objects + obj1
- To say that there are no files: no c.objects
  ↳ for initialisation
- lt[s, s'] s is lower then s'

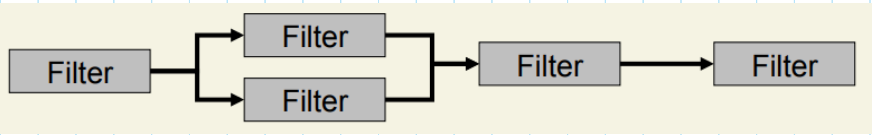
Modularity

- Coupling: measures independence between different modules
- Tightly coupled:
  - cannot developed, tested, changed, understood or reused in isolation
  - Exposing internal data is indicator for tightly coupled
- Data representation is difficult to change during maintenance
  - ↳ use getter/setter methods instead to hide implementation
    - non-functional requirements

Avoid exposure of sub-objects

- Do not return references to sub-objects
- Do not store arguments as sub-objects
- Facade Pattern
  - Use an API to access system
  - Encapsulates subsystem from client
  - Offers easy methods to client
- Flyweight Pattern:
  - Creating an object for each character in the document is costly
  - Define flyweight object that can be shared and is immutable

- Pipes and Filter Pattern
  - Used for data-streams



  - Filter: - Read data from input port, calculate something and output it into the pipe
  - Pipe: - Streams
    - First-in-first-out buffer
  - Data is processed incrementally as it arrives
  - No shared data => filters are independant
  - Not appropriate for interactive data
  - Filters can be easily added or replaced
  - Potential for parallelism

Tightly coupled methods

- If one module (caller) calls another module (callee)
- Callers cannot be reused without callee modules
- Change in callee => potential change in caller

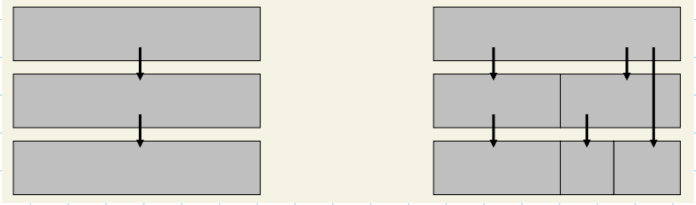
→ Solutions: - Move code
- Duplicate functionalities to avoid dependencies
- Event-based architecture
  - components generate events
  - e.g. User interfaces
- Implemented by Observer pattern

- Observer pattern: · Subject maintains a list of its dependents, called observers and notifies them of any state change by calling one of their methods => generation of an event

- Model-view-controller: · Model: -contains core functionality and data
  · View(s): -display information to user
  - Bar, charts, tables, etc.
  · Controller(s): - handle user input
  · Updates via events

- Event based style: · Strong support for reuse, plug in new components by registering for events
  · Add/remove components with no effect on other components
  · What component will respond to what event?
  ↳ loss of control

- Restricting calls: · Enforce a policy that restricts which other modules a module may call
  · Example: layered architecture



· Example: - three-tier architecture
↳ Front end → Back end → Database

Inheritance

- Extend functionality of a class by creating a subclass
  ↳ Introduces dependencies

- Aggregation: · If we need to override a lot of functionality use aggregation
  · class Cat {
    Eat = dog.eat
    Walk = dog.Walk } } Aggregation

- Using interfaces: · Replace class name by most general supertype that offers all required operations
  ↳ TreeMap<Ident, Type> => Map<Ident, Type>
  ↳ SymbolTable() {
    types = new TreeMap<Ident, Type>
  } convert to

  SymbolTable(Map<Ident, Type> t) {
    types = t
  } to shift problem to client

- Delegation Allocation: · Hand over responsibility for a task to another class (=Fabric)
  · AbstractFactory is called for a particular object
  · AbstractFactory calls a concreteFactory to generate concrete object

Adaptation

- Sometimes need to change software architecture
- For example new interfaces / features
- Parameterization:
  - Prepare modules for changes
  - Use variable values instead of constant ones
  - StringStream f1 => StringStream [] streams
  - Use generic types
    - ↳ Filter<D> [] filters
- Static:
  - Nur ein mal in ganzer Klasse, also alle Objekte zeigen auf gleiche Speicherstelle } Variable
  - Static method can be accessed without the creation of the corresponding object } Methods
  - Can access static variables
- Static Binding:
  - Can be resolved at compile time
  - static / private / final methods
  - Cannot be overridden
  - Compiler has no difficulties determining object of class
  - Static methods / variables cannot be overriden by subclasses
- Dynamic Binding:
  - Is resolved at run-time
  - Used for case distinction

```
class RegularMovie extends Movie {
    int getCost() { return 10; }
```

  - Dynamic method binding is a case distinction on the dynamic type of the receiver object.

Testing

- Unit Test (JUnit) for detailed design module

```
[ Test ]  ← @Test
public void withdrawTest( int balance, int amount ) {
  SavingsAccount target = new SavingsAccount();
  target.deposit( balance );
  target.withdraw( amount );
  Assert.IsTrue( target.getBalance( ) == balance – amount );
}
```

  - assertEquals(x, y)
  - assertTrue(bool)
  - assertArrayEquals([5],[5])

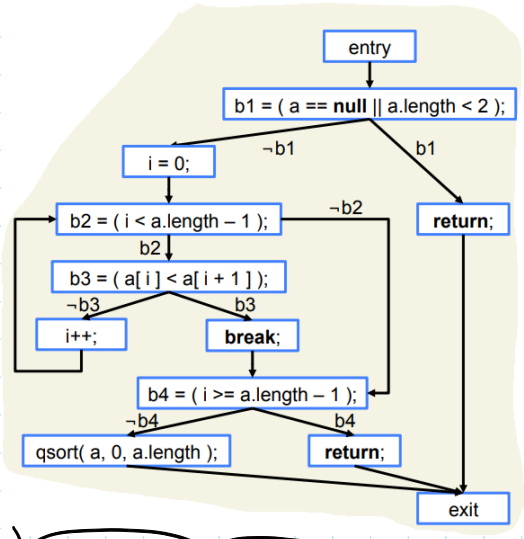
- Functional testing: Black-box for covering all requirements
- Structural testing: White-box for covering all code
- Random testing: Black-box for covering all corner-cases
- For exceptions: @Test(expected = exception.class)
- you can add many assertEquals(1, method(2)) into one method.

Structural Testing

- use design knowledge to cover most of the code

```
public void sort( int[ ] a ) {
  if( a == null || a.length < 2 )
    return;
  int i;
  for( i = 0; i < a.length – 1; i++ ) {
    if( a[ i ] < a[ i + 1 ] )
      break;
  }
  if( i >= a.length – 1 )
    return;
  qsort( a, 0, a.length );
}
```

Basic blocks

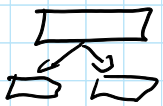


Control flow graph (= CFG)

- The more parts of a CFG are executed, the higher the chance is, to find a bug
- statement coverage: $\frac{\#\text{executed blocks}}{\#\text{total blocks}}$

  - entry/exit not counted as block

- Branch coverage: $\frac{\#\text{executed branches}}{\#\text{total branches}}$

  - Branch:  : No branch 

- Path coverage: $\frac{\#\text{executed paths}}{\#\text{total paths}}$
  - A path is a sequence of blocks from n_1 = entry to n_k = exit
  - An arbitrary large number of test cases is needed to complete path coverage for loops
- Loop coverage: $\frac{\#\text{of executed loops with 0, 1, and more than one iteration}}{\#\text{number of loops} * 3}$
- Loop coverage => path coverage => branch coverage => statement coverage
  - ↳ if 100%
- Exceptions
  - Treat as block with arrow to exit
  - Checked exception:
    - Need to be catched
    - At compile time
    - e.g. fileNotFound

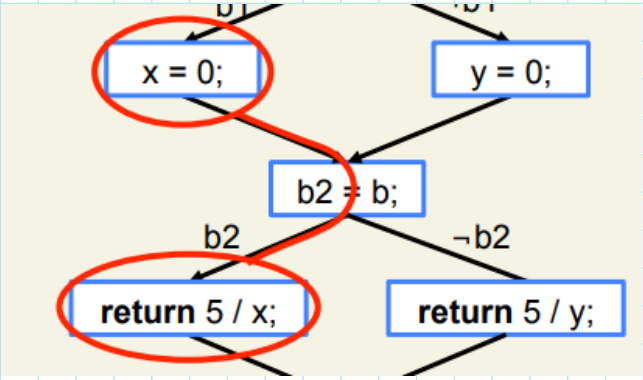
    } Regular control flow

  - Unchecked exception:
    - At run-time
    - division by zero
    - Not supposed to occur

    } Ignore this type of exception

- Definition clear path
  - No altering of a specific variable
  - n_1 is the definition of the variable
  - n_k alters the variable

- Definition-use pair (=DU-pair)
  - Variable v
  - Pair of nodes (d, u) such there is a definition-clear path

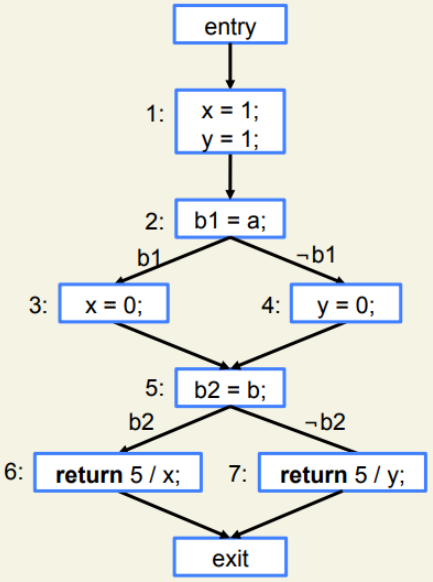


- Static reaching-definition analysis

- The reaching definitions at a node n are:
  - The reaching definitions of n's predecessors in the CFG
  - minus the definitions killed by one of n'd predecessors
  - plus the definitions made by one of n'd predecessors

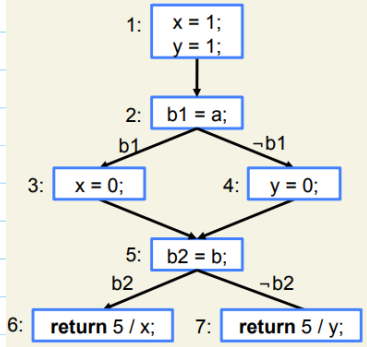
- pred(n) = { m | (m,n,c) is an edge in the CFG }
- succ(m) = { n | (m,n,c) is an edge in the CFG }
- gen(n) = { v_n | n is a variable definition for v }
- kill(n) = { v_m | n is a variable definition for v and m ≠ n }

- Reach(n): The reaching definitions at the beginning of n
- ReachOut(n): The reaching definitions at the end of n



n	Reach(n)	ReachOut(n)
1	∅	x_1, y_1
2	x_1, y_1	x_1, y_1
3	x_1, y_1	x_3, y_1
4	x_1, y_1	x_1, y_4
5	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4
6	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4
7	x_1, x_3, y_1, y_4	x_1, x_3, y_1, y_4

- The set of DU-pairs is easily determined as
  { (d,u) | u is a variable use for v and $v_d \in$ Reach(u) }



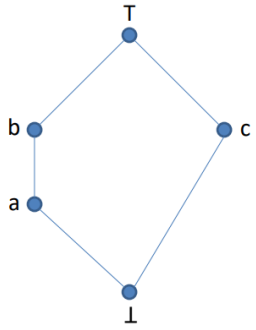
n	Reach(n)
1	∅
2	x_1, y_1
3	x_1, y_1
4	x_1, y_1
5	x_1, x_3, y_1, y_4
6	x_1, x_3, y_1, y_4
7	x_1, x_3, y_1, y_4

- DU-pairs for x: (1,6), (3,6)
- DU-pairs for y: (1,7), (4,7)

Joins ($= A \sqcup B$)

- When we have two abstract elements A and B, we can join them
- $A \sqsubseteq A \sqcup B$ && $B \sqsubseteq A \sqcup B$
- Cannot always reach a fixed point => Widening

Poset (=Partially ordered set)

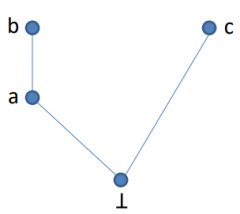


- T : Top
  - Includes everything else
- ⊥ : Bottom

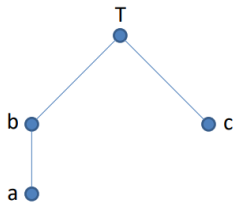
{ (⊥ ,a), (⊥ ,c), (a, b), (b, T), (c, T), (a, a), (b, b), (c, c), (T, T), (⊥ ,⊥), (⊥ ,b), (⊥ , T), (a, T) }

$\left.\begin{array}{l} \bot \sqsubseteq a \\ \bot \sqsubseteq c \end{array}\right\}$ T contains everything, $\bot \sqsubseteq \bot$

- Least/greatest element may not exist, but if they do, they are unique



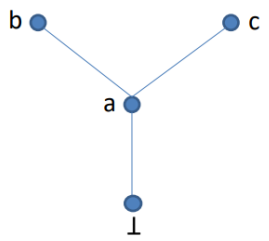
No greatest element



No least element

} a and c are on the same level

Bounds in Posets



{b, c} has no upper bound

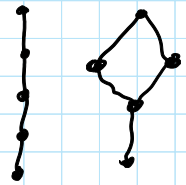
{b, c} has 2 lower bounds: a and ⊥

- {c} · has as upper bound {a, b}
  - · has as lower bound {d, f}
  - · has as least upper bound {b} (⊔)
  - · has as greatest lower bound {d} (⊓)

⇒ we write $p \sqcap q$ for $\sqcap \{p, q\}$

Complete Lattice

- A Poset in which each element has a ⊓ & ⊔



Monotone Function

- $d \sqsubseteq d' \Rightarrow f(d) \sqsubseteq f(d')$

Not a monotone function

Fixed Point

- x is a fixed point, iff $f(x) = x$
- x is a post-fixedpoint, iff $f(x) \sqsubseteq x$ $(= f(x) \in x)$

All fixed points are post-fixed points

Least fixed point

- Smallest fixed point
- Fixed point which is less than all other fixed points

Tarki's fixed point theorem

- Complete Lattice + monotone function ⇒ Least-fixed point exists

Approximating a function

- Concrete C
- Abstract A (= approximates the concrete function)
- $\alpha : C \to A$ (= abstraction function)
- $\gamma : A \to C$ (= concretization function)

Abstract Interpretation

1. Select abstract domain
   - e.g. interval domain

- $[a,b] \sqsubseteq_i [c,d]$ if $c \leq a$ and $b \leq d$
- $[a,b] \sqcup_i [c,d] = [\min(\{a,c\}), \max(\{b,d\})]$
- $[a,b] \sqcap_i [c,d] = \text{meet}(\max(\{a,c\}), \min(\{b,d\}))$
  where $\text{meet}(a,b)$ returns $[a,b]$ if $a \leq b$ and $\bot_i$ otherwise

$$\alpha^i : \wp(\Sigma) \to (\text{Lab} \to (\text{Var} \to L^i))$$
$$\gamma^i : (\text{Lab} \to (\text{Var} \to L^i)) \to \wp(\Sigma)$$

(Lab: current label; Var: variable; L^i: value of variable in abstract domain)

Actual program states

$$\alpha^i(\{\langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto -2\}\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto -2\}\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto -2\}\rangle,$$
$$\langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto 4\}\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto 4\}\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto 4\}\rangle\})$$
$$= 1 \to (x \mapsto [1,8],\ y \mapsto [9,9],\ q \mapsto [-2,4])$$

Abstracted to intervals

$$\gamma^i(1 \to (x \mapsto [1,8],\ y \mapsto [9,9],\ q \mapsto [-2,4]))$$
$$= \{\langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto -2\}\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto -2\}\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto -2\}\rangle,$$
$$\langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto 4\}\rangle, \langle 1,\{x\mapsto 5,y\mapsto 9,q\mapsto 4\}\rangle, \langle 1,\{x\mapsto 8,y\mapsto 9,q\mapsto 4\}\rangle,$$
$$\langle 1,\{x\mapsto 7,y\mapsto 9,q\mapsto 3\}\rangle, \langle 1,\{x\mapsto 3,y\mapsto 9,q\mapsto 4\}\rangle, \langle 1,\{x\mapsto 1,y\mapsto 9,q\mapsto -1\}\rangle,$$
$$\ldots, \ldots, \ldots\}$$

can be many more actual states

2. Select abstract semantics

Next state where transformer is applied

$$F^{\#}: (Lab \to A) \to (Lab \to A)$$

Transformer map

$$F^{\#}(m)\ell = \begin{cases} \top & \text{if } \ell \text{ is initial label} \\ \bigsqcup_{(\ell', action, \ell)} [\![action]\!](m(\ell')) & \text{otherwise} \end{cases}$$

Approximation function for concrete program

- $(\ell', action, \ell) \triangleq \langle \nu', \sigma' \rangle \to \langle \nu, \sigma \rangle$
  - $x := a$

$$[0,4] \leq [3,5] = ([0,4] \sqcap_i [-\infty,5], [0,\infty] \sqcap_i [3,5]) = ([0,4], [3,5])$$

> Transformer definition

3. Iterate over abstract domain with transformer until we reach fixed point

```
foo (int i) {

1: int x :=5;
2: int y :=7;

3: if (i ≥ 0) {
4:    y := y + 1;
5:    i := i - 1;
6:    goto 3;
   }
7:
}
```

Not initialized $= \top$

1: $x \to [-\infty,\infty]$, $y \to [-\infty,\infty]$, $i \to [-\infty,\infty]$

2: $x \to [5,5]$, $y \to [-\infty,\infty]$, $i \to [-\infty,\infty]$

3: $x \to [5,5]$, $y \to [7,7]$, $i \to [-\infty,\infty]$

4: $x \to \bot_i$, $y \to \bot_i$, $i \to \bot_i$

5: $x \to \bot_i$, $y \to \bot_i$, $i \to \bot_i$

6: $x \to \bot_i$, $y \to \bot_i$, $i \to \bot_i$

7: $x \to \bot_i$, $y \to \bot_i$, $i \to \bot_i$

- If we can't reach fixed point => Widening

Widening Operator (∇)

- If we see that an endpoint is unstable, we move it to the extreme

- $[1, 2] \nabla [0, 2] = [-\infty, 2]$
- $[0, 2] \nabla [1, 2] = [0, 2]$
- $[1, 5] \nabla [1, 5] = [1, 5]$
- $[2, 3] \nabla [2, 4] = [2, \infty]$

last state / current state

Pointer Analysis

- All objects allocated at the same program point (=label) get represented by a single abstract object.
- Flow sensitive:
  - Respect the program order
  - Separate set of points-to pairs for every program point
  - Set represents all possible may-alliases
- Flow insensitive:
  - All execution orders are possible
  - Good for concurrency
- We have two mappings:
  - Pointer variable → set abstract objects
  - Variable abstract objects → set of abstract objects

Flow Sensitive

label; pointer to object; set symbol; objects pointer can point to

$$Labs \rightarrow ((PtrVar \rightarrow \wp(AbsObj)) \times (AbsObj \times Field \rightarrow \wp(AbsObj)))$$

label; object; variable of object

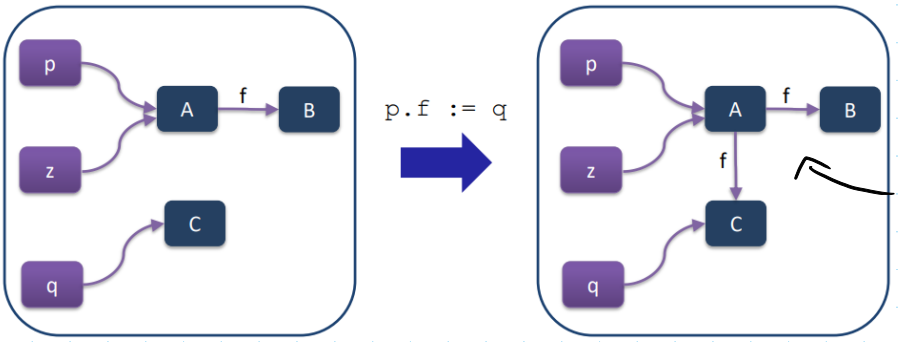
$$1 \rightarrow (p \rightarrow \{a_5, a_{10}\}, a_5.f \rightarrow \{a_6, a_9\})$$

pointer p; variable of object a_5

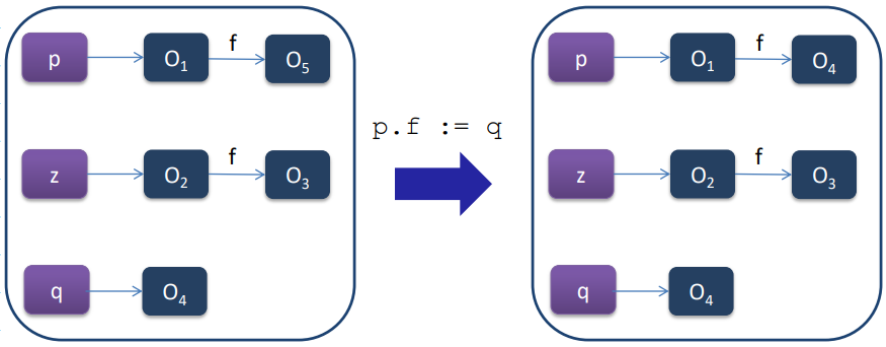
$$1 \rightarrow (p \rightarrow \{a_5, a_{10}\}, a_5.f \rightarrow \{a_6, a_9\})$$

$$\sqsubseteq$$

$$1 \rightarrow (p \rightarrow \{a_5, a_{10}, a_{15}\}, a_5.f \rightarrow \{a_6, a_9, a_{52}\})$$



Need to take the union



- Initial state: Top = {A1, A2, null}
- Unreachable object: Bottom = ∅

```
p :=alloc1 ; // A1
q :=alloc2; // A2
if  p=q 3 then
   z:=p 4
else
   z:=q 5
```

```
p ↦ {A1, A2, null},
q ↦ {A1, A2, null},
z ↦ {A1, A2, null}
```

```
p ↦ {A1}, q ↦ {A1, A2, null},
, z ↦ {A1, A2, null}
```

```
p ↦ {A1}, q ↦ {A2},
z ↦ {A1, A2, null}
```

```
p ↦ ∅, q ↦ ∅, z ↦ {A1, A2, null}
```

```
p ↦ ∅, q ↦ ∅, z ↦ ∅
```

```
p ↦ {A1}, q ↦ {A2},
z ↦ {A1, A2, null}
```

```
p ↦ {A1}, q ↦ {A2}, z ↦ {A2}
```

Flow sensitive

Flow insensitive

- Does not keep information per label

```
(PtrVar → ℘(AbsObj)) ×
(AbsObj × Field → ℘(AbsObj))
```

```
p :=alloc1; // A1
q :=alloc2; // A2
if  p=q 3 then
   z:=p 4
else
   z:=q 5
```

⇒

```
p :=alloc1; // A1
q :=alloc2; // A2
z:=p 4
z:=q 5
```

First, look at all object creation terms

⇒ Output of Analysis:

```
p ↦ {A1},q ↦ {A2},z ↦ {A1, A2}
```

← Then, look at other things and add them

Symbolic execution

- Instance of under-approximation and over-approximation
- We give each variable a symbolic value and get big constraint formula for each one
- During execution we keep two sets:
  - symbolic store: - Input: Variable
    - Output: symbolic value
    - ↳ σ_s: $x \to x_0$, $y \to y_0$, $z \to x_0 + y_0$
  - Path constraints: - Records the history of all branches taken so far
    - To initialize: pct: true
    - ↳ pct: $x_0 > 10 \wedge x_0 > y_0 + 1$
- We explore all paths of the program
  - At each branch we duplicate the current state of the analysis
  - Exponentially in the number of branches
- At the end we can ask the SMT-solver for a satisfying assignment to the pct formula
- Unbounded loops: - Underapproximate them
  - Bound iteration by some number
- Caching: - Save formula → solution mapping
  - If same/weaker formula reappears, don't use SMT, just use cached solution
  - If stronger formula appears, check first, if cached solution trust the stronger formula has some subparts in it, works.

Concolic Execution

- SMT-solver can't solve non-linear functions well
- Combine concrete and symbolic execution
- Run program with concrete values but keep constraints

```
int twice(int v) {
  return 2 * v;
}

void test(int x, int y) {
  z = twice(y);
  if (x == z) {
    if (x > y + 10)
      ERROR;
  }
}

int main() {
  x = read();
  y = read();
  test(x,y);
}
```

Hence, we get:

bad decision for values because we don't reach ERROR

σ : x ↦ 22, y ↦ 7, z ↦ 14

σ_s : x ↦ x0, y ↦ y0, z ↦ 2*y0

pct : x0 ≠ 2*y0

- Wenn die values nicht richtig sind, um die richtigen branch-paths zu nehmen:
  - Negate pct, dann rufen wir den SMT solver um concrete values zu bekommen
  - Wenn wir diese values bei der ernenten Ausführung wählen, nehmen wir die andere branch predicting
- Wenn es sich um non-linear constraints handelt, setzen wir concrete values in die formula ein um so eine einfachere Formel zu bekommen

1. $x \neq y0 \cdot y0$ / negate it
2. $x = y0 \cdot y0$ / insert actual values
3. $x = 8 \cdot 8$
4. $x = 64$