Abstract Interpretation

(Cousot, Cousot 1977)

also known as

Data-Flow Analysis

(Kildall 1973)

Example: Constant propagation
Here is why it is useful

```
int a, b, step, i;
 boolean c;
 a = 0;
 b = a + 10;
 step = -1;
 if (step > 0) {
  i = a;
 } else {
  i = b;
 }
 c = true;
 while (c) {
  print(i);
  i = i + step;    // can emit decrement
  if (step > 0) {
   c = (i < b);
  } else {
   c = (i > a); // can emit better instruction here
 } // insert here (a = a + step), redo analysis
}
```

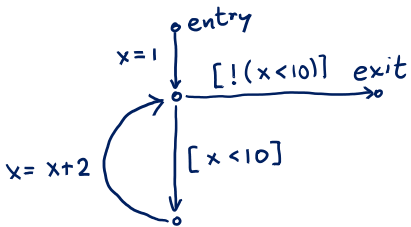
Goal of Data-Flow Analysis

Automatically compute information about the program

- Use it to report errors to user (like type errors)
- Use it to optimize the program

Works on control-flow graphs:
(like flow-charts)

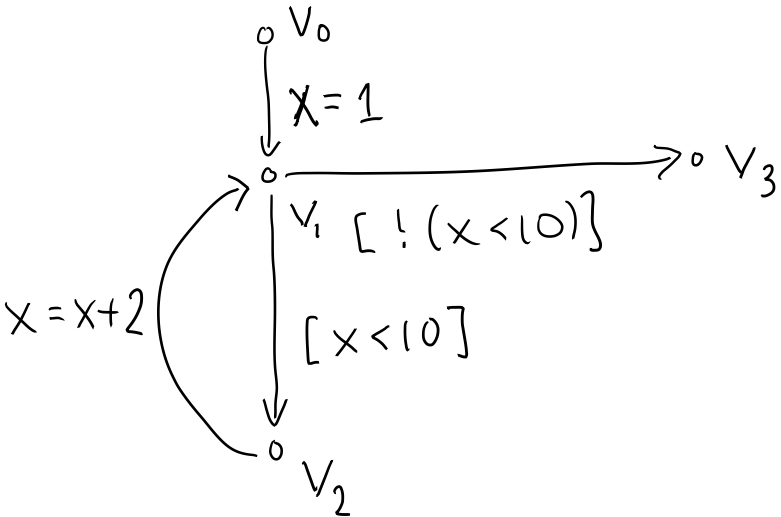
```
x = 1
while (x < 10) {
  x = x + 2
}
```



Interpretation and Abstract Interpratation

- Control-Flow graph is similar to AST
- We can
  - interpret control flow graph
  - generate machine code from it (e.g. LLVM, gcc)
  - abstractly interpret it: do not push values, but **approximately compute supersets of possible values** (e.g. intervals, types, etc.)

Compute Range of x at Each Point



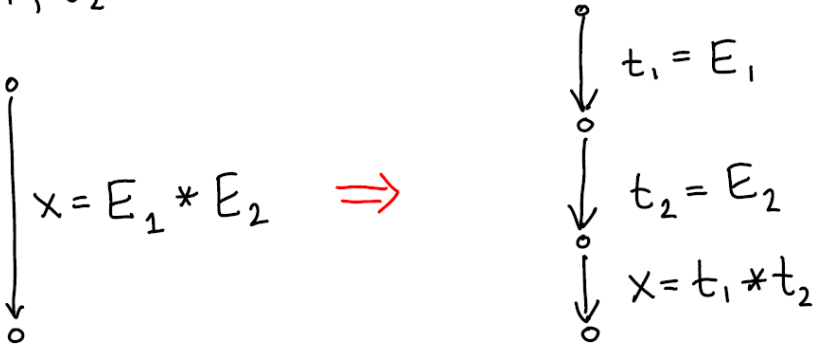
Generating Control-Flow Graphs

- Start with graph that has one entry and one exit node and label is entire program
- Recursively decompose the program to have more edges with simpler labels
- When labels cannot be decomposed further, we are done

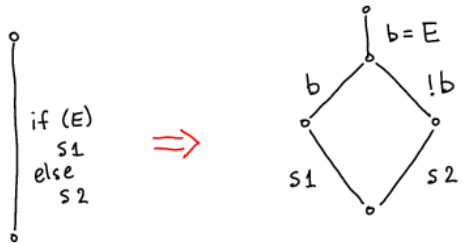
Flattening Expressions
for simplicity and ordering of side effects

E_1, E_2 – complex expressions

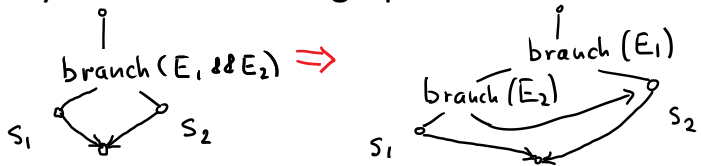
t_1, t_2 – fresh variables



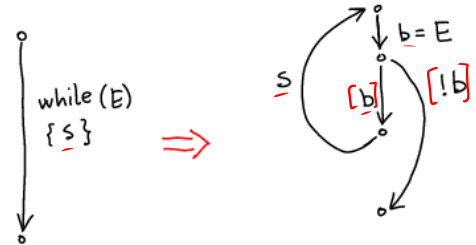
If-Then-Else



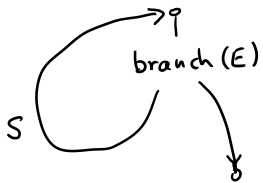
Translation using "branch" instruction works nicely for control flow graphs: two destinations



While



Better translation uses the "branch" instruction



Example 1: Convert to CFG

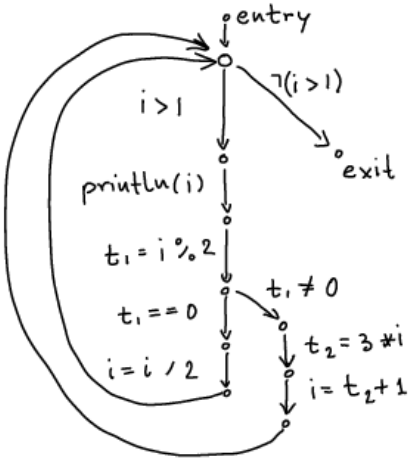
```
while (i < 10) {
  println(j);
  i = i + 1;
  j = j +2*i + 1;
}
```

Example 2: Convert to CFG

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```

Example 2 Result

```
int i = n;
while (i > 1) {
  println(i);
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3*i + 1;
  }
}
```



Translation Functions

$[\ s_1\ ;\ s_2\ ]\ v_{source}\ v_{target}$ =

$[\ s_1\ ]\ v_{source}\ v_{fresh}$

$[\ s_2\ ]\ v_{fresh}\ v_{target}$

insert $(v_s,stmt,v_t)$=

cfg = cfg + $(v_s,stmt,\ v_t)$

$[$ **branch**$(x<y)\ ]\ v_{source}\ v_{true}\ v_{false}$ =

insert$(v_{source},[x<y],v_{true})$;

insert$(v_{source},[!(x<y)],v_{false})$

$[\ x=y+z\ ]\ v_s\ v_t$ = **insert**$(v_s,x=y+z,\ v_t)$

when y,z are constants or variables

Abstract Interpretation:
Analysis Domain (D)
Lattices

Lattice

Partial order: binary relation $\leq$ (subset of some D^2) which is

- reflexive: $x \leq x$
- anti-symmetric: $x \leq y \wedge y \leq x \rightarrow x = y$
- transitive: $x \leq y \wedge y \leq z \rightarrow x \leq z$

Lattice is a partial order in which every **two-element** set has **least among its upper bounds** and **greatest among its lower bounds**

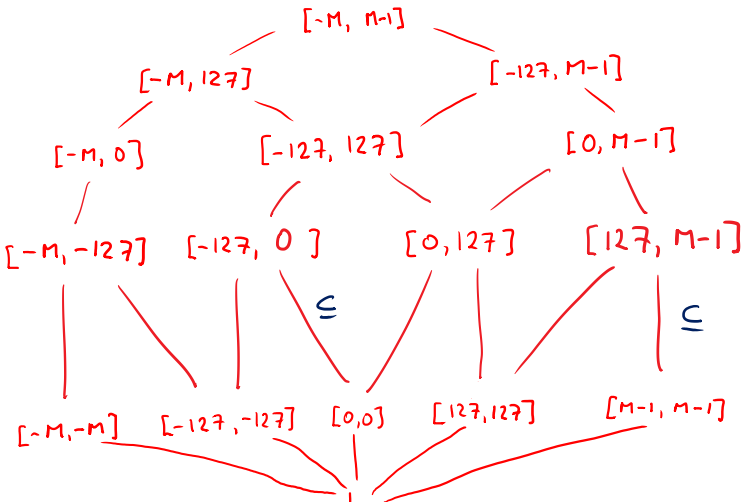
- Lemma: if $(D, \leq)$ is lattice and D is finite, then lub and glb exist for every finite set

$\bigsqcap \quad \bigsqcup \qquad \bigsqcup \{a, b, c\}$

Graphs and Partial Orders

- If the domain is finite, then partial order can be represented by directed graphs
  - if $x \leq y$ then draw edge from x to y
- For partial order, no need to draw $x \leq z$ if $x \leq y$ and $y \leq z$. So we only draw non-transitive edges
- Also, because always $x \leq x$, we do not draw those self loops
- Note that the resulting graph is acyclic: if we had a cycle, the elements must to be equal

Domain of Intervals [a,b] where a,b∈{-M,-127,0,127,M-1}



Defining Abstract Interpretation

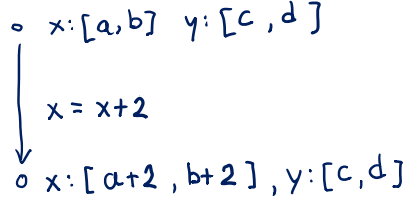
Abstract Domain D describing which information to compute – this is often a lattice

- inferred types for each variable: x:T1, y:T2
- interval for each variable x:[a,b], y:[a',b']

Transfer Functions, [[**st**]] for each statement **st**, how this statement affects the facts $D \to D$

- Example:

$[\![x = x+2]\!](x:[a,b], \ldots) = (x:[a+2, b+2], \ldots)$

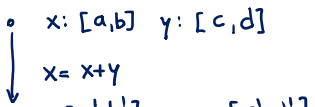


For now, we consider arbitrary integer bounds for intervals

- Thus, we work with BigInt-s
- Often we must analyze machine integers
  - need to correctly represent (and/or warn about) overflows and underflows
  - fundamentally same approach as for unbounded integers
- For efficiency, many analysis do not consider arbitrary intervals, but only a subset of them W
- We consider as the domain
  - empty set (denoted $\perp$, pronounced “bottom”)
  - all intervals [a,b] where a,b are integers and $a \leq b$, or where we allow $a = -\infty$ and/or $b = \infty$
  - set of all integers $[-\infty, \infty]$ is denoted T , pronounced “top”

Find Transfer Function: Plus

Suppose we have only two integer variables: x,y



If $a \le x \le b$ $c \le y \le d$

and we execute **x= x+y**

then $x' = x+y$

$y' = y$

so

$\le x' \le$

$\le y' \le$

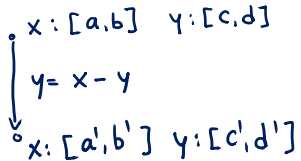
So we can let

a'= a+c b' = b+d

c'=c d' = d

Find Transfer Function: Minus

Suppose we have only two integer variables: x,y



If

and we execute **y= x-y**

then

So we can let

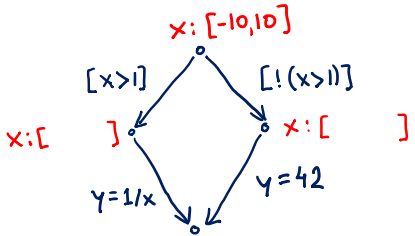
a'= a b' = b

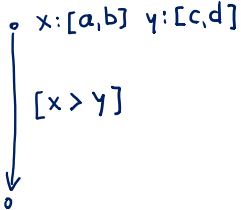
c'= a - d d' = b - c

Transfer Functions for **Tests**

Tests e.g. [x>1] come from translating if,while into CFG

```
x : [-10,10]
if (x > 1) {
  x :
  y = 1 / x
} else {
  x :
  y = 42
}
```

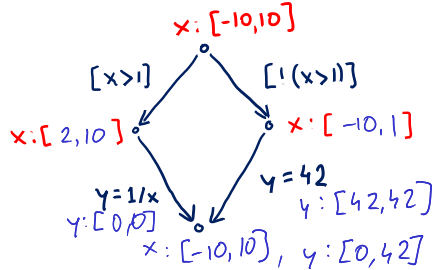


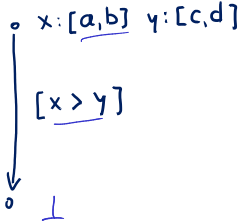


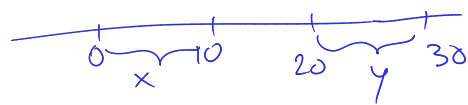
Transfer Functions for **Tests**

Tests e.g. [x>1] come from translating if,while into CFG

```
x : [-10,10]
if (x > 1) {
  x : [2, 10]
  y = 1 / x
} else {
  x : [-10, 1]
  y = 42
}
```



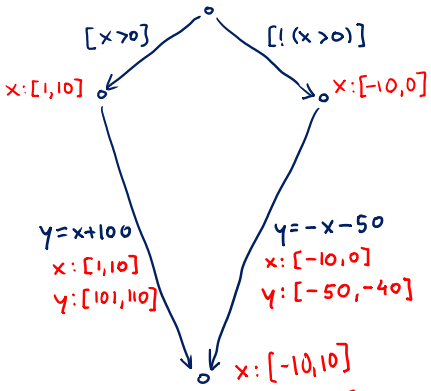




Joining Data-Flow Facts

x: [-10,10] y: [-1000,1000]

```
if (x > 0) {
   x:                y:
   y = x + 100
   x:                y:
} else {
   x:                y:
   y = -x – 50
   x:                y:
}
x:                   y:
```



join ⊔

Joining Data-Flow Facts

x: [-10,10] y: [-1000,1000]

```
if (x > 0) {
```
x: [1, 10] y: [-1000, 1000]

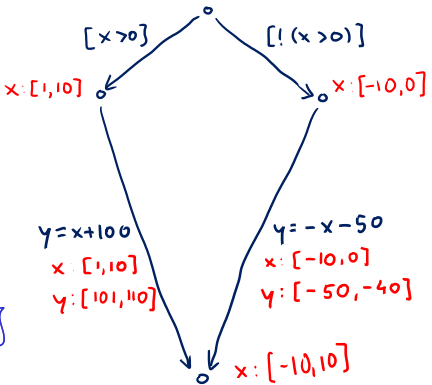
```
  y = x + 100
```
x: [1,10] y: [101,110]

```
} else {
```
[x ≤ 0]

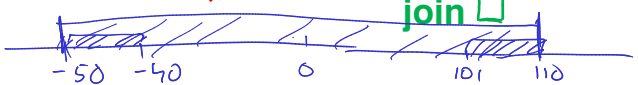
x: [-10,0] y: [-1000,1000]

```
  y = -x – 50
```
x: [-10,10] y: [-50,-40]

```
}
```

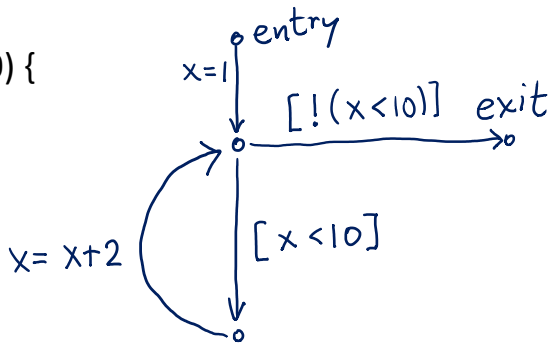


x: y: join ⊔



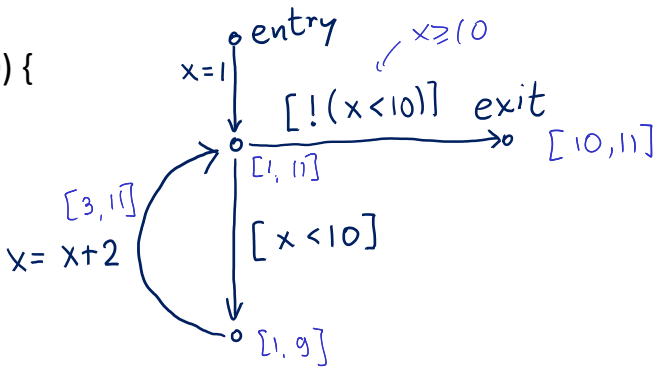
Handling Loops: Iterate Until Stabilizes

```
x = 1

while (x < 10) {

  x = x + 2

}
```



Handling Loops: Iterate Until Stabilizes

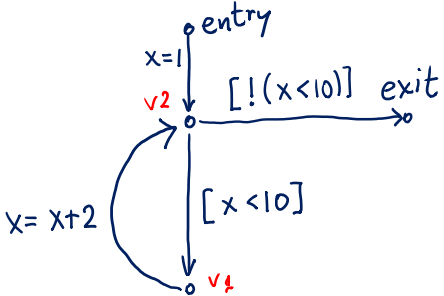
```
x = 1
[1,1]
while (x < 10) {
  [1,1]
  x = x + 2
  [3,3]
}
```



Analysis Algorithm

```
var facts : Map[Node,Domain] = Map.withDefault(empty)
facts(entry) = initialValues
while (there was change)
  pick edge (v1,statmt,v2) from CFG
        such that facts(v1) has changed
  facts(v2) = facts(v2) join transferFun(statmt, facts(v1))
}
```

Order does not matter for the end result, as long as we do not permanently neglect any edge whose source was changed.



```
var facts : Map[Node,Domain] = Map.withDefault(empty)
var worklist : Queue[Node] = empty

def assign(v1: Node, d: Domain) = if facts(v1) != d then
  facts(v1) = d
  for (stmt, v2) <- outEdges(v1) do worklist.add(v2)

assign(entry, initialValues)
while (!worklist.isEmpty) do
  var v2 = worklist.getAndRemoveFirst
  update = facts(v2)
  for (v1, stmt) <- inEdges(v2)
    do update = update join transferFun(facts(v1), stmt)
  assign(v2, update)
```

Work-List Version

Exercise: Run range analysis, prove that **error** is unreachable

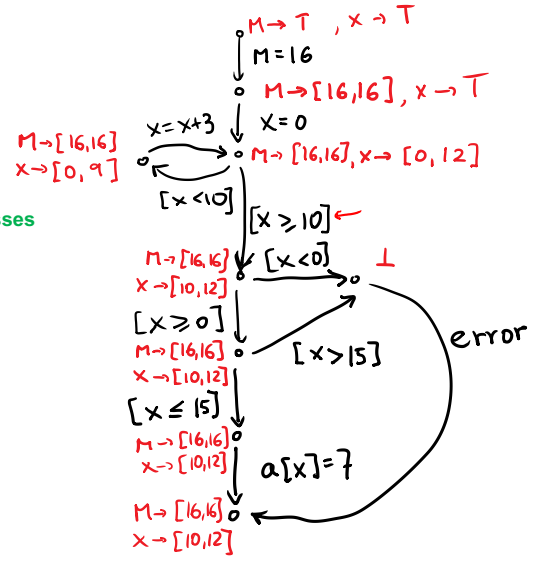
```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
     error;
} else {
   error;
}
```

checks array accesses

Range analysis results

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

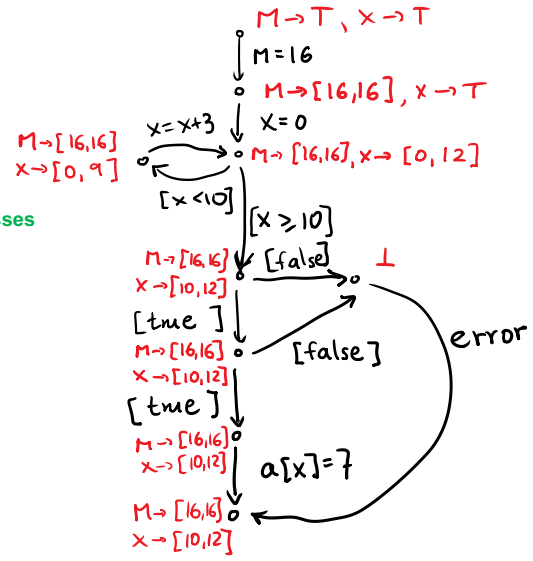
checks array accesses



Simplified Conditions

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
   error;
}
```

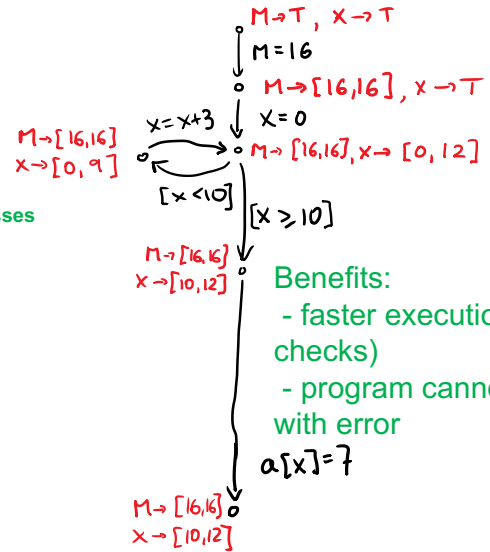
checks array accesses



Remove Trivial Edges, Unreachable Nodes

```
int M = 16;
int[M] a;
x := 0;
while (x < 10) {
  x := x + 3;
}
if (x >= 0) {
  if (x <= 15)
    a[x]=7;
  else
    error;
} else {
  error;
}
```

checks array accesses



Benefits:
- faster execution (no checks)
- program cannot crash with error

Constant Propagation Domain

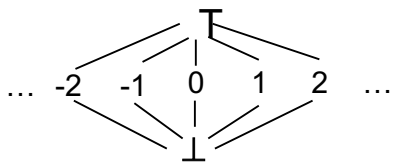
Domain values D are:

- intervals [a,a], denoted simply ‘a’
- empty set, denoted ⊥ and set of all integers T

Formally, if **Z** denotes integers, then

$$D = \{\bot, T\} \cup \{ a \mid a \in \mathbf{Z} \}$$

D is an infinite set



Constant Propagation Transfer Functions

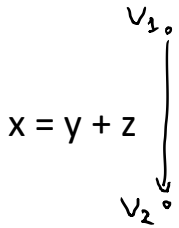


table for +:

z \ y	$\bot$	C_y	$\top$
$\bot$	$\bot$	$\bot$	$\bot$
C_z	$\bot$	C_y+C_z	$\top$
$\top$	$\bot$	$\top$	$\top$

For **each variable** (x,y,z) and **each CFG node (program point)** we store: ⊥ , a constant, or T

```
enum Element:
  case Top
  case Bot
  case Const(v: Int)
var facts : Map[Nodes,Map[VarNames,Element]]
  what executes during analysis of x=y+z:
oldY = facts(v1)("y")
oldZ = facts(v1)("z")
newX = tableForPlus(oldY, oldZ)
facts(v2) = facts(v2) join facts(v1).updated("x", newX)
```

```
def tableForPlus(y:Element, z:Element)
= (x,y) match
   case (Const(cy),Const(cz)) =>
                    Const(cy+cz)
   case (Bot, _) => Bot
   case (_, Bot) => Bot
   case (Top, Const(cz)) => Top
   case (Const(cy), Top) => Top
```

Run Constant Propagation

What is the number of updates?

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

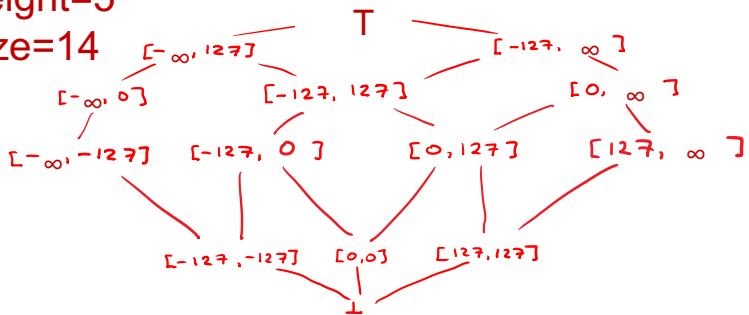
```
x = 1
n = readInt()
while (x < n) {
  x = x + 2
}
```

Observe

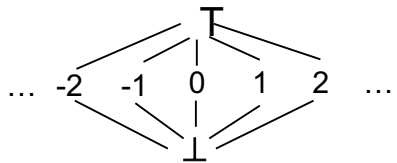
- Range analysis with end points
  W = {-128, 0, 127} has a finite domain
- Constant propagation has infinite domain
  (for every integer constant, one element)
- Yet, constant propagation finishes sooner!
  - it is not about the size of the domain
  - it is about the height

Height of Lattice: Length of Max. Chain

height=5
size=14



height=2
size =∞

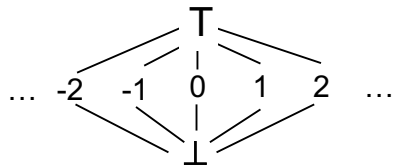


Chain of Length n

- A set of elements $x_0, x_1, ..., x_n$ in D that are linearly ordered, that is $x_0 < x_1 < ... < x_n$
- A lattice can have many chains. Its **height** is the maximum n for all the chains
- If there is no upper bound on lengths of chains, we say lattice has **infinite height**
- Any monotonic sequence of distinct elements has length at most equal to lattice height
  - including sequence occuring during analysis!
  - **such sequences are always monotonic**

In constant propagation, each value can change only twice

height=2
size =∞



consider value for x before assignment

- Initially: ⊥
- changes 1st time to: 1
- change 2nd time to: T

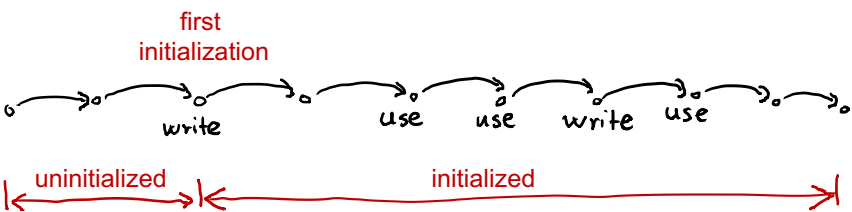
total changes: two (height)

```
x = 1
n = 1000
while (x < n) {
  x = x + 2
}
```

Total number of changes bounded by: **height·|Nodes| ·|Vars|**

var facts : **Map[Nodes,Map[VarNames,Element]]**

Initialization Analysis



What does javac say to this:

```
class Test {
   static void test(int p) {
         int n;
         p = p - 1;
         if (p > 0) {
            n = 100;
         }
         while (n != 0) {
            System.out.println(n);
            n = n - p;
         }
   }
}
```

```
Test.java:8: variable n might not have been initialized
         while (n > 0) {
                ^
1 error
```

Program that compiles in java

```
class Test {
   static void test(int p) {
         int n;
         p = p - 1;
         if (p > 0) {
            n = 100;
         }
         else {
            n = -100;
         }
         while (n != 0) {
            System.out.println(n);
            n = n - p;
         }
   }
}
```

We would like variables to be initialized on all execution paths.

Otherwise, the program execution could be undesirably affected by the value that was in the variable initially.

We can enforce such check using initialization analysis.

What does javac say to this?

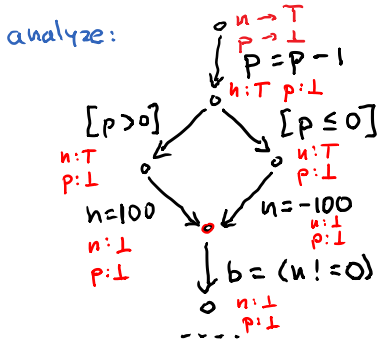
```
static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
           n = 100;
        }
        System.out.println("Hello!");
        if (p > 0) {
           while (n != 0) {
              System.out.println(n);
              n = n - p;
           }
        }
}
```

Initialization Analysis

```
class Test {
   static void test(int p) {
         int n;
         p = p - 1;
         if (p > 0) {
            n = 100;
         }
         else {
            n = -100;
         }
         while (n != 0) {
            System.out.println(n);
            n = n - p;
         }
   }
}
```

T indicates presence of flow from states where variable was not initialized:

- If variable is **possibly uninitialized**, we use T
- Otherwise (initialized, or unreachable): ⊥



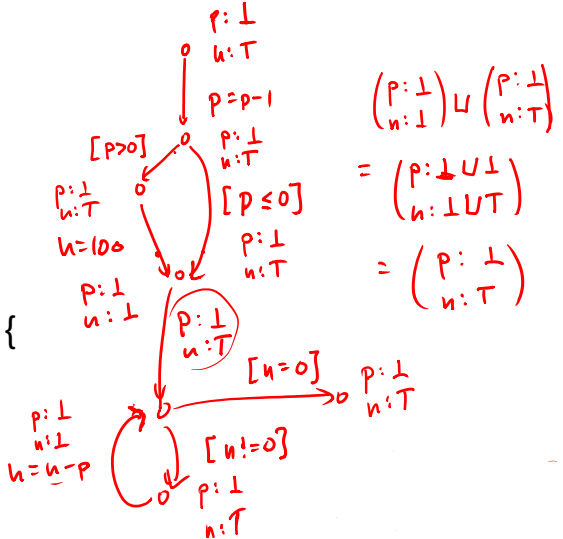
If var occurs anywhere but left-hand side of assignment and has value T, report error

Sketch of Initialization Analysis

- Domain: for each variable, for each program point: D = {⊥,T}
- At program entry, local variables: T ; parameters: ⊥
- At other program points: each variable: ⊥
- An assignment x = e sets variable x to ⊥
- lub (join, ⊔) of any value with T gives T $T \sqcup \bot = T$
  - uninitialized values are contagious along paths
  - ⊥ value for x means there is definitely no possibility for accessing uninitialized value of x

Run initialization analysis Ex.1

```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
while (n != 0) {
    n = n - p;
}
```

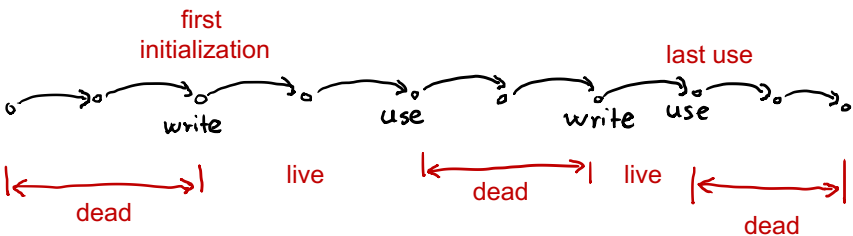


Run initialization analysis Ex.2

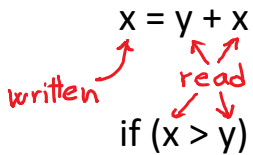
```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
if (p > 0) {
    n = n - p;
}
```

Liveness Analysis

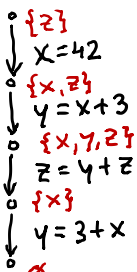
Variable is dead if its current value will not be used in the future.
If there are no uses before it is reassigned or the execution ends,
then the variable is surely dead at a given point.



What is Written and What Read



Example:



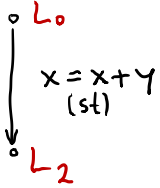
Purpose:

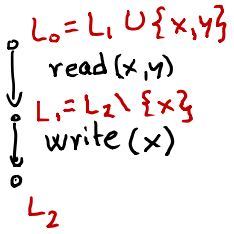
Register allocation:

find good way to decide which variable should go to which register at what point in time.

How Transfer Functions Look

L_0 - set of live variables





$$L_0 = (L_2 \setminus \{x\}) \cup \{x, y\}$$

Generally

$$L_0 = (L_2 \setminus def(st)) \cup use(st)$$

Initialization: Forward Analysis

while (there was change)
  pick edge (v1,statmt,v2) from CFG
        such that facts(v1) has changed
  facts(v2)=facts(v2) **join** transferFun(statmt, facts(v1))
}

Liveness: Backward Analysis

while (there was change)
  pick edge (v1,statmt,v2) from CFG
        such that facts(v2) has changed
  facts(v1)=facts(v1) **join** transferFun(statmt, facts(v2))
}

Example

x, y, xy, z, yz, xz, res1

∅

```
x = m[0]
```

{x}

```
y = m[1]
```

{x, y}

```
xy = x * y
```

{x, y, xy}

```
z = m[2]
```

{x, z, y, xy}

```
yz = y*z
```

{x, z, xy, yz}

```
xz = x*z
```

{xz, xy, yz}

```
res1 = xy + yz
```

{res1, xz}

```
m[3] = res1 + xz
```

∅

Memory and Its Management

Kinds of Memory in Compiled Programs

Program Data	Typical Machine Representation
intermediate values	registers, stack
local variables, parameters	registers, stack
return addresses of function calls	stack (+ 1 register)
global variables, constants	data segment, pre-allocated
compiled program (instructions)	code segment, pre-allocated
algebraic data type values, objects	dynamic heap
closures (first class functions)	dynamic heap

Pre-allocated memory has fixed size at compile time
Stack can grow, but must shrink in the LIFO way

Heap is most general: allocate and deallocate in any order

- if we never deallocate (as in the project), can use a stack separate from the stack for locals and returns
- but never deallocating leads to out-of-memory errors

Memory as Array

Traditionally, languages like C give full access to program memory

through pointers that can be manipulated (can even write to stack)

In C, the heap can be implemented as a library with **malloc** and **free**, using operating system calls to obtain pages of free memory, then using them as large arrays of bytes.

Memory as Array

Traditionally, languages like C give full access to program memory

through pointers that can be manipulated (can even write to stack)

In C, the heap can be implemented as a library with **malloc** and **free**, using operating system calls to obtain pages of free memory, then using them as large arrays of bytes.

```
typedef struct node {              // size 8 bytes
    int content;                   // offset 0
    struct node *next;             // offset 4
} node_t;
```

Memory as Array

Traditionally, languages like C give full access to program memory

through pointers that can be manipulated (can even write to stack)

In C, the heap can be implemented as a library with **malloc** and **free**, using operating system calls to obtain pages of free memory, then using them as large arrays of bytes.

```
typedef struct node {                // size 8 bytes
    int content;                     // offset 0
    struct node *next;               // offset 4
} node_t;

head = malloc(sizeof(node_t));       // head points to 8 bytes on heap
head -> content = 42;                // RAM[head] = 42
second = malloc(sizeof(node_t));     // second points to 8 more bytes
head -> next = second;               // RAM[head + 4] = second
```

Malloc and Free Using Free List

Need to know which memory is still used and which can be reused.

Allocation and deallocation happen in any order

$\Rightarrow$ memory array has interleaved regions of allocated and free memory.

Malloc and Free Using Free List

Need to know which memory is still used and which can be reused.
Allocation and deallocation happen in any order
$\Rightarrow$ memory array has interleaved regions of allocated and free memory.

Approach:

- allocated memory is responsibility of the program
- create a list of free blocks using only free memory!

Malloc and Free Using Free List

Need to know which memory is still used and which can be reused.
Allocation and deallocation happen in any order
$\Rightarrow$ memory array has interleaved regions of allocated and free memory.

Approach:

- allocated memory is responsibility of the program
- create a list of free blocks using only free memory!

Free/unused memory for the application is a linked list data structure for the allocator

- list elements are variable length: size stored in each block
- allocation: find sufficient block, split it, update free list, return pointer to split part
- deallocation: insert the block into list; possibly merge with adjacent free blocks

See also:

- Lectures of David August at This Link
- D. Knuth, The Art of Computer Programming, "Dynamic Storage Allocation"

Lack of Memory Safety

Using pointers is flexible and easy to compile:

Emit memory access instructions and library calls to malloc and free.

Lack of Memory Safety

Using pointers is flexible and easy to compile:

Emit memory access instructions and library calls to malloc and free.

But it is not memory-safe!

Lack of Memory Safety

Using pointers is flexible and easy to compile:

Emit memory access instructions and library calls to malloc and free.

But it is not memory-safe!

```
long* x = malloc(...);
*x = 9876543;
free(x);
// x is now dangling pointer
long* y = malloc(...);
*y = 1234567;
// y might use part of same memory as x
*x = 0;
// now *y may be changed and even corrupted
```

Lack of Memory Safety

Using pointers is flexible and easy to compile:

Emit memory access instructions and library calls to malloc and free.

But it is not memory-safe!

```
long* x = malloc(...);
*x = 9876543;
free(x);
// x is now dangling pointer
long* y = malloc(...);
*y = 1234567;
// y might use part of same memory as x
*x = 0;
// now *y may be changed and even corrupted
```

To ensure memory safety: cannot allow developer to use 'free' arbitrarily

- use either *automated memory management* or *type-checked memory management*

Automated Memory Management

Reference counting: maintain field in each heap object that counts how many references to this object exist.

```
x.f = y
```

becomes:

```
x.f.count--;
if (x.f.count == 0) deallocate(x.f);
y.count++;
x.f = y
```

Deallocation also decrements references and can recursively deallocate other objects.

Automated Memory Management

Reference counting: maintain field in each heap object that counts how many references to this object exist.

```
x.f = y
```

becomes:

```
x.f.count--;
if (x.f.count == 0) deallocate(x.f);
y.count++;
x.f = y
```

Deallocation also decrements references and can recursively deallocate other objects.

Works as long as ***there are no cycles!***

Automated Memory Management

Reference counting: maintain field in each heap object that counts how many references to this object exist.

```
x.f = y
```

becomes:

```
x.f.count--;
if (x.f.count == 0) deallocate(x.f);
y.count++;
x.f = y
```

Deallocation also decrements references and can recursively deallocate other objects.

Works as long as ***there are no cycles!*** Not very efficient (high overhead)

Automated Memory Management

Reference counting: maintain field in each heap object that counts how many references to this object exist.

```
x.f = y
```

becomes:

```
x.f.count--;
if (x.f.count == 0) deallocate(x.f);
y.count++;
x.f = y
```

Deallocation also decrements references and can recursively deallocate other objects.

Works as long as ***there are no cycles!*** Not very efficient (high overhead)

See: automatic Reference Counting in Swift;
compile time reference counting in Rust: Ownership, References and Borrowing

Garbage Collection

To automatically collect unused, possibly-cyclic data structures

Convenient for functional programming with data sharing
introduced in LISP around 1960!

Garbage Collection

To automatically collect unused, possibly-cyclic data structures

Convenient for functional programming with data sharing
introduced in LISP around 1960!

Periodically mark all *reachable* objects as *used*; free up the rest as garbage

reachable: from global variables and local variables on the stack

Garbage Collection

To automatically collect unused, possibly-cyclic data structures

Convenient for functional programming with data sharing
introduced in LISP around 1960!

Periodically mark all *reachable* objects as *used*; free up the rest as garbage

reachable: from global variables and local variables on the stack

Two main types of garbage collection algorithms:

- **Mark and Sweep:** mark all reachable objects and put them in a free list
  (good if there is little garbage, but suffers from fragmentation)
- **Copying Collector:** use twice the space; after marking, copy all useful data into a separate region, compacting memory (put blocks next to each other)

Garbage Collection

To automatically collect unused, possibly-cyclic data structures

Convenient for functional programming with data sharing
introduced in LISP around 1960!

Periodically mark all *reachable* objects as *used*; free up the rest as garbage

reachable: from global variables and local variables on the stack

Two main types of garbage collection algorithms:

- **Mark and Sweep:** mark all reachable objects and put them in a free list
  (good if there is little garbage, but suffers from fragmentation)
- **Copying Collector:** use twice the space; after marking, copy all useful data into a separate region, compacting memory (put blocks next to each other)

Generational collector: organize objects by generations; collect newly allocated objects more often; if they survive multiple collections, promote them to older gen.
Typically used in Java: generational parallel and concurrent copying collector

Compiler/Runtime Support for Garbage Collection

Garbage collector needs to know:

- how to find roots in global variables, stack, registers (or ensure references are never only in registers)
- how to follow (non-weak) references through objects

Compiler/Runtime Support for Garbage Collection

Garbage collector needs to know:

- how to find roots in global variables, stack, registers
  (or ensure references are never only in registers)
- how to follow (non-weak) references through objects

For this, some amount of run-time type information is needed.

Generational GC may need to traverse all older generations

to know what is alive in new generation

To speed this up, GC can use information that ensures certain groups of objects do not point to newer generation.

To maintain this information, compiler may instrument writes to object fields
(similar to reference counting)

Dynamic Dispatch

Dynamic dispatch is key to object-oriented *and* functional languages.

$\Rightarrow$ Used to implement method calls and higher-order functions.

Dynamic Dispatch

Dynamic dispatch is key to object-oriented *and* functional languages.

$\Rightarrow$ Used to implement method calls and higher-order functions.

```
class Animal:
  def noise = "squeak!"
  def muchNoise = noise + noise

class Dog extends Animal:
  override def noise = "aw!"

val d: Animal = new Dog
d.muchNoise // == "aw!aw!"
```

Dynamic Dispatch

Dynamic dispatch is key to object-oriented *and* functional languages.

⇒ Used to implement method calls and higher-order functions.

```
class Animal:
  def noise = "squeak!"
  def muchNoise = noise + noise

class Dog extends Animal:
  override def noise = "aw!"

val d: Animal = new Dog
d.muchNoise // == "aw!aw!"
```

Compilation of `muchNoise` cannot make direct call to virtual method `noise`

must invoke whatever method is defined by dynamic type of object

⇝ virtual method table

Dynamic Dispatch Implementation

```
struct Animal { val vtable : Array[FunPtr] }
def Animal_noise(this: Animal) = return "squeak!"
def Animal_muchNoise(this: Animal) =
  this.vtable(0)(this) + this.vtable(0)(this)

struct Dog { val vtable: Array[FunPtr] }
def Dog_noise(this: Dog) = return "aw!"

val Animal_vtable = Array(Animal_noise, Animal_muchNoise)
val Dog_vtable = Array(Dog_noise, Animal_muchNoise)

val d = malloc(Dog)
d.vtable = Dog_vtable
d.vtable(1).apply(d) // 1 is the index of muchNoise
```

Virtual methods calls have one extra indirection (even more for multiple inheritance)

First-Class Functions as Objects: Captured Variables

```
def f(x: Int) =
   ...
   ((y: Int) => x + y) // Closure_1

f(42)(20)
```

First-Class Functions as Objects: Captured Variables

```
def f(x: Int) =
  ...
  ((y: Int) => x + y) // Closure_1

f(42)(20)
```

becomes:

```
???
```

```
def f(x: Int) =
  ...
  ???

f(42)???
```

given:

```
abstract class Function[-A, +B] { def apply(x:A): B }
```

First-Class Functions as Objects: Captured Variables

```
def f(x: Int) =
  ...
  ((y: Int) => x + y) // Closure_1

f(42)(20)
```

becomes:

```
class Closure_1(x:Int) extends Function[Int, Int]:
  def apply(y: Int): Int = x + y

def f(x: Int) =
  ...
  new Closure_1(x)

f(42).apply(20)
```

given:

```
abstract class Function[-A, +B] { def apply(x:A): B }
```

Captured Variables

```
val f = // Block_2
   var x = 42
   (y: Int) => { x += 1; x + y } // Closure_2

f(1) + f(1) // computes: 43 + 44
```

becomes:

Captured Variables

```
val f = // Block_2
   var x = 42
   (y: Int) => { x += 1; x + y } // Closure_2

f(1) + f(1) // computes: 43 + 44
```

becomes:

```
class Block_2_Vars { var x: Int = _ }
class Closure_2(block: Block_2_Vars) extends Function[Int, Int]:
   def apply(y: Int): Int = { block.x += 1; block.x + y }
val f =
   val block2 = new Block_2_Vars
   block2.x = 42
   new Closure_2(block2)

f.apply(1) + f.apply(1)
```

Code Specialization (Scala.js, GHC Haskell, Rust, C++, ...)

By *partially evaluating* program at compile time,

we can specialize its parts and generate more efficient code.

Code Specialization (Scala.js, GHC Haskell, Rust, C++, ...)

By *partially evaluating* program at compile time,

we can specialize its parts and generate more efficient code.

Such transformation can be done automatically or under user control

using, for example, *staged computation*, *macros*, *templates*...

```
def fold(l: List[A], b: B, f : (A,B) => B): B = l match
   case Nil => b
   case x :: xs => f(x, fold(xs, b, f))
fold(l, 0, _ + _)
```

```
def foldZeroPlus(l: List[A]): B = l match
  case Nil => 0
  case x :: xs => x + foldZeroPlus(xs) // no closure call
foldZeroPlus(l)
```

Algebraic Transformations (Used in Glasgow Haskell Compiler)

Higher-order combinators such as `map` satisfy many laws

which can be used for optimization, including parallel execution.

Typically, these laws hold only when functions are pure

```
list.map(f).map(g) == list.map(x => g(f(x)))
```

Algebraic Transformations (Used in Glasgow Haskell Compiler)

Higher-order combinators such as `map` satisfy many laws

which can be used for optimization, including parallel execution.

Typically, these laws hold only when functions are pure

```
list.map(f).map(g) == list.map(x => g(f(x)))
```

Type systems and program analyses for purity are active areas of research.

If a language has mutable objects and allows their sharing (aliasing),

it is particularly difficult to prove that a function behaves as pure;

requires reasoning about possible heap configurations

(see: **shape analysis**, **alias analysis**).

The End

The End

No more lectures and tutorials!

The End

No more lectures and tutorials!

Next Project: Custom Compiler Extension (**Optional**)

By **teams** of 2 to 3
Choose a cool way of extending the Amy language & compiler
Grading scheme: **bonus points**; can make up for half of previous lab grades

The End

No more lectures and tutorials!

Next Project: Custom Compiler Extension (**Optional**)

By **teams** of 2 to 3
Choose a cool way of extending the Amy language & compiler
Grading scheme: **bonus points**; can make up for half of previous lab grades

Final Exam

Date/time: 13-Dec-2021, 04:30PM - 07:30PM
Location: LG5 Multi-function Room
Will give you additional exercise material to prepare

The End

No more lectures and tutorials!

Next Project: Custom Compiler Extension (**Optional**)

By **teams** of 2 to 3
Choose a cool way of extending the Amy language & compiler
Grading scheme: **bonus points**; can make up for half of previous lab grades

Final Exam

Date/time: 13-Dec-2021, 04:30PM - 07:30PM
Location: LG5 Multi-function Room
Will give you additional exercise material to prepare

Give Your Feedback

So I can **improve** the course in future years
Please be gentle! It was my first time giving it

Hope you enjoyed!