Constant Propagation Transfer Functions

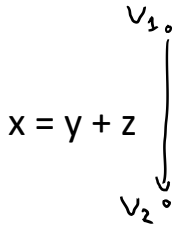


table for +:

?	⊥	C_y	⊤
⊥	⊥	⊥	⊥
C_z	⊥	C_y+C_z	⊤
⊤	⊥	⊤	⊤

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

```
abstract class Element
case class Top extends Element
case class Bot extends Element
case class Const(v:Int) extends Element
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
}
```

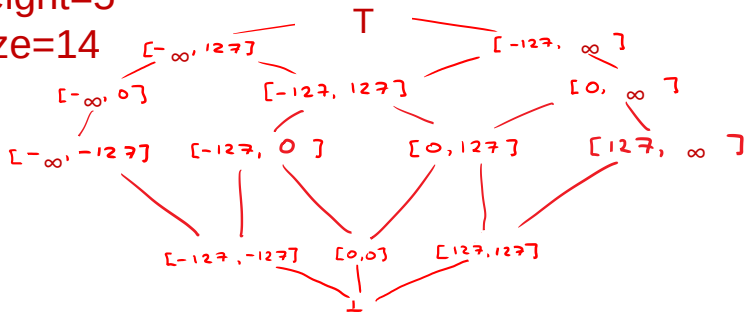
```
x = 1
n = readInt()
while (x < n) {
  x = x + 2
}
```

Observe

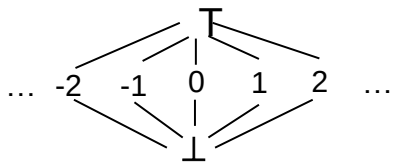
- Range analysis with end points
  W = {-128, 0, 127} has a finite domain
- Constant propagation has infinite domain
  (for every integer constant, one element)
- Yet, constant propagation finishes sooner!
  - it is not about the size of the domain
  - it is about the height

Height of Lattice: Length of Max. Chain

height=5
size=14



height=2
size =∞

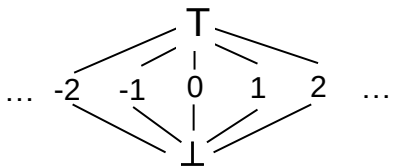


Chain of Length n

- A set of elements $x_0, x_1, \ldots, x_n$ in D that are linearly ordered, that is $x_0 < x_1 < \ldots < x_n$
- 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]]**

Exercise

$\mathbf{B_{32}}$ – the set of all 32-bit integers

What is the upper bound for number of changes in the entire analysis for:

- 3 variables,
- 7 program points

for these two analyses:

1) constant propagation for constants from $\mathbf{B}_{32}$

2) The following domain D:

$$D = \{\bot\} \cup \{ [a,b] \mid a,b \in \mathbf{B_{32}}\ , a \leq b\}$$

Height of B_{32}

$D = \{\bot\} \cup \{ [a,b] \mid a,b \in \mathbf{B_{32}}, a \leq b\}$

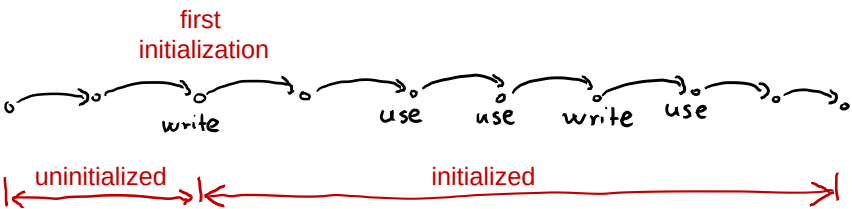
One possible chain of maximal length:

$\bot$

…

[MinInt,MaxInt]

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?

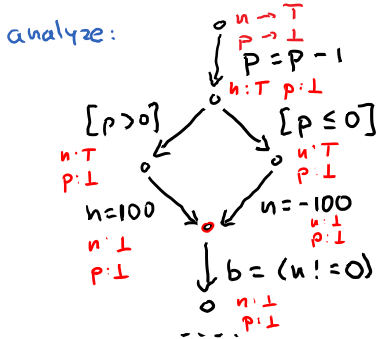
```
static void test(int p) {
        int n;
        p = p - 1;
        if (p > 0) {
           n = 100;
        }
        System.out.println("Hello!");
        if (p > 0) {
           while (n != 0) {
              System.out.println(n);
              n = n - p;
           }
        }
}
```

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): ⊥



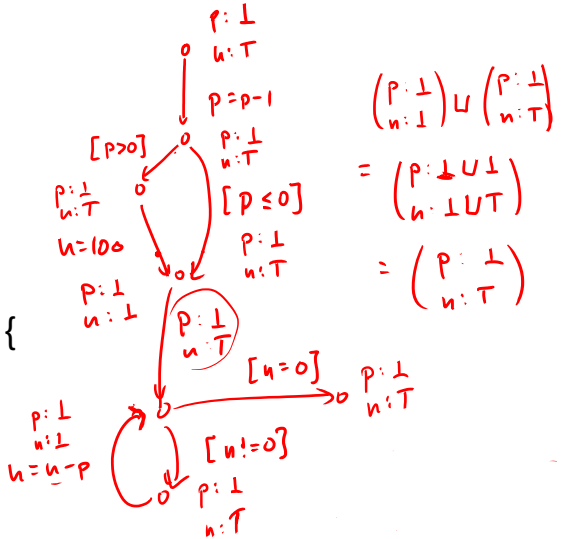
If var occurs anywhere but left-hand side of assignment and has value T, report error

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;
}
```

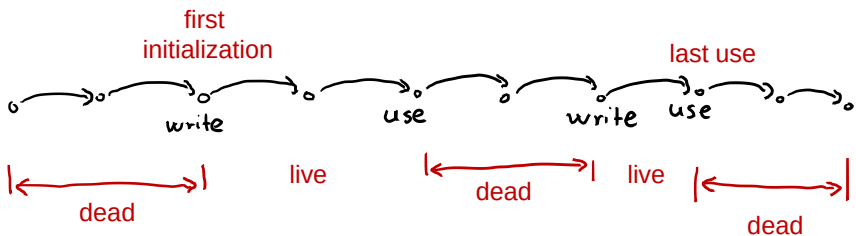


Run initialization analysis Ex.2

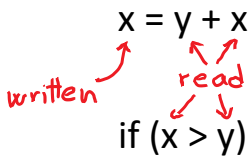
```
int n;
p = p - 1;
if (p > 0) {
    n = 100;
}
if (p > 0) {
    n = n - p;
}
```

Liveness Analysis

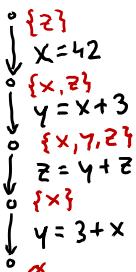
Variable is dead if its current value will not be used in the future.
If there are no uses before it is reassigned or the execution ends, then the variable is surely dead at a given point.



What is Written and What Read



Example:



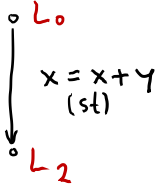
Purpose:

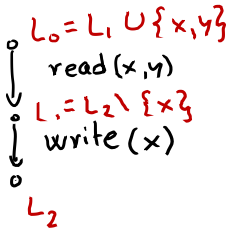
Register allocation:

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
```

∅

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	data segment, pre-allocated
algebraic data type values	dynamic heap
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 de-allocate (as in the project), can use a stack separate from the stack for locals and returns
- but never de-allocating leads to out-of-memory errors

Memory as Array

Languages like C traditionally 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**, and that uses operating system calls to obtain large blocks of available memory, then treats 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 = 8 bytes on heap
head -> content = 42;                     // RAM[head] = 42
second = malloc(sizeof(node_t));          // second = get 8bytes
head -> next = second;                    // RAM[head + 4] = second
```

Malloc and Free Using Free List

Need to know which memory is used and which is fresh.
Because allocation and de-allocation is in any order, 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!

What is free and 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 a sufficient block, split it, update the free list, return the split of part
- deallocation inserts the block into list, if possible merge with adjacent blocks

See also:

- Lectures of David August at This Link
- D. Knuth, The Art of Computer Programming, Vol. 1, "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.

- 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

- we want automated memory management

Automated Memory Management

Reference counting: maintain a 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. This works as long as there are no cycles.
See: Automatic Reference Counting in Swift

Forms of compile time reference counting in Rust: Ownership, References and Borrowing

Garbage Collection

To automatically collect cyclic data structures and convenient functional programming with sharing data we use garbage collection (already introduced in LISP).
Periodically mark all objects reachable from global and local variables of all stack frames, free up the rest as garbage

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 and 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 generation.
Typically used in Java: generational parallel and concurrent copying collector

Compiler 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 that certaing groups of objects do not point to newer generation. To maintain that information, compiler may need to instrument all writes of object fields, with worst-case overhead similar to that of reference counting.

Dynamic Dispatch

Dynamic dispatch is key to object-oriented languages.
It can also be used to implement higher-order functions.

```
class Animal {
  def noise = ”squeak!”
  def muchNoise = noise + noise
}
class Dog extends Animal {
  override def noise = ”aw!”
}
d = new Dog
d.muchNoise

res0: String = aw!aw!
```

Compilation of `muchNoise` cannot make a direct call to method that returns "squeak!" but must invoke whatever method is most specific to the dynamic type of the object given by new declaration.
$\rightsquigarrow$ virtual method table

Dynamic Dispatch Implementation

```
type Animal = struct { vtable : FunPtrs[] }
def Animal_noise(this:Animal) = return "squeak!"
def Animal_muchNoise(this:Animal) =
  (this -> vtable)[0](this) +
  (this -> vtable)[0](this)

type Dog = struct { vtable : FunPtrs[] }

def Dog_noise(this:Dog) = return "aw!"

Animal_vtable[] = { Animal_noise, Animal_muchNoise }
Dog_vtable[] = { Dog_noise, Animal_muchNoise }

d = malloc(Dog)
d -> vtable = Dog_vtable
(d -> vtable)[1](d) // 1 is the index of muchNoise
```

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

First-Class Functions as Objects: Capturing Vals

```
val f = {
  val x = 42
  ((y:Int) => x + y) // Closure_1
}
f(20)
```

becomes:

```
abstract class Function[A-,B+] {
  def apply(x:A): B
}
class Closure_1(x:Int) extends Function {
  def apply(y: Int): Int = x + y
}
val f = {
   val x = 42
   new Closure_1(x)
}
f.apply(20)
```

Capturing Vars

```
val f = { // Block_2
  var x = 42
  ((y:Int) => x + y; x++) // Closure_2
}
f(20) + f(0)
```

becomes:

```
class Block_2_Vars { var x: Int = _ }
class Closure_2(block: Block_2_Vars) extends Function {
  def apply(y: Int): Int = { block.x + y; block.x++ }
}
val f = {
   val block2 = new Block_2_Vars
   block2.x = 42
   new Closure_2(block2)
}
f.apply(20) + f.apply(0)
```

Code Specialization (Used in Scala.js)

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
}
foldZeroPlus(l)
```

Algebraic Transformations (Used in Glasgow Haskell Compiler)

Higher-order combinators such as map satisfy many laws that 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 an active areas of research.

If a language has mutable objects and allows their sharing, it is particularly difficult to prove that a function behaves as pure: knowing if a modification is to auxiliary objects or externally observable objects requires reasoning about possible heap configurations (**shape analysis**, **alias analysis**).