Constant Propagation

Class notes
Program Analysis course given by Prof. Mooly Sagiv
Computer Science Department, Tel Aviv University
second lecture 8/3/2007

Osnat Minz and Mati Shomrat

Introduction

This lecture focuses on the Constant Propagation Analysis. This is just one of many types of analyses that can be applied to programs. We use this example as an incentive for the introduction of the mathematical foundation that will serve us throughout the course. We focus on partially ordered sets, lattices and their relation to monotone functions. We later use those to formalize our analysis technique and prove its correctness.

We start by introducing the Constant Propagation problem. The aim of our analysis is to determine for each program point, whether a variable has a constant value whenever the execution reaches that point. Furthermore, if a variable is constant at some point we would like to know its value.

Information about constants can be used, for example, in the process of optimization, where all uses of a variable may be replaced by the constant value.

Informal Example

We describe an informal algorithm for the computation of constants in order to give the reader some intuition as to how the algorithm works before delving into the mathematical world.

Before describing the algorithm let us define two values:

Top denoted $\top$, will serve us to indicate that a variable is potentially non-constant and

Bottom denoted $\bot$, is the most accurate value that can be assigned. It captures the case where the set of represented states is empty.

```
1. z := 3
2. x := 1
3. while (x > 0) do
4.    if (x = 1)
5.       y := 7
6.    else
7.       y := z + 4
8.    x := 3
9.    print y
```

Fig. 1: A simple example program

Consider the simple C like program shown in figure 1. We can make several observations about the program:

- In line 2, z has the value 3
- In line 5, x has the value 1
- In lines 8,9, y has the value 7, thus the print statement can be replaced by `print 7`
- In line 4, x either has the value 1 or 3, hence it is not a constant.

Let us now describe an informal algorithm for the Constant Propagation problem. The algorithm simply follows the program's flow. At each statement we remember the environment, a mapping between the program's variables to their values in $\mathbb{Z} \cup \{\bot, \top\}$. It is an iterative algorithm in the sense that if the analyzed program contains loops it will follow them. The algorithm stops once no more changes are detected.

For the program in Figure 1 the steps of the algorithm are as follow:

1. Initialization: assign initial values to the program's variables $[x \mapsto 0, y \mapsto 0, z \mapsto 0]$
2. `z = 3`, change the value of z $[x \mapsto 0, y \mapsto 0, z \mapsto 3]$
3. `x = 1`, change the value of x $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$
4. `while (x >0)`, at this point we have $x \mapsto 1$ so we have no choice but t enter the loop.
5. `if (x = 1)`, indeed this is the case, so the algorithm only follows the `then` branch.

6. `y = 7`, change the value of x $[x \mapsto 1, y \mapsto 7, z \mapsto 3]$

7. `x = 3`, change the value of x $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$

8. `print y`, at this point we currently have $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$. We have now completed our first iteration of the loop and we return to the `while` statement.

9. `while (x >0)`, from our initial iteration we have at this program point $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$, but now we have $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$. The state at this point should have been $[x \mapsto \{1, 3\}, y \mapsto \{0, 7\}, z \mapsto 3]$. Since we represent states in a conservative way we get $[x \mapsto \top, y \mapsto \top, z \mapsto 3]$

   Again we enter the loop, but now since `x` is non-constant we will execute both branches of the if statement.

10. First, we follow the `then` branch. We note that on that path `x` is 1, we get $[x \mapsto 1, y \mapsto 7, z \mapsto 3]$

11. Continuing down that path `x = 3` and by the time we reach the print statement we get $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$

12. We go back to the `else` branch were we have $[x \mapsto \top, y \mapsto 7, z \mapsto 3]$

13. Following that path as well, by the time we reach the print statement we have $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$

14. Termination, since both paths lead to the same environment in the print statement, and since this environment is the same as the one calculated on our previous journey through that point in the program the algorithm terminates.

In the remainder of this document we will formulate the above algorithm, giving it firm mathematical foundation and then prove its correctness.

A (more) Formal Example

For a program S the Constant Propagation algorithm contains the following stages:

1. Construct a control flow graph (CFG) of the program S.

2. Associate transfer functions with the edges of the CFG. The transfer function of an edge reflects the semantics of the atomic statements at its source node.

3. At every node (program point) we maintain the values of the program's variables at that point. We initialize those to $\perp$.

4. Iterate until the values of the variables stabilize.

Later we will show that the algorithm always terminate and that no matter how we chose to traverse the CFG the outcome of the algorithm will be unique. Note, however, that while the order of traversal does not affect the correctness of the algorithm it will influence its cost, that is the number of iterations.

Control Flow Graph

A control flow graph (CFG) is a graph representation of all execution paths that might occur in a program. Each node in the graph represents an atomic statement; directed edges are used to represent transfers in the control flow. The graph is finite and its size is proportional to the program size. The graph can be built by a compiler. There might be paths in the CFG that are unreachable at run time.

Figure 2 shows the CFG for our working example.

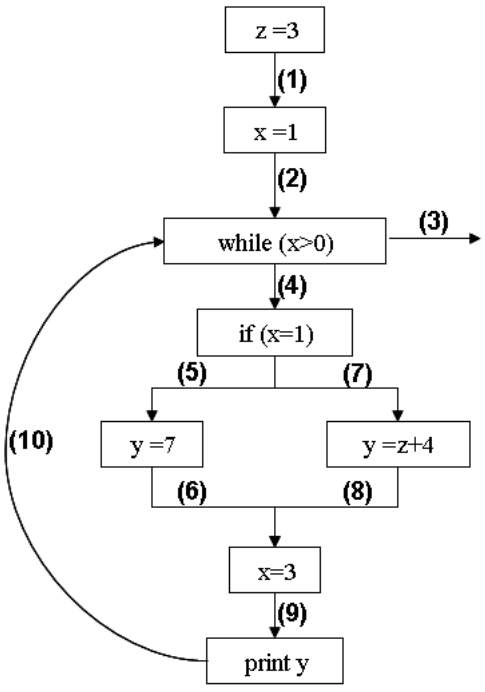

Fig. 2: Control Flow Graph of the example program

Transfer Functions

At every node in the graph we store the known environment. We assign a *transfer function* to each edge in the control flow graph. The transfer function is a mapping from one environment to the other that corresponds to the semantics of its source node.

We will use lambda expressions to describe the transfer functions. For example, $\lambda e.e$, the identity function, corresponds to the `skip` statement. $\lambda e.[x \mapsto 2]$ is the function corresponding to the assignment statement `x = 2`. The function modifies only the value of the variable `x` in the environment.

The binary function *meet* ($\sqcap$) and *join* ($\sqcup$) are two elements in the set of transfer functions. The meet function can be thought of as taking the intersection of two mappings, and the join function as taking the union. Table 1 shows the output of the two functions when applied to values from $\mathbb{Z} \cup \{\bot, \top\}$.

$\bigsqcap$	$\top$	$n \in \mathbb{N}$	$m \neq n$	$\bot$
$\top$	$\top$	n	m	$\bot$
$n \in \mathbb{Z}$	n	n	$\bot$	$\bot$
$m \neq n$	m	$\bot$	m	$\bot$
$m = n$	m	$\bot$	m	$\bot$
$\bot$	$\bot$	$\bot$	$\bot$	$\bot$

(a)

$\bigsqcup$	$\top$	$n \in \mathbb{N}$	$m \neq n$	$\bot$
$\top$	$\top$	$\top$	$\top$	$\top$
$n \in \mathbb{Z}$	$\top$	n	$\top$	n
$m \neq n$	$\top$	$\top$	m	m
$m = n$	m	$\bot$	m	$\bot$
$\bot$	$\top$	$\top$	m	$\bot$

(b)

Tab. 1: Meet and Join functions

In Figure 2 a label is assigned to each of the graph's edges. We assign a transfer function to each such edge:

(1) $\lambda e.e[z \mapsto 3]$
(2) $\lambda e.e[x \mapsto 1]$
(3) $\lambda e.$ if $e(x) \leq 0$ then e else $\bot$
(4) $\lambda e.$ if $e(x) > 0$ then e else $\bot$
(5) $\lambda e.e \sqcap [x \mapsto 1, y \mapsto \top, z \mapsto \top]$
(6) $\lambda e.e[y \mapsto 7]$
(7) $\lambda e.$ if $e(x) \neq 1$ then e else $\bot$
(8) $\lambda e.e[y \mapsto e(z) + 4]$
(9) $\lambda e.e[x \mapsto 3]$
(10) $\lambda e.e$

The revised algorithm

The algorithm then progresses in a similar way to the informal algorithm we described earlier.

- **Initialization.** The environment $e_0 = [x \mapsto 0, y \mapsto 0, z \mapsto 0]$ is assigned to the starting node v_0. We initialize the environments in all other nodes to be $\bot$.

- **Graph traversal.** The algorithm traverses the graph in a depth first search (DFS) manner. As it advances from node v to u it joins the environment in u with the new environment obtained by applying the transfer function associated with the edge (v, u) to the environment in v.

- **Termination.** Once the iterative application of the transfer functions yields no more changes to the environments the algorithm terminates.

Notice, that while for this more formal example we specified the traversal scheme to be DFS this is not a requirement of the algorithm. Different traversal scheme may be applied. The outcome is unique and does not depend on the order of traversal. However, different traversal schemes may converge in different rates, affecting the efficiency of the algorithm.

In the above examples we provided an informal and intuitive description of an algorithm that compute the constants of a program. Before continuing to further formalize our algorithm and to prove its correctness we now establish mathematical foundation on which we can build.

Mathematical Foundation

Partially ordered sets and complete lattices play a crucial role in program analysis, we shall summarize some of their properties. We review the basic approaches for

to construct complete lattices from other complete lattices and state the central properties of partially ordered sets satisfying the ascending and descending chain condition. We then review the classical results about least and greatest fix points.

Partially Ordered Sets (Poset)

A partial ordering is a binary relation $\sqsubseteq$: $L \times L \rightarrow \{\texttt{True}, \texttt{False}\}$ that is:

- **Reflexive** $\forall l \in L : l \sqsubseteq l$
- **Transitive** $\forall l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_3 \Rightarrow l_1 \sqsubseteq l_3$
- **Anti-symmetric** $\forall l_1, l_2 \in L : l_1 \sqsubseteq l_2 \wedge l_2 \sqsubseteq l_1 \Rightarrow l_1 = l_2$

A *partially ordered set* (or poset) is a set equipped with a partial ordering relation. This relation formalizes the intuitive concept of an ordering, sequencing, or arrangement of the set's elements.

For the sake of simplicity we will use the following notation in the remainder of the document:

- $l_1 \sqsubseteq l_2 \Leftrightarrow l_2 \sqsupseteq l_1$
- $l_1 \sqsubset l_2 \Leftrightarrow l_1 \sqsubseteq l_2 \wedge l_1 \neq l_2$
- $l_1 \sqsubset l_2 \Leftrightarrow l_2 \sqsupset l_1$

A few examples of partially ordered sets are:

- The set of natural numbers equipped with the lesser than relation $(\mathbb{N}, \leq)$
- Power sets - set of subsets of a given set S along with the subset or superset relations. $(P(S), \subseteq) or (P(S), \supseteq)$
- Program environment - We say that one program environment is smaller than another environment if it is smaller on all program variables. For example: $[x \mapsto \bot, y \mapsto \bot] \sqsubseteq [x \mapsto 5, y \mapsto \bot] \sqsubseteq [x \mapsto 5, y \mapsto 7] \sqsubseteq [x \mapsto \top, y \mapsto \top]$

We draw partially ordered sets as *Hasse diagrams*. Hasse diagrams are pictures of a finite partially ordered set forming a drawing of the transitive reduction of the partial order. The Hasse diagram of a poset (X, R) is the directed graph whose vertex set is X and whose arcs are the covering pairs (x, y) in the poset. We usually draw the Hasse diagram of a finite poset in the plane in such a way that, if y covers x, then the point representing y is higher than the point representing x. No arrows are required in the drawing, since the directions of the arrows

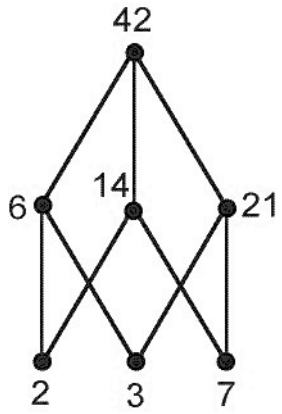

Fig. 3: Hasse diagram for the divisors of 42

are implicit. Figure 3 shows the Hasse diagram for the divisors of 42, using "divisibility" as partial ordering.

Upper and Lower Bounds. An upper bound of a subset S of some partially ordered set $(P, \sqsubseteq)$ is an element of P which is greater than or equal to every element of S. The term lower bound is defined dually as an element of P which is lesser than or equal to every element of S.

More formally, if $(P, \sqsubseteq)$ is a poset than the following hold:

- $l \in P$ is a lower bound of a subset $S \subseteq P$ if for all $s \in S : l \sqsubseteq s$
- $u \in P$ is an upper bound of a subset $S \subseteq P$ if for all $s \in S : u \sqsupseteq s$
- $l_0 \in P$ is a greatest lower bounds of a subset $S \subseteq P$ if l_0 is a lower bound of S and for all lower bounds l of S $l_0 \sqsupseteq l$.
- $u_0 \in P$ is a least upper bounds of a subset $S \subseteq P$ if u_0 is an upper bound of S and for all upper bounds u of S $u_0 \sqsubseteq u$.

Note that subset S of a partially ordered set P need not have a least upper bounds nor a greatest lower bounds, but when they exist they are unique (follows from the anti-symmetry of $\sqsubseteq$). We denote $\bigsqcap P$ (meet) and $\bigsqcup P$ (join) as the greatest lower bounds and least upper bounds respectively.

As an example notice that for every subset of the natural numbers zero serves as a lower bound. Every finite subset of the natural number also has a greatest upper bounds.

Complete Lattices

A *complete lattice* is a partially ordered set $(L, \sqsubseteq) = (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$ where all subsets $L' \subseteq L$ has a least upper bounds and a greatest lower bounds. We denote

$\bot$ to be the least element, $\bot = \bigsqcup \emptyset = \bigsqcap L$, and $\top$ to be the greatest element, $\top = \bigsqcap \emptyset = \bigsqcup L$.

As an example of a complete lattice consider the powerset $\mathcal{P}(S)$ of some set S. If we take $\sqsubseteq$ to be $\subset$ then $\bigsqcup S = \bigcup S, \bigsqcap L' = \bigcap S, \bot = \emptyset$ and $\top = S$. If we take $\sqsubseteq$ to be $\supseteq$ then $\bigsqcup S = \bigcap S, \bigsqcap S = \bigcup S, \bot = S$ and $\top = \emptyset$. In case where $S = \{1, 2, 3\}$ figure 4 shows the two complete lattices.

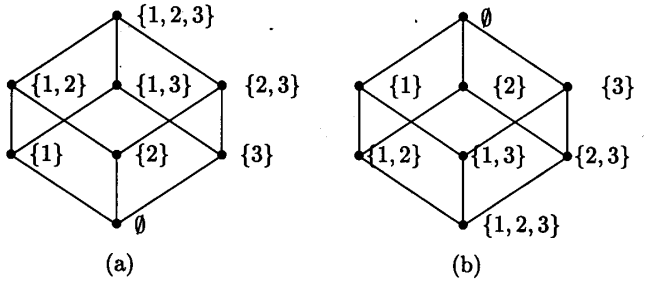

Fig. 4: Two complete lattices

We can turn $(\mathbb{N}, \leq)$ into a complete lattice by adding an artificial greatest element ∞.

Lemma 1. *For every partially ordered set $\mathcal{L} = (L, \sqsubseteq)$ the following conditions are equivalent:*

(i) L is a complete lattice.

(ii) Every subset of L has a least upper bounds.

(iii) Every subset of L has a greatest lower bounds.

Proof. From the definition of complete latticea we have that *(i)* implies *(ii)* and *(iii)*. To show that *(ii)* implies *(iii)* we construct the meet operation using the join operation. This suffice, because condition *(ii)* means that the join operation exists, and condition *(iii)* is implied by the existence of the meet operation.

The construction of $\bigsqcap$ is as follows: Let $L' \subseteq L$, we denote $\widetilde{\bigsqcap}$ our candidate $\bigsqcap$ and define

$$\widetilde{\bigsqcap} L' = \bigsqcup \{l \in L \mid \forall l' \in L' : l \sqsubseteq l'\}$$

.

We need to show that $\widetilde{\bigsqcap} L'$ is indeed the greatest lower bound of L'. We note that all elements on the right hand side of the definition are lower bounds of L', the join of all of them must also be a lower bound of L'. Since any lower bound of L' will be in the set it follows that $\widetilde{\bigsqcap} L' = \bigsqcap L'$.

To prove that *(iii)* implies *(ii)* we follow analogous arguments only this time we are constructing the join from the assumed meet. □

Construction of Complete Lattices

Complete lattice can be combine to construct other lattices.

Cartesian Product. Let $(L_1, \sqsubseteq_1) = (L_1, \sqsubseteq_1, \bigsqcup_1 \bigsqcap_1, \bot_1, \top_1)$ be a complete lattice and let $(L_2, \sqsubseteq_2) = (L_2, \sqsubseteq_2, \bigsqcup_2 \bigsqcap_2, \bot_2, \top_2)$ be another complete lattice. We define

$$L = \{(l_1, l_2) \mid l_1 \in L_1 \wedge l_2 \in l_2\}$$

and

$$(l_{11}, l_{2,1}) \sqsubseteq (l_{1,2}, l_{2,2}) \text{ iff } l_{1,1} \sqsubseteq_1 l_{2,1} \wedge l_{1,2} \sqsubseteq_1 l_{2,2}$$

We write $L = L_1 \times L_2$ and call it the *cartesian product* of L_1 and L_2. It is straightforward to verify that L itself is a complete lattice. We can generalize the construction to construct a lattice $L = L_1 \times L_2 \times \ldots \times L_n$.

Example. The Cartesian product of the thirteen-element set of standard playing card ranks $\{Ace, King, Queen, Jack, 10, 9, 8, 7, 6, 5, 4, 3, 2\}$ and the four- element set of card suits $\{\clubsuit, \diamondsuit, \heartsuit, \spadesuit\}$ is the 52-element set of playing cards $\{(Ace, \clubsuit), \ldots, (2, \clubsuit), \ldots, (Ace, \spadesuit), \ldots, (2, \spadesuit)\}$. The Cartesian product has 52 elements because that is the product of 13 times 4.

Finite Maps. Let $(L_1, \sqsubseteq_1) = (L_1, \sqsubseteq_1, \bigsqcup, \bigsqcap, \bot, \top)$, be a complete lattice and V be finite set. We define a poset $L = (f : V \to L_1, \sqsubseteq)$ where:

$$f_1 \sqsubseteq f_2 \text{ if for all } v \in V, f_1(v) \sqsubseteq_1 f_2(v)$$

It is straightforward to verify that L is a complete lattice.

Using combination of lattices will enable us to handle programs with multiple variables, which is the the normal case for programs and otherwise we would only be able to deal with unrealistic toy programs.

Monotone Functions. Let $(L, \sqsubseteq)$ be a poset. We say $f : L \to L$ is *monotone* (also called order preserving) if for every two elements $l_1, l_2 \in L$, $l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$

Lemma 2. *Given a lattice L then $f : L \to L$ is monotone iff*

$$\forall X \subseteq L : \bigsqcup\{f(z) \mid z \in X\} \sqsubseteq f(\bigsqcup\{z \mid z \in X)\}$$

Proof. Let f be monotone. From the definition of join we have that for all $z \in X$, $z \sqsubseteq \bigsqcup\{x \mid x \in X\}$. Applying f and using its monotonicity we get $f(z) \sqsubseteq f(\bigsqcup\{x \mid x \in X\})$. Hence, $f(\bigsqcup\{x \mid x \in X\})$ is an upper bound of $f(z)$, and from the definition of join as a least upper bound we get $\bigsqcup\{f(z) \mid z \in X\} \sqsubseteq f(\bigsqcup\{x \mid x \in X\})$. We leave it as an exercise for the reader to prove the other direction. □

Intuitively, this lemma means that when working locally, traversing each path on its own rather than considering all paths, we lose information. However, the information loss is conservative.

We will require that out transfer functions be monotone, this requirement implies that increase in our knowledge about the input result in an increase of our knowledge about the output (or at least the our knowledge does not decrease).

Distributive Functions. We say f is *distributive* if for all $X \subseteq L : \bigsqcup\{f(x) \mid x \in X\} = f(\bigsqcup\{x \mid x \in X\}$. In this case our iterative algorithm will give an exact result, since no information loss occurs in the course of executing the algorithm.

Chains

Definition. A subset $L' \in L$ in a poset $(L, \sqsubseteq)$ is a called a *chain* if every two elements in L' are ordered. That means, for all $l_1, l_2 \in L' : l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$.

- An *ascending chain* is a sequence of values: $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \ldots$
- A *strictly ascending chain* is a sequence of values: $l_1 \sqsubset l_2 \sqsubset l_3 \sqsubset \ldots$
- A *descending chain* is a sequence of values: $l_1 \sqsupseteq l_2 \sqsupseteq l_3 \sqsupseteq \ldots$
- A *strictly descending chain* is a sequence of values: $l1 \sqsupset l_2 \sqsupset l_3 \sqsupset \ldots$
- We say L has a *finite height* if every chain in L is finite.

Example. Consider the complete binary tree and use "prefix of" as partial ordering. We get that every path in the tree is a chain.

Lemma 3. *A poset $(L, \sqsubseteq)$ has finite height if and only if every strictly decreasing and strictly increasing chains are finite.*

Fixed Points

We say $l \in L$ is a *fixed point* of a function $f : L \to L$ if $f(l) = l$.

Given a function $f : L \to L$ and a complete lattice $\mathcal{L} := (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$ we define the following three sets:

- The fixed points set $Fix(f) = \{l \mid f(l) = l\}$
- The reductive set $Red(f) = \{l \mid f(l) \sqsubseteq l\}$
- The extensive set $Ext(f) = \{l \mid f(l) \sqsupseteq l\}$

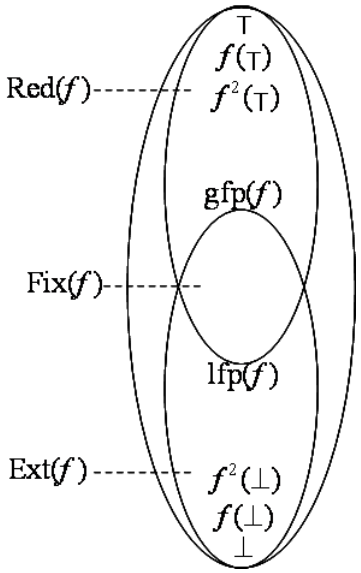

Fig. 5: Fixed Points

Figure 5 depicts the fixed points of f.

In order to see how our example program relates to the fixed points diagram we claim (and prove shortly) that the outcome of our algorithm is the $\mathrm{lfp}(f)$. Since our example program was quite trivial, we will modify it slightly by adding the variable w initializing it to 0 and within the while loop we add the statement `w = w + 1` (see Figure 6). It is not hard to see that the final outcome of our algorithm (the $\mathrm{lfp}(f)$) is now $[w \mapsto \top, x \mapsto 3, y \mapsto 7, z \mapsto 3]$.

```
0. w = 0
1. z := 3
2. x := 1
3. while (x > 0) do
3+     w = w + 1
4.     if (x = 1)
5.        y := 7
6.     else
7.        y := z + 4
8.     x := 3
9.     print y
```

Fig. 6: `Slightly modified example program`

The state $[w \mapsto \top, x \mapsto \top, y \mapsto 7, z \mapsto 3]$ is placed above the $\mathrm{lfp}(f)$ in the

diagram. If that would have been our algorithm's outcome it would have been a correct (sound) solution since every variable the algorithm claim is constant is indeed so, but obviously this is not the best solution.

On the other hand if the outcome would have been $[w \mapsto 5, x \mapsto 3, y \mapsto 7, z \mapsto 3]$ we would say it is not sound, since though w is said to be a constant it is not so. The state is depicted below the $\mathrm{lfp}(f)$ in the diagram.

We have:

$$\bot \sqsubseteq [w \mapsto 5, x \mapsto 3, y \mapsto 7, z \mapsto 3] \sqsubseteq \mathrm{lfp} \sqsubseteq [w \mapsto \top, x \mapsto \top, y \mapsto 7, z \mapsto 3] \sqsubseteq \top$$

Tarski's Theorem

Let $\mathcal{L} := (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$ be a complete lattice. If $f : L \to L$ is monotone then $\mathrm{lfp}(f)$ and $\mathrm{gfp}(f)$ satisfy:

$$\mathrm{lfp}(f) = \bigsqcap \mathrm{Red}(f) \in \mathrm{Fix}(f)$$

$$\mathrm{gfp}(f) = \bigsqcup \mathrm{Ext}(f) \in \mathrm{Fix}(f)$$

Proof. To prove the claim for $\mathrm{lfp}(f)$ we define $l_0 = \bigsqcap Red(f)$. We first show that $f(l_0) \sqsubseteq l_0$ so that $l_0 \in Red(f)$. Since $l_0 \sqsubseteq l$ for all $l \in Red(f)$ and f is monotone we have

$$f(l_0) \sqsubseteq f(l) \sqsubseteq l \text{ for all } l \in Red(f)$$

and hence $f(l_0) \sqsubseteq l_0$. To prove that $l_0 \sqsubseteq f(l_0)$ we observe that $f(f(l_0)) \sqsubseteq f(l_0)$ showing that $f(l_0) \in Red(f)$ and hence $l_0 \sqsubseteq f(l_0)$ by definition of l_0. Together this shows that l_0 is a fixed point of f so $l_0 \in Fix(f)$. To see that l_0 is least in $Fix(f)$ simply note that $Fix(f) \subseteq Red(f)$. It follows that $\mathrm{lfp}(f) = l_0$.
The claim for $\mathrm{gfp}(f)$ is proved in a similar way. □

We have a special interest in $\mathrm{lfp}(f)$, since we would like to find the best solution without damaging the soundness of our algorithm. $\mathrm{lfp}(f)$ is that point, below it our solution is not conservative. We sometimes will give a point above $\mathrm{lfp}(f)$ as a result from efficiency considerations.

Calculating $\mathrm{lfp}(f)$

Now that we've established the mathematical foundations we are interested in finding constructive algorithm to compute the $\mathrm{lfp}(f)$ of every point in out program. We're interested in the $\mathrm{lfp}(f)$ since it give us the minimum number of non-constant and the maximum number of $\bot$.

```
x = ⊥
while x ≠ f(x) do
    x = f(x)
```

Consider the following algorithm:

We show that if this iterative algorithm terminates, it will compute $\mathrm{lfp}(f)$. Note that if the algorithm terminate x = f(x), meaning that the end value is in $Fix(f)$. In order to prove that the algorithm indeed calculate $\mathrm{lfp}(f)$ all is left to show is that the end value is a lower bound of $Fix(f)$. We take advantage of the fact that f is monotone, and prove our claim using induction on the number of loop iterations.

Our base case is trivially correct, since $\bot$ is obviously a lower bound. Assume that after $i-1$ iterations x is a lower bound. In the i^{th} iteration we calculate the new value of x which is $f(x)$. Since x is a lower bound we have that for all $l \in Fix(f)$ x $\sqsubseteq l$. From the monotonicity of f we get $f(x) \sqsubseteq f(l) = l$, and the new value of x is still a lower bound.

This is true only if the algorithm terminates, which in the general case is not guaranteed. Notice, however, that if the lattice L on which f is defined has a finite height the algorithm is guaranteed to terminate.

A similar analysis can be done starting with x = $\top$ and calculating $\mathrm{gfp}(f)$.

Chaotic Iteration

Given a lattice $\mathcal{L} := (L, \sqsubseteq, \bigsqcap, \bigsqcup, \bot, \top)$ with finite strictly increasing chains, we denote $L^n = L \times L \times \ldots \times L$. Let $\underline{f} : L^n \to L^n$ be a monotone function. We would like to compute $\mathrm{lfp}(\underline{f})$, which is the simultaneous fixed point of the system $\{x[i] = \underline{f}_i(x)\}$

A naïve algorithm for finding the $\mathrm{lfp}(\underline{f})$ of the combined system is a slight modification of the above mentioned algorithm to handle vector of variables instead of a single one.

$$\begin{array}{l} \overrightarrow{x} = \{\bot, \bot, \bot, \ldots, \bot\} \\ \texttt{while } \overrightarrow{x} \neq f(\overrightarrow{x}) \texttt{ do} \\ \quad \overrightarrow{x} = f(\overrightarrow{x}) \end{array}$$

Fig. 7: A naïve algorithm

This algorithm, however correct, is extremely inefficient taking $O(n)$ time at each step.

The Algorithm

An improved algorithm should structure its traversal method to conservatively include only those steps that might influence the final outcome and to skip those who don't.

At all times in the algorithm we maintain a work-list (WL) that holds the tasks to be done. Initially, since no computation has been made, the list contains all the statements. As the algorithm progresses, a task is removed from the list and processed. Processing a task might result in other tasks being added to the list. The process continues until the work-list is empty - there is no more work to be done. In our context, the tasks are to update our knowledge of constants at a point in the program. If by processing a task, applying the transfer function, changes are made to the following state we will add to the list all the states that might be influenced. When the algorithm converges – no more changes are made – it will terminate. The algorithm is displayed in Figure 8.

```
Step 1 Initialization
for i:=1 to n do
   x[i] = ⊥
WL = {1,2,...,n}
Step 2 Iteration (updating of WL and the values array)
while (WL ≠) do
   select and remove an element i ∈ WL
   new := f_i(x)
   if (new ≠ x[i]) then
      x[i] := new
      add all indexes that directly depend on i to WL
```

Fig. 8: Chaotic Iteration

Notice that this is more a framework rather than a specific algorithm since the details of how the work-list is maintained, as well as the specifics of the extraction operation are omitted. Various concrete algorithms can be derived from this framework. All algorithms will have the same final outcome, which is $\mathrm{lfp}(\underline{f})$, however, their complexity differs based on the exact details of implementation.

Specialized Chaotic Iteration

A variant of the chaotic iteration algorithm is defined in terms of graph rather than the vector representation of the above algorithm. As with the above algorithm we maintain a work-list of the nodes we still need to traverse. This will fit nicely with a CFG representation of a program. The algorithm is defined in figure 9.

```
Chaotic(G(V,E):Graph, v0:Node, L:Lattice, ι:L, f : E → (L → L))
    for each v ∈ V do
        cf_entry[v] = ⊥
    cf_enry[v0] = ι
    WL = {v0, v1, v2, ..., vn}
    while (WL ≠) do
        select and remove element u ∈ WL
        for each v s.t.  (u, v) ∈ E do
            temp = f(e)(cf_entry[u])
            new := cf_entry[v] ⊔ temp
            if (new ≠ cf_entry[v]) then
                cf_entry[v] := new
                WL := WL ∪ {v}
```

Fig. 9: `Specialized Chaotic Iteration`

We assume that there is a a designated control flow node v_0 that is the start node and has no incoming edges.

For a program S the graph is the CFG of S, v_0 the starting node and ι is an initialization value chosen, for example, by the implementors of the compiler. Also, the lattice L is the environments, namely a mapping from the program variables to their value at a program point. f associates a transfer function with every edge of the CFG.

The algorithm is equivalent to the following system of equations:

$$S := \begin{cases} cf_{entry}[s] = \iota \\ cf_{entry}[v] = \sqcup\{f(u,v)(cf_{entry}[u]) \mid (u,v) \in E\} \end{cases}$$

and in vector representation:

$$F_S : L^n \to L^n := \begin{cases} F_S(X)[s] = \iota \\ F_S(X)[v] = \bigsqcup\{f(u,v)(X[u]) \mid (u,v) \in E\} \end{cases}$$

The solution is the same in both representations, i.e. $\mathrm{lfp}(S) = \mathrm{lfp}(F_S)$

The number of equations is the same as the number of nodes in the graph, and a minimal solution to the system is our desired result.

Soundness, Completeness and Complexity

We now turn to discuss the characteristics of the chaotic iteration algorithm.

Complexity. Let n be the number of nodes in the graph (CFG), n is proportional in the size of our program which may be quite large. In practice, though, the analysis will be applied to a single procedure at a time, thus maintaining a manageable size. k is the maximum outdegree and is dependant on the programming language. In our examples the maximum outdegree is 2, though in a language such as `C` that has a `switch` statement the outdegree is unbound. h is the height of the lattice which is proportional in the number of nested loops, and while in theory it is unbounded, in practice Knuth reported of a maximum nesting of seven. c is the maximum cost of applying $f_{(e)}, \sqcup$ and L comparisons. From all the above we get that the complexity of the algorithm is $O(n \cdot h \cdot c \cdot k)$

Soundness. The chaotic iteration algorithm is *sound* which means that any constant it will report of is indeed such, though there might be constants it will fail to detect. In that sense it is conservative. For an error detection algorithm soundness implies that it will detect every possible error, though some of them may not occur at runtime. In general, the least fixed points computed represents at least all occurring run time states.

The soundness of the algorithm can be proved by first proving that the transfer function of each semantic construct behave as it should. The soundness of the algorithms than follows since we already know that it computes the least fixed points, which represent the maximum number of constants.

Completeness. Another desirable property is *completeness*. In constant propagation that means our algorithm detects every constant in the program. As is usually the case the the algorithm is not complete. Completeness means that every state represented by the least fixed point is reachable for some input.

Next Week on Program Analysis...

Though we've mentioned that the chaotic algorithm is sound a formal proof is still required. To so we have to introduce the field of abstract interpretation, explaining Galois connections (and insertions) and collecting semantics and how it relates to structural operational semantics. These topics will be covered in depth on the next lecture. We give here a (very) short introduction.

Abstract Interpretation

Abstract Interpretation is the foundation of program analysis. Formalized by Patrick Cousot and Radhia Cousot [CC76], its goals are to establish the soundness of a given program analysis algorithm and to guide the design of new analysis algorithms.

The technique relates each step in the algorithm to a step in structural operational semantics and establish global correctness using a general theorem proved in CC76] following directly from Tarski's theorem.

Galois Connection

Let A and C be two lattice, $\alpha : C \to A$ and $\gamma : A \to C$. The pair of functions (α, γ) form a *Galois connection* if:

- α and γ are monotone
- for all $a \in A, \alpha(\gamma(a)) \sqsubseteq a$
- for all $c \in C, c \sqsubseteq \gamma(\alpha(c))$

Alternatively, the pair (α, γ) is a Galois connection if for all c in C and for all a in A, $\alpha(c) \sqsubseteq a$ iff $c \sqsubseteq \gamma(a)$.

Figure 10 gives a schematic representation of Galois connection.

In order to prove soundness of the constant propagation algorithm we need to:

- Define an "appropriate" structural operation semantics
- Define "collecting" structural operation semantics
- Establish a Galois connection between collecting states and the abstract states
- (Local correctness) Show that the abstract interpretation of every atomic statement is sound with respect to the collecting semantics
- (Global correctness) Conclude that the analysis is sound according to [CC76]

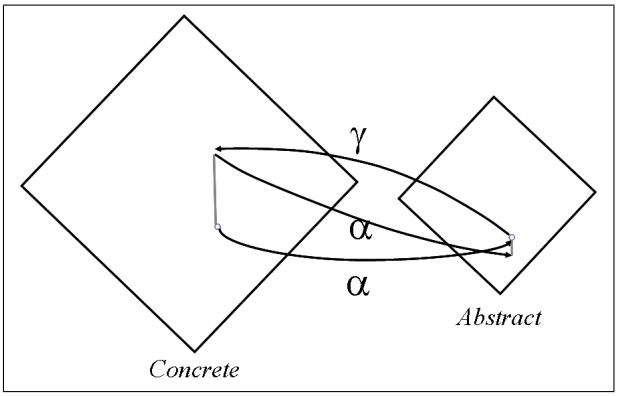

Fig. 10: Galois Connection

Collecting Semantics

The collecting semantics is used to define a collection of states that can occur at any program point during the course of an execution. Since we don't know what the program's input will be we have to analyze the program for all possible inputs. For example, while executing the constant propagation we reached a state where the value of x could have been either 1 or 3. In the structural operational semantics we mapped this to the state $[x \mapsto \top]$ whereas in collecting semantics the state will be the collection of the possible states, namely $\{[x \mapsto 1], [x \mapsto 3]\}$.

Summary

In this lecture we have described the Constant Propagation Analysis. We formalized the algorithm using partially ordered sets, lattices and monotone functions. We have begun to introduce the concepts that will later serve us to prove the correctness of the algorithm and other algorithms in the future. Those are the abstract interpretation technique, Galois connections and the collecting semantics. These concepts will be explained in greater length future lectures.

References

[1] Lecture slides

[2] Fleming Nielson, Hanne Riis Nielson and Chris Hankin *Principels of Program Analysis*, Springer-Verlag, 1999 ,

[3] Wikipedia, The Free Encyclopedia, www.wikipedia.org