Advanced Program Analysis – Spring Semester 2007

Lecture III – Monotone Frameworks (15/03/2007)

Summarized by Omer Kidron, ID 040805202, omer.kidron@gmail.com

This summery is **addition** to the notes, available at
http://www.cs.tau.ac.il/~msagiv/courses/pa07/mon.pdf

Fixed Point

We discussed in the previous lecture Constant Propagation method for program analysis:

We are looking for the minimal fixed point. We don't want just any fixed point, but the **minimal** one, because it's the most accurate solution. Later on the course we'll show that LFP is the most reasonable point to select: It's result is sound.

Least Fixed Point

Let's take a look over the next example program:
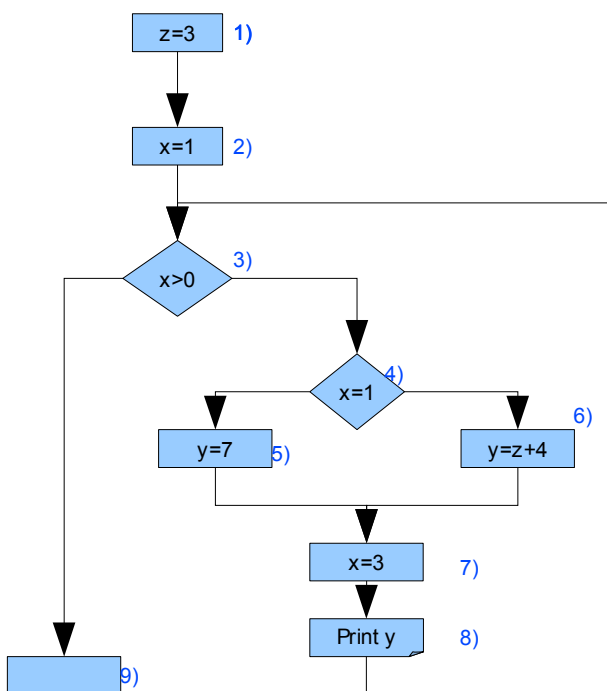
We are looking for the minimal fixed point. The minimal solution guarantees that the solution is the most precise among the fixed points. But why are we interested in computing fixed points?

As we will see later any fixed point is a sound solution under certain solution.

However, there may exist sound solution which not fixed points but those are usually hard to compute in iterative manner.

We want to find some assignments to the equations-system, that will solve it. Our solutions are vectors of 9 elements, where each element is a triple of the values of (x, y, z).

The following table shows some vectors where each major column (1, 2, ...) is number of the element in the vector (=step in the program), and each minor column (x, y, z) is the value of the x, y and z variables in that point of the program.

#	1			2			3			4			5			6			7			8			9		
	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z	x	y	z
a	0	0	0	0	0	3	T	T	3	T	T	3	1	T	3	T	T	3	T	7	3	3	7	3	3	7	3
b	0	0	0	0	0	3	T	T	5	T	T	5	1	T	5	T	T	5	T	7	5	3	7	5	3	7	5
c	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T

Solution **a** is the minimal fixed point, found by the algorithm. From all the 9 elements of vector a, the solution, which also a LFP, is (x = T, y = T, z = 3). That's because this triplet is a sound solution to the equations-system.

Looking at solution **b**, where we gave a higher bound (5) to the variable z, the solution would be (x=T, y=T, z=5). This also solves the system, but it's not minimal nor tight.

Solution **c** is trivial: it's obvious that (x=T, y=T, z=T) solves the equation-system, but it doesn't help us.

In order to find additional fixed points, one may add more variable, such as t. t will have a constant value at all the points, say t=18. The point will be fixed everywhere, but not necessarily minimal one.

Monotone Forward Frameworks

Intuitively, we start from the beginning of the program, with the first line (entry) and initial state, and for each line we activate the matching transition function over the previous state.

Formally, this framework suggest the next model for analysis:

- We use the complete lattice $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ to describe the information over the variables.

- Initial value is noted by $i \in L$

- The effect of every control flow edge e is described by a monotone function $f_e : L \to L$ (transfer function)

- We compute the LFP of the equation system:

- DF(entry) = i // The initial value of first line
- DF(v) = $\sqcup$ {f (u,v)(DF(u) | (u, v) $\in$E} // The DF of each line is the join of all the lines entering it.

Looking at the previous example program, using this method, we get the next DF:
- DF(1)=ι // Initial value
- DF(2)=DF(1)[z→3] // Over the previous state, x assigned 3
- DF(3)=DF(2)[x→1] $\sqcup$DF(8) // x assigned y join the state there will be when returning from the while loop
- DF(4)= if DF(3)x >0 then DF(3) else $\bot$ // if we came from the while condition, then x remains as it was, otherwise, we can tell what x is.
- DF(5)= DF(4) $\sqcap$[x→1, y→$\top$, z→$\top$] // If we reached line 5, x is 1 for sure
- DF(6)= if DF(4)x $\neq$1 then DF(4) else $\bot$ // As after the while condition (line 4)
- DF(7)= DF(5) [y→7] 7DF(6)[y→e(z)+4] // Line 7 might be reached from 2 different line, 5 or 6, so we join the possible values.
- DF(8)= DF(7)[x→3]
- DF(9)=if DF(3)x≤0 then DF(3) else $\bot$

Chaotic Iteration for Forward Problem

We can use the next algorithm to implement the forward system:

    // Init every var
    for v ∈N **do**
            DF(v) := ⊥

    DF(entry) := ι
    WL= N // Init the working line
    while WL != Ø **do**
            Select and remove an arbitrary u ∈WL
            for every edge (u, v) ∈E **do**
                    temp = DF(v) ⊔ f_e(DF(u)) // Assign to temp the info we have so far about v's
                                variables, joined with effect of the vertices
                                leading to it
                    if (temp != DF(v)) // If the last move didn't add new info about the edge
                            DF(v) := temp // We are in a stable state
                            WL := WL ∪{v}

– Complexity might reach $O(n^2)$
– Assigning in temp could be expensive, because of the Join operation.

Monotone Backward Frameworks

Intuitively, we start from the end of the program, with the last line (exit/return line) and initial state, and we go backwards in to code, towards the 1st line, for each line we activate the matching transition function over the previous state. We can look at the control-graph forward and backwards. This way we can anticipate future values of the computation.

Formally, this framework suggest the next model for analysis:

– We use the complete lattice $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ to describe the information over the variables.

– Initial value is noted by $i \in L$

– The effect of every control flow edge e is described by a monotone function $f_e : L \to L$ (transfer function.

– We compute the LFP of the equation system:

  – DF(exit) = i // The initial value of first line

  – DF(v) = $\sqcup \{f(u,v)(DF(u) \mid (v, u) \in E\}$ // The DF of each line is the join of all the lines it enters.

Chaotic Iteration for Backward Problem

We can use the next algorithm to implement the forward system:

> // Init every var
>
> **for** v $\in$ N **do**
>
> > DF(v) := $\bot$
>
>
> DF(exit) := ι
>
> WL= N // Init the working line
>
> **while** WL != $\varnothing$ **do**
>
> > Select and remove an arbitrary u $\in$ WL
> >
> > **for** every edge (v, u) $\in$ E **do**
> >
> > > temp = DF(v) $\sqcup$ f_e(DF(u)) // Assign to temp the info we have so far about v's variables, joined with effect of the vertices it leads to
> > >
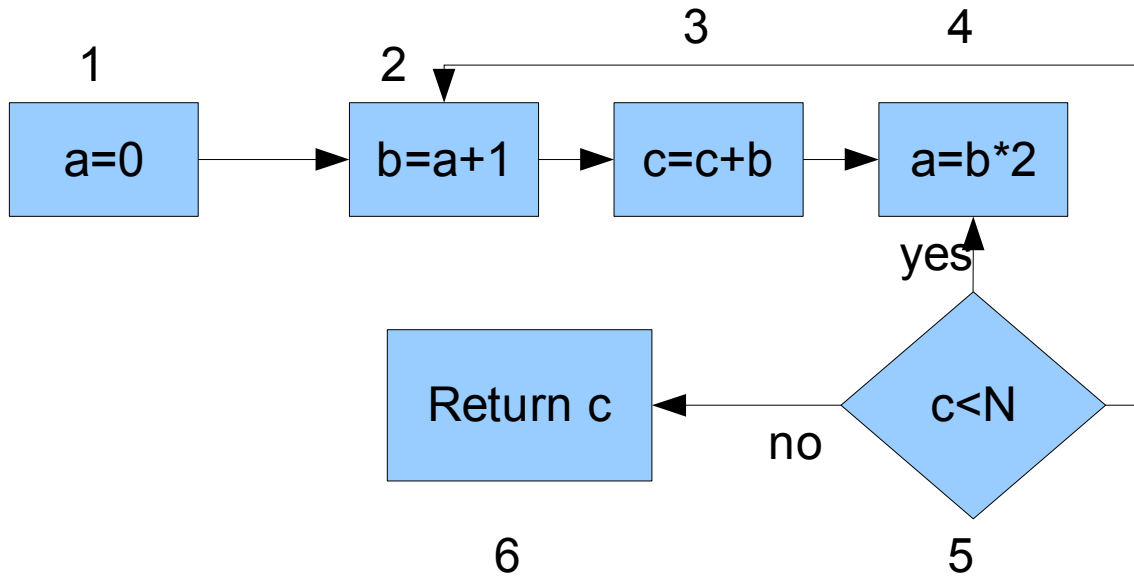> > > **if** (temp != DF(v)) // If the last move didn't add new info about the edge
> > >
> > > > DF(v) := temp // We are in a stable state
> > > >
> > > > WL := WL $\cup$ {v}

– This is exactly the same algorithm of the forward problem, except for the opposite direction of the edges.

General Monotone Frameworks

Control graph:

The lattice would be (P(var), $\subseteq$, $\cup$, $\cap$, $\emptyset$, Var), where P(Var) is the power set of the variables in the program.

$Df_1 = Df_2 - \{a\}$ // All the variables before line 1

$Df_2 = Df_3 - \{b\} \cup \{a\}$ // We subtract all that from the left wing of the assignment

$Df_3 = Df_4 - \{c\} \cup \{b, c\}$ // Adding what's in the right wing of the assignment

$Df_4 = Df_5 - \{a\} \cup \{b\}$

$Df_5 = \underline{Df_6 \cup \{c\}} \cup \underline{Df_2 \cup \{c\}}$

      arc to L6   arc to L1

$Df_6 = \emptyset$

As we in the previous paragraph, the general model is called "Gen/Kill". Because at each transition we remove a constant set or add a constant set (or both, as in the last example). These are the simple cases:
– Reaching Definitions
– Available Expressions
– Live Variables
– $\sqcup = \cup$ or $\sqcup = \cap$
– $f_e(entry)) = (entry\text{-}kill(e)) \cup gen(e)$

Maybe Garbage Variable

We want to know when variable t might contain "garbage" value (not initialized). This is important, because in a case like the following example (from the slide), variable y won't be initialized:
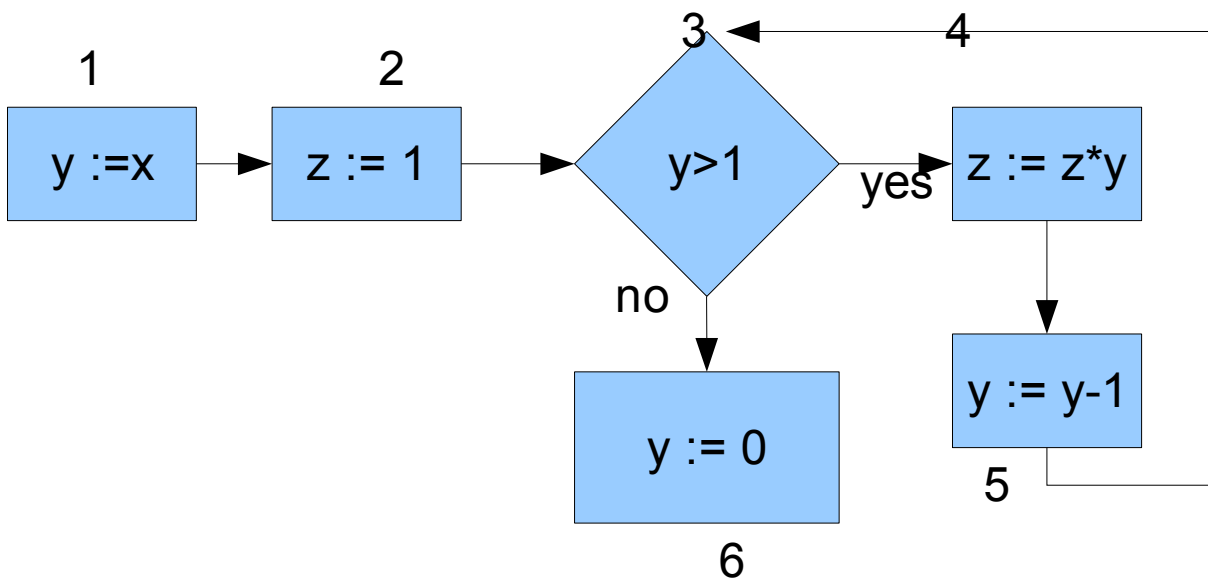
     x := 5 ;

if z > 2

        then y := 17;

        else skip;

  t := y + x;

The model we'll use is:

- $L = (P(Var), \subseteq, \cup, \cap, \emptyset, Var)$

- Initial value $\iota = Var^*$

- Transfer functions $f_e(DF)$:

  - x := a if $FV(a) \cap DF \neq \emptyset$ then

    - $DF \cup \{x\}$

    - else $DF - \{x\}$

  - skip DF

  - b DF

Let's look at the control graph of an example program:

Var = Df_0 = {x, y, z} // No variable is initialized

Df_1 = {x, y, z} // If x is not certainly initialized then also y now

Df_2 = {x, y} // No z is initialized. But it will change after 1 iteration of the while

Df_3 = Df_2 // No assignment

Df_4 = {x, y, z}

Df_5 = {x, y, z}

Now we return (or might return, actually) to line 3, and because Df_2 is dependent on all entry points, Df_2 is {x, y, z}

Over Conservative Solution

This is an example of a blind spot for this algorithm: It doesn't know that z is initialized in the other branch.:

```
    if y >1
            then z := 1;
    ...
    if y >1
            then y := z;
```

There are algorithms that remember that y>1, but keeping this amount of information affects the performance badly.

Point-To Analysis

In C language, a variable is a pointer, and you can't tell it's content and what a change in it will affect (examples in slide 21). Therefor we use matrix and store which variable points to which variable.

Code	Transition Function	Explanation
x=&y	$DF - \{(x, z) \mid z \in Var\} \cup \{x, y\}$	Remove all previous references of x and add the current
x=y	$DF - \{(x, z) \mid z \in Var\} \cup \{(x, z) \mid (y, z) \in DF\}$	Adds to x everything that y points to.
x=*y	$DF - \{(x, z) \mid z \in Var\} \cup \{(x, z) \mid (y, w), (w, z) \in DF\}$	Add a pointer to what's y points to. (y points to w, w points to z, hence x points to z)
*x=y	$DF - \varnothing \cup \{(w, t) \mid (x, w), (y, t) \in DF\}$	We can't remove all the pointer x pointed to.

We can improve performance and complexity: if instead of referring to assignment as initialization, meaning the line x=y is treated as x←y and y←x, then we can use Tarjan algorithm. This allows us to process long programs.

When using this method, we encounter the usage of other methods, such as Constant Propagation (seen in the previous lecture), like the next example:

```
    x = 5;
    *p = 7;
```

Example

Let's look at the next program:

1. t := &a;
2. y := &b;
3. z := &c;
4. if x> 0;

5.   then p:= &y;

6.   else p:= &z;

7. p := t;

Using the Point-To we'll look at the matrix:

#	Before	Line	After
1.	Ø	T:=&a	{(t, a)}
2.	{(t, a)}	Y:= &b	{(t, a), (y, b)}
3.	{(t, a), (y, b)}	Z:=&c	{(t, a), (y, b), (z, c)}
4.	{(t, a), (y, b), (z, c)}	If X>0	{(t, a), (y, b), (z, c), (p, y), (p, z)}
5.		P:=&y	{(t, a), (y, b), (z, c), (p, y)}
6.		P:=&z	{(t, a), (y, b), (z, c), (p, z)}
7.	{(t, a), (y, b), (z, c), (p, y), (p, z)}	P:=t	{(t, a), (y, b), (y, c), (p, y), (p, z), (y, a), (z, a)}

Flow Insensitive

In class we saw Andersen's algorithm for points-to analysis. This algorithm ignores the flow control and uses one graph for the entire program. For each pointer we keep a list of all the other pointers/variables it points to.

1. t = &a

2. y = &b

3. z = &c

4. if x>0

5.   p = &y

6. else

7.   p = &z

8. p = t

The algorithm keeps track of all pointers

t = {a} // Line 1

y = {b} // Line 2

z = {c} // Line 3

p = {y, b, z, c, t} // y because of line 5, b because of y, z because of line 7 (flow insensitive), c because of z, t because of line 8

The complexity of the algorithm is $O(n^2)$ in time and space.

Steengaard suggested a method which is flow-insensitive and assignment-direction insensitive (meaning it treats a=b as (a=b) and (b=a)). Every time the algorithm sees an assignment, it build the equality class of the variables: x=y becomes [x]=[y] and x points to all the variables y pointed to and vice versa.

An example from the article:

```
a = &x
b = &y
if p then
    y = &z;
else
    y = &x
fi
c = &y
```
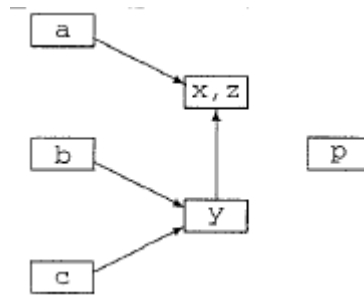
The complexity is linear in space, and nearly linear in time (because of the union find which is done when uniting equality classes).

A good article, comparing Andersen and Steengaard can be found [here](#)

Chaotic Iterations Analysis

In computing the complexity of the Chaotic Iterations methods, we use the following parameters:

-   N – No. of nodes.
-   k – Maximal out degree in the graph.
-   h – Height of the lattice (length of longest chain in the lattice)
-   c – Some cost, which is the max cost of:
    -   Applying f_l
    -   Join ($\sqcup$)
    -   L comparisons

The overall complexity of a chaotic iterations algorithm $\in O(N * h * k * c)$

Precision of Chaotic Iterations

-   Optimal – Since it runs over all possibilities, it finds the best solution.
-   "Join-over-all-path" - In a given node, we'll take all the possible (infinite number of) paths to it and join them (union). This is equals to using the Least Fixed Point.

It gives completeness, because it can discover paths with problems (because it contains all the paths), even though they are actually not running.

Join Over All Path (JOP)

-   Every path is a collection of edges.
-   The transition function are run serially along the path.
-   JOP usually is not important.

$JOP \subseteq DF(v)$ is true for each fixed point, because JOP goes over long paths. Since the functions are monotonic, we can show it with induction.

- correction to the slide: in the 2ⁿᵈ bullet it should be "edge e" instead of "labels l".

Additive (Distributive) Monotone Problems

- Kill/Gen Problems, as we explained before (page 6)

- May be uninitialized, as we explained before (page 6)

- Truly Live – Whether there's a use of variable's value.

- Linear constant propagation – When we deal with linear expressions, of the form x=ay+b, where x,y are variables and a,b are literals.

- Points-To with one level pointer – a point b, b points to a primitive value (not other variable)

Non Additive (Non-Distributive) Monotone Problems

- Points-To Analysis – When there's more than 1 level of pointing, the function cannot be distributive.

- Constant Propagation on arbitrary expression.

Converting Into Distributive Frameworks

We saw before how to take a problem and solve it using the distributive framework.

1. Define a finite lattice, denoted by $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$

2. We define $i \in L$ to be the initial value for entry.

3. The effect of every edge at e is described by a monotone function $f_e : L \to L$ (transfer function)

4. The distributivity will result from defining a distributive over $(P(L), \subseteq, \cup, \cap, \emptyset, L)$ function as follows:

$$F(X) = \{f_1(x) : x \in X\}$$

It's distributive, because for
$$A, B \subseteq L, F(A \cup B) = \{f_1(x) : x \in A \cup B\} = \{f_1(x) : x \in A\} \cup \{f_1(x) : x \in B\} = F(A) \cup F(B)$$

5. Solve the following system of equations:

   1. DF(entry) = {i}
   2. DF(v) = $\cup \{f_{(u, v)}(x) \mid (u, v) \in E, x \in DF(u)\}$

Constant Propagation

It is undecidable to find the JOP in the constant propagation problem.

Here is a Sketch of a proof:

        while (*cond*)

```
if (*cond*) x_1 = x_1 + 1;

if (*cond*) x_2 = x_2 + 1;

...

if (*cond*) x_n = x_n + 1;
y = truncate (1/ (1 + p2(x_1, x_2, ..., x_n))   // Is y=0 here?
```

Static Analysis problems beyond Monotone Frameworks

Problems that can't be handled with Monotone Frameworks:

– Infinite heights – the methodology doesn't stop drilling down the tree. Infinite height is undecidable. Beside, the maximal height is part of the complexity.

– Bi-Directional Problems – When a refer to b but b is allowed to refer back to a, some dependencies created, which monotone can't handle. For example:

   – x := b[z]

   – a[b[y]] := x // Data type question

Historical Perspective

The articles are referenced (for websites). Accessing them requires login, can be achieved via TAU proxy.

- [1973 Kildall](#) - Defined the basic framework but required distributive frameworks
- [1976 Kam & Ulmann](#) - Defined Monotone Framework
- [1980 Tarjan](#) - Suggested an almost linear time algorithm for reducible flow graph
- [1980 Rosen](#) - Suggested a linear time algorithm for high level language

Conclusions

- Chaotic iterations is a powerful technique
- Easy to implement
- Rather precise
- But expensive
    - More efficient methods exist for structured programs