Lecture Notes on Program Analysis and Verification

Deepak D'Souza and K V Raghavan
Department of Computer Science and Automation
Indian Institute of Science, Bangalore.

7 August 2013

Contents

1 Lattices and the Knaster-Tarski Theorem **2**
1.1 Why study lattices in program analysis 2
1.2 Partial orders and lattices 3
1.3 Monotonic functions and the Knaster-Tarski Theorem 6
1.4 Computing the LFP 10

2 Interprocedural Analysis **12**
2.1 Motivation 12
2.2 Interprocedurally valid paths 14
2.3 Call-Strings approach 15
2.4 Correctness of call-strings approach 18
2.5 Computing JOP/LFP 18
2.5.1 Bounded call-string method 20

Chapter 1

Lattices and the Knaster-Tarski Theorem

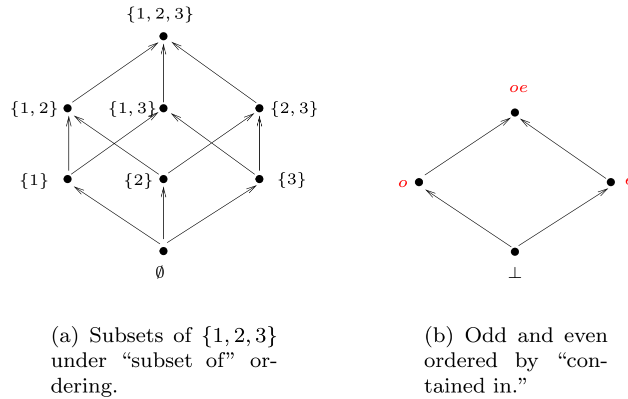
In this chapter we recall some of the basic concepts from lattice theory that we make use of later in these lectures. We begin with some motivation for why we need lattices, introduce and illustrate the basic definitions, and finally state and prove the well-known Knaster-Tarski fixpoint theorem.

1.1 Why study lattices in program analysis

In program analysis we are typically interested in finding a "safe" approximation (or an "over-approximation") of the set of concrete states that may arise at a program point due to different executions of the program. A natural way to obtain this "collecting state" at a point N in a program is to take the union of the set of states reached along each (initial) path in the program leading to the point N. For example, in the program of Fig. 1.2(a), the (single) execution of the program visits point 5 several times, leading to concrete states $\{(5,2),(6,4),(7,6),(8,8)\}$. We represent a concrete state in which `x` is mapped to 5 and `y` is mapped to 2, by simply $(5,2)$.

When dealing with abstract states, we want to collect the abstract states reached by "abstractly executing" or "interpreting" each path in the program that leads to point N, and then take a union of the set of concrete states they represent. This latter step corresponds to taking the "join" of the abstract states collected at point N. If the abstract states have a "complete" lattice structure, then this join is guaranteed to exist. This value at point N is called the "join over all paths" (JOP) at point N. For example, we could interpret the program using the lattice of abstract values shown in Fig. 1.2(b), where an abstract value of the form (o, oe) represents the set of concrete states in which `p` is mapped to an odd value and `q` is mapped to any (odd or even) value. Thus, along the path 12345, the resulting abstract value at point 5 is (e,e). The only other abstract value at point 5 is (o,e) (via path 12345345 for example). The join over all paths at point 5 is just the join of the elements (e,e) and (o,e), which is the abstract element (oe,e).

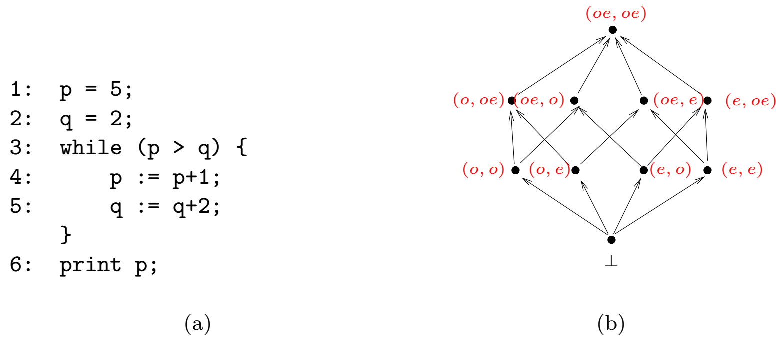
Why are we interested in certain functions on lattices having fixpoints? It turns out that instead of finding the JOP values for a given program and abstract



(a) Subsets of $\{1, 2, 3\}$ under "subset of" ordering.

(b) Odd and even ordered by "contained in."

Figure 1.1: Some example lattices.



(a)

(b)

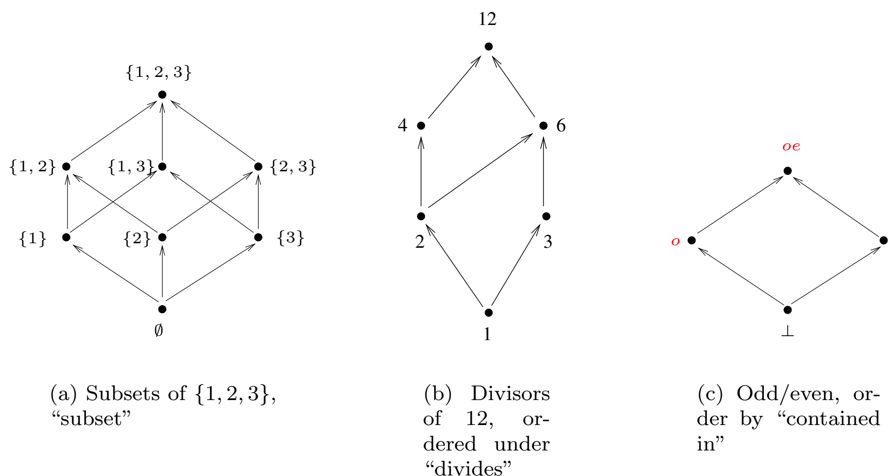
Figure 1.2: (a) Example program and (b) parity lattice.

analysis, it is more convenient to find a fixpoint of a certain function (induced by the abstract analysis and given program) on a lattice. Under certain conditions this fixpoint is guaranteed to over-approximate the actual JOP value. The Knaster-Tarski theorem gives us sufficient conditions under which a function on a complete lattice is guaranteed to have a fixpoint. It also tells us the structure of these fixpoints and gives us a characterisation of the greatest and least fixpoints.

1.2 Partial orders and lattices

An order is a relationship among elements of a set. The usual order on numbers, $1 \leq 2 \leq 3$, is called a *total* order, since each pair of elements is ordered. Some domains are naturally "partially" ordered, as shown in Fig. 1.3: for example the "subset of" or "containment" ordering on the the set of all subsets of a three element set. Notice that the subsets $\{1, 2\}$ and $\{2, 3\}$ are unordered as neither is a subset of the other.

A *partially ordered set* is a non-empty set D along with a partial order $\leq$ on



(a) Subsets of $\{1, 2, 3\}$, "subset"

(b) Divisors of 12, ordered under "divides"

(c) Odd/even, order by "contained in"

Figure 1.3: Some example partial orders

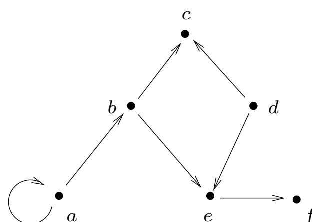
D. Thus $\leq$ is a binary relation on D satisfying:

- $\leq$ is reflexive ($d \leq d$ for each $d \in D$)
- $\leq$ is transitive ($d \leq d'$ and $d' \leq d''$ implies $d \leq d''$)
- $\leq$ is anti-symmetric ($d \leq d'$ and $d' \leq d$ implies $d = d'$).

It is convenient to view a partial order as a graph, or what is commonly called its "Hasse diagram." To begin with, we can view a binary relation on a set as a *directed graph*. For example, the binary relation

$$\{(a, a), (a, b), (b, c), (b, e), (d, e), (d, c), (e, f)\}$$

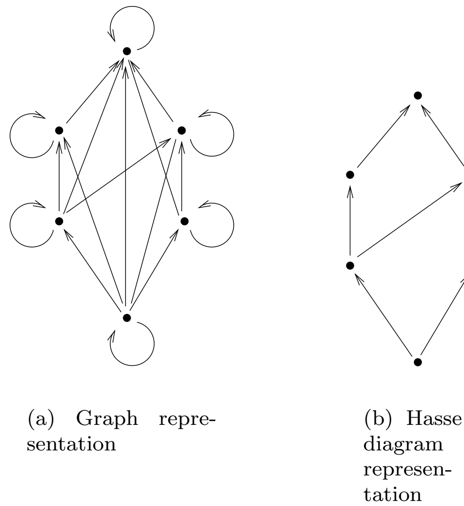
can be represented as the graph:



A partial order is then a special kind of directed graph, as shown in Fig. 1.4(a), in which reflexivity means a self-loop on each node, antisymmetry means no non-trivial cycles, and transitivity means "transitivity" of edges.

Let $(D, \leq)$ be a partially ordered set.

- An element $u \in D$ is an *upper bound* of a set of elements $X \subseteq D$, if $x \leq u$ for all $x \in X$.
- u is the *least upper bound* (or *lub* or *join*) of X if u is an upper bound for X, and for every upper bound y of X, we have $u \leq y$. We write $u = \bigsqcup X$.



(a) Graph representation

(b) Hasse diagram representation

Figure 1.4: Graph and Hasse diagram representation of the divisors of 12 poset.

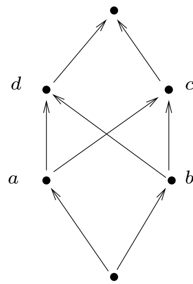


Figure 1.5: An example partial order

- Similarly, $v = \sqcap X$ (v is the *greatest lower bound* or *glb* or *meet* of X).

The above definitions are well-illustrated in the example partial order of Fig. 1.5: the pair of elements a and b have both d and c as upper bounds, but have *no* lub.

A *lattice* is a partially order set in which every pair of elements has an lub and a glb. A *complete* lattice is a lattice in which every *subset* of elements has a lub and glb.

The examples of Fig. 1.3 are all complete lattices. So also is the parity lattice of Fig. 1.2(b). Fig. 1.6 has some examples that illustrate the definitions.

We note that a complete lattice must have a *least* element, which we denote $\bot$, being the glb of the whole set D, and similarly a *greatest* element, denoted $\top$, which is the lub of the empty set.

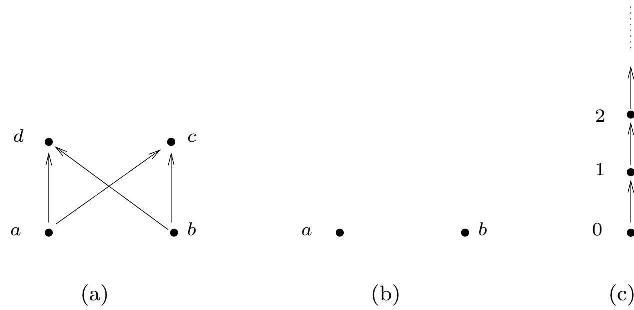


Figure 1.6: (a) A partial order that is not a lattice, (b) the simplest partial order that is not a lattice, and (c) a lattice that is not complete.

1.3 Monotonic functions and the Knaster-Tarski Theorem

Let $(D, \leq)$ be a partially ordered set, and X be a non-empty subset of D. Then X induces a partial order, which we call the partial order *induced by* X in $(D, \leq)$, and defined to be $(X, \leq \cap (X \times X))$. As an example, the partial order induced by the set $X = \{2, 3, 12\}$ in the partial order of Fig. 1.3(b) is shown below in Fig. 1.7:

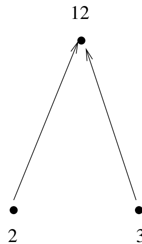


Figure 1.7: The partial order induced by $\{2, 3, 12\}$ in the partial order of Fig. 1.3(b).

Let $(D, \leq)$ be a partially ordered set. A function $f : D \to D$ is *monotonic* or *order-preserving* (wrt $(D, \leq)$) if for each $x, y \in D$, whenever $x \leq y$ we have $f(x) \leq f(y)$. Fig. 1.8 illustrates a monotone function on the divisor lattice.

Let f be a function on a partial order $(D, \leq)$. Then:

- A *fixpoint* of a function $f : D \to D$ is an element $x \in D$ such that $f(x) = x$. In Fig. 1.8, the fixpoints of the function are a and f.
- A *pre-fixpoint* of f is an element x such that $x \leq f(x)$. In Fig. 1.8, the pre-fixpoints are a, b, d, and f.
- A *post-fixpoint* of f is an element x such that $f(x) \leq x$. In Fig. 1.8, the post-fixpoints are b, e, and f.

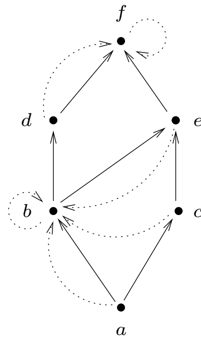


Figure 1.8: A monotone function on a partial order. The image of an element under the function is shown by a dotted arrow.

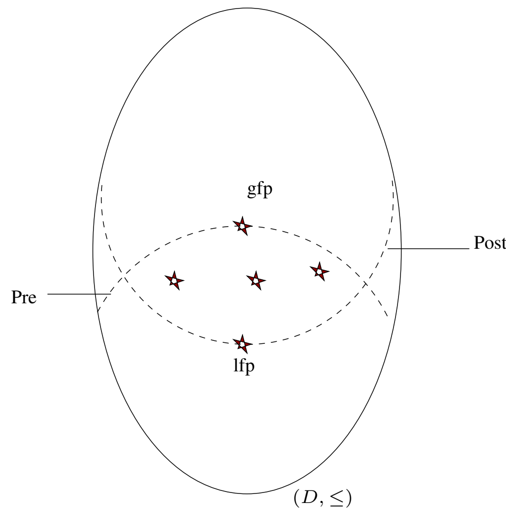


Figure 1.9: The structure of fixpoints (denoted by stars) according to the Knaster-Tarski theorem.

Theorem 1 (Knaster-Tarski [2]) *Let* $(D, \leq)$ *be a complete lattice, and* $f : D \to D$ *a monotonic function on* $(D, \leq)$*. Then:*

(a) f *has a fixpoint. In fact,* f *has a least fixpoint which coincides with the glb of the set of postfixpoints of* f*, and a greatest fixpoint which coincides with the lub of the prefixpoints of* f*.*

(b) The set of fixpoints P *of* f *itself forms a complete lattice under* $\leq$*. More precisely, the partial order induced by* P *in* $(D, \leq)$ *forms a complete lattice.*

Fig. 1.9 illustrates the Knaster-Tarski theorem. The fixpoints of f are shown as stars.

Exercise Consider the complete lattice and monotone function f of Fig. 1.8

1. Mark the pre-fixpoints with up-triangles ($\triangle$).
2. What is the lub of the pre-fixpoints?
3. Mark post-fixpoints with down-triangles ($\triangledown$).

The fixpoints are the stars (⧖). $\square$

We now prove the Knaster-Tarski theorem. Let $(D, \leq)$ be a complete lattice and $f : D \rightarrow D$ be a monotone function on D. We first prove part (a) of the theorem. Let *Pre* denote the set of pre-fixpoints of f. Note that *Pre* is non-empty since $\perp$ is a pre-fixpoint of f. Let g be the lub of *Pre*, which is guaranteed to exist since $(D, \leq)$ is a complete lattice. We show that g is the greatest fixpoint of f.

We first show that g is a fixpoint of f (i.e. $f(g) = g$). We first argue that $g \leq f(g)$. It is sufficient to show that $f(g)$ is an upper bound of *Pre*, since then by virtue of g being the lub of *Pre*, we must have $g \leq f(g)$. To show that f is an upper bound of *Pre*, let x be any element of *Pre*. Thus x is a pre-fixpoint of f, and hence $x \leq f(x)$ (see Fig. 1.10(a)). Now, since $x \leq g$ (remember that g is the lub of *Pre*), by monotonicity of f we have $f(x) \leq f(g)$. By transitivity it follows that $x \leq f(g)$.

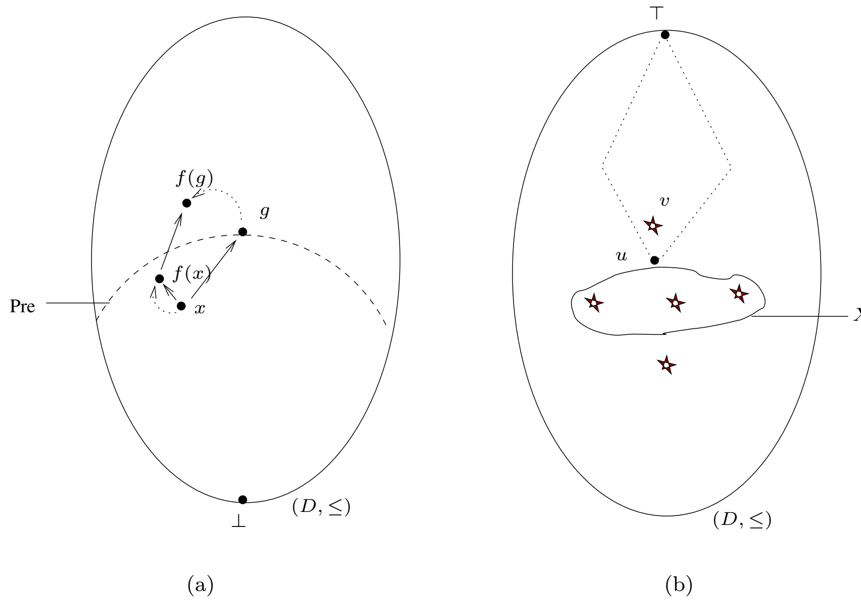


Figure 1.10: Proof of Knaster-Tarski theorm Part (a) and (b).

We now argue that $f(g) \leq g$. It is sufficient to show that $f(g)$ is a pre-fixpoint of f. We have just shown that g is itself a pre-fixpoint since $g \leq f(g)$. But then by monotonicity of f, we must have $f(g) \leq f(f(g))$. This completes the proof that g is a fixpoint of f.

We now argue that g must be the greatest fixpoint of f. Let h be any other fixpoint of f. Then in particular, h is a pre-fixpoint of f, and hence $h \in$ *Pre*, and hence $h \leq g$ which is the lub of *Pre*. Thus g dominates all other fixpoints of f, and hence is the greatest fixpoint of f.

In a similar manner one can show that the glb of the set *Post* of post-fixpoints of f is the least fixpoint of f.

Coming now to Part (b) of the theorem, Let P be the set of fixpoints of f, which we know to be non-empty by the first part of the theorem. We show that $(P, \leq)$ is a *complete* lattice. Firstly, $(P, \leq)$ forms a partial order, since restricting the ordering of a partial order to any subset of it maintains the partially ordered structure. To argue that $(P, \leq)$ is a complete lattice, we need to show that each subset of P has a lub and glb.

Let $X \subseteq P$. We first show there is an lub of X in $(P, \leq)$. Let u be the lub of X in $(D, \leq)$. Consider the "interval" of elements I between u and $\top$:

$$I = [u, \top] = \{x \in D \mid u \leq x\}.$$

Then it is easy to see that $(I, \leq)$ is also a complete lattice. Further, f restricted to I is a function on I, in the sense that for any $x \in I$, $f(x) \in I$. To see this, let $x \in I$, and consider $f(x)$. We will show $f(x)$ is an upper bound of the set X: this follows since for any $p \in X$, we have $p \leq x$ (since x dominates u which in turn dominates each $p \in X$), and hence $p = f(p) \leq f(x)$ by monotonicity of f. But since u is the lub of X, we must have $u \leq f(x)$, and hence $f(x) \in I$.

Further, $f : I \to I$ is a monotonic function on $(I, \leq)$. Hence, by Part (a) of the theorem, f has a least fixpoint in I, say v. We can now argue that v is the lub of X in $(P, \leq)$. To begin with v is clearly an upper bound of X since u is an upper bound of X and $u \leq v$. To see that v is the least of all fixpoints that are also upper bounds of X, let v' be any other fixpoint upper bound of X. Then v' must also be in I and hence a fixpoint of f in $(I, \leq)$. By choice of v, we must have $v \leq v'$.

In a similar way we can show that X has a glb as well. Thus $(P, \leq)$ forms a complete lattice. This completes the proof of part (b) of the Knaster-Tarski theorem. $\square$

1.4 Computing the LFP

- A *chain* in a partial order $(D, \leq)$ is a totally ordered subset of D.
- An *ascending chain* is an infinite sequence of elements of D of the form:

$$d_0 \leq d_1 \leq d_2 \leq \cdots.$$

- An ascending chain $\langle d_i \rangle$ is *eventually stable* if there exists n_0 such that $d_i = d_{n_0}$ for each $i \geq n_0$.
- $(D, \leq)$ has *finite* height if each chain is finite.
- $(D, \leq)$ has *bounded* height if there exists k such that each chain in D has height at most k (i.e. the number of elements in each chain is at most $k + 1$.)

Characterising lfp's and gfp's of a function f in a complete lattice $(D, \leq)$:

- f is *continuous* if for any ascending chain X in D,

$$f(\bigsqcup X) = \bigsqcup (f(X)).$$

- If f is *continuous* then

$$\mathit{lfp}(f) = \bigsqcup_{i \geq 0} (f^i(\bot)).$$

- If f is *monotonic* and $(D, \leq)$ has *finite height* then we can compute $\mathit{lfp}(f)$ by finding the stable value of the asc. chain

$$\bot \leq f(\bot) \leq f^2(\bot) \leq f^3(\bot) \leq \cdots.$$

Monotonicity, distributivity, and continuity:

- f is *monotone*:

$$x \leq y \Rightarrow f(x) \leq f(y).$$

- f is *distributive*:

$$f(x \sqcup y) = f(x) \sqcup f(y).$$

- f is *continuous*: For any asc chain X:

$$f(\bigsqcup X) = \bigsqcup (f(X)).$$

- f is *infinitely distributive*: For any $X \subseteq D$:

$$f(\bigsqcup X) = \bigsqcup (f(X)).$$

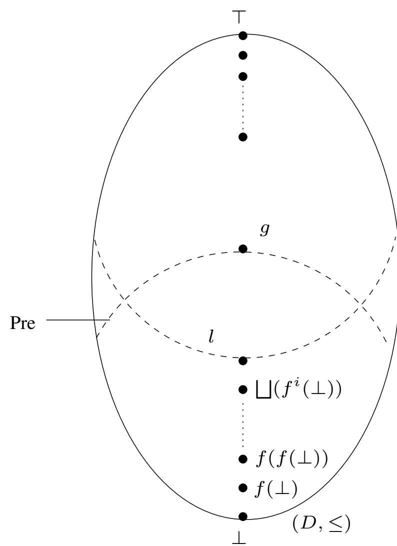


Figure 1.11: Computing the LFP.

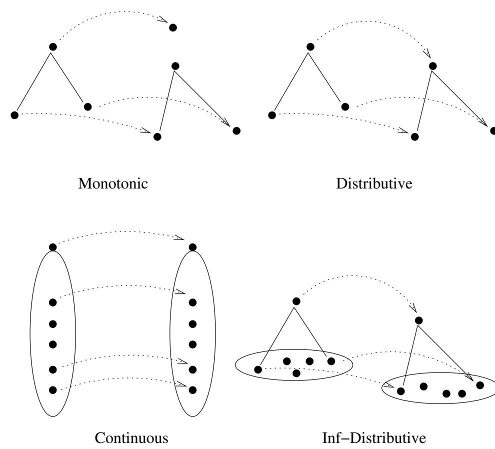


Figure 1.12: Illustrating the definitions of monotonicity, distributivity, continuity and infinite distributivity.

Chapter 2

Interprocedural Analysis

2.1 Motivation

In interprocedural data-flow analysis we are interested in analysing programs with procedures. As before we would like to come up with data-flow facts that safely over-approximate the set of program states that could arise at a program point via different executions of the program.

```
main() {          f() {          g() {
  x := 0;           x := x+1;      f();
  f();              return;        return;
  g();            }              }
  print x;
}
```

Figure 2.1: Example program with two procedures `f` and `g`.

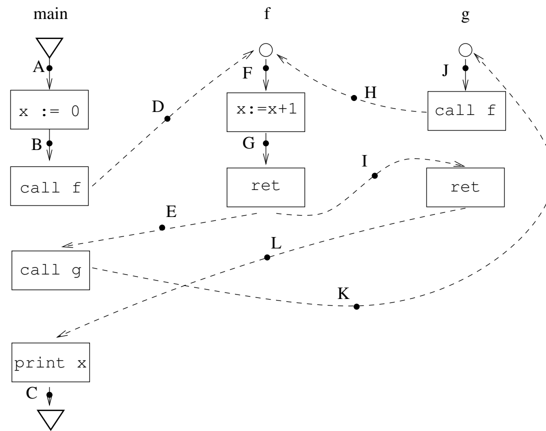


Figure 2.2: The extended CFG for the program of Fig. 2.1.

As a first attempt, we could extend a given data-flow analysis to programs with procedures as follows: To begin with, we build an extended CFG for the given program in which we add edges from call statements to the beginning of the called function (so-called "call edges"), and edges from return statements to the "return sites" in the calling procedure (the so-called "return edges"). These new edges can be given appropriate transfer functions. Fig. 2.1 shows an example program with procedures, and Fig. 2.2 shows the extended CFG for it. We can now compute the JOP for the given analysis in this extended control-flow graph.

The problem with this approach is that while it gives us a sound result, it is far too *imprecise* in general. To illustrate this, suppose we want to do a "collecting state" analysis for the example program of Fig. 2.1, starting with the initial abstract state $\{\mathtt{x} \mapsto 0\}$. Since there is only one (maximal) execution of the program corresponding to the path *ABDFGEKJHFGIL*, the collecting state at point C is simply $\{\mathtt{x} \mapsto 2\}$. However, if we compute the JOP for this analysis, we will get the infinite set of states $\{\mathtt{x} \mapsto 0, \mathtt{x} \mapsto 1, \mathtt{x} \mapsto 3, \ldots\}$ at C. This is because there are several paths (in this case infinitely many) that *don't* correspond to any execution of the program, for example the path *ABDFGEKJHFGEKJHFGIL* (which produces the state $\mathtt{x} \mapsto 3$). The first path is *interprocedurally valid*, while the second is *interprocedurally invalid*, in the sense that it takes the return edge E when it should have taken I given that the last pending call was H. The path *ABDFGILC* is another interprocedurally invalid path which produces the state $\mathtt{x} \mapsto 1$.

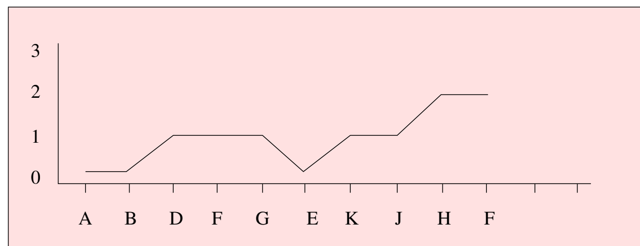


Figure 2.3: A path $\rho = ABDFGEKJHF$ in $IVP_{G'}$ for the example program in Fig. 2.2. The call-string $cs(\rho)$ associated with ρ is KH. For $\rho = ABDFGEK$ we have $cs(\rho) = K$, and for $\rho = ABDFGE$ we have $cs(\rho) = \epsilon$.

Instead, what we would like to compute is the "Join over Interprocedurally Valid Paths" (JVP) where we consider only interprocedurally valid paths that reach the given program point.

2.2 Interprocedurally valid paths

We define the JVP more formally in this section. We begin by defining interprocedurally valid paths and their associated "call-strings." Informally, a path ρ in the extended CFG G' is inter-procedurally valid if every return edge in ρ "corresponds" to the most recent "pending" call edge. For example, in the example program the return edge E *corresponds* to the call edge D. The call-string of an interprocedurally valid path ρ is a subsequence of call edges which have not "returned" as yet in ρ. For example, the call-string associated with the path *ABDFGEKJHF*, written $cs(ABDFGEKJHF)$, is "KH". Fig. 2.3 shows an interprocedurally valid path in the extended CFG of Fig. 2.2. The y-axis plots the number of pending calls for each prefix of the path.

Definition 1 (Interprocedurally valid paths and their call-strings) *Let ρ be a path in an extended CFG G'. We define when ρ is* interprocedurally valid *(and we say $\rho \in IVP(G')$) and what is its* call-string $cs(\rho)$, *by induction on the length of ρ.*

- *If $\rho = \epsilon$ then $\rho \in IVP(G')$. In this case $cs(\rho) = \epsilon$.*
- *If $\rho = \rho' \cdot N$ then $\rho \in IVP(G')$ iff $\rho' \in IVP(G')$ with $cs(\rho') = \gamma$ say, and one of the following holds:*
  1. *N is neither a call nor a ret edge.*
     In this case $cs(\rho) = \gamma$.
  2. *N is a call edge.*
     In this case $cs(\rho) = \gamma \cdot N$.
  3. *N is ret edge, and γ is of the form $\gamma' \cdot C$, and N corresponds to the call edge C.*
     In this case $cs(\rho) = \gamma'$.
- *We denote the set of (potential) call-strings in G' by Γ. Thus $\Gamma = \mathcal{C}^*$, where $\mathcal{C}$ is the set of call edges in G'.*

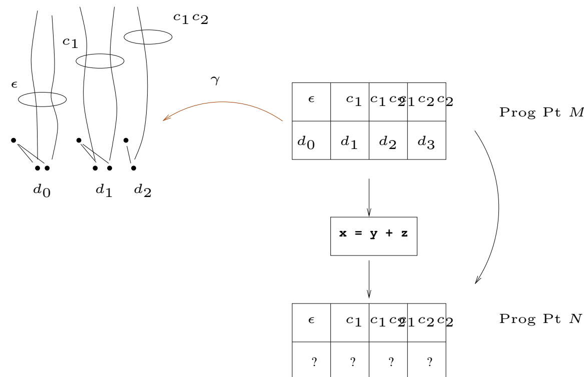


Figure 2.4: The meaning of a call-string table

Definition 2 (Join over interprocedurally-valid paths (JVP)) *Let* $\mathcal{A} = ((D, \leq), f_{MN}, d_0)$ *be a given abstract interpretation and let* G' *be an extended CFG. Let* $path_{I,N}(G')$ *be the set of paths from the initial point* I *to point* N *in* G'*. Then we define the* join over all interprocedurally valid paths *(JVP) at point* N *in* G' *to be:*

$$\bigsqcup_{\rho \in path_{I,N}(G') \cap IVP(G')} f_\rho(d_0).$$

2.3 Call-Strings approach

We now describe the first of several approaches to interprocedural analysis proposed by Sharir and Pnueli [1]. One approach to obtain the JVP is to find the JOP over same graph, but modify the abstract interpretation. We can modify the transfer functions for call/ret edges to detect and invalidate (interprocedurally) invalid edges. We need to augment underlying data values with some information for this. Natural thing to try is “call-strings”.

The abstract data elements of the call-string analysis will be maps (or a “table”) from call-strings to abstract data values of the underlying analysis. A call-string table ξ at program point N represents the fact that, for each call-string γ, there are some (initial) paths with call-string γ reaching N, and the join of the abstract states (obtained by propagating d_0) along these paths is dominated by $\xi(\gamma)$. This meaning is illustrated in Fig. 2.4. It will be useful to keep this meaning in mind, while designing the transfer functions of the call-string analysis.

The overall plan is to define an abstract interpretation $\mathcal{A}'$ which extends the given abstract interpretation, say $\mathcal{A}$, with call-string data. We then show that the JOP of $\mathcal{A}'$ on G' coincides with the JVP of $\mathcal{A}$ on G'. We could use Kildall (or any other technique) to compute the LFP of $\mathcal{A}'$ on G'. This value is guaranteed to over-approximate the JVP of $\mathcal{A}$ on G'.

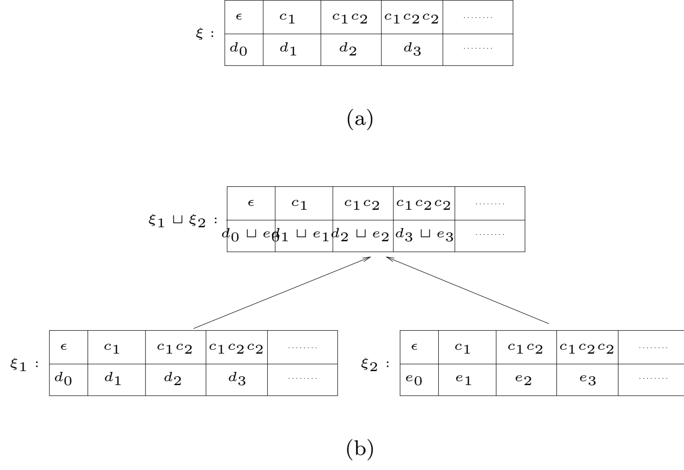


Figure 2.5: The call-string-tagged data values and their join.

ξ_0 :

ϵ	c_1	c_1c_2	$c_1c_2c_2$	....
d_0	$\bot$	$\bot$	$\bot$	....

Figure 2.6: The intial value of the call-string-tagged analysis.

The call-string-tagged abstract interpretation $\mathcal{A}'$ is defined as follows. The lattice is $(D', \leq')$ where elements of D' are maps $\xi : \Gamma \to D$. The ordering $\leq'$ on D' is the pointwise extension of $\leq$ in D. That is $\xi_1 \leq' \xi_2$ iff for each $\gamma \in \Gamma$, $\xi_1(\gamma) \leq \xi_2(\gamma)$. This induces a join operation that is the point-wise join of the table entries, and is illustrated in Fig. 2.5. It is easy to check that $(D', \leq')$ is also a complete lattice.

The initial value ξ_0 of the analysis $\mathcal{A}'$ Initial value ξ_0 is given by

$$\xi_0(\gamma) = \begin{cases} d_0 & \text{if } \gamma = \epsilon \\ \bot & \text{otherwise.} \end{cases}$$

It is illustrated in Fig. 2.6

The transfer functions of the analysis $\mathcal{A}'$ are given as follows.

- Transfer functions for non-call/ret edge N:

$$f'_{MN}(\xi) = f_{MN} \circ \xi.$$

- Transfer functions for call edge N:

$$f'_{MN}(\xi) = \lambda\gamma. \begin{cases} \xi(\gamma') & \text{if } \gamma = \gamma' \cdot N \\ \bot & \text{otherwise} \end{cases}$$

- Transfer functions for ret edge N whose corresponding call edge is C:

$$f'_{MN}(\xi) = \lambda\gamma.\xi(\gamma \cdot C)$$

Note that the transfer functions f'_{MN} are monotonic (distributive) if each f_{MN} is monotonic (distributive). Hence $\mathcal{A}'$ forms a valid abstract interpretation.

ϵ	c_1	$cs(\rho)$	$c_1 c_2 c_2$	....
$\perp$	$\perp$	d	$\perp$	....

Figure 2.7: Proving correctness of the call-string analysis.

Example: Transfer functions f'_{MN} for example program

- Non-call/ret edge B:
$$\xi_B = f_{AB} \circ \xi_A.$$
- Call edge D:
$$\xi_D(\gamma) = \begin{cases} \xi_B(\gamma') & \text{if } \gamma = \gamma' \cdot D \\ \perp & \text{otherwise} \end{cases}$$
- Return edge E:
$$\xi_E(\gamma) = \xi_G(\gamma \cdot D).$$

Exercise

1. Let $\mathcal{A}$ be the standard collecting state analysis, and consider the program of Fig. 2.2. For brevity, represent a set of concrete states as $\{0, 1\}$ (meaning the 2 concrete states $x \mapsto 0$ and $x \mapsto 1$). Assume an initial value $d_0 = \{0\}$.

Show the call-string tagged abstract states (in the lattice $\mathcal{A}'$) along the paths

(a) ABDFGEKJHFGIL (interprocedurally valid)
(b) ABDFGIL (interprocedurally invalid).

2. Use Kildall's algo to compute the LFP of the $\mathcal{A}'$ analysis for the example program of Fig. 2.2. Start with initial value $d_0 = \{0\}$.

$\square$

2.4 Correctness of call-strings approach

Without loss of generality, we assume that the transfer functions of the underlying analysis satisfy the property that the $\perp$ element of the $(D, \leq)$ lattice is mapped to $\perp$ (i.e. $f_{MN}(\perp) = \perp$ for each f_{MN}).

Theorem 2 *Let N be a point in an extended graph G'. Then*

$$JVP_{\mathcal{A}}(N) = \bigsqcup_{\gamma \in \Gamma} JOP_{\mathcal{A}'}(N)(\gamma).$$

Proof. Use following lemmas to prove that LHS dominates RHS and vice-versa. $\square$

Lemma 3 *Let ρ be a path in $IVP_{G'}$. Then*

$$f'_\rho(\xi_0) = \lambda\gamma. \begin{cases} f_\rho(d_0) & if \gamma = cs(\rho) \\ \perp & \text{otherwise.} \end{cases}$$

Proof. By induction of length of ρ. $\square$

Lemma 4 *Let ρ be a path* not in *$IVP_{G'}$. Then*

$$f'_\rho(\xi_0) = \lambda\gamma.\perp.$$

ϵ	c_1	c_2	$c_1 c_2 c_2$	....
$\perp$	$\perp$	$\perp$	$\perp$	....

Proof. Since ρ is invalid, it must be the case that ρ has an invalid prefix. Consider the smallest such prefix $\alpha \cdot N$. Then it must be the case that α is valid and N is a return edge not corresponding to $cs(\alpha)$. Using the previous lemma it follows that $f'_{\alpha \cdot N}(\xi_0) = \lambda\gamma.\perp$. But then all extensions of α along ρ must also have transfer function $\lambda\gamma.\perp$. $\square$

2.5 Computing JOP/LFP

The problem with the call-strings approach above is that D' is infinite in general (even if D were finite). So we cannot use Kildall's algo to compute an over-approximation of JOP. In this section we give two methods to *bound* the number of call-strings. The first uses "approximate" call-strings, while the second uses a safe bound on length of call-strings needed.

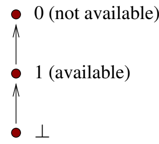


Figure 2.8: Lattice for available expression analysis.

Consider only call-strings of upto length $\leq l$, that may additionally be prefixed by a "$*$". A "$*$" prefix means that we have left out some initial calls. For example, for $l = 2$, call strings can be of the form "c_1c_2" or "$*c_1c_2$" etc. So each table ξ is now a finite table.

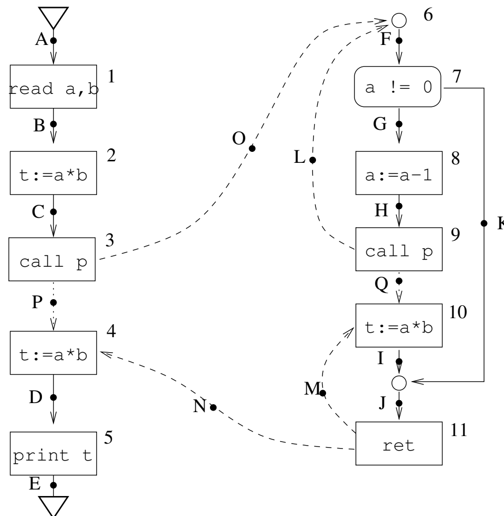
The transfer functions for non-call/ret edges remain same. For a call edge C: Shift γ entry to $\gamma \cdot C$ if $|\gamma \cdot C| \leq l$; else shift it to $* \cdot \gamma' \cdot C$ where γ is of the form $A \cdot \gamma'$, for some call A.

For a return edge N:

- If $\gamma = \gamma' \cdot C$ and N corresponds to call edge C, then shift $\gamma' \cdot C$ entry to γ' entry.
- If $\gamma = *$ then copy its entry to $*$ entry at the return site.

Exercise An expression (like `a*b` in the program below) is *available* at a point N in an execution ρ if there is a point before N in ρ where the expression is computed and since then till N none of its constituent variables (like `a` or `b` in the example expression above) are written to. In an "available expression" analysis, we want to say whether an expression is available at a given program point (meaning that in all executions of the program reaching that point the expression is available), or not. If we are interested in the availability of a single expression, we could use a lattice like in Fig. 2.8.

Consider the program whose extended graph is shown below:



Is `a*b` available at program point N? Yes it is, if we consider interprocedurally valid paths only.

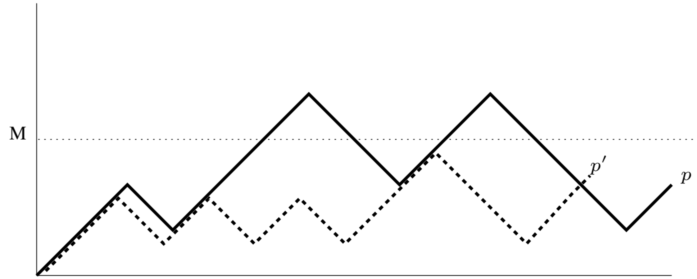


Figure 2.9: Paths with bounded call-strings

Do an interprocedural analysis for the availability of the expression `a*b`, using approximate call-strings (assume a length of 2) for the program below. Use Kildall's algo to compute the ξ table values representing the LFP of the analysis. Start with initial value $d_0 = 0$. □

2.5.1 Bounded call-string method

When the underlying lattice $(D, \leq)$ is finite, it is possible to bound the length of call strings Γ we need to consider.

For a number l, we denote the set of call strings (for the given program P) of length at most l, by Γ_l. Define a new analysis $\mathcal{A}''$ (called the bounded call-string analysis) in which the call-string tables have entries only for Γ_M for a certain constant M. We will show that $\text{JOP}(G', \mathcal{A}'') = \text{JOP}(G', \mathcal{A}')$. This is illustrated in Fig. 2.12.

Let k be the number of call sites in the given program P. In the example program of Fig.2.2 for example there are 3 call sites.

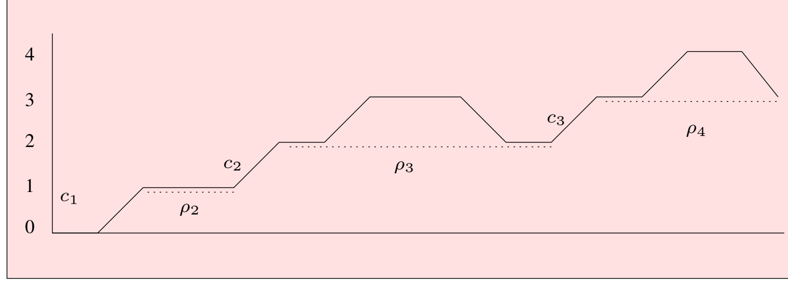
Lemma 5 *For any path p in $IVP(r_1, N)$ such that $|cs(q)| > M = k|D|^2$ for some prefix q of p, there is a path p' in $IVP_{\Gamma_M}(r_1, N)$ with $f_{p'}(d_0) = f_p(d_0)$.*

Proof. It is sufficient to prove that for any path p in $IVP(r_1, N)$ with a prefix q such that $|cs(q)| > M$, we can produce a *smaller* path p' in $IVP(r_1, N)$ with $f_{p'}(d_0) = f_p(d_0)$. Since, if $|p| \leq M$ then $p \in IVP_{\Gamma_M}$.

A path ρ in $IVP(r_1, n)$ can be decomposed as

$$\rho_1 \| (c_1, r_{p_2}) \| \rho_2 \| (c_2, r_{p_3}) \| \sigma_3 \| \cdots \| (c_{j-1}, r_{p_j}) \| \rho_j.$$

where each ρ_i $(i < j)$ is a *valid and complete* path from r_{p_i} to c_i, and ρ_j is a *valid and complete* path from r_{p_j} to n. Thus $c_1, \ldots, c_j$ are the unfinished calls at the end of ρ.



To prove the subclaim, Let p_0 be the first prefix of p where $|cs(p_0)| > M$. Let the decomposition of p_0 be

$$\rho_1 \| (c_1, r_{p_2}) \| \rho_2 \| (c_2, r_{p_3}) \| \sigma_3 \| \cdots \| (c_{j-1}, r_{p_j}) \| \rho_j.$$

Tag each unfinished-call c_i in p_0 by $(c_i, f_{q \cdot c_i}(d_0), f_{q \cdot c_i q' e_{i+1}}(d_0))$ where e_{i+1} is corresponding return of c_i *in* p. If there is no return for c_i in p tag it with $(c, f_{q \cdot c_i}(d_0), \bot)$.

The number of possible distinct such tags is $k \cdot |D|^2$. So there must be two calls qc and $qcq'c$ with same tag values. There are two cases: both calls don't return (that is their tag values are $\bot$) or they do return and abstract value at that point is d'. In both cases we can argue that we can cut out a portion of the path, as shown in the Figs.2.10 and 2.11, and preserve the final value computed along the paths.

□

We can argue that the LFP of $\mathcal{A}''$ is no less precise than that of $\mathcal{A}'$. Consider any fixpoint V' (a vector of tables) of $\mathcal{A}'$. Truncate each entry of V' to (call-strings of) length M, to get V''. Clearly V' dominates V''. Further, observe that V'' is a *post-fixpoint* of the transfer functions for $\mathcal{A}''$. By the Knaster-Tarski characterisation of LFP, we know that V'' dominates LFP($\mathcal{A}''$). This is illustrated in Fig. 2.12

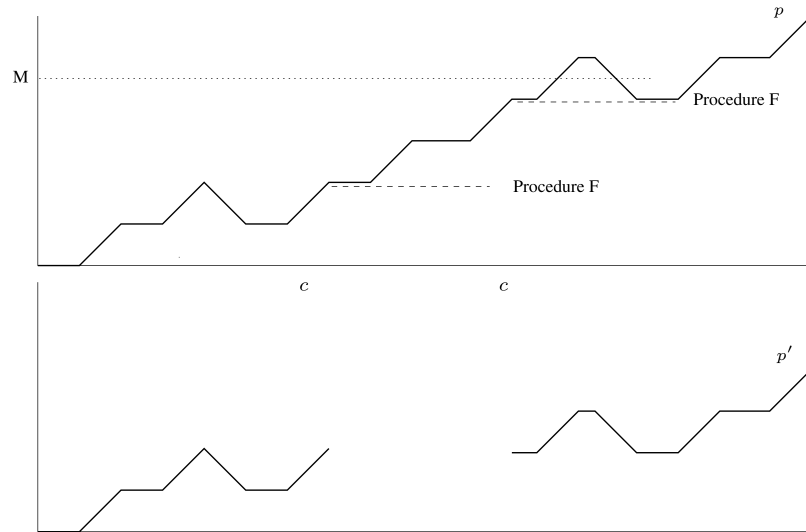


Figure 2.10: Proving subclaim – tag values are ⊥.

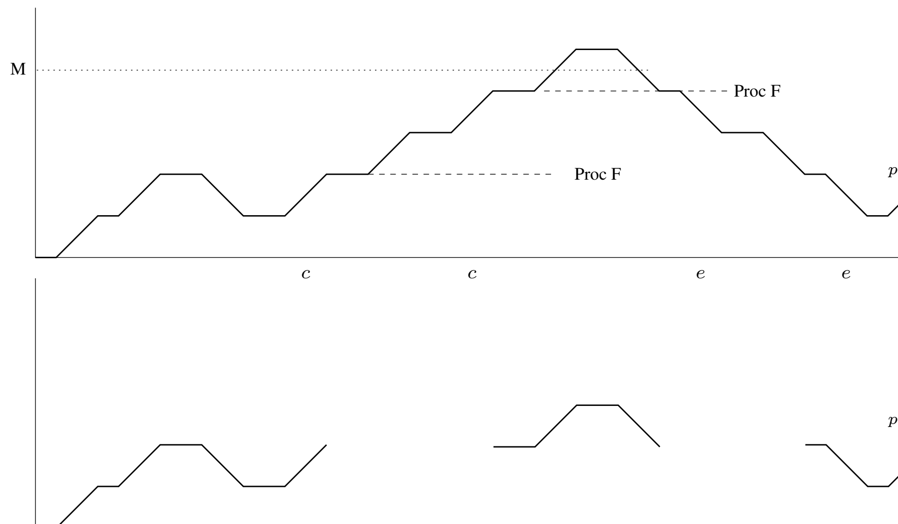


Figure 2.11: Proving subclaim – tag values are not ⊥.

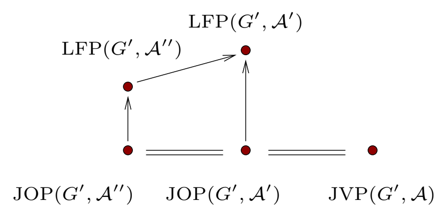


Figure 2.12: Relation between the JOP's and LFP of the different call-string based analyses.

Bibliography

[1] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In Steven S. Muchnick and Neil D. Jones, editors, *Program Flow Analysis: Theory and Applications*, chapter 7, pages 189–234. Prentice-Hall, Englewood Cliffs, NJ, 1981.

[2] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5:285–309, 1955.