Class Notes on The Mathematical Foundations of Iterative Program Analysis

Omer Tripp and Guy Ilany

March 28, 2011

Scope These notes present the mathematical underpinnings of iterative program analysis. This background enables declarative definition of the solution computed by an analysis, such that different solutions can be compared and the notion of a "best" (or "most exact") solution can be treated formally.

1 Lattice Theory

1.1 Posets

A *partially-ordered set* (poset) is a binary relation $\sqsubseteq$ over a set L that is reflexive, antisymmetric and transitive. For all a, b and c in L we have that:

- $a \sqsubseteq a$ (reflexivity);
- $a \sqsubseteq b \wedge b \sqsubseteq c \Rightarrow a \sqsubseteq c$ (transitivity); and
- $a \sqsubseteq b \wedge b \sqsubseteq a \Rightarrow a = b$ (antisymmetry).

We denote a poset by $\langle L, \sqsubseteq \rangle$.

Intuitively, a poset formalizes the intuitive concept of ordering the elements of a set. Here are some examples:

1. The total order according to weak inequality on natural numbers: $\langle \mathbb{N}, \leq \rangle$. (Note that strong inequality ($<$) does not induce a poset, since it is not reflexive.)
2. The powerset of set S, $\mathcal{P}(S)$, where $X \sqsubseteq Y \Leftrightarrow X \subseteq Y$: $\langle \mathcal{P}(S), \subseteq \rangle$.
3. The powerset of set S, $\mathcal{P}(S)$, where $X \sqsubseteq Y \Leftrightarrow X \supseteq Y$: $\langle \mathcal{P}(S), \supseteq \rangle$. In this case, $\bot = S$ and $\top = \emptyset$.

In program analysis, the ordering imposed on a set of elements corresponds to the accuracy of the analysis' solution:

$$
\begin{aligned}
&l_1 \sqsubseteq l_2 \\
&\qquad \Leftrightarrow \quad l_1 \text{ is more precise than } l_2 \\
&\qquad \Leftrightarrow \quad l_1 \text{ represents fewer concrete states than } l_2
\end{aligned}
$$

We saw the example of constant propagation, where the following poset is used:

- $L = \mathbb{N} \cup \{\bot, \top\}$; and
- $\sqsubseteq = \{\langle \bot, n \rangle \mid n \in \mathbb{N}\} \cup \{\langle n, \top \rangle \mid n \in \mathbb{N}\}$.

$\bot$ represents an unitialized value, and is thus the least element in $\langle L, \sqsubseteq \rangle$. $\top$ is used if the analysis concludes that a variable can be assigned more than one integral value, and thus represents the least precise solution. This notion of comparability is used to impose an ordering on program environments: Environment env_1 is smaller than environment env_2 if the value assigned to each variable in env_1 is smaller than its value in env_2. Consider for example a program with two variables, x and y. Then:

$$[x \mapsto \bot, y \mapsto \bot] \sqsubseteq [x \mapsto \bot, y \mapsto 7] \sqsubseteq [x \mapsto 2, y \mapsto 7] \sqsubseteq [x \mapsto 2, y \mapsto \top] \sqsubseteq [x \mapsto \top, y \mapsto \top].$$

In what follows, we use the following notations:

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

1.2 Upper and Lower Bounds

Given poset $\langle L, \sqsubseteq \rangle$ and subset L' of L ($L' \subseteq L$), we say that $l \in L$ is a *lower bound* of L' if

$$\forall l' \in L'.\ l \sqsubseteq l'.$$

Analogously, we say that $u \in L$ is an upper bound of L' if

$$\forall l' \in L'.\ l' \sqsubseteq u.$$

$l_0 \in L$ is a *greatest lower bound* of $L' \subseteq L$ if (i) l_0 is a lower bound of L', and (ii) for any lower bound l of L', $l \sqsubseteq l_0$. Analogously, $u_0 \in L$ is a *least upper bound* of L' if (i) u_0 is an upper bound of L', and (ii) for any upper bound u of L', $u_0 \sqsubseteq u$.

Lemma Given poset L, for every subset L' of L:

- If the greatest lower bound of L' exists, then it is unique. In this case, we denote it by $\sqcap L'$, and refer to it as the *meet* over L'.
- If the least upper bound of L' exists, then it is unique. In this case, we denote it by $\sqcup L'$, and refer to it as the *join* over L'.

Proof We show the proof for the greatest lower bound. The proof for the least upper bound is analogous. Let L be a poset and $L' \subseteq L$. Assume that u and u' both satisfy (i) and (ii) above. Then because u is a lower bound and u' is a greatest lower bound, (ii) prescribes that $u \sqsubseteq u'$. Symetrically, since u' is a lower bound and u is a greatest lower bound, (ii) prescribes that $u' \sqsubseteq u$. We now use the fact that L is a poset, and thus $\sqsubseteq$ is antisymmetrical, to conclude that $u \sqsubseteq u' \wedge u' \sqsubseteq u \Rightarrow u = u'$. This concludes our proof.

In what follows, we sometimes abuse notation and write $a \sqcap b$ (*resp.* $a \sqcup b$) to refer to the meet $\sqcap\{a, b\}$ (*resp.* join $\sqcup\{a, b\}$) of two elements, a and b.

1.3 Complete Lattices

Poset $\langle L, \sqsubseteq\rangle$ is a *complete lattice* if every subset of L has a least upper bound as well as a greatest upper bound. We denote a complete lattice as a tuple $\langle L, \sqsubseteq, \sqcup, \sqcap, \bot, \top\rangle$, where

- $\bot = \sqcup\emptyset = \sqcap L$; and
- $\top = \sqcup L = \sqcap\emptyset$.

The least element is the greatest lower bound over all the elements in L, and thus $\forall l \in L.\ \bot \sqsubseteq l$. Similarly, the greatest element is the least upper bound over all the elements in L, and thus $\forall l \in L.\ l \sqsubseteq \top$.

The examples we gave above of posets ($\langle \mathbb{N}, \leq\rangle$, $\langle \mathcal{P}(S), \subseteq\rangle$, $\langle \mathcal{P}(S), \supseteq\rangle$, and the constant-propagation poset) are all also complete lattices. However, if we omit $\top$ from the constant-propagation lattice and consider $L = \mathbb{N} \cup \{\bot\}$ and $\sqsubseteq = \{\langle \bot, n\rangle \mid n \in \mathbb{N}\}$ instead, then $\langle L, \sqsubseteq\rangle$ is no longer a complete lattice. For example, the least upper bound of $\mathbb{N}_{odd}$ does not exist.

The following claim asserts that there is redundancy in the above definition of a complete lattice, in that it suffices to require the existence of either a least upper bound or a greatest lower bound for every subset of L:

Claim Let $\langle L, \sqsubseteq\rangle$ be a poset. Then

$$\begin{array}{cl} & L \text{ is a complete lattice} \\ \Leftrightarrow & \text{every subset of } L \text{ has a least upper bound} \\ \Leftrightarrow & \text{every subset of } L \text{ has a greatest lower bound.} \end{array}$$

Proof We prove the equivalence between the first and second conditions. The equivalence between the first and third conditions is proved analogously. Since the first condition immediately entails the second condition, we need only prove that if every subset of L has a least upper bound, then L is a complete lattice.

Thus, we assume the second condition, and let L' be a subset of L. We shall show that L' has a greatest lower bound. Consider set

$$S = \{l \in L \mid \forall l' \in L'.\ l \sqsubseteq l'\}\ .$$

By definition, S is the set of all lower bounds of L'. Now, since the second condition holds for L, the join over S, $u = \sqcup S$, is defined. We shall prove that u is also the greatest lower bound of L':

- First, we prove that u is a lower bound of L by choosing some arbitrary $l \in L$ and proving that $l \sqsupseteq u$. Observe that $\forall s \in S.s \sqsubseteq l$, since each $s \in S$ is not greater than all the elements in L, and in particular, l. This implies that l is an upper bound of S. Hence, by definition of the least upper bound, it holds that u is not greater than any other upper bound of S, and thus $u \sqsubseteq l$.

- Next, we observe that u is the greatest lower bound of L'. This follows from the fact that u is the least upper bound of S, and thus an upper bound of S, which implies that $\forall s \in S.\ s \sqsubseteq u$. That is, there is no lower bound of L' that is greater than u.

Based on these two properties of u, we conclude that u is the greatest lower bound of L', which completes our proof.

1.4 Constructors for Complete Lattices

In many program analyses, we are interested in the approximation of multiple values at multiple program locations. For example, in constant propagation, our objective is to compute a safe approximation of the (integral) values assigned to *each* variable at *each* program location. This motivates the following constructors for complete lattices:

Cartesian product Let $\langle L_1, \sqsubseteq_1 \rangle = \langle L_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1 \rangle$ and $\langle L_2, \sqsubseteq_2 \rangle = \langle L_2, \sqsubseteq_2, \sqcup_2, \sqcap_2, \bot_2, \top_2 \rangle$ be complete lattices. Then poset $L = \langle L_1 \times L_2, \sqsubseteq \rangle$ where

$$\langle x_1, y_1 \rangle \sqsubseteq \langle x_2, y_2 \rangle \Leftrightarrow x_1 \sqsubseteq x_2 \wedge y_1 \sqsubseteq y_2$$

is also complete lattice, as we now prove formally. (Note that $\sqsubseteq$ is overloaded. We disambiguate which overload is used where needed.)

Assuming that L_1 and L_2 are complete lattices, we show for $L = \langle L_1 \times L_2, \sqsubseteq \rangle$ that (i) L is a poset, and (ii) every subset of L has a least upper bound. For (i), we show that $\sqsubseteq$ is reflexive, transitive and antisymmetric:

- **Reflexivity** $\langle x, y \rangle \sqsubseteq_L \langle x, y \rangle \Leftrightarrow x \sqsubseteq_{L_1} x \wedge y \sqsubseteq_{L_2} y \Leftrightarrow T$, since both $\sqsubseteq_{L_1}$ and $\sqsubseteq_{L_2}$ are reflexive.

- **Transitivity** Assume that $\langle x_1, x_2 \rangle \sqsubseteq_L \langle y_1, y_2 \rangle$ and $\langle y_1, y_2 \rangle \sqsubseteq_L \langle z_1, z_2 \rangle$. This implies that

  $x_1 \sqsubseteq_{L_1} y_1 \wedge y_1 \sqsubseteq_{L_1} z_1$, and thus, from the transitivity of L_1: $x_1 \sqsubseteq_{L_1} z_1$
  $x_2 \sqsubseteq_{L_2} y_2 \wedge y_2 \sqsubseteq_{L_2} z_2$, and thus, from the transitivity of L_2: $x_2 \sqsubseteq_{L_2} z_2$

  Now, since $x_1 \sqsubseteq_{L_1} z_1$ and $x_2 \sqsubseteq_{L_2} z_2$, we conclude that $\langle x_1, x_2 \rangle \sqsubseteq_L \langle z_1, z_2 \rangle$.

- **Antisymmetry** Assume that (1) $\langle x_1, y_1 \rangle \sqsubseteq_L \langle x_2, y_2 \rangle$ and (2) $\langle x_2, y_2 \rangle \sqsubseteq_L \langle x_1, y_1 \rangle$. Then

$$
\begin{array}{ll}
x_1 \sqsubseteq_{L_1} x_2 \text{ (by (1))} \wedge x_2 \sqsubseteq_{L_1} x_1 \text{ (by (2))} & \Rightarrow x_1 = x_2 \text{ (by the antisymmetry of } L_1) \\
y_1 \sqsubseteq_{L_2} y_2 \text{ (by (1))} \wedge y_2 \sqsubseteq_{L_2} y_1 \text{ (by (2))} & \Rightarrow y_1 = y_2 \text{ (by the antisymmetry of } L_2),
\end{array}
$$

which implies $\langle x_1, x_2 \rangle = \langle z_1, z_2 \rangle$.

For (ii), let $X \subseteq L$. We define $X_1 = \{l_1 \in L_1 \mid \exists l_2 \in L_2. \langle l_1, l_2 \rangle \in X\}$ and $X_2 = \{l_2 \in L_2 \mid \exists l_1 \in L_1. \langle l_1, l_2 \rangle \in X\}$. Then the least upper bounds of X_1 and X_2, $x_1 = \sqcup X_1$ and $x_2 = \sqcup X_2$, are well defined. We claim that $x = \langle x_1, x_2 \rangle$ is the least upper bound of X. First, we observe that x is an upper bound of X. Now, let $y = \langle y_1, y_2 \rangle$ be an upper bound of X. Then

$$
\begin{array}{ll}
\forall z = \langle l_1, l_2 \rangle \in X. z \sqsubseteq y & \Rightarrow \\
\quad l_1 \sqsubseteq y_1 \wedge l_2 \sqsubseteq y_2 & \Rightarrow \\
\quad y_1 \text{ is an upper bound of } X_1 \wedge y_2 \text{ is an upper bound of } X_2 & \Rightarrow \\
\quad y_1 \sqsupseteq x_1 \wedge y_2 \sqsupseteq x_2 & \Rightarrow \\
\quad y \sqsupseteq x. &
\end{array}
$$

This concludes our proof.

Using the cartesian-product constructor, we can simultaneously represent the constant-propagation solution for multiple variables. Each component in the cartesian product corresponds to a specific variable in the program. We still need, however, to distinguish between program locations. Hence the following.

Finite maps Let $\langle L_1, \sqsubseteq_1 \rangle = \langle L_1, \sqsubseteq_1, \sqcup_1, \sqcap_1, \bot_1, \top_1 \rangle$ be a complete lattice and V a finite set. Then poset $L = \langle V \to L_1, \sqsubseteq \rangle$ where

$$e_1 \sqsubseteq e_2 \Leftrightarrow \forall v \in V.\ e_1\ v \sqsubseteq e_2\ v \tag{1}$$

is a complete lattice, as we now prove.

To show that L is indeed a complete lattice, we need to show that (i) L is a poset and (ii) every subset of L has a least upper bound. We leave (i) as a simple exercise, and move straight to (ii). Let $X \subseteq L$. For each $v \in V$, we define $X_v = \{l \in L_1 \mid \exists e \in X. e\ v = l\}$. Then $x_v = \sqcup X_v$ is well defined. We claim that $e = \lambda v \in V.\ x_v$ is the least upper bound of X. First, note that e is an upper bound of X. Next, let e' be an upper bound of X. Then given $v \in V$, $\forall e'' \in X. e'\ v \sqsupseteq e''\ v$, and thus $e'\ v$ is an upper bound of X_v, which implies that $e'\ v \sqsupseteq x_v$. Thus, according to (1), $e' \sqsupseteq e$. We conclude that e is the least upper bound of X, which concludes our proof.

In constant propagation, V represents the (finite) set of program variables. Every program variable is mapped via the solution computed by the constant-propagation analysis to an abstract value. Since L is a poset, we can compare between different constant-propagation solutions according to the condition in (1).

2 Fixed Points

Iterative program analysis relies a notion of convergence, which we now treat formally.

2.1 Chains

Let $L = \langle L, \sqsubseteq \rangle$ be a poset. Then subset Y of L is a *chain* if every two elements in Y are ordered:

$$\forall l_1, l_2 \in Y.\ l_1 \sqsubseteq l_2 \vee l_2 \sqsubseteq l_1.$$

Sequence $\langle l_1, l_2, \ldots \rangle$ of values is considered an *ascending chain* (*resp. descending chain*) if $l_1 \sqsubseteq l_2 \sqsubseteq \ldots$ (*resp.* $l_1 \sqsupseteq l_2 \sqsupseteq \ldots$). $\langle l_1, l_2, \ldots \rangle$ is a *strictly ascending chain* (*resp. strictly descending chain*) if $l_1 \sqsubset l_2 \sqsubset \ldots$ (*resp.* $l_1 \sqsupset l_2 \sqsupset \ldots$).

We say that poset $L = \langle L, \sqsubseteq \rangle$ has a *finite height* if every chain in L is finite. For example, the constant-propagation lattice has a finite height of 3. Every maximal strictly ascending chain there is of the form: $\langle \bot, n, \top \rangle$ $(n \in \mathbb{N})$.

Lemma Poset $L = \langle L, \sqsubseteq \rangle$ has a finite height iff every strictly ascending and strictly descending chain in L is finite.

Proof We need only show that if every strictly ascending and strictly descending chain in L is finite, then L has a finite height. Let us falsely assume that this is not the case. Then by definition, there is an infinite chain $Y = \langle l_1, l_2, \ldots \rangle$ in L. Since $\sqsubseteq$, when restricted to the elements in Y, is a total order, we can transform Y into a strictly ascending chain $Y' = \langle l_{i_1}, l_{i_2}, \ldots \rangle$ (where $l_{i_1} \sqsubset l_{i_2} \sqsubset \ldots$), contrary to our assumption that there are no infinite strictly ascending chains in L. This concludes our proof.

Intuitively, chains bound the amount of iterations that an iterative analysis would perform in the worst case: If all the chains are finite, then the height of the poset is finite, and so the analysis is guaranteed to converge at some point assuming that it behaves monotonically, which brings us to our next definition.

2.2 Monotone Functions

Let $L = \langle L, \sqsubseteq \rangle$ be a poset. Function $f \colon L \to L$ is *monotone* if

$$\forall l_1, l_2 \in L.\ l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2).$$

A simple class of monotone functions is those mapping all values in L is a constant value: $f_k \colon L \to L = \lambda l \in L.\ k$. f_k is monotone since $\forall l_1 \sqsubseteq l_2 \in L.\ f_k(l_1) = f_k(l_2) \Rightarrow f_k(l_1) \sqsubseteq f_k(l_2)$.

An example we saw in program analysis is constant propagation, where the transfer functions defined on the edges of the program's control-flow graph are monotone. Here are some examples:

- **Nop** The transfer function corresponding to a `skip` statement is the identity function, $id = \lambda e.\ e$. This function is visibly monotone, since $e_1 \sqsubseteq e_2 \Rightarrow id(e_1) = e_1 \sqsubseteq id(e_2) = e_2$.
- **Assignment** The transfer function corresponding to assignment statement `x=2` is $ass = \lambda e.\ e[x \mapsto 2]$. Consider environments e_1 and e_2, such that $e_1 \sqsubseteq e_2$. Then $ass(e_1) \sqsubseteq ass(e_2)$, since

$$\forall v \in Var \setminus \{x\}.\ ass(e_1)\ v \sqsubseteq ass(e_2)\ v \wedge ass(e_1)\ x = ass(e_2)\ x.$$

2.3 Tarski's Theorem

We say that $l \in L$ is a *fixed point* of function $f\colon L \to L$ if $f(l) = l$.
(Notice that in the general case, the least fixed point (greatest fixed point) does not allways exist, but when L is a complete Lattice, both allways exist.)
We are interseted in finding the *least* and *greatest* fixed points of f.

For example:

$f : \mathcal{P}(U) \to \mathcal{P}(U)$	Least Fixed Point	Greatest Fixed Point
$f(X) = X$	$\emptyset$	U
$f(X) = \emptyset$	$\emptyset$	$\emptyset$
$f(X) = X \setminus A$	$\emptyset$	$U \setminus A$
$f(X) = X \cup A$	A	U

Another simple example of Constant Propogation is sketched below:

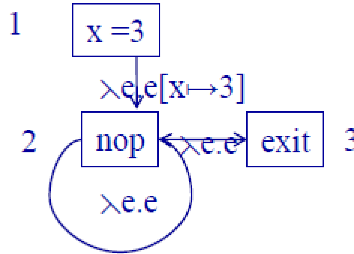

Figure 1: A simple constant propogation Control Graph

This simple example has 3 states $CP1, CP2, CP3$ (each of which contains a single variable x), with 3 transitions:

- $CP1 = [X \longmapsto 0]$
- $CP2 = CP1[X \longmapsto 0] \sqcup\ CP2$
- $CP3 = CP3$

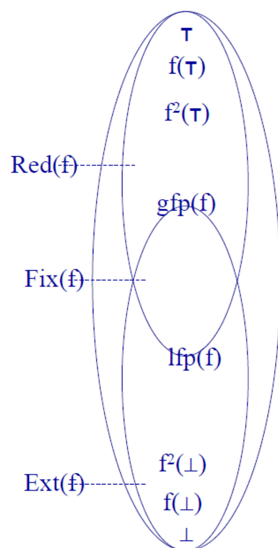

Figure 2: Visualization of Tarski's theorem

So we can look at f as a function that operates on a triplet of states and returns a triplet of states. A least fixed point of this function needs to be least fixed point simultaneously in SP1, SP2 and SP3.

It is easy to see that:

$(SP1, SP2, SP3)$	is a fixed point	is lfp
$(0, 3, 3)$	Yes	Yes
$(0, \top, \top)$	Yes	No
$(\top, \top, \top)$	No	No
$(0, \bot, \top)$	No	No

Given monotone function $f : L \to L$, where $L = \langle L, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ is a complete lattice, we define:

- $Fix(f) = \{l \mid l \in L.\ f(l) = l\}$ (the *fixed-points* set)
- $Red(f) = \{l \mid l \in L.\ f(l) \sqsubseteq l\}$ (the *reductive* set)
- $Ext(f) = \{l \mid l \in L.\ l \sqsubseteq f(l)\}$ (the *extensive* set)

Tarski's Theorem, 1955 Let $L = \langle L, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ be a complete lattice, and $f : L \to L$ a monotone function. Then the *least fixed point* and the *greatest fixed point* of f, denoted by $lfp(f)$ and $gfp(f)$ respectively, satisfy:

- $lfp(f) = \bigsqcap Fix(f) = \bigsqcap Red(f) \in Fix(f)$

- $gfp(f) = \bigsqcup Fix(f) = \bigsqcup Ext(f) \in Fix(f)$

A visualization of Tarski's theorem is presented in Figure 2.

Proof We prove the claim for $lfp(f)$. The proof for $gfp(f)$ is analogous. Let $a = \sqcap Red(f)$. Notice that a is well defined, since L is a lattice and $\{l \in L \mid f(l) \sqsubseteq l\} \subseteq L$, and so $a = \sqcap\{l \in L \mid f(l) \sqsubseteq l\}$ exists and is unique.

We show both $f(a) \sqsubseteq a$ and $a \sqsubseteq f(a)$, thus concluding that $a = f(a)$ (since $\sqsubseteq$ is antisymmetric). This implies that $a \in Fix(f)$. To see that $a = lfp(f)$, notice that since $\sqsubseteq$ is reflexive, it follows that $Fix(f) \sqsubseteq Red(f)$, and by the definition of a, we get that $a = \bigsqcap Fix(f)$ and is the least fixed point of f, thus proves Tarski's theorem.

We first show that $f(a) \sqsubseteq a$, so that $a \in Red(f)$: Since (i) $a \sqsubseteq l$ for all $l \in Red(f)$ and (ii) f is monotone, we have

$$\begin{aligned} \forall l \in Red(f).\ a \sqsubseteq l \Rightarrow & \quad (\textit{employing } f\textit{, which is monotone}) \\ f(a) \sqsubseteq f(l) \Rightarrow & \quad (\text{by definition, } \forall l \in Red(f). f(l) \sqsubseteq l) \\ f(a) \sqsubseteq l & \end{aligned}$$

Thus, $f(a)$ is a lower bound of $Red(f)$. Since a is the greatest lower bound of $Red(f)$, by definition, we conclude that $f(a) \sqsubseteq a$.

In the other direction, we observe that since f is monotone

$$f(a) \sqsubseteq a \Rightarrow f(f(a)) \sqsubseteq f(a),$$

which implies that $f(a) \in Red(f)$. Now, again, since a is the greatest lower bound of $Red(f)$, and in particular, a lower bound of $Red(f)$, we conclude that $a \sqsubseteq f(a)$, which completes our proof.

Computing $lfp(f)$ From the perspective of program analysis, $lfp(f)$ represents the most precise fixed-point solution. This solution may not be computable in general, but as Tarski's theorem shows, it is guaranteed to exist. We claim that the following (simple) algorithm computes $lfp(f)$ if it terminates:

$$\begin{aligned} & x = \bot \\ & \textit{while } f(x) \neq x \\ & \qquad x := f(x) \end{aligned}$$

A similar algorithm can be defined for computing $gfp(f)$, this time starting with $x = \top$.

First, note that if the algorithm terminates, then $x = f(x)$, meaning that the final value is in $Fix(f)$. In order to prove that the algorithm indeed converges on $lfp(f)$, we show that it satisfies a stronger property: At each point along the computation, x 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 of $Fix(f)$. Thus, in the i-th iteration:

$$\begin{aligned} \forall l \in Fix(f).\ x \sqsubseteq l \Rightarrow & \qquad (f \text{ is monotone}) \\ f(x) \sqsubseteq f(l) \Rightarrow & \qquad (l = f(l)) \\ f(x) \sqsubseteq l & \end{aligned}$$

and since the i-th iteration ends with the assignment $x := f(x)$, we conclude that after the i-th iteration, x is still a lower bound of $Fix(f)$.

Recall that this is true only if the algorithm terminates, which in general is not guaranteed. Notice, however, that if the lattice L over which f is defined has a finite height, then the algorithm is guaranteed to terminate.

2.4 Chaotic Iterations

Given a lattice $L = \langle L, \sqsubseteq, \sqcup, \sqcap, \bot, \top \rangle$ with finite strictly increasing chains: $L^n = L \times L \times L \times \cdots \times L$ and a monotone function $f : L^n \to L^n$, it is desirable to compute $lfp(f)$ (which as stated before, must exist). Notice that $lfp(f)$ is defined as the set of simultaneous least fixed points of each dimention: $\{x_i \mid x_i = lfp(f_i), i = 1 \dots n\}$

Thus if we have a program with n statements, and we wish to calculate the least fixed point of the program (which means calculating the least fixed point of every statement), we can simply look for the least fixed point of the function f ($f : L^n \to L^n$).

One naive way for searching is by using the generic algorithm we've seen earlier:

$$\begin{aligned} & \overline{x} = (\bot, \bot, \dots, \bot) \\ & \mathit{while}\ f(\overline{x}) \neq \overline{x} \\ & \qquad \overline{x} := f(\overline{x}) \end{aligned}$$

(Notice the slightly different vector form of the algorithm.)
This algorithm, in case of convergence, indeed finds $lfp(f)$, but is extremely inefficient, taking O(n) time at each step due to the vector form, so that if the height of the control graph is h, the number of steps until a solution is found could be $O(h * n^2)$, which is not good enough since it does not follow the control flow of the program and recomputes the program property associated with every control point at each iteration step. An improved algorithm should structure its traversal method to conservatively include only those steps that might influence the final outcome and to skip those that cannot exert any influence.

We now take a look at a simple scheme which basically breaks the operations on vectors to operating on a single variable (of a single dimention), and finds

the lfp of that dimention. From that point on, this dimention could be ignored while searching for the lfp of another dimention. This process continues until each dimention's lfp is found. Notice that the solution is obviously equal to that of the vector form, according to the definition.

The algorithm maintains a worklist (WL) which holds the indices (corresponding to statements), where lfp was not already found. Initially, since no computation has been made, the list contains all indices. As the algorithm progresses, an index is removed from the list and processed. Processing an index might result in other nodes 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 work that is done for each index is to update our knowledge of constants at a point in the program. If, when processing an index (this is done by applying the transfer function), changes are made to the following state, we will add to the list all the indices of those states that might be influenced. When the algorithm converges (so no more changes are made, and $f(\overline{x}) = \overline{x}$) it will terminate. The

```
for i :=1  to n do
    x[i] = ⊥
WL = {1, 2, …, n}
while (WL ≠  Ø )  do
  select and remove an element i ∈ WL
  new := f_i(x)
  if (new ≠ x[i]) then
       x[i] := new;
       Add all the indexes that directly depends on i to WL
```

Figure 3: Chaotic Iteration algorithm

general algorithm, in its vector form, is presented in Figure 3.

This algorithm is a generic scheme due to the selection made at each iteration. Although the final outcome is the same no matter which selection scheme is used, ordering has considerable impact on performance. This algorithm produces the minimum number of non-constants (due the least fixed point) and maximum number of $\perp$ due to the initialization stage.

The control-flow graph of a program is a natural way of representing dependencies between elements in the worklist. The indices that directly depend on

i correspond to the control-flow locations immediately succeeding the location represented by i (and i itself may be among them if it is a successor of itself).

For example, given the program:

$$
\begin{array}{l}
x := 0 \\
\textit{while } x \leq 10 \textit{ do begin}: \\
\quad y := 0 \\
\quad \textit{while } y \leq x \textit{ do}: \\
\quad \quad y := y + 1 \\
\textit{end}
\end{array}
$$

Figure 4 represents its control-flow graph, which discloses the dependencies

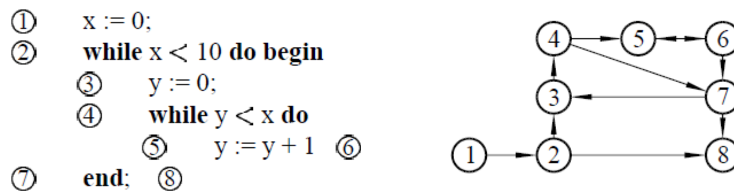

Figure 4: A cotrol-flow graph example

between (potential) worklist elements.

A similar algorithm that is adapted for the control-graph form, is presented in Figure 5. Notice that the worklist is initialized with the starting node s alone. This might result in fewer iterations, in case the control graph contains unreachable nodes.

Another representation of the algorithm, in both forms, is via a system of equations:

$$
S = \begin{cases} df_{entry}[s] = \tau \\ df_{entry}[v] = \bigsqcup\{f(u,v)(df_{entry}[u]) \mid (u,v) \in E\} \end{cases}
$$

```
Chaotic(G(V, E): Graph, s: Node, L: Lattice, ι: L, f: E →(L →L) )
  for each v in V to n do df_entry[v] := ⊥
  df[s] = ι
  WL = {s}
  while (WL ≠ ∅ ) do
    select and remove an element u ∈ WL
    for each v, such that. (u, v) ∈E do
        temp = f(e)(df_entry[u])
        new := df_entry(v)⊔ temp
        if (new ≠ df_entry[v]) then
            df_entry[v] := new;
            WL := WL ∪{v}
```

Figure 5: Chaotic Iteration algorithm, CFG form

for representing the CFG form, and:

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

for representing the vector form, where τ is an initial assigment of the start node s

The solution is the same in both representations, i.e. $lfp(S) = lfp(F_S)$. The number of equations is the same as the number of nodes in the graph, and the goal is a minimal solution for the system.

What follows is an example of the algorithm's operation on a familiar sample program. Figure 6 presents the CFG and transfer functions for the example, and Table 1 describes the state changes during the algorithm's operation.

Below is a short description of the algorithm's operation corresponding to the example presented in Figure 6 and Table 1:

1. The initial state.
2. Node 2 has been added to WL (node 1 was removed), and the state $df_{entry}[2]$ is updated with $z \mapsto 3$ according to the transfer function.
3. Node 3 has been added to WL (node 2 was removed), and the state $df_{entry}[3]$ is updated with $x \mapsto 1$ according to the transfer function.

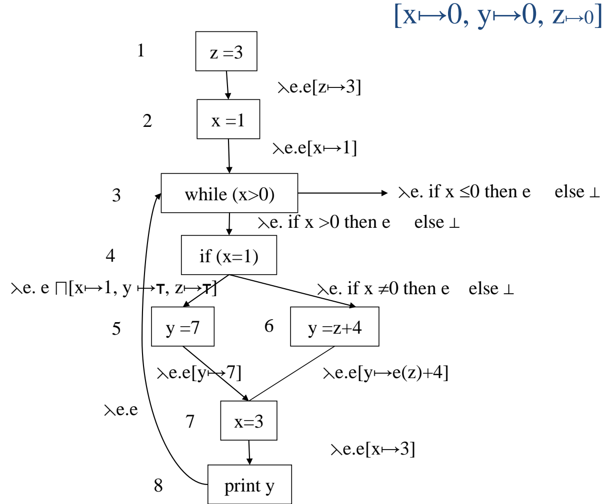

Figure 6: CFG and its corresponding transfer functions

4. Node 4 has been added to WL (node 3 was removed), and the state $df_{entry}[4]$ is $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$.

5. Node 5 has been added to WL (node 4 was removed), and the state $df_{entry}[5]$ is $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$ as well.

6. Both edges $(5, 6)$ and $(5, 7)$ are traversed, but since the transfer function corresponding the edge $(5, 6)$ yields a fixed point, node 6 is not added to the WL while node 7 does. The value of y changes accordingly.

7. Node 8 has been added to WL (node 7 was removed), and the vale of x has changed to 3 according to the transfer function.

8. Node 3 has been added to WL (node 8 was removed), and the values of x and y are updated in $df_{entry}[3]$ (the value of z does not change).

9. Node 4 has been added to WL (node 3 was removed), and the values of x and y are updated in $df_{entry}[4]$ (the value of z does not change).

10. Both nodes 5 and 6 have been added to WL (node 4 was removed). The value of y is updated in $df_{entry}[5]$ (the value of x and z do not change).

11. $df_{entry}[6]$ is updated with $[x \mapsto 1, y \mapsto \top, z \mapsto 3]$. Node 5 has been removed.

Table 1: State changes while running the algorithm

	WL	$df_{entry}[v]$
1	$\{1\}$	$df_{entry}[1] = [x \mapsto 0, y \mapsto 0, z \mapsto 0]$
2	$\{2\}$	$df_{entry}[2] = [x \mapsto 0, y \mapsto 0, z \mapsto 3]$
3	$\{3\}$	$df_{entry}[3] = [x \mapsto 1, y \mapsto 0, z \mapsto 3]$
4	$\{4\}$	$df_{entry}[4] = [x \mapsto 1, y \mapsto 0, z \mapsto 3]$
5	$\{5\}$	$df_{entry}[5] = [x \mapsto 1, y \mapsto 0, z \mapsto 3]$
6	$\{7\}$	$df_{entry}[7] = [x \mapsto 1, y \mapsto 7, z \mapsto 3]$
7.	$\{8\}$	$df_{entry}[8] = [x \mapsto 3, y \mapsto 7, z \mapsto 3]$
8	$\{3\}$	$df_{entry}[3] = [x \mapsto \top, y \mapsto \top, z \mapsto 3]$
9	$\{4\}$	$df_{entry}[4] = [x \mapsto \top, y \mapsto \top, z \mapsto 3]$
10	$\{5, 6\}$	$df_{entry}[5] = [x \mapsto 1, y \mapsto \top, z \mapsto 3]$
11	$\{6, 7\}$	$df_{entry}[6] = [x \mapsto \top, y \mapsto \top, z \mapsto 3]$
12	$\{7\}$	$df_{entry}[7] = [x \mapsto \top, y \mapsto 7, z \mapsto 3]$

12. The value of x in $df_{entry}[7]$ is updated to $\top$. While employing the transition function of edge $(7, 8)$, the algorithm reaches a fixed point of $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$, and the work-list is empty.