Program Analysis/ Mooly Sagiv, Noam Rinetzky
Lecture #4, 15-April-2004: Posets, Lattices and Constant Propagation
Notes by: Roi Barkan

Introduction

In this lecture, mathematical background will be given, about the notion of posets and lattices. We will make use of the constant propagation example to demonstrate the ideas. We will start with a short example of constant propagation problem, and with an intuitive description of the algorithm used to solve it, and continue by creating the mathematical basis, and formalizing the intuitive algorithm.

Constant Propagation

Problem Definition

Given a program, we would like to know for every point in the program (after every statement), which variables have constant values, and which do not. We say that a variable has a constant value at a certain point if every execution that reaches that point, gives that variable the same constant value.

A Simple Example Program

```
1   z = 3
2   x = 1
3   while (x > 0) (
4   if (x = 1) then
5           y = 7
6   else
7           y = z + 4
8   x = 3
9   print y
10  )
```

For this short example there are some simple constant propagation results we can notice:

- In line 2 the variable z has the value of 3.
- In line 5 the variable x has the value of 1.
- In lines 8 and 9 the variable y has the value of 7. Thus, the print statement can be replaced by print 7.
- In line 4 the variable x does not have constant value. Specifically, it can take two values 1 and 3.

Constant Propagation Algorithm

The algorithm we shall present basically tries to find for every statement in the program a mapping between variables, and values of $N \cup \{T, \perp\}$. If a variable is mapped to a constant number, that number is the variables value in that statement on every execution. If a variable is mapped to T (top), its value in the statement is not known to be constant,

and in the variable is mapped to $\perp$ (bottom), its value is not initialized on every execution, or the statement is unreachable.
The algorithm for assigning the mappings to the statements is an iterative algorithm, that traverses the control flow graph of the algorithm, and updates each mapping according to the mapping of the previous statement, and the functionality of the statement. The traversal is iterative, because non-trivial programs have circles in their control flow graphs, and it ends when a "fixed-point" is reached – i.e., further iterations don't change the mappings
The execution of the algorithm for the given example program:

1. Assign the mapping $[x \mapsto 0, y \mapsto 0, z \mapsto 0]$ to statement 1, look at statement 2.
2. Assign $[x \mapsto 0, y \mapsto 0, z \mapsto 3]$ to statement 2, move to statement 3.
3. Assign $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$ to statement 3, move to statement 4.
4. Assign $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$ to statement 4, move to statement 5.
5. Assign $[x \mapsto 1, y \mapsto 0, z \mapsto 3]$ to statement 5, move to statement 8.
6. Assign $[x \mapsto 1, y \mapsto 7, z \mapsto 3]$ to statement 8, move to statement 9.
7. Assign $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$ to statement 9, move to statement 4.
8. Change the mapping of statement 4 to be $[x \mapsto T, y \mapsto T, z \mapsto 3]$, move to 5 (and 7).
9. Change 5 to be $[x \mapsto 1, y \mapsto T, z \mapsto 3]$, move to 7 (and 8).
10. Assign $[x \mapsto T, y \mapsto T, z \mapsto 3]$ to statement 7, move to 8.
11. Change 8 to be $[x \mapsto T, y \mapsto 7, z \mapsto 3]$, move to 9.
12. The mapping of statement 9 stays $[x \mapsto 3, y \mapsto 7, z \mapsto 3]$ - the algorithm terminates since no further changes are possible.

The Algorithm – More Formally

The constant propagation algorithm has, like described the following stages:

1. Construct a control flow graph (CFG).
2. Associate transfer functions with the edges of the graph. The transfer functions depend on the statement or the program condition itself (the type of statement determines which functions need to be associated to each of its outgoing edges).
3. Iterate until no more changes occur

We will later show that the algorithm always stops, and that the solution is unique, no matter what the order of traversal is. Note that while the solution is unique, and does not depend on the order of traversal – different traversal schemes might result in different performance characteristics of the algorithm, i.e., the number of iterations of the algorithm depends on the traversal order.

Transfer Functions

The transfer functions assigned to each edge in the graph are functions from the group of all constant mappings to the group of all constant mappings (includes mapping to top and bottom), and describe the modification that traversing through an edge does to a mapping. For example, if the mapping of a statement S is $[x \mapsto 0, y \mapsto 1]$, and S itself is **x:=2**, then it is obvious that after moving through S, the mapping should be $[x \mapsto 2, y \mapsto 1]$, and thus the Transfer function on the outgoing edge of S should reflect that, and be $\lambda e.e[x \mapsto 2]$ (this notation is equivalent to $f(e) = e[x \mapsto 2]$).

In order to easily describe more complex transfer functions, we use two basic binary functions: meet ($\sqcap$), and join ($\sqcup$). The meet function takes the intersection of two mappings, and the join function takes the union.

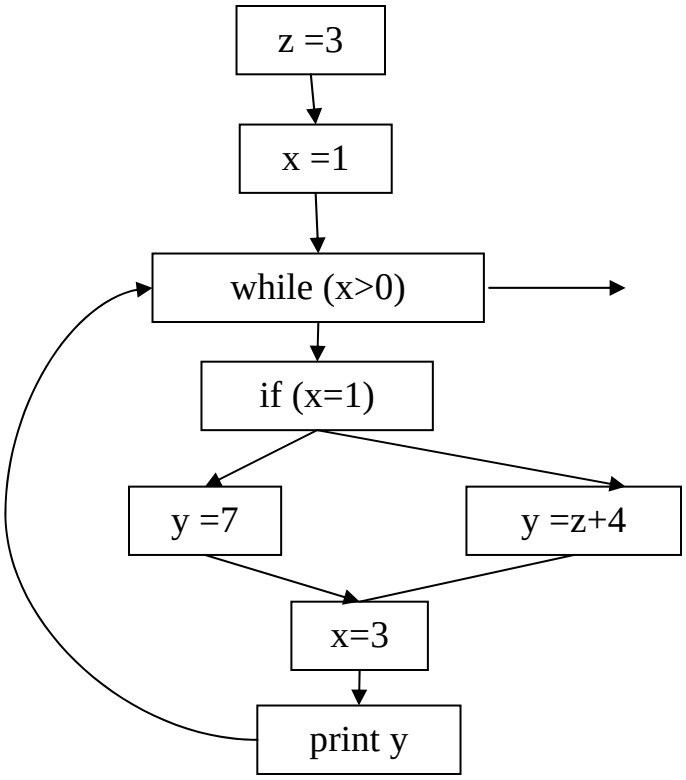
Formally, both functions take two mappings and return mapping. Both functions are "variable-independent", which means that the result mapping for variable x depends only on the mapping of x in the two input mappings. The "truth table" of both functions is as follows:

$\sqcap$	T	$n \in N$	$m(\neq n)$	$\perp$
T	T	n	M	$\perp$
$n \in N$	n	n	$\perp$	$\perp$
$m(\neq n)$	m	$\perp$	M	$\perp$
$\perp$	$\perp$	$\perp$	$\perp$	$\perp$

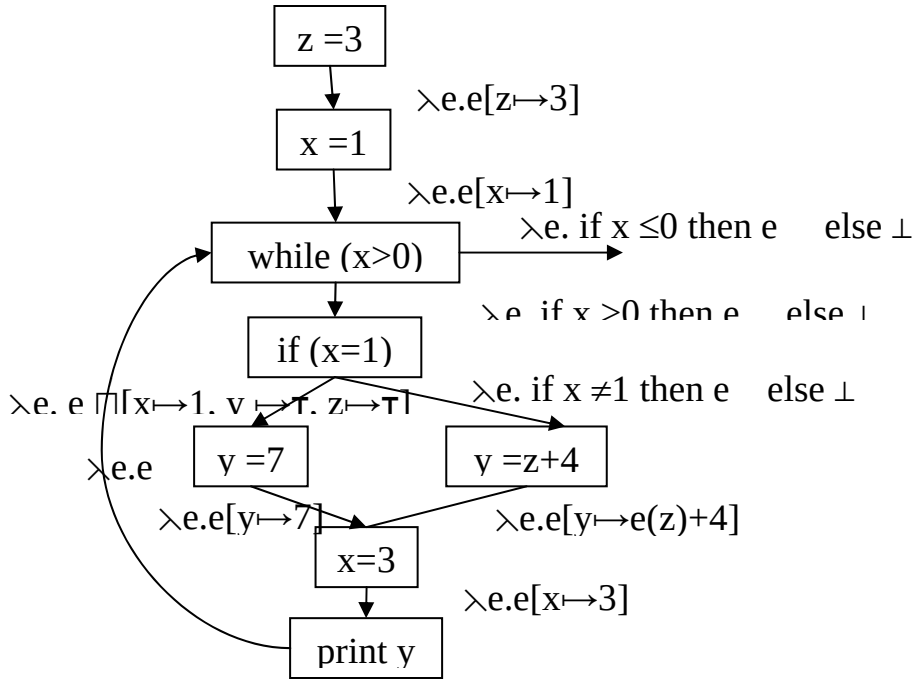
$\sqcup$	T	$n \in N$	$m(\neq n)$	$\perp$
T	T	T	T	T
$n \in N$	T	n	T	n
$m(\neq n)$	T	T	m	m
$\perp$	T	n	m	$\perp$

Back to the Example

Control Flow Graph



Associating Transfer Functions



Mathematical Background

We will need the mathematical background to accurately define the analysis we will make, its result, the uniqueness of the result, and its correctness.

Posets

A partial ordering is a binary relation $\sqsubseteq : L \times L \rightarrow \{false, true\}$

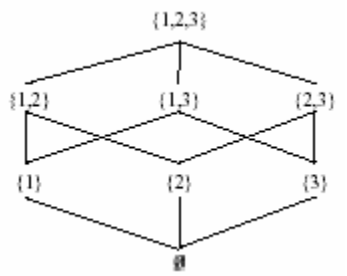
- For all $l \in L : l \sqsubseteq l$ (Reflexive)
- For all $l_1, l_2, l_3 \in L : l_1 \sqsubseteq l_2, l_2 \sqsubseteq l_3 \rightarrow l_1 \sqsubseteq l_3$ (Transitive)
- For all $l_1, l_2 \in L : l_1 \sqsubseteq l_2, l_2 \sqsubseteq l_1 \rightarrow l_1 = l_2$ (Anti-Symmetric)

Denoted by $(L, \sqsubseteq)$

Examples for poset:

- In program analysis
  $l_1 \sqsubseteq l_2 \Leftrightarrow l_1$ is more precise than (or equally precise as) l_2
  $\Leftrightarrow l_1$ represents a subset of the concrete states represented by l_2
- Total orders $(N, \leq)$
- Powersets $(P(S), \subseteq)$
- Powersets $(P(S), \supseteq)$
- Constant propagation

Here's a graphic example of the partial order of $\left(P\left(\{1,2,3\}\right), \subseteq\right)$. A path going up from X to Y implies $X \sqsubseteq Y$.



In order to ease our work, we add several simple notations:

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

Upper and Lower Bounds

Consider a poset $(L, \sqsubseteq)$

- A subset $L' \subseteq L$ has a lower bound $l \in L$ if for all $l' \in L' : l \sqsubseteq l'$
- A subset $L' \subseteq L$ has an upper bound $u \in L$ if for all $l' \in L' : l' \sqsubseteq u$
- A greatest lower bound of a subset $L' \subseteq L$ is a lower bound $l_0 \in L$ such that $l \sqsubseteq l_0$ for any lower bound l of L'
- A lowest upper bound of a subset $L' \subseteq L$ is an upper bound $u_0 \in L$ such that $u_0 \sqsubseteq u$ for any upper bound u of L'

For every subset $L' \subseteq L$:

- The greatest lower bound of L' is unique if at all exists, and its value is $\sqcap L'$
  - If both l and l' are greatest lower bounds of L' than $l \sqsubseteq l'$ and $l' \sqsubseteq l$ thus l=l'
- The lowest upper bound of L' is unique if at all exists, and its value is $\sqcup L'$

Complete Lattices

A poset $(L, \sqsubseteq)$ is a complete lattice if every subset has least upper bounds and a greatest lower bounds.

- It is denoted: $L = (L, \sqsubseteq) = (L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$.
- $\bot = \sqcup \emptyset = \sqcap L$. This is the least element.
- $\top = \sqcup L = \sqcap \emptyset$. This is the greatest element.

Examples

- Powersets $(P(S), \subseteq)$
- Powersets $(P(S), \supseteq)$
- Constant propagation

Note that the total order $(N, \leq)$ is not a complete lattice, because it has no greatest element. It is possible to add an artificial element that represents infinity, to classify $(N \cup \{\infty\}, \leq)$ as a complete lattice.

Lemma: for every poset $(L, \sqsubseteq)$ the following conditions are equivalent:

i. $(L, \sqsubseteq)$ is a complete lattice.
ii. Every subset of L has a least upper bound.
iii. Every subset of L has a greatest lower bound.

Proof:

First, we note that (i) is equivalent to (ii) and (iii) together.

Now, to prove that (ii) ➔ (iii) we will show how to implement the meet operation, by using the join operation. This is suffice, because condition (ii) means that the join operation exists, and condition (iii) is implied by the existence of the meet operation.

The claim is that for every $Y \subseteq L$, the following equation holds:

$\sqcap Y = \sqcup \{ l \in L \mid \forall y \in Y : l \sqsubseteq y \}$. To prove that, we need to show that $\sqcup \{ l \in L \mid \forall y \in Y : l \sqsubseteq y \}$ is indeed the greatest lower bound of Y. Because every member in $\{ l \in L \mid \forall y \in Y : l \sqsubseteq y \}$ is a lower bound of Y, the join of all of them must also be a lower bound of Y (as, by definition of least upper bound, it is smaller than all elements in Y); and because all the lower bounds of Y are in $\{ l \in L \mid \forall y \in Y : l \sqsubseteq y \}$, the join operator guarantees that $\sqcup \{ l \in L \mid \forall y \in Y : l \sqsubseteq y \}$ will indeed be the greatest lower bound of Y.

The proof that (iii) ➔ (ii) is similar

Cartesian Products

Given A complete lattice $(L_1, \sqsubseteq_1) = (L_1, \sqsubseteq, \sqcup_1, \sqcap_1, \bot_1, \top_1)$, and another complete lattice $(L_2, \sqsubseteq_2) = (L_2, \sqsubseteq, \sqcup_2, \sqcap_2, \bot_2, \top_2)$, we can define a Poset $L = (L_1 \times L_2, \sqsubseteq)$ where

- $(x_1, x_2) \sqsubseteq (y_1, y_2)$ if and only if
  - $x_1 \sqsubseteq x_2$ and
  - $y_1 \sqsubseteq y_2$

One can easily see that L is a complete lattice, because there is no influence of the two components of the Cartesian product on each-other.

Finite Maps

Given a complete lattice $(L_1, \sqsubseteq_1) = (L_1, \sqsubseteq, \sqcup_1, \sqcap_1, \bot_1, \top_1)$, and a finite set V, we can define a Poset $L = (V \to L_1, \sqsubseteq)$ where

- $e_1 \sqsubseteq e_2$ if for all $v \in V$ $e_1 v \sqsubseteq e_2 v$

One can easily see that L is a complete lattice, because a finite map is basically a Cartesian product of L_1 with itself |V| times.

Chains

1. A subset $Y \subseteq L$ in a poset $(L, \sqsubseteq)$ is a chain if every two elements in Y are ordered: For all $l_1, l_2 \in Y$: $l_1 \sqsubseteq l_2$ or $l_2 \sqsubseteq l_1$
2. An ascending chain is a sequence of values: $l_1 \sqsubseteq l_2 \sqsubseteq l_3 \sqsubseteq \ldots$
3. A strictly ascending chain is a sequence of values: $l_1 \sqsubset l_2 \sqsubset l_3 \sqsubset \ldots$
4. A descending chain is a sequence of values: $l_1 \sqsupseteq l_2 \sqsupseteq l_3 \sqsupseteq \ldots$
5. A strictly descending chain is a sequence of values: $l_1 \sqsupset l_2 \sqsupset l_3 \sqsupset \ldots$
6. L has a finite height if every chain in L is finite

Lemma : A poset $(L, \sqsubseteq)$ has finite height if and only if every strictly decreasing and strictly increasing chains are finite
Most complete Lattices used in Program Analysis have a finite height.

Monotone Functions

Given a poset $(L, \sqsubseteq)$, a function $f: L \to L$ is called monotone if for every $l_1, l_2 \in L$: $l_1 \sqsubseteq l_2 \Rightarrow f(l_1) \sqsubseteq f(l_2)$.

Galois Connections

Given 2 Lattices C and A and 2 functions $\alpha: C \to A$ and $\gamma: A \to C$,
The pair of functions (α, γ) form a Galois connection if:
$\forall c \in C, \forall a \in A$: $\alpha(c) \sqsubseteq a$ iff $c \sqsubseteq \gamma(a)$
Alternatively:

1. α and γ are monotonic
2. $\forall a \in A: \alpha(\gamma(a)) \sqsubseteq a$
3. $\forall c \in C : c \sqsubseteq \gamma(\alpha(C))$

It can be proved that α and γ uniquely determines each other.

Fixed Points

Fixed points of a function, are points x, for which $f(x) = x$. The fixed points notion is helpful for us when designing iterative algorithms, because it helps us prove where an iterative algorithm will complete, and whether or not it completes at all.
Given a monotone function $f: L \to L$, where $(L, \sqsubseteq, \sqcup, \sqcap, \bot, \top)$ is a complete lattice we shall define the following three groups:

1. $Fix(f) = \{l \mid l \in L, f(l) = l\}$.

2. $\mathrm{Re}d(f)=\{l \mid l\in L, f(l)\sqsubseteq l\}$
3. $Ext(f)=\{l \mid l\in L, l\sqsubseteq f(l)\}$

The following theorem will show us how to find the upper and lower bound of $Fix(f)$. Denote $lfp(f)=\sqcap Fix(f)$, and $gfp(f)=\sqcup Fix(f)$. The acronyms LFP and GFP stand for lowest fixed point and greatest fixed point, respectively, much as their definitions imply.

Theorem (Tarski, 1995): if f is monotone then

- $lfp(f)=\sqcap \mathrm{Re}d(f)$
- $gfp(f)=\sqcup Ext(f)$

Proof:

First, note that $\sqsubseteq$ is reflexive, and thus $Fix(f)\subseteq \mathrm{Re}d(f)$. Define $l_0=\sqcap \mathrm{Re}d(f)$. To prove that l_0 is lfp(f), we need to prove two claims:

1. l_0 is a lower bound for Fix(f).
2. $l_0\in Fix(f)$.

l_0 is a lower bound for all of Red(f), and $Fix(f)\subseteq \mathrm{Re}d(f)$. Thus l_0 is a lower bound for Fix(f), and this proves the first claim.

To show the second claim, we will prove that $f(l_0)=l_0$. We will use the anti-symmetry of $\sqsubseteq$, and prove separately that $l_0\sqsubseteq f(l_0)$, and that $f(l_0)\sqsubseteq l_0$. For every $l\in \mathrm{Re}d(f)$, we know that $l_0\sqsubseteq l$, and because f is monotone we can deduce that $f(l_0)\sqsubseteq f(l)$. Because l belongs to Red(f) we know that $f(l)\sqsubseteq l$. Now apply transitivity of $\sqsubseteq$, and get $f(l_0)\sqsubseteq l$ for every $l\in \mathrm{Re}d(f)$. That means that $f(l_0)$ is a lower bound for Red(f), and thus must be lower than the greatest lower bound of Red(f). Thus we get $f(l_0)\sqsubseteq l_0$.

To prove the other side of the equality, $l_0\sqsubseteq f(l_0)$, we will use fact that f is monotone on the recently proved expression of $f(l_0)\sqsubseteq l_0$, and reveal that $f(f(l_0))\sqsubseteq f(l_0)$. But this, according to the definition of Red(f) means that $f(l_0)\in \mathrm{Re}d(f)$. l_0 is a lower bound for all of Red(f), and thus $l_0\sqsubseteq f(l_0)$.

The proof that $gfp(f)=\sqcup Ext(f)$ is similar.

A little intuition

Although we've just proved Tarski's Theorem completely, we would like to show another proof, for the specific case where the iterative algorithm that follows ends.

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

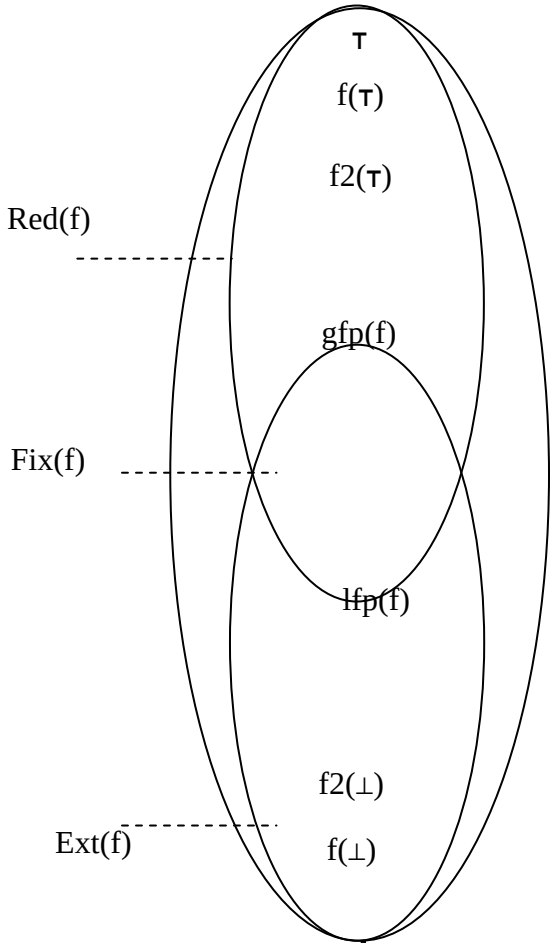
We will prove that the algorithm calculates lfp(f).

First, we note that because the algorithm ends, the final value it calculates is in Fix(f). Now, all that is left is to show that the calculated value is a lower bound for Fix(f).

That is shown by an iterative use of the fact that f is monotone, throughout the algorithm.
We start with the obvious fact that $\perp$ is a lower bound for Fix(f),
Now we (inductively) claim that if a certain x is a lower bound for Fix(f), then f(x) must also be a lower bound of Fix(f). The reason is that for every $l \in Fix(f)$ we have $x \sqsubseteq l$. now because f is monotone, we get $f(x) \sqsubseteq f(l)$, and because $l \in Fix(f)$, $f(x) \sqsubseteq l$.
Thus throughout the algorithm, the variable x always holds a lower bound for Fix(f), and if the algorithm ends – x still has a lower bound.
The same analysis can be made for the algorithm starting with x=T, that calculates gfp(f).

Note that when L has a finite height the above algorithm always terminate.

A graphic description of fixed points



Chaotic Iterations

Given a lattice $L = (L, \sqsubseteq, \sqcup, \sqcap, \perp, \top)$ with finite strictly increasing chains:
$L^n = L \times L \times \ldots \times L$, and a monotone function $\underline{f}: L^n \rightarrow L^n$

It is easy to see that lfp($\underline{f}$) is the simultaneous least fixed of the system $\{x[i] = \underline{f_i}(x) : 1 \leq i \leq n \}$.

Thus if we have a program with N statements, and we wish to calculate the least fixed point of a function on every statement, we can simply look for the least fixed point of the function f by using the naïve algorithm given earlier:

```
x := (⊥,⊥,…⊥)
while(x≠f(x))
        x := f(x)
```

Note that f operates on a vector of size N, where there is one entry in the vector for every program point.

A more structured algorithm will traverse the N statements:

```
for i:=1 to N do
        x[i] := ⊥
WL={1}
while (WL is not empty) do
        select and remove an element i from WL
        new := f_i(x)
        if ( new ≠ x[i]) then
                x[i] = new
                add all the indices that directly depend on i to WL
        end-if
end-while
```

This algorithm is exactly the algorithm shown previously for constant propagation, only now we know a few provable facts about it.

Chaotic Iterations for Constant Propagation

We have returned, finally to the same algorithm we have shown previously for constant propagation, only now we know it is a **chaotic iterations** algorithm. This gives us proof for a number of claims

- The algorithm always terminates
- For the given lattice we chose, the algorithm computes the lfp of the transfer functions. This means that the algorithm:
    - Results in a minimum number of non-constants.
    - Results in a maximum number of ⊥ values.

Soundness of the Algorithm

The algorithm shown here is sound, which means it gives conservative, but always true results:

- Every detected constant is indeed a constant
- Every error will be detected

- The least-fixed-point computed will represent all runtime states

The soundness of the algorithm can be proved simply by showing that the transfer function assigned to each semantic construct behaves as it semantically should be. Once that is shown – the correctness of algorithm is achieved because we already know that chaotic iterations lead to the lowest fixed point, which indeed represents maximum number of constants.

Completeness of the Algorithm

As always, our algorithm is conservative, and not complete. This means that:

- There may be undetected constants.
- There may be errors that will be detected although they will not occur at runtime.
- The may be states represented by the algorithm which will not be reachable at runtime.

There are many reasons for the lack of completeness. They mainly revolve around the fact that we are working in an abstract representation of the program, and not actually executing it. The value of T does not actually exist in the real program semantics, and our use of it leads to lack of completeness.