CS 5/6110, Software Correctness Analysis, Spring 2022

Ganesh Gopalakrishnan
School of Computing
University of Utah
Salt Lake City, UT 84112

How I got hooked on this topic: Reading Manna, Chapter-5, these examples!

Now consider the recursive definition:

$$F(x,y) \;=\; if\ x=y\ then\ y+1\ else\ F(x,F(x-1,y+1)).$$

$$\begin{aligned} f_1 &= \lambda(x,y)\ .\ if\ x=y\ then\ y+1\ else\ x+1 \\ f_2 &= \lambda(x,y)\ .\ if\ x\geq y\ then\ x+1\ else\ y-1 \\ f_3 &= \lambda(x,y)\ .\ if\ x\geq y\ and\ x-y\ is\ even\ then\ x+1\ else\ \perp \end{aligned}$$

We can plug-in f1, f2, or f3 in lieu of F and “solve” the equation

Which solutions are computed when you

* Experiment with it in an I/O-gathering session
- Normally (eagerly, as in Python)
- Lazily (as in Haskell)

Why does running order determine the function computed?

What I'm trying to do

- Tell you that an elephant can be viewed from many sides
  - A tube
  - A pancake
  - A pokey thing
- Fixpoint theory is in many areas of CS
  - Context-free Languages
  - PL semantics
  - Static analysis
  - CTL model checking
  - Even CMOS transistor simulation
- Learning it in "just one class" may give you the view that elephants are pokey objects ☺

What fixed-point/fixpoint theory is aimed at

- Solving equations
  - Some equations make sense, some don't
    - X = 2 – yes
    - X = X + 1 – no, in int
    - F(x) = F(x) – yes , but too many solutions
    - F(x) = F(x+1) - <your answer>
    - F(x) = F(x) + 1 - <your answer_

What fixed-point/fixpoint theory is aimed at

- Solving equations
  - Some equations make sense, some don't
    - X = 2 – yes
    - X = X + 1 – no, in int
    - F(x) = F(x) – yes , but too many solutions
    - F(x) = F(x+1) – sure it can
    - F(x) = F(x) + 1 - no

What fixed-point/fixpoint theory is aimed at

- Solving equations
  - Some equations make sense, some don't
    - X = 2 – yes
    - X = X + 1 – no, in int
    - F(x) = F(x) – yes , but too many solutions
    - F(x) = F(x+1) – sure it can
    - F(x) = F(x) + 1 – no
- Main points
  - Recursion / circularity is natural
  - How do we solve when we have circular situations?

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | SS | epsilon
    - Versus
  - S -> aSbS | bSaS | epsilon
- How do we view the above as language equations?

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | epsilon
  - Solve the recursive language equation

- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e}

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | epsilon
  - Solve the recursive language equation
- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e}
- What do we get when we iterate from L_S = {} “upwards” ?

But for the grammar with the | … | SS rule

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | SS | epsilon

- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e} U L_S L_S

- We have two solutions!
  - What are they?

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | SS | epsilon
- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e} U L_S L_S
- One is the "usual solution"
  - Context-free rewrite schemes (productions) go after the LFP
    - "equal a's and b's"
  - We also have Sigma* as a solution
    - For the case we have S -> … | SS

Uniqueness of Solutions

- People in general like things when solutions are unique
  - They sleep well at night
  - They are kinder to strangers, even smile at them
  - They remember to floss well at night
  - …
- How about these situations : unique or not?
  - $X^2 = 4$
  - $X^2 = -4$
  - Quadratic equations
  - …

Uniqueness of Solutions

- People in general like things when solutions are unique
- A lot of Fixpoint Theory in CS and programming is aimed at seeking uniqueness

- This is why equation systems on monotone lattices are important

Monotone is a “betterness order”

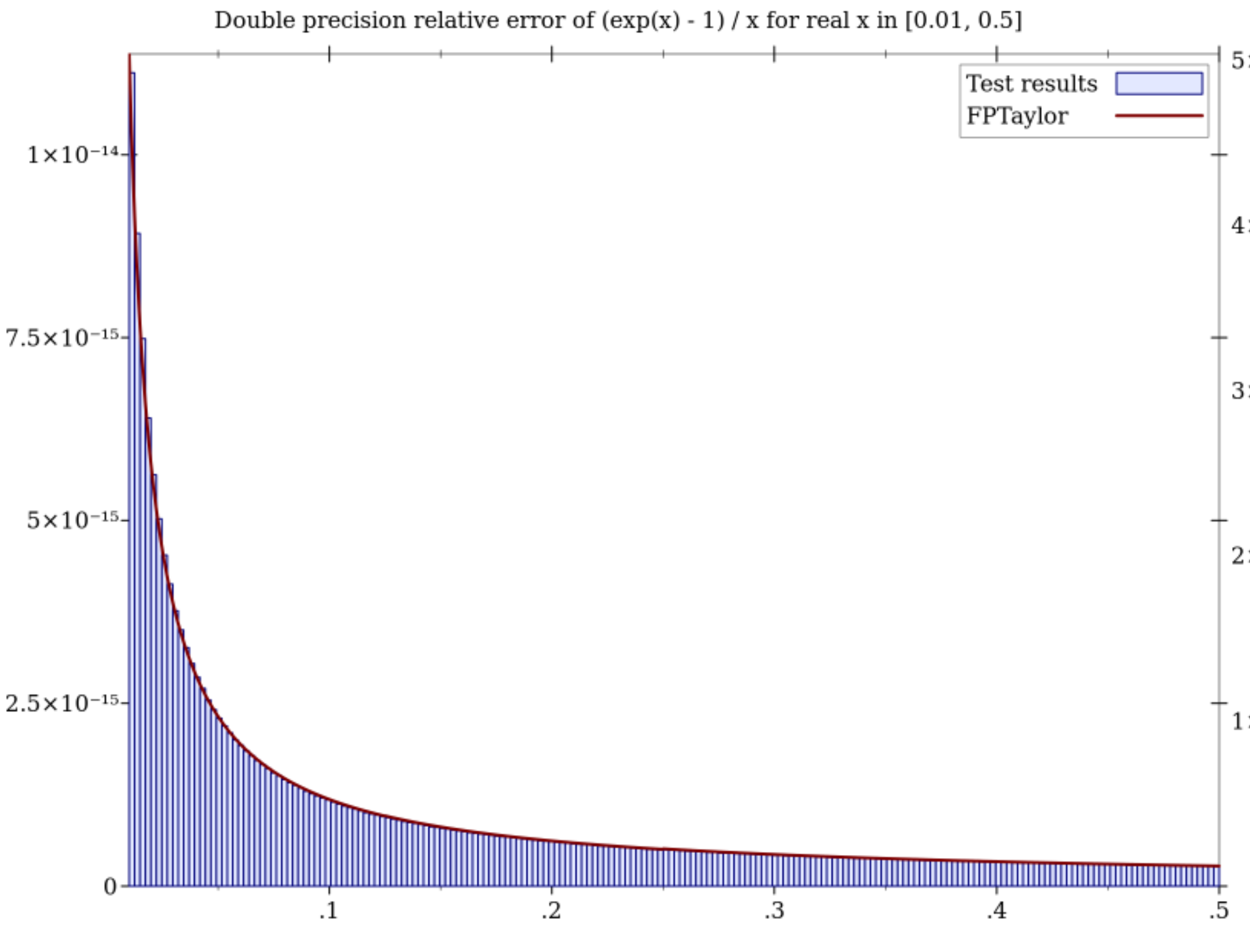
- Or a worseness order
  - A <= B means
    - A is a better component than B (in static analysis at least)
    - A is a tighter approximation than B
- Example
  - You can get resistors with 10% tolerance (a silver band on them)
  - Or a 5% tolerance (gold band on them)
    - 5ohms@5% <= 5ohms@10%
- Resistor parallel composition respects betterness
  - R1 || R2 = (R1*R2) / (R1 + R2)
    - Here if you initially use R2 == R2@10%
    - And stick in R2@5%
    - The whole ckt gets better
- This is monotonicity
  - Not all systems are monotonic
  - Hence causes huge debugging headaches when monotonicity is violated!

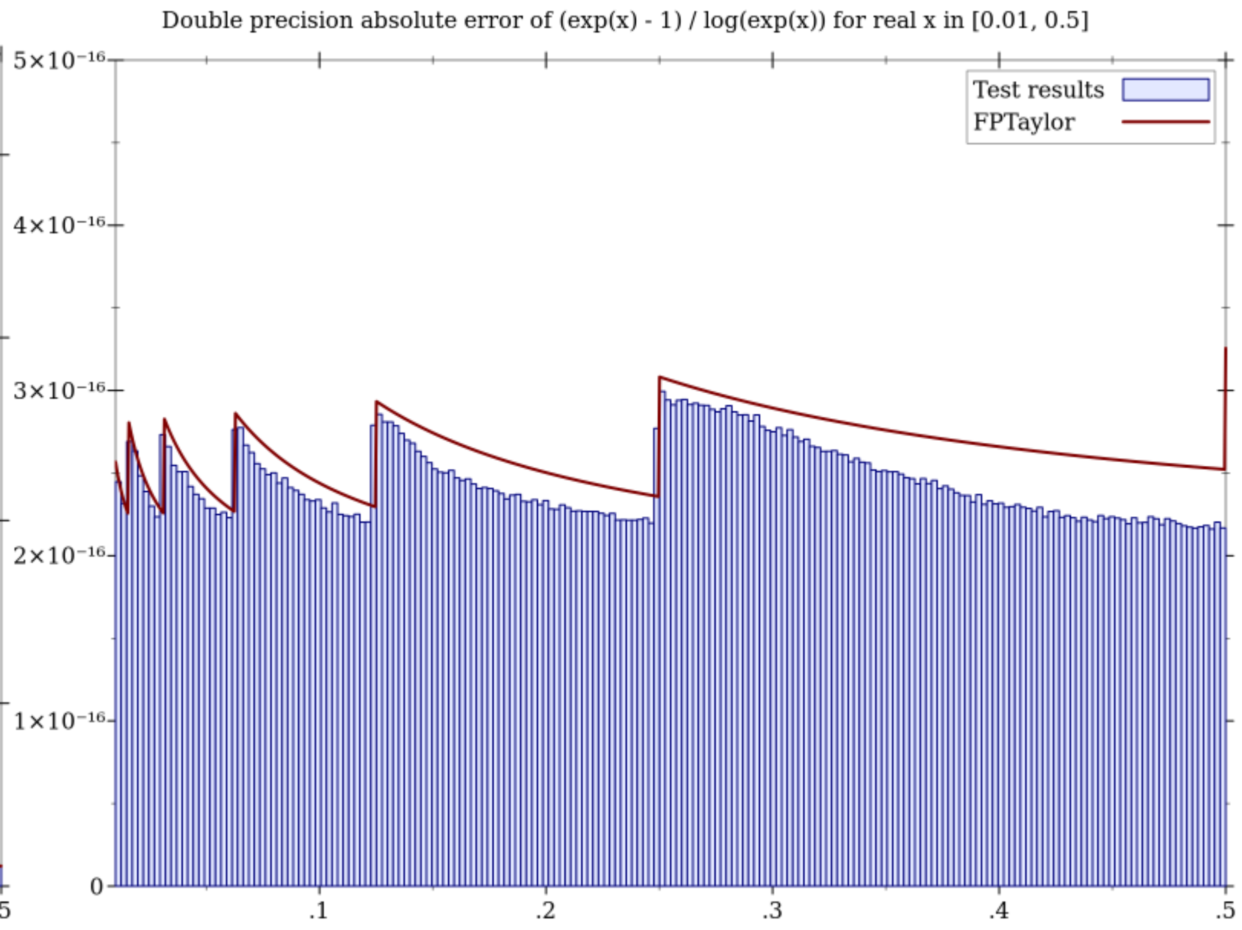
Floating-point error behaves non-monotone

- If you plug-in a component that introduces worse error, the overall error can decrease!
- See plots next slide!

Example of non-monotonicity (FP example)

- This example was discussed last class – here are the plots





Solving for a pair of unknowns is natural

- Mutual recursion, i.e. defining two languages
  - S -> epsilon | (W S
  - W -> (W W |)

- Solve
- (L_S, L_W) = ({e} U {(} L_W L_S , {(} L_W L_W U {)})

Fixpoint Theory to Explain CFGs

- Then discuss the system
  - S -> epsilon | (W S
  - W -> (W W |)

- Solve
- (L_S, L_W) = ({e} U {(} L_W L_S   ,   {(} L_W L_W   U {)})

- What fixpoint obtained by iterating up from ({} , {}) ?
- What is the lattice ordering?

Circular situations in circuits

- Two inverters in a loop
  - X = not(not(X))
- Three (or an odd number of) inverters in a loop
  - X = not(not(not(X)))
- What are possible fixpoints?
  - Is there a “least” fixpoint?
- This is again another glimpse of “fixpoint thinking”

Fixpoint theory in HW design

- Oral history : Randy Bryant
  - https://youtu.be/1Ch-wjFT9VE

Connection w. (flow-sensitive) static-analysis

- Flow-sensitive Static analysis ends up solving such mutually recursive situations

E.g., Analysis (from Moller/Schwartzbach)

5.4 Live Variables Analysis

A variable is *live* at a program point if there exists an execution where its value is read later in the execution without it being written to in between. Clearly undecidable, this property can be approximated by a static analysis called live variables analysis (or liveness analysis). The typical use of live variables analysis is optimization: there is no need to store the value of a variable that is not live. For this reason, we want the analysis to be conservative in the direction where the answer "not live" can be trusted and "live" is the safe but useless answer.

We use a powerset lattice where the elements are the variables occurring in the given program. This is an example of a *parameterized* lattice, that is, one that depends on the specific program being analyzed. For the example program

```
var x,y,z;
x = input;
```

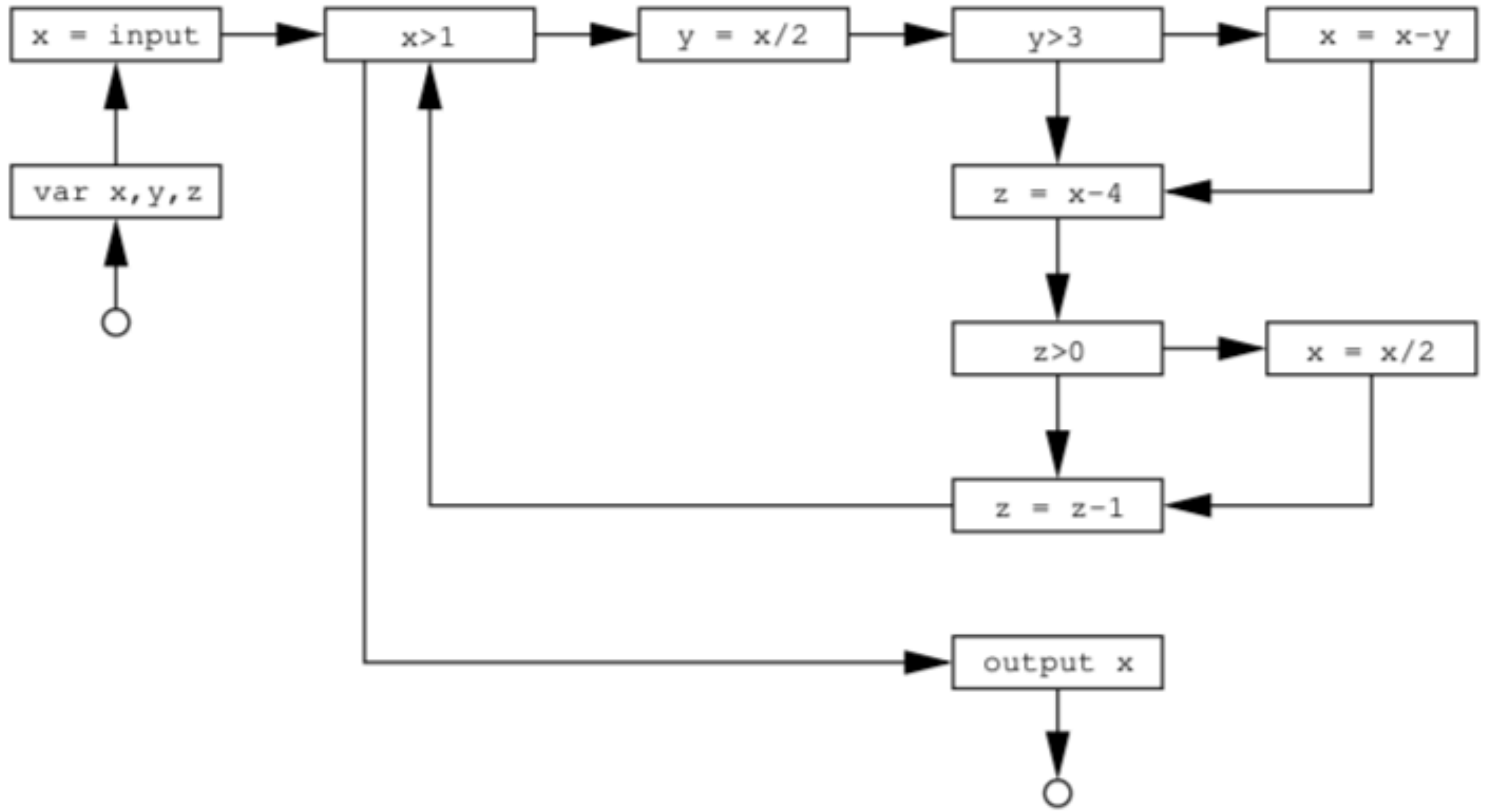
E.g., Analysis (from Moller/Schwartzbach)

```
while (x>1) {
  y = x/2;
  if (y>3) x = x-y;
  z = x-4;
  if (z>0) x = x/2;
  z = z-1;
}
output x;
```

the lattice modeling abstract states is thus:[4]

$$States = (\mathcal{P}(\{\mathtt{x},\mathtt{y},\mathtt{z}\}), \subseteq)$$

The corresponding CFG looks as follows:



E.g., Analysis (from Moller/Schwartzbach)

$$JOIN(v) = \bigcup_{w \in succ(v)} [\![w]\!]$$

$X = E$: $\quad [\![v]\!] = JOIN(v) \setminus \{X\} \cup vars(E)$

This rule models the fact that the set of live variables before the assignment is the same as the set after the assignment, except for the variable being written to and the variables that are needed to evaluate the right-hand-side expression.

> **Exercise 5.23**: Explain why the constraint rule for assignments, as defined above, is sound.

Branch conditions and output statements are modelled as follows:

$$\left.\begin{array}{l}\texttt{if }(E)\texttt{:}\\ \texttt{while }(E)\texttt{:}\\ \texttt{output }E\texttt{:}\end{array}\right\} \quad [\![v]\!] = JOIN(v) \cup vars(E)$$

where $vars(E)$ denotes the set of variables occurring in E. For variable declarations and exit nodes:

var $X_1, \ldots, X_n$: $\quad [\![v]\!] = JOIN(v) \setminus \{X_1, \ldots, X_n\}$

$$[\![exit]\!] = \emptyset$$

For all other nodes:

$$[\![v]\!] = JOIN(v)$$

E.g., Analysis (from Moller/Schwartzbach)

Our example program yields these constraints:

$$
\begin{aligned}
&[\![\texttt{var x,y,z}]\!] = [\![\texttt{x=input}]\!] \setminus \{\texttt{x},\texttt{y},\texttt{z}\}\\
&[\![\texttt{x=input}]\!] = [\![\texttt{x>1}]\!] \setminus \{\texttt{x}\}\\
&[\![\texttt{x>1}]\!] = ([\![\texttt{y=x/2}]\!] \cup [\![\texttt{output x}]\!]) \cup \{\texttt{x}\}\\
&[\![\texttt{y=x/2}]\!] = ([\![\texttt{y>3}]\!] \setminus \{\texttt{y}\}) \cup \{\texttt{x}\}\\
&[\![\texttt{y>3}]\!] = [\![\texttt{x=x-y}]\!] \cup [\![\texttt{z=x-4}]\!] \cup \{\texttt{y}\}\\
&[\![\texttt{x=x-y}]\!] = ([\![\texttt{z=x-4}]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x},\texttt{y}\}\\
&[\![\texttt{z=x-4}]\!] = ([\![\texttt{z>0}]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{x}\}\\
&[\![\texttt{z>0}]\!] = [\![\texttt{x=x/2}]\!] \cup [\![\texttt{z=z-1}]\!] \cup \{\texttt{z}\}\\
&[\![\texttt{x=x/2}]\!] = ([\![\texttt{z=z-1}]\!] \setminus \{\texttt{x}\}) \cup \{\texttt{x}\}\\
&[\![\texttt{z=z-1}]\!] = ([\![\texttt{x>1}]\!] \setminus \{\texttt{z}\}) \cup \{\texttt{z}\}\\
&[\![\texttt{output x}]\!] = [\![\mathit{exit}]\!] \cup \{\texttt{x}\}\\
&[\![\mathit{exit}]\!] = \emptyset
\end{aligned}
$$

See the recursion here!

E.g., Analysis (from Moller/Schwartzbach)

whose least solution is:

$$
\begin{aligned}
[\![entry]\!] &= \emptyset \\
[\![\texttt{var x,y,z}]\!] &= \emptyset \\
[\![\texttt{x=input}]\!] &= \emptyset \\
[\![\texttt{x>1}]\!] &= \{\texttt{x}\} \\
[\![\texttt{x=x-y}]\!] &= \{\texttt{x}, \texttt{y}\} \\
[\![\texttt{z=x-4}]\!] &= \{\texttt{x}\} \\
[\![\texttt{z>0}]\!] &= \{\texttt{x}, \texttt{z}\} \\
[\![\texttt{x=x/2}]\!] &= \{\texttt{x}, \texttt{z}\} \\
[\![\texttt{z=z-1}]\!] &= \{\texttt{x}, \texttt{z}\} \\
[\![\texttt{output x}]\!] &= \{\texttt{x}\} \\
[\![exit]\!] &= \emptyset
\end{aligned}
$$

E.g., Analysis (from Moller/Schwartzbach)

From this information a clever compiler could deduce that y and z are never live at the same time, and that the value written in the assignment z=z-1 is never read. Thus, the program may safely be optimized into the following one, which saves the cost of one assignment and could result in better register allocation:

```
var x,yz;
x = input;
while (x>1) {
  yz = x/2;
  if (yz>3) x = x-yz;
  yz = x-4;
  if (yz>0) x = x/2;
}
output x;
```

Uniqueness of Least Fixpoints

- Least fixpoints exist and are unique when Tau is
  - Monotonic
  - Continuous
- For infinite lattices
  - Continuity implies Monotonicity
- For finite lattices
  - Monotonicity implies Continuity

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
    - S -> aSbS | bSaS | SS | epsilon
        - Versus
    - S -> aSbS | bSaS | epsilon

- Then discuss the system
    - S -> epsilon | (W S
    - W -> (W W |)

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | epsilon
  - Solve the recursive language equation

- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e}

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | epsilon
  - Solve the recursive language equation
- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e}
- What do we get when we iterate from L_S = {} “upwards” ?

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | SS | epsilon
- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e} U L_S L_S

Fixpoint Theory to Explain CFGs

- Context-free Grammars re-interpreted as recursive equations
  - S -> aSbS | bSaS | SS | epsilon

- L_S = {a} L_S {b} L_S U {b} L_S {a} L_S U {e} U L_S L_S

- Are there 2 fixpoints? Which is found using iteration using {} as the bottom (going up)?

Fixpoint Theory to Explain CFGs

- Then discuss the system
  - S -> epsilon | (W S
  - W -> (W W |)

- Solve
- (L_S, L_W) = ({e} U {(} L_W L_S   ,   {(} L_W L_W   U {)})

Fixpoint Theory to Explain CFGs

- Then discuss the system
  - S -> epsilon | (W S
  - W -> (W W |)
- Solve
- (L_S, L_W) = ({e} U {(} L_W L_S   ,   {(} L_W L_W   U {)})
- What fixpoint obtained by iterating up from ({} , {}) ?
- What is the lattice ordering?

Fixpoint Theory to understand Recursion

Now consider the recursive definition:

$$F(x, y) \;=\; if\ x = y\ then\ y + 1\ else\ F(x, F(x-1, y+1)).$$

$$\begin{aligned} f_1 &= \lambda(x,y)\ .\ if\ x = y\ then\ y+1\ else\ x+1 \\ f_2 &= \lambda(x,y)\ .\ if\ x \geq y\ then\ x+1\ else\ y-1 \\ f_3 &= \lambda(x,y)\ .\ if\ x \geq y\ and\ x-y\ is\ even\ then\ x+1\ else\ \perp \end{aligned}$$

Function f3 corresponds to lim_i { Tau^i [Bottom_fn] }

Where Tau for “F” above is: ...fill this... and is called
the “functional underlying the recursive definition (in Manna’s book)

In Chapter 18 of Book-3, it is called the “pre” function (e.g. PreFact etc) on
which the Y combinator is applied.

Applying the Y combinator gives the same effect as computing the limit of this chain of functions

What does Tau^1[Bottom] correspond to? What about Tau^2 ? Tau^3 ? ...fill this...

Now discuss notes in this directory

- Manna’s work on interpreting these functions
- Which function can we “experimentally compute”?
  - If we keep experimenting with F in say Python , what function table can we fill ?
  - What if we did it in a different (lazy) language)?

- That is, if we compute according to a fixpoint computation rule, we will get the “true answer”

- None of this is largely of concern for finite lattices
  - Many static-analysis situations
- But the general story is important to know.

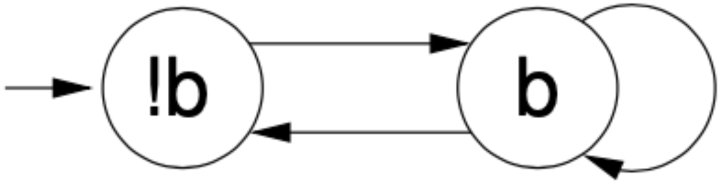
CTL Model Checking

- Least fixpoints exist and are unique when Tau is
  - Monotonic
  - Continuous
- For infinite lattices
  - Continuity implies Monotonicity
- For finite lattices
  - Monotonicity implies Continuity

State-Space Travel via BDDs

- We will use BDDs to represent Kripke Structure
- We will model Transition Relations using BDDs
- Use Boolean operations to obtain the set of reachable states

State Transition Systems via BDDs



$$\lambda(b, b').(b + b').$$

Fig. 11.4. Simple state transition system (example SimpleTR)

The values of b and b' for which this relation is satisfied represent the present and next states in our example. In other words,

– a move where b is false now and true in the next state is represented by $\neg bb'$.
– a move where b is true in the present and next states is represented by bb'.
– finally, a move where b is true in the present state and false in the next state is represented by $b\neg b'$.

The set of reachable states defined by “P”

In other words, we can introduce a predicate P such that a state x is in P if and only if it is reachable from the initial state I through a finite number of steps, as dictated by the transition relation T. The above recursive recipe is encoded as

$$P(s) = (I(s) \vee \exists x.(P(x) \wedge T(x, s))).$$

This can be computed via fixpoint iteration

Rewriting again, we have

$$P = (\lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s))))) \; P.$$

In other words, P is a fixed-point of

$$\lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s)))).$$

Let us call this Lambda expression H:

$$H = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s)))).$$

$$P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s))))P_0$$

Etc…

This can be computed via fixpoint iteration

- $I = \lambda b.\neg b.$
- $T = \lambda(b, b^{'}).\ (b + b^{'}).$
- $P_0 = \lambda s.false$, which encodes the fact that "we've reached nowhere yet!"
- $P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x, s))))P_0.$
  This simplifies to $P_1 = I$, which is, in effect, an assertion that we've "just reached" the initial state, starting from P_0.
- Let's see the derivation of P_1 in detail. Expanding T and P_0, we have
  $P_1 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge\ (x + s)\ )))\ (\lambda x.false).$

This can be computed via fixpoint iteration

- The above simplifies to $\neg b$.
- By this token, we are expecting P_2 to be all states that are zero or one step away from the start state. Let's see whether we obtain this result.
- $P_2 = \lambda G.(\lambda s.(I(s) \vee \exists x.(G(x) \wedge T(x,s))))P_1$.
  $= \lambda s.(\neg s \vee \exists x.(\neg x \wedge (x + s)))$.
  $= \lambda s.1$.

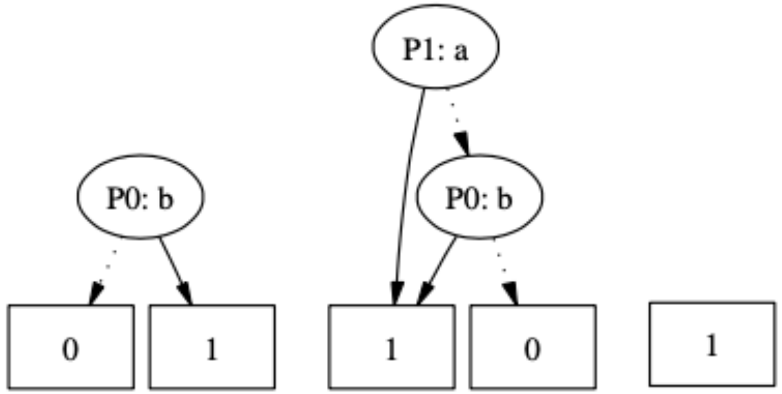
Forward Reahability via the BDD tool called "BED"

```
var b bp                 % Declare b and b'
let I = !b               % Declare init state
let t1 = !b and bp       % 0 --> 1
upall t1                 % Build BDD for it
view t1                  % View it
let t2 = b and bp        % 1 --> 1
let t3 = b and !bp       % 1 --> 0
let T = t1 or t2 or t3   % All three edges
upall T                  % Build and view the BDD
view T                   %

let   P0 = false
upall P0
view  P0

let   P1 = I or ((exists b. (P0 and T))[bp:=b])
upall P1
view  P1

let   P2 = I or ((exists b. (P0 and T))[bp:=b])
upall P2
view  P2
```



P0 P1 P2, the least fixed-point

Fig. 11.5. BED commands for reachability analysis on SimpleTR, and the fixed-point iteration leading up to the least fixed-point that denotes the set of reachable states starting from I

Forward Reahability via the BDD tool called "BED": another example

```
var a ap b bp

let T = (a  and b  and ap  and bp)  or /* S0 -> S0 */
        (!a and b  and !ap and bp)  or /* S1 -> S1 */
        (a  and !b and ap  and !bp) or /* S2 -> S2 */
        (!a and !b and !ap and !bp) or /* S3 -> S3 */
        (!a and b  and ap  and !bp) or /* S1 -> S2 */
        (a and !b  and !ap and bp)  or /* S2 -> S1 */
        (!a and b  and ap  and bp)  or /* S1 -> S0 */
        (a and !b  and ap  and bp)     /* S2 -> S0 */

upall T
view T              /* Produces BDD for TREL 'T' */

let I = a and b
let P0 = b
let P1 = I or ((exists a. (exists b. (P0 and T)))[ap:=a][bp:=b])

upall P1
view P1
```

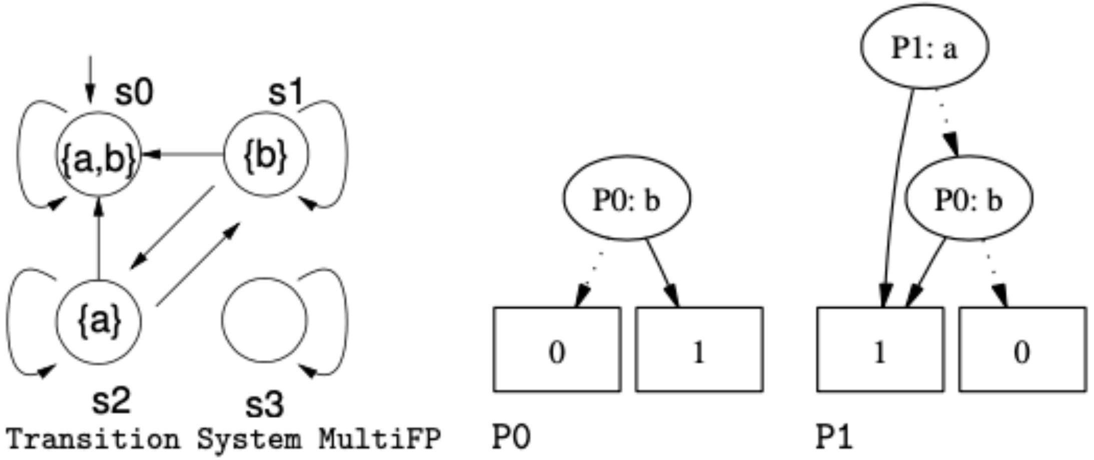


Fig. 11.6. Example where multiple fixed-points exist. This figure shows attainment of a fixed-point $a \vee b$ which is between the least fixed-point of $a \wedge b$ and the greatest fixed-point of 1. The figure shows the initial approximant P0 and the next approximant P1

Now, for the CTL Logic, and Reachability using BDDs

CTL formulas are Kripke structure classifiers

Given a CTL formula φ, all possible computation trees fall into two bins—*models* and *non-models*.[5] The computation trees in the *model* ('good') bin are those that satisfy φ while those in the *non-model* ('bad') bin obviously falsify φ.
Consider the CTL formula AG (EF (EG a)) as an example. Here,

- 'A' is a *path quantifier* and stands for *all paths* at a state
- 'G' is a *state quantifier* and stands for *everywhere along the path*
- 'E' is a *path quantifier* and stands for *exists a path*
- 'F' is a *state quantifier* and stands for *find* (or *future*) along a path
- 'X' is a *state quantifier* and stands for *next* along a path

The truth of the formula AG (EF (EG a)) can be calculated as follows:

- In all paths, everywhere along those paths, EF (EG a) is true
- The truth of EF (EG a) can be calculated as follows:
  - There exists a path where we will find that EG a is true.
  - The truth of EG a can be calculated as follows:
    * There exists a path where a is globally true.

Now, for the CTL Logic, and Reachability using BDDs

CTL formulas γ are inductively defined as follows:

$\gamma \to x$	a propositional variable	
$\mid \neg\gamma$	negation of γ	
$\mid (\gamma)$	parenthesization of γ	
$\mid \gamma_1 \vee \gamma_2$	disjunction	
$\mid \mathrm{AG}\ \gamma$	on all paths,	everywhere along each path
$\mid \mathrm{AF}\ \gamma$	on all paths,	somewhere on each path
$\mid \mathrm{AX}\ \gamma$	on all paths,	next time on each path
$\mid \mathrm{EG}\ \gamma$	on some path,	everywhere on that path
$\mid \mathrm{EF}\ \gamma$	on some path,	somewhere on that path
$\mid \mathrm{EX}\ \gamma$	on some path,	next time on that path
$\mid \mathrm{A}[\gamma_1\ \mathrm{U}\ \gamma_2]$	on all paths,	γ_1 until γ_2
$\mid \mathrm{E}[\gamma_1\ \mathrm{U}\ \gamma_2]$	on some path,	γ_1 until γ_2
$\mid \mathrm{A}[\gamma_1\ \mathrm{W}\ \gamma_2]$	on all paths,	γ_1 weak-until γ_2
$\mid \mathrm{E}[\gamma_1\ \mathrm{W}\ \gamma_2]$	on some path,	γ_1 weak-until γ_2

Now, for the CTL Logic, and Reachability using BDDs

$$\mathrm{EG}\ p = p \wedge (\mathrm{EX}\ (\mathrm{EG}\ \mathrm{p}))$$

```
bed> var a a1 b b1
var a a1 b b1
bed> let TREL =
    (not(a) and b and a1 and not(b1)) or (a and not(a1) and b1) or
    (a and not(b) and b1)             or (a and not(b) and a1)
bed> upall TREL
Upall( TREL ) -> 53
bed> view TREL ... (displays the BDD)
```

Now, for the CTL Logic, and Reachability using BDDs

$$\text{EG } p = p \wedge (\text{EX } (\text{EG p}))$$

- In the BED syntax, $a \oplus b$ is written `a != b`. Now we perform the fixed-point iteration assisted by BED. We construct variable names that mnemonically capture what we are achieving at each step:

```
EG_a_xor_b_0 = true -- first approximant

EG_a_xor_b_1 = (a != b) and (EX true) -- second approximant
```

This simplifies to `(a != b)`, as `(EX true)` is `true`.

Now, in order to determine `EG_a_xor_b_2`, we continue the fixed-point iteration process, and write

```
EG_a_xor_b_2 = (a != b) and EX (a != b)
```

At this juncture, we realize that we need to calculate `EX (a != b)`. This can be calculated using BED as follows:

```
bed> let EX_a_xor_b = exists a1. exists b1. (TREL and (a1 != b1))
bed> upall EX_a_xor_b
bed> view EX_a_xor_b
```

Now, for the CTL Logic, and Reachability using BDDs

$$\mathrm{EG}\ p = p \wedge (\mathrm{EX}\ (\mathrm{EG\ p}))$$

Calculating AX

If we have to calculate **`AX p`**, we would employ duality and write it as

```
!(EX !p)
```

This approach will be used in the rest of this book.

Now, for the CTL Logic, and Reachability using BDDs

$$\mathrm{A[pUq]} = q \vee (p \wedge \mathrm{AX}(\mathrm{A}[pUq]))$$

```
bed> var p p1 q q1
bed> let TREL = (p and not(q) and p1 and not(q1))
             or (p and not(q) and p1 and q1)
             or (p and q and not(p1) and q1)
             or (not(p) and q and p1 and not(q1))
             or (p and not(q) and not(p1) and q1)
             or (p and not(q) and p1 and not(q1))

bed> upall TREL
Upall( TREL ) -> 67
bed> view TREL
bed> let A_p_U_q_0 = false
bed> let AX_A_p_U_q_0 = false
bed> let A_p_U_q_1 = (q or (p and AX_A_p_U_q_0))
```

Now, for the CTL Logic, and Reachability using BDDs

$$A[pUq] = q \vee (p \wedge AX(A[pUq]))$$

```
bed> upall A_p_U_q_1
Upall( A_p_U_q_1 ) -> 3
bed> view A_p_U_q_1
bed> let EX_not_q = exists p1. exists q1. (TREL and !q1)
bed> upall EX_not_q
Upall( EX_not_q ) -> 80
bed> view EX_not_q
bed> let AX_q = !EX_not_q
bed> upall AX_q
Upall( AX_q ) -> 82
bed> view AX_q
bed> let A_p_U_q_2 = (q or (p and AX_q))
bed> upall A_p_U_q_2
Upall( A_p_U_q_2 ) -> 3
bed> view A_p_U_q_2 --> gives ‘‘q’’, hence denotes {S1,S2} -- LFP
```

Now, for the CTL Logic, and Reachability using BDDs

$$A[pUq] = q \lor (p \land AX(A[pUq]))$$

23.2.5 GFP for Until

```
bed> let A_p_U_q_0 = true
bed> let AX_A_p_U_q_0 = true
bed> let A_p_U_q_1 = (q or (p and AX_A_p_U_q_0))
bed> upall A_p_U_q_1
Upall( A_p_U_q_1 ) -> 72
view A_p_U_q_1
bed> let EX_not_p_or_q = exists p1. exists q1. (TREL and !(p1 or q1))
bed> upall EX_not_p_or_q
Upall( EX_not_p_or_q ) -> 0
bed> let AX_p_or_q = !EX_not_p_or_q
bed> upall AX_p_or_q
Upall( AX_p_or_q ) -> 1
bed> view A_p_U_q_1
bed> let A_p_U_q_2 = (q or (p and AX_p_or_q)) --> reached
     Fixed-point (q or p) which denotes {S0,S1,S2,S3}
```

Summary

- Fixpoint theory is everywhere in CS
  - Static analysis
  - Recursive program analysis
  - CFG explanation
  - CTL model-checking
- Finding lattices and monotonic + continuous functionals is key
- Once set up this way, we usually go after the least fixpoint
- Greatest fixpoints also “make sense”
  - But sometimes they are useless
  - as in the CFG example S -> aSbS | bSaS | SS | epsilon

Summary