CS 5/6110, Software Correctness Analysis, Spring 2021

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

16

Least Fixpoint of Functions, Computational Rules
Monotonicity, Continuity
Y Combinators

Where we are

	NO ASG NOW ON…	AS WE HAVE 6 WKS	BUT QUIZZES YES!!
W 3/17	Wrap up fixed-point theory -- AND/OR show how fixed-points work in the context of CTL model checking		
M 3/22	Fixpoint Theory used to realize Computational Tree Logic (CTL) Model Checking		Quiz10
W 3/24	Office Hours		

f1, f2, and f3 are solutions for F below. We can arrive at f3 (the least fixpoint by iterating up from ”Bottom”

Now consider the recursive definition:

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

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

Here, the “bottom” in f3 is the bottom value

For LFP, we have to substitute the bottom function in place of “F” and iterate f1, f2, and f3 are solutions for F below. We can arrive at f3 (the least fixpoint by iterating up from "Bottom"

Now consider the recursive definition:

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

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

The same f3 would also be obtained if we solve for the recursion of "F" using the Y combinator

```
Y = (lambda x. (lambda h. x(h h)) (lambda h. x(h h)))
```

This demo of the use of Y will be presented in later slides
plus using the Jove demo's on the use of Y

Modeling Partial Functions

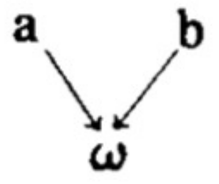
- Discussions from manna-fp-theory-2.pdf
- Manna uses “omega” instead of Bottom (for the undefined value) and Capital Omega for the undefined function
- Manna calls second-order functions by the term “functional”

Modeling Partial Functions

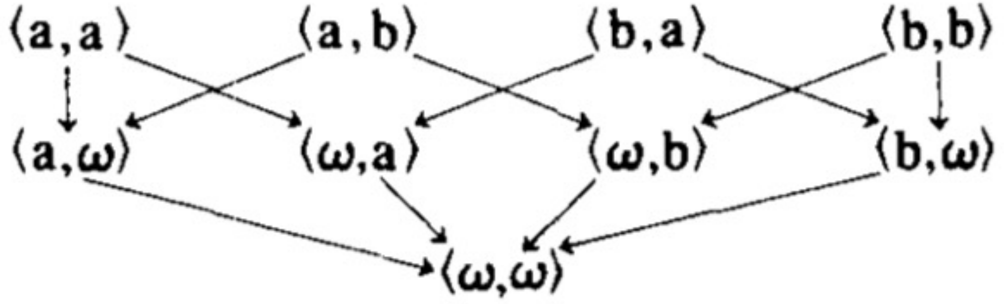
In developing a theory for handling partial functions it is convenient to introduce the special element ω to represent the value undefined. We let D^+ denote $D \cup \{\omega\}$, assuming $\omega \notin D$ by convention; when D is the Cartesian product $A_1 \times \cdots \times A_n$, we let D^+ be $A_1^+ \times \cdots \times A_n^+$. Any partial function f mapping $D_1 = A_1 \times \cdots \times A_n$ into D_2 may then be considered as a total function mapping D_1 into D_2^+: if f is undefined for $\langle d_1, \ldots, d_n \rangle \in D_1$, we let $f(d_1, \ldots, d_n)$ be ω.

Lattice Ordering

- How values are ordered in the information order is shown
- How pairs of values are ordered is shown
- No reflexive and transitive edges shown for clarity



D^+



$(D \times D)^+ = D^+ \times D^+$

Monotonic functions act as per info order

- Monotonic functions “respect the information order ” of their arguments

Monotonic Functions

Any function f computed by a program has the property that whenever the input x is less defined than the input y, the output $f(x)$ is less defined than $f(y)$. We therefore require that the extended function f from D_1^+ into D_2^+ be *monotonic*, i.e.

$$x \sqsubseteq y \text{ implies } f(x) \sqsubseteq f(y) \quad \text{for all } x, y \in D_1^+.$$

We let $(D_1^+ \to D_2^+)$ denote the set of all monotonic functions from D_1^+ into D_2^+.

Monotonic functions act as per info order

- For multiple arguments, there is a “natural extension” – useful for languages with the call-by-value semantics (“evaluate the arguments before applying the function body”)

> lowing we denote such a constant function just by c. If f has many arguments, i.e. $D_1 = A_1 \times \cdots \times A_n$, it may have many different monotonic extensions. A particularly important extension of any function is called the *natural extension*, defined by letting $f(d_1, \ldots, d_n)$ be ω whenever at least one of the d_i is ω. This corresponds intuitively to the functions computed by programs which must know all their inputs before beginning execution (i.e. ALGOL “call by value”).

Example where Call By Value makes a diff.

- The call by value evaluation rule computes less than the least fixpoint – can terminate even when another computation rule can discover the answer – example from paper manna-fp-theory-1.pdf is below
- Those other evaluation rules are
  - “normal order” evaluation rules: Example:
  - Leftmost Outermost
  - Popularly, these are known as Lazy Evaluation Order
- See next slide for details!

Example where Call By Value makes a diff.

Let us consider, for example, the following recursive program over the integers

$$P_1 : F(x, y) \Leftarrow \textit{if } x = 0 \textit{ then } 1 \textit{ else } F(x - 1, F(x, y)).$$

The least fixpoint f_{P_1} can be shown to be

$$f_{P_1}(x, y) : \textit{if } x \geq 0 \textit{ then } 1 \textit{ else undefined}.$$

However, the computed function C_{P_1}, where C is "call by value," turns out to be

$$C_{P_1}(x, y) : \textit{if } x = 0 \textit{ then } 1 \textit{ else undefined}.$$

Thus C_{P_1} is properly less defined than f_{P_1}—e.g. $C_{P_1}(1, 0)$ is *undefined* while $f_{P_1}(1, 0) = 1$.

We construct increasing chains of monotonic functions from the functional obtained from the body of a recursive definition. Such chains have unique least fixpoints

The Ordering $\subseteq$ on Functions

Let f and g be two monotonic functions mapping D_1^+ into D_2^+. We say that $f \subseteq g$, read "f is less defined than or equal to g," if $f(x) \subseteq g(x)$ for any $x \in D_1^+$; this relation is indeed a partial ordering on $(D_1^+ \to D_2^+)$. We say that $f \equiv g$, read "f is equal to g," if $f(x) \equiv g(x)$ for each $x \in D_1^+$ (that is, $f \equiv g$ iff $f \subseteq g$ and $g \subseteq f$). We denote by Ω the function which is always undefined: $\Omega(x)$ is ω for any $x \in D_1^+$. Note that $\Omega \subseteq f$ for any function f of $(D_1^+ \to D_2^+)$.

Infinite increasing sequences $f_0 \subseteq f_1 \subseteq f_2 \subseteq \cdots$ of functions in $(D_1^+ \to D_2^+)$ are called *chains*. It can be shown that any chain has a *unique limit function* in $(D_1^+ \to D_2^+)$, denoted by $\lim_i \{f_i\}$, which has the characteristic properties that $f_i \subseteq \lim_i \{f_i\}$ for every i, and for any function g such that $f_i \subseteq g$ for every i, we have $\lim_i \{f_i\} \subseteq g$.

Example 4. Consider the sequence of monotonic functions $f_0, f_1, f_2, \ldots$ over the natural numbers defined by

$$f_i(x) \equiv (\textbf{if } x \leq i \textbf{ then } x! \textbf{ else } \omega).$$

This sequence is a chain, as $f_i \subseteq f_{i+1}$ for every i; $\lim_i \{f_i\}$ is the factorial function. □

Derivation of Least Fixpoint of Functionals

- We need monotonicity and continuity
- For finite lattices, monotonicity implies continuity
- For infinite lattices, it does not

- So let's first understand monotonicity and continuity thru more examples before we dive in!

Monotonicity ensures the absence of “composition surprises”

- Example:
  - Suppose the lattice ordering is reflective of the quality of an object
  - Example: a 5% tolerance resistor is better than a 10% tolerance resistor
- Does it then mean that a circuit where we clip out a 10% resistor and solder-in a 5% resistor also improves?
- Think of a circuit as a Lambda function and reason out …

Example

- Parcomp(R1, R2) = R1R2 / (R1+R2)

- Now we can ask: if R2’ [= R2”

  - Does this hold : Parcomp(R, R’) [= Parcomp(R, R’’) for any R?

- If so, Parcomp is a monotonic map

- With this property preserved, we can improve one resistor and the whole circuit as a result improves

Monotonicity is sometimes violated!

- Instead of Parcomp, what if it is a floating-point program where you optimize ONE expression?
  - The accuracy of the whole program can reduce, sometimes
  - Example: the introduction of the fused-multiply-add by a compiler may improve local accuracy, but can lead to the overall computation’s accuracy reducing
- What if it is a real-time system where we make ONE component faster?
  - By one module’s output arriving faster, the whole system may slow down
- Thus, “local improvements” may not be a global improvement
  - Monotonicity is what ensure this – and hence, highly preferred

Continuity: “no limit-surprise”

- Most functions (and functionals) are continuous
- Lack of continuity evidenced by the following:
  - One requires an “infinite amount of information” before emitting a finite (small) piece of information
  - If this behavior occurs, the function is not continuous

- Continuity is manifested by situations where “as you provide more input”, the ”output grows more”

- The lack of continuity is also evident in situations where you ”need to solve the halting problem” before you can answer something

Example from actual computations

- Think of stream-based computations
- A computer node can often be viewed as a stream function
- Let the node be a factorial node!
  - Initial input (at time t0) = bottom → output = bottom
  - Input at t1: 1, bottom → 1, bottom
  - Input at t2: 1,2,bottom → 1,2,bottom
  - Input at t3: 1,2,3, bottom → 1,2,6,bottom
  - …
  - I.e. you are progressively providing more input to the node, the node computes more
- Thus you have a stream-to-stream factorial function which is continuous
- But in mathematics, you don't need to do this: you can have a non-continuous and weird function
  - "I want all of Nat before I output anything"
- In this sense, continuity is deeply tied to computability on actual machines
- These are things I learned from Prof. Eugene Stark when I took his programming semantics course during my PhD. These things must be written down somewhere… but these days, not too many people talk about it.
- I also learned these ideas from Prof. Prateek Misra, also an instructor during my PhD

Example of a non-monotonic fn from Manna

(i) The natural extension (*weak equality*), denoted by $=$, yields the value ω whenever at least one of its arguments is ω. The weak equality predicate is of course monotonic.

(ii) Another extension (*strong equality*), denoted by $\equiv$, yields the value **true** when both arguments are ω and **false** when exactly one argument is ω; in other words, $x \equiv y$ if and only if $x \sqsubseteq y$ and $y \sqsubseteq x$. The strong equality predicate is *not* a monotonic mapping from $D^+ \times D^+$ into $\{\mathbf{true}, \mathbf{false}\}^+$, since $\langle \omega, d\rangle \sqsubseteq \langle d, d\rangle$ but $(\omega \equiv d) \not\sqsubseteq (d \equiv d)$ (i.e. **false** $\not\sqsubseteq$ **true**) for $d \in D$.

Defn of continuity

Continuous Functionals

We now consider a function τ mapping the set of functions $(D_1^+ \to D_2^+)$ into itself, called a *functional*; that is, τ takes any monotonic function f as its argument and yields a monotonic function $\tau[f]$ as its value. As in the case of functions, it is natural to restrict ourselves to *monotonic* functionals, i.e. τ such that $f \subseteq g$ implies $\tau[f] \subseteq \tau[g]$ for all f and g in $(D_1^+ \to D_2^+)$. For our purposes, however, we consider only functionals satisfying a stronger property, called *continuity*. A functional τ is said to be continuous if for any chain of functions

$$f_0 \subseteq f_1 \subseteq f_2 \subseteq \ldots$$

we have

$$\tau[f_0] \subseteq \tau[f_1] \subseteq \tau[f_2] \subseteq \cdots$$

and

$$\tau[\lim_i \{f_i\}] \equiv \lim_i \{\tau[f_i]\}.$$

Every continuous functional is clearly monotonic.

Example of a non-continuous fn from Manna

(b) The functional over the natural numbers ($N^+ \to N^+$) defined by

$$\tau[F](x) \equiv \textbf{if } \forall x[F(x) = x] \textbf{ then } F(x) \textbf{ else } \omega$$

is monotonic but not continuous; if we consider the chain $f_0 \subseteq f_1 \subseteq \cdots$ where $f_i(x) \equiv \textbf{if } x < i \textbf{ then } x \textbf{ else } \omega$, $\tau[f_i] \equiv \Omega$ for any i so that $\lim_i \{\tau[f_i]\} \equiv \Omega$, whereas $\tau[\lim_i \{f_i\}]$ is the identity function. $\square$

Theorem (Kleene, Others)

- Continuous functions "tau" on lattices (or other structures such as CPOs) have a unique least fixpoint given by

  tau^i (bottom)

* Derivation in class

Relationship with Lambda Calculus

- One can compute least fixpoints also using “Y”
- One can see that “YF” and “least fixpoint iteration” both behave similarly
- Derivation in class

Demo using Jupyter (Jove) notebooks

- See Chapter 18's material from https://github.com/ganeshutah/Jove.git