The Cutest and Fuzziest Computer Program

Or: How I Learned to Stop Worrying and Love Lambda Calculus

Alex Grabanski

2/24/2017

Define Fuzzy? Define Cute?

For our purposes:

- A program is ”cute” if it’s short.
- A program is ”fuzzy” if running it actually runs several programs non-deterministically.

A God Complex

Imagine that you're God*

- Suppose that you're lazy.

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.
- You're a programmer, so might as well write a computer program.

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.
- You're a programmer, so might as well write a computer program.
- Due to feature bloat, may as well invent a multiverse.

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.
- You're a programmer, so might as well write a computer program.
- Due to feature bloat, may as well invent a multiverse.
- How would you do it?

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.
- You're a programmer, so might as well write a computer program.
- Due to feature bloat, may as well invent a multiverse.
- How would you do it?
- One solution: In an elegant programming language, write something that interprets commands for a very minimal language.

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.
- You're a programmer, so might as well write a computer program.
- Due to feature bloat, may as well invent a multiverse.
- How would you do it?
- One solution: In an elegant programming language, write something that interprets commands for a very minimal language.
- Then, generate every possible program for the minimal language.

---

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A God Complex

Imagine that you're God*

- Suppose that you're lazy.
- Really lazy.
- But you need to invent the universe.
- You're a programmer, so might as well write a computer program.
- Due to feature bloat, may as well invent a multiverse.
- How would you do it?
- One solution: In an elegant programming language, write something that interprets commands for a very minimal language.
- Then, generate every possible program for the minimal language.
- Run them all simultaneously!

[0] existence, uniqueness proofs/disproofs left as exercises to interested audience members

A Very Elegant Language: Lambda Calculus

- Language based on *reduction* of *terms* to other terms.

Terms:

- Any parameter name (x/y/z/foo/bar) is a term
- For any collection of terms $f, t_1, t_2, ..., (f\ t_1\ t_2...)$ is a term, thought of as the application of the function f to the other terms.
- For any term t and parameter name x, $\lambda x.t$ is a term. Within t, x is said to be *bound*

Reduction:

- $(\lambda x.t)\ y \rightarrow sub(t, x, y)$ where $sub(t, x, y)$ means "substitute all free occurrences of x with y in term t"
- A variable appears *free* in a term if it is not *bound*.

Elegant and Natural Code Samples: Definitions

- To get ourselves off the ground, we first define natural numbers.
- We do so using so-called Church encoding.
- The idea: A natural number n may be identified and *defined* by its action on functions via the n^{th} iterate.
- Some numbers:

```
ZERO := λf.λx. x
ONE := λf.λx. f x
TWO := λf.λx. f (f x)
THREE := λf.λx. f (f (f x))
FOUR := λf.λx. f (f (f (f x)))
```

Elegant and Natural Growing Code Samples

$$\begin{aligned}
&\text{SUCC} := \lambda\text{num}.\lambda\text{f}.\lambda\text{x}.\ \text{f}\ (\text{num}\ \text{f}\ \text{x})\\
&\text{ADD} := \lambda\text{num1}.\lambda\text{num2}.\ \lambda\text{f}.\lambda\text{x}.\ \text{num1}\ \text{f}\ (\text{num2}\ \text{f}\ \text{x})\\
&\text{MUL} := \lambda\text{num1}.\lambda\text{num2}.\ \lambda\text{f}.\lambda\text{x}.\ \text{num1}\ (\text{num2}\ \text{f})\ \text{x}
\end{aligned}$$

Elegant and Truthy Code Samples: Definitions

- Now, for Church-encoded Booleans (true/false values)
- Define true/false values by how they ”if/then/else” things.
- Some booleans:

$$\text{TRUE} := \lambda action1.\lambda action2.\ action1$$
$$\text{FALSE} := \lambda action1.\lambda action2.\ action2$$

Elegant and Truthy Code Samples: Logic

```
IF := λp.λaction1.λaction2. p action1 action2
NOT := λp. IF p FALSE TRUE
AND := λp1.λp2. IF p1 p2 FALSE
ZERO? := λnum. num (λx. FALSE) TRUE
```

Note the use of the property that the zeroth iterate of a function is the identity!

Elegant and Romantic Code Samples

- We Church-encode Pairs of terms by how we index the two elements
- We index the two elements with TRUE/FALSE!

```
PAIR := λe1.λe2.λpred.pred e1 e2
FIRST := λp. p TRUE
SECOND := λp. p FALSE
```

Elegant and Natural Shrinking Code Samples

```
PRED := λnum. SECOND (num (λp.
                  IF (FIRST p)
                     (PAIR TRUE (SUCC (SECOND p)))
                     (PAIR TRUE (SECOND p)))
                (PAIR FALSE ZERO))
```

```
SUB := λnum1.λnum2.num2 PRED num1
```

Note that subtraction ”bottoms out” at zero!

Code Samples: Comparative Studies

```
LEQ? := λnum1.λnum2. ZERO? (SUB num1 num2)
EQ? := λnum1.λnum2. AND (LEQ? num1 num2)
                        (LEQ? num2 num1)
```

Endless Code Samples

$$\text{REC} := \lambda f . \; f \; f$$

Exercise: Try to reduce (REC REC) until you can't reduce it any more!

Code Samples: A Nation...

```
COUNTWHILE := λp. REC (λself.λaccum.
                        IF (p accum)
                           ((REC self) (SUCC accum))
                           (PRED accum)
                        ) ZERO

DIV := λnum1.λnum2. COUNTWHILE
                      (λaccum. LEQ? (MUL accum num2) num1)

DIVREM := λnum1.λnum2.
            PAIR (DIV num1 num2)
                 (SUB num1 (MUL num2 (DIV num1 num2)))
```

```
break;
```

time!

Implementing the Multiverse

- Now that we've got that standard library stuff outta the way, time to implement the multiverse.
- But first, we must introduce the original sin of creation...

A Very F***ed-Up Language: P”

- More f***ed-up than Brainf***
- But very similar – P”!

Setting:

- Finite *Program Tape* containing a sequence of valid instructions
- Bi-Infinite *Data Tape* of boolean (FALSE/TRUE) (0/1) cells.

Instructions:

- ”$\ll$”: Move the data tape’s read/write head one space to the left
- ”$\gg$”: *Flip the bit under the data tape’s head*, then move the data tape’s head one space to the right
- ”[” : No-op. Marks a potential jump destination for...
- ”]” : If the current symbol under the data tape is 1/TRUE, jump back in the program tape to the matching ””.

Abominable Code Samples

Flip the bit under the data tape head:

- FLIPBIT := $\ll \gg$

Set the bit under the data tape head to 0:

- SET0 := [FLIPBIT]

Move one space to the right *without* flipping the current bit:

- R := FLIPBIT $\gg$

Move the current bit one space to the right:

- RMOV := R SET0 FLIPBIT $\ll$ [FLIPBIT R FLIPBIT $\ll$] R FLIPBIT $\ll$
- Painful, right?
- But it's Turing complete!

Wrapping Up: Row, Row, Row Your Boat

```
RCONSTSTREAM := λconst. REC
                  (λself.PAIR const (REC self))
LCONSTSTREAM := λconst. REC
                 (λself.PAIR (REC self) const)

MAP := λf. REC (λself.λstream.
           PAIR (f (FIRST stream))
                ((REC self) (SECOND stream)))

NATURALS := REC (λself.λaccum.
          PAIR accum (REC self (SUCC accum))) ZERO
```

Wrapping Up: Operator?

LBRACKET := ZERO
RBRACKET := ONE
LSHIFTOP := TWO
RSHIFTOP := THREE

Wrapping Up: We Caught Everything On

$$\text{CONSTTAPE} := \lambda\text{const}.\ \text{PAIR}\ (\text{LCONSTSTREAM const})\ (\text{RCONSTSTREAM const})$$

$$\text{EMPTYPROGRAMTAPE} := \text{CONSTTAPE LBRACKET}$$

$$\text{EMPTYDATATAPE} := \text{CONSTTAPE FALSE}$$

Wrapping Up: Red

```
READ := λtape. SECOND (FIRST tape)
LSHIFT := λtape. PAIR (FIRST (FIRST tape))
                      (PAIR (READ tape) (SECOND tape))
RSHIFT := λtape. PAIR (PAIR (FIRST tape)
                            (FIRST (SECOND tape)))
                      (SECOND (SECOND tape))
SET := λtape.λval. PAIR (PAIR (FIRST (FIRST tape))
                              (SECOND tape)

FLIPBIT := λtape. SET tape (NOT (READ tape))
```

Wrapping Up: Enumerating Programs

```
INITPROGTAPE :=
λnum. (REC (λself.λtape.λdivResult.
    IF (ZERO? (FIRST divResult))
       (SET tape (SECOND divResult))
       (LSHIFT
          ((REC self)
             (RSHIFT (SET tape (SECOND divResult)))
             (DIVREM (FIRST divResult) FOUR)))
) EMPTYPROGRAMTAPE (PAIR num ZERO))
```

Wrapping Up: Might As Well

```
JUMP :=
(REC (λself.λtape.
     IF (EQ? (READ (LSHIFT tape)) LBRACKET)
          (LSHIFT tape)
          (IF (EQ? (READ (LSHIFT tape)) RBRACKET)
               ((REC self) ((REC self) (LSHIFT tape)))
               ((REC self) (LSHIFT tape))
          )
     ))
```

Wrapping Up: One Small

```
STEP :=
λstatePair. (λprogTape.λdataTape.λinstruction.

    IF (EQ? instruction LSHIFTOP)
        (PAIR (RSHIFT progTape) (LSHIFT dataTape))
    (IF (EQ? instruction RSHIFTOP)
        (PAIR (RSHIFT progTape)
        (RSHIFT (FLIPBIT dataTape)))
    (IF (EQ? instruction RBRACKET)
        (IF (READ dataTape)
            (PAIR (JUMP progTape) dataTape)
            (PAIR (RSHIFT progTape) dataTape)
        )
        (PAIR (RSHIFT progTape) dataTape)
    )))
  (FIRST statePair) (SECOND statePair)
  (READ (FIRST statePair))
```

He's Beginning to Believe

```
SIMULATE_MACHINE :=
λmachineNum. REC (λself. λmachineState.
                    REC self (STEP machineState))
                 (PAIR (INITPROGTAPE machineNum)
                       EMPTYDATATAPE)
```

And AC said, ”LET THERE BE LIGHT!”

```
ALL_MACHINES := NATURALS
MULTIVERSE := MAP SIMULATE_MACHINE ALL_MACHINES
```

A Programmer's Dream

With just that, we just created a single expression which evaluates isomorphic copies of:

- The program which runs forever iff ZFC is consistent
- Microsoft Windows (TM) ($\mathbb{R}$) ($\mathbb{C}$) ($\mathbb{Z}$) ($\mathbb{N}$)
- Club Penguin
- TurboTax
- WinRAR
- Runescape (circa 2007)
- A Lambda Calculus interpreter
- A Lambda Calculus interpreter, interpreting the universal expression itself

All for just one payment of 715 bytes! [zipped]

Whoa, Dude, Pass the Bong, Man!

- Like, dude, what if our universe may be accurately simulated by a Turing machine?
- Bruh, you could be right.
- If so, wouldn't it appear as one of our programs?

'Tis a Gift

- What code fragments should we expect to be executed more often by our program?
- Shorter ones
- Intuition: More likely to be embedded as subroutines
- If programs are viewed as concrete explanations of events, this means that simple explanations are more likely.
- We just invented *Occam's Razor*
- But we also just invented...

Strong AI

- One possible definition of strong AI:
- Something which is able to take any sequence of *percepts* generated by a set of rules, learn the rule as it goes, and use that to come up with actions (which may have an effect on the sequence) which are closer and closer to an optimal strategy.
- This doesn't account for stochastically-generated inputs, but we can modify the approach we'll develop
- W.l.o.g, our percept stream is a binary stream
- For us, we'll assume that we're just playing a "guess the next bit in the stream" game to simplify things.

Solomonoff Induction

- Using the same principles of our universal program...
- Maintain the infinite program state stream
- BUT define a new operator for P", "*", pronounced "exhibits"
- As we receive new percepts...
- Filter out partially-evaluated programs from our stream if the symbol under the data head when the next "exhibits" is hit fails to match the current percept
- Then, we always use the shortest program remaining to determine how to act.
- Problem: We don't know if we'll ever hit another "exhibits" operation! Could loop forever!
- Solution: We approximate (AIXI-t-l) by only simulating up to a constant number of steps before giving up.

But is that a good definition for intelligence?

- No clue.
- Also, AIXI-t-I is horribly inefficient
- But it's pretty

Exercises

Note: For the following exercises, you are allowed to pick your favorite model of computation to work with. Answers may depend on the model of computation used!

- 1. Prove whether or not we live in a simulation.
- 2. If yes to 1, construct the shortest program guaranteed to simulate an isomorphic copy of our universe.
- 3. Define intelligence.
- 4. If yes to 1, use 2 and Occam's Razor to argue for/against the existence of an intelligent prime mover within an environment with a "root" simulation.

Questions?

```
((λq.λy.λn.λu.λo.λv.λi.(λw.λj.λh.λk.λe.(λg.((λf. o (λs.λt.
          q (f (t y)) (o s (t n))))  (λm. o (λs. λa.
                        o s ((λs. (λp.λd.λa.(λr.λm.
                  (e a)
                      (q r (k d))
                  (e m
                      (q r (h ((λt. (λt.λa. q (q (t y y) a)
                     (t n))  t ((λp. p n y)  (j t)))  d)))
                  (e (w m)
                      (j d
                          (q ((o (λs.λt.(λl.
               (e (j l))
                   l
                   (e (w (j l))
                       (o s (o s l))
                       (o s l)
                   )
               ) (k t) ))  p) d)
                          (q r d)
                       )
                       (q r d)
                   ))) (h p) (w a))
                   (s y) (s n)
                     (j (s y)))  a))
                     (q ((o (λs.λt.λd.(λa.λb.
                         (e a)
                            b
                            (k (o s (h b)
                                      ((λa.λb.(λd.
         q d (g a (v b d))) ((λa.λb. (λp. o (λs.λa.
                       (p a)
                           (o s (u a))
                           (w a)
                       ) n)  (λa. (λa.λb. e (g a b))  (v a b) a))  a b))  a i)))
                     ) (d y) ((λt.λa. q (q (t y y) a)
                      (t n))  t (d n))) ((λc. q ((λc. o (λs.q (o s) c))  c) ((λc. o (λs.q c (o s)))  c))
(w i) )  (q n n))  m) ((λc. q ((λc. o (λs.q (o s) c))  c) ((λc. o (λs.q c (o s)))  c))  n) ))  ((o (λs.λa.
               q a (o s (u a))) n) ) )) (λa.λb.b w a)) (λa. (a (λp.(λs.
            (p y)
               (q y (u s))
               (q y s))
            (p n))
          (q n n)) n) (λt. t y n) (λt. q (q (t y) (t n y))
                    (t n n)) (λt. q (t y y) (q (j t) (t n))) (λa. a (λx. n) y)  ) (λa.λb.λp.p a b) (λa.λb
. a) (λa.λb. b) (λa.λf.λx. f (a f x)) (λf. f f) (λa.λb. λf.λx. a (b f) x) (λf.λx. f (f (f (f x)))) )
```