JPL

The Halting Problem

Erann Gat

Section 368

Outline

- The halting problem
  - Definition
  - Relevance for V&V
  - Approaches to a solution
- Chaitin's Omega
  - and it's relation to the halting problem
- Take home messages:
  - Constraints matter
  - Newer is not necessarily better

The Halting Problem

- Does a program halt or run forever?

```
For(int I=1; I<10; I++) printf("%d", I);
```

Halts

```
While(1) printf("Hello\n");
```

Runs Forever

```
Int I = readInteger();
While(I!=1) {
  if (I&1) I = 3*I+1;
  else I = I>>1;
}
```

???

Note that we can't answer the question by running the code!

Why should we care?

JPL

- Software V&V:

```
Main () {
   ...
   if (condition)
     error();  /* Does this ever happen? */
   ...
}
```

- This problem (determining whether arbitrary code reaches a particular control point) is equivalent to the halting problem.

The ultimate question

JPL

- (Virtually) every mathematical question can be cast as an instance of the halting problem.
  - This is usually done by constructing a program that systematically searches for counterexamples. If the theorem is true, the program never halts.

There do not exist integers a, b and c S.T. a^n+b^n=c^n for n>2

- So if we had a solution to the halting problem then we would have a solution to all mathematical questions!
  - This fact alone should lead us to suspect that the halting problem cannot be solved. And indeed we can prove that this is so…

The Halting Theorem

JPL

- Theorem (Turing, 1936): It is not possible to tell in general whether a program halts or runs forever.
- Proof in three and a half bullets:
  - Assume there exists a procedure H which will tell you if a program P halts when run on input I. In other words, H(P,I) returns TRUE if P halts on input I, FALSE otherwise.
  - Construct a program B which takes as input a program P and calls H(P,P). If H(P,P) returns TRUE, then B enters an infinite loop, otherwise B halts. (In other words, B does the *opposite* of what P would do if run on a copy of itself as input.)
  - What happens if we run B with a copy of itself as the input?
    - B(B) halts if and only if B(B) does not halt — a contradiction. Therefore, H cannot exist.

Proof detail

- Assume: H(P,I) returns TRUE if P(I) halts, FALSE otherwise
- Define B(P): if H(P,P) then loop-forever else halt
- Compute: B(B)
- B(P) calls H(P,P), so compute H(B,B)
- H(B,B) returns TRUE iff B(B) halts.
- B(B) halts iff H(B,B) returns FALSE

Consequences

- It is not possible to tell (in general) whether a program reaches any particular control point.
- It is not possible to tell (in general) if two programs produce the same output.
- It is not possible to tell (in general) if a program conforms to a formal specification.

Sounds like bad news! But…

A way out

- These results apply “in general”, that is, to *unconstrained* code. What if we constrained the code?
- Example: If we eliminate “while” loops the halting problem is solvable!
- Unfortunately, we also lose a lot of capability. (For example, we need a while loop to write a program that solves the halting problem for programs without while loops!)
- There is an important application area that obeys this constraint…
- Spacecraft sequences don’t have while loops!

Whither the while loop?

JPL

- We can solve the halting problem for some programs despite the fact that they contain while loops.
- Example: a program that solves the halting problem for programs without while loops. (We can prove it halts.)
- Simply eliminating WHILE loops does lead to a solution of the halting problem, but it's too crude. We lose more capability than we really need to. Is there a middle ground?
- Possible alternatives
  - Finite-state automata (FSA's)
  - Push-down automata (PDA's)
  - Non-deterministic PDA's
  - Goal nets? VML?

Shameless plug

- ESL (Execution Support Language) was designed to allow expressive “sequences” while maintaining formal analyzability.
- Able to formally prove certain useful invariants based on the concept of cognizant failure, i.e. you can’t prove that the code will work, but you can prove that either the code will work or inform you that it has not.
- Use this property of “cognizant failure” to build provably probabilistically reliable systems
  - Use formal statistical methods to prove statements of the form, “The probability that this code will fail is provably less than X.”
- Fielded examples: Remote Agent, Rovers

JPL

And now for something completely different…

Data compression

JPL

- Two programs that produce the same output:

```
printf("01010101010101010101010101010101");
```

```
for (int I=0;I<16;I++) printf("01");
```

Can this be compressed?

```
243F6A88 85A308D3 13198A2E 03707344 A4093822 299F31D0 082EFA98 EC4E6C89 452821E6
38D01377 BE5466CF 34E90C6C C0AC29B7 C97C50DD 3F84D5B5 B5470917 9216D5D9 8979FB1B
D1310BA6 98DFB5AC 2FFD72DB D01ADFB7 B8E1AFED 6A267E96 BA7C9045 F12C7F99 24A19947
B3916CF7 0801F2E2 858EFC16 636920D8 71574E69 A458FEA3 F4933D7E 0D95748F 728EB658
718BCD58 82154AEE 7B54A41D C25A59B5 9C30D539 2AF26013 C5D1B023 286085F0 CA417918
B8DB38EF 8E79DCB0 603A180E 6C9E0E8B B01E8A3E D71577C1 BD314B27 78AF2FDA 55605C60
E65525F3 AA55AB94 57489862 63E81440 55CA396A 2AAB10B6 B4CC5C34 1141E8CE A15486AF
7C72E993 B3EE1411 636FBC2A 2BA9C55D 741831F6 CE5C3E16 9B87931E AFD6BA33 6C24CF5C
7A325381 28958677 3B8F4898 6B4BB9AF C4BFE81B 66282193 61D809CC FB21A991 487CAC60
5DEC8032 EF845D5D E98575B1 DC262302 EB651B88 23893E81 D396ACC5 0F6D6FF3 83F44239
2E0B4482 A4842004 69C8F04A 9E1F9B5E 21C66842 F6E96C9A 670C9C61 ABD388F0 6A51A0D2
D8542F68 960FA728 AB5133A3 6EEF0B6C 137A3BE4 BA3BF050 7EFB2A98 A1F1651D 39AF0176
66CA593E 82430E88 8CEE8619 456F9FB4 7D84A5C3 3B8B5EBE E06F75D8 85C12073 401A449F
56C16AA6 4ED3AA62 363F7706 1BFEDF72 429B023D 37D0D724 D00A1248 DB0FEA
```

Data compression

JPL

- Yes…

```
(defun compute-pi-hex (n &aux (p 0) r)
  (dotimes (i n)
    (incf p (- (/ 4 (+ 1 (* 8 i)))
               (/ 2 (+ 4 (* 8 i)))
               (/ 1 (+ 5 (* 8 i)))
               (/ 1 (+ 6 (* 8 i)))))
    (multiple-value-setq (r p) (truncate p 16))
    (format t "~X" r)
    (if (= (mod i 8) 1) (princ #\space))
    (setf p (* p 16))))
```

Can any data be compressed?

- No. Uncompressible data exists. (In fact, most data is uncompressible.)
- Proof is by a simple counting argument: for N bits of output, there are 2^N possible combinations, and therefore 2^N different programs required to produce them all. To make 2^N distinct programs, at least one of those programs must have at least N bits.

Can we distinguish compressible data?

JPL

- Can we tell (in general) whether a particular data stream is compressible? i.e. given a program P that produces a particular output, does there exist a shorter program that produces the same output?
- This is also unsolvable in general. Specifically, it is unsolvable for all programs longer than some threshold size. (Chaitin, 1975)
- Definition: A program P is *elegant* if no program shorter than P produces the same output.
  - Elegant == uncompressible
- There are an infinite number of elegant programs, one (or more) for each possible computable output.

Chaitin's Proof

- Assume we have an elegance tester E. E(P) returns TRUE if P is elegant, FALSE otherwise.
- Construct a program B that takes as input a number N and systematically enumerates all possible programs longer than N bits. For each such program P, B computes E(P) until it finds a program for which E returns TRUE. B then runs that program.
- Now run B with N set to the length of B plus 1. Since there are an infinite number of elegant programs, B must eventually find one and produce its output. But the program it finds must be longer than B, so it cannot be elegant since B, which is shorter, produced the same output!
- Therefore E was wrong! QED.

Consequences

- Optimal compression is not possible.
  - Strictly speaking, it is not possible to prove that a particular compression scheme is optimal.
- It is not possible to know in general if a particular data stream is compressible!
  - Pi, for example, appears random but is in fact infinitely compressible
  - Can't know if quantum randomness is "really" random!
- It is not possible to accurately estimate the effort required for a large-scale software development effort.
  - http://www.idiom.com/~zilla/Work/kcsest.pdf
  - This claim is somewhat controversial.

An uncompressible number

JPL

- Despite the fact that we can't tell in general whether a particular data stream is compressible, we can define a provably uncompressible number!
- The probability that a randomly generated program (for a particular computational model) will halt is called Ω.
  - It is not a universal constant, but rather a parameter of the particular computational model being used.
- If we knew (any) Ω with sufficient accuracy (a few thousand bits is enough) we could solve the halting problem!
- Therefore, all Ω's are uncomputable (and therefore uncompressible).

The ultimate answer!

JPL

- Conduct a systematic empirical search for halting programs.
  - Use this data to produce a lower bound for Ω.
  - This lower bound increases monotonically: every time we find a halting program, the lower bound increases.
- For any particular program P eventually one of two things will happen: either we will discover empirically that P halts, or we will discover enough other halting programs that we can know that P doesn't halt (if we know Ω).
  - Eventually, the difference between the (assumed known) actual value and the lower bound will become so small that if P were to halt it would make the lower bound greater than the actual value.

Computing Ω

JPL

- Ω is uncomputable (because if we could compute it we could solve the halting problem).
- Ω is (provably) uncompressible (because if we could compress it we could compute it)
- Nonetheless, it is possible to determine a few dozen bits of Ω!
  - http://apru.nus.edu.sg/pdf%20files/student/Shu%20abstract.pdf
  - The first 80 exact bits of Ω (for a particular register-machine computational model) are:

0.00000000000000000010000010000001000000100000011100100111000101000101000…

Ω ≅ 4.80591783357695780 x 10^-7

Summary and Conclusions

JPL

- The halting problem is not solvable for arbitrary code, but it is solvable for constrained code.
- Therefore, code that needs to be reliable needs to be constrained. Getting the design of these constraints right is important… and hard.
- Anyone who writes code (or designs architectures) for mission-critical systems should be familiar with the fundamental "physics" of software.
- Not everything in Computer Science becomes obsolete in three years.