Introduction to Computer Science
Theory of Computation
Dr. Antonio Siochi

Outline

- Turing Machines
- Bare Bone Language
- A Noncomputable Function

Abstract Computer

- control unit + memory, finite alphabet of symbols
- control unit examines/writes a memory cell depending upon the control unit's state (values of its registers)
- control unit can halt or continue
- program = specification of what to do for each possible combination of cell values and control unit states

Turing Machine

- an abstract computer
- unlimited amount of memory
- generalization of real machines
- theoretically more powerful than real machines
- thus, if TM has any limits, then so will real machines

Components of TM

- memory = tape of infinite length
- symbols on tape can be read or written
- cpu = control mechanism with read/write head
- head can be moved left or right
- finite set of states, some of which are start or halt
Computable Functions

- need a method of measuring computing power
- function: associates one output with an input (example: incr X: incr associates X+1 with X)
- define functions either by table lookup or rule
- the "more" functions a machine can calculate, the "more" powerful it is
- a function is TM-computable if there exists a TM that computes that function

Outline

- Turing Machines
- Bare Bone Language
- A Noncomputable Function

Bare bones Language

- given a model computer: memory and CPU design a programming language that will not restrict the capabilities of the computer (sufficient)
- minimal language: only those statements necessary to write any program (compute any function)
- no conveniences, only necessities

Data Types

- since memory actually stores 1's and 0's, any data type such as character or integer is a convenience for the programmer
- therefore only need one data type: bit string
- therefore no need for declarations!
- variable names are strictly alphabetical
Procedural Statements

- only three assignment statements
  - clear X; // sets X to the bit pattern all zeroes
  - incr X; // adds 1 to the bit pattern
  - decr X; // subtracts one from the bit pattern

Examples

- set X to 1
clear X;
icr X;

- set X to 3
clear X;
icr X;
icr X;
icr X;

Control Statement

- only one control statement
  
  while variableName not 0 do;
    s1;
s2;
...
end;

  (must include decr variableName if loop is to terminate)

Example I

// move the value of X to Z

clear Z;
while X not 0 do;
icr Z;
decr Z;
end;

X | Z
---|---
100 | 000
011 | 001
010 | 010
001 | 011
000 | 100

Expressive Power

easy to see that barebones == Zcomputer instructions
we can translate any of the BB instructions into a series of Zcomputer instructions
so any BB program can be translated into an equivalent Zcomputer program

Example II

// copy Today to Tomorrow

clear Aux;
clear Tomorrow;
while Today not 0 do;
icr Aux;
decr Today;
end;
while Aux not 0 do;
icr Today;
icr Tomorrow;
decr Aux;
end;
Expressive Power

we claim that any algorithm can be expressed in BB
(so if we can’t write a BB program for it, no one could
write it in ANY language)

THUS
given a problem, if we can write a bare bones program to
solve it, the problem is computable in bare bones,
 hence on our Zcomputer

Goal

• what can be computed, not how efficiently
• definition of computability

Equivalence of TM and BB

• any BB program can be thought of as computing
  a function (the program takes inputs and produces
  an output)
• some BB programs compute partial functions:
  while X not 0 do;
  end;
• can be shown that we can create a TM for any BB
  and that we can write a BB for any TM

A minimal language

• since TM is theoretically more powerful than any
  real computer
• since BB language can write any program
  computable on TM
• then BB can write any program computable on a
  real machine
• thus any real language that has at least BB’s
  statements can be used to write any computable
  program on a real computer

Outline

- Turing Machines
- Bare Bone Language
- A Noncomputable Function

Noncomputable function

• Church-Turing thesis: a function is computable if it
  is TM-computable
• are all functions computable?
• is there a TM that determines whether any arbitrary
  TM will halt? (the halting problem)
• is there a program (function) that computes whether
  any arbitrary program is computable?
The Halting Problem

- the answer is no
- intuition: is the statement “This statement is false.” true or false?
- prove by assuming that a program exists that can solve the halting problem
- then show that because of this assumption, we are left with only contradictions
- thus, the assumption was false

Assume it exists

assume a function H(f) exists that if given the text of any program, f, will return 1 if f halts, 0 else
example: if f is the program incr X; then H(f) = 1
if instead f is the following program:
clear X;
incr X;
while X not 0 do;
end;
then H(f) = 0

Create a new function

define a new function G(f):
X = H(f);  // call H() to see if f halts
while X not 0 do;  // G will not halt if X is not 0
end;
Thus we see that G(f) will halt if X==0,
but not if X==1
That is G(f) halts if f doesn’t halt.
G(f) doesn’t halt if f halts.

Contradictions

G(f) is a function whose input is the text of any function.
We can thus feed G(f) to itself: f = G(f) so G(f) = G(G(f)).
Substituting in the previous result:
That is, G(f) halts if G(f) doesn’t halt.
G(f) doesn’t halt if G(f) halts.

Conclusions

- no such function H(f) exists, since if we assume it does, we obtain contradictions
- there is no step by step method (algorithm, computable function) that will determine if an arbitrary program halts
- there is no program which will detect an infinite loop in an arbitrary program with arbitrary inputs
- since this is a limit of TM, it is a limit for all computers