Announcements

• Reading
  – 7th edition: p. 201
  – 6th edition: p. 177
  – 5th edition: p. ?
• Answer Catalyst Survey about which time you will take the final exam (by Sunday).
  – Review session Saturday/Sunday
  – List of Final Exam Topics and sampling of some typical kinds of exam questions on the web

Last lecture highlights

• Turing machine definition
  – Intuitive justification, Church-Turing Thesis
• Programs ≡ Turing machines
  – Distinction between the executing program $P$ and its code $<P>$
• Program Interpreter $U$ (Universal TM)
  – Takes as input: $(<P>, x)$ where $<P>$ is the code of a program and $x$ is an input string
  – Simulates $P$ on input $x$

Last lecture highlights

• Halting Problem
  – Input: the code of a program $P$ and an input $x$ for $P$, i.e. given $(<P>, x)$
  – Output: 1 if $P$ halts on input $x$
    0 if $P$ does not halt on input $x$
• Theorem (Turing): There is no program that solves the halting problem. It is “undecidable”
  • Proof idea: “diagonalization”
    – Table of the Halting Problem answers
    – Each row is a fingerprint of its program
    – If there is a program $H$ for the Halting problem then we can create a new program $D$ that can’t be in any row

Flipped Diagonal

Want to create a new program whose halting properties are given by the flipped diagonal.
More on the proof

- The Halting Problem takes exactly the same kind of input as the Universal machine $U$ does.
  - Though $H$ can’t exist, we know that $U$ does.

- Why can’t we apply the same diagonalization trick to a table of what $U$ does?
  - We’ll just write a 1 in the table if $U$ halts rather than write the output of $U$.

Another view of the proof

- We can forget the table and just create the code for $D$ assuming that the code for $H$ exists.

  Function $D(x)$:
  - if $H(x)=1$ then
    - while (true); /* loop forever */
  - else
    - no-op; /* do nothing and halt */
  - endif

- $D$’s fingerprint is different from every row of the table.
  - $D$ can’t be a program so $H$ cannot exist!

Another view of the proof

Does $D$ halt on input $<D>$?

- $D$ halts on input $<D>$
  - $H$ outputs 1 on input $<D>$, $<D>$
    - since $H$ solves the halting problem and so $H(<D>,x)$ outputs 1 if $D$ halts on input $x$
  - $D$ runs forever on input $<D>$
    - since $D$ goes into an infinite loop on $x$ if $H(x)=1$
Another view of the proof

Function $D(x)$:
- if $H(x,x)=1$ then
  - while (true); /* loop forever */
- else
  - no-op; /* do nothing and halt */
- endif

Does $D$ halt on input $<D>$?

$D$ halts on input $<D>$ if $H$ outputs 1 on input $(<D>,<D>)$

[since $H$ solves the halting problem and so $H(x,x)$ outputs 1 if $D$ halts on input $x$]

$D$ runs forever on input $<D>$ if $D$ goes into an infinite loop on $x$ iff $H(x,x)=1$

SCOOPING THE LOOP SNOOPER
A proof that the Halting Problem is undecidable
by Geoffrey K. Pullum (U. Edinburgh)

No general procedure for bug checks succeeds.
Now, I won’t just assert that, I’ll show where it leads:
I will prove that although you might work till you drop,
you cannot tell if computation will stop.

For imagine we have a procedure called $P$
that for specified input permits you to see
whether specified source code, with all of its faults,
defines a routine that eventually halts.

You feed in your program, with suitable data,
and $P$ gets to work, and a little while later
(in finite compute time) correctly infers
whether infinite looping behavior occurs...

...Here’s the trick that I’ll use—and it’s simple to do.
I’ll define a procedure, which I will call $Q$,
that will use $P$’s predictions of halting success
to stir up a terrible logical mess.

And this program called $Q$ wouldn’t stay on the shelf;
I would ask it to forecast its run on itself.
When it reads its own source code, just what will it do?
What’s the looping behavior of $Q$ run on $Q$?

Full poem at:
http://www.lel.ed.ac.uk/~gpullum/loopsnoop.html

Using undecidability of the halting problem

- We have one problem that we know is impossible to solve
  - Halting problem
- Showing this took serious effort
- We’d like to use this fact to derive that other problems are impossible to solve
  - don’t want to go back to square one to do it

Another undecidable problem

The “always halts” problem
- Given: $<Q>$, the code of a program $Q$
- Output: 1 if $Q$ halts on every input
  0 if not.

Claim: the “always halts” problem is undecidable
Proof idea:
- Show we could solve the Halting Problem if we had a solution for the “always halts” problem.
- No program solving for Halting Problem exists $implies$ no program solving the “always halts” problem exists

What we would like

- To solve the Halting Problem need to handle inputs of the form $(<P>,x)$
- Our program will create a new program code $<Q>$ so that
  - if $P$ halts on input $x$
    - then $Q$ always halts
  - if $P$ runs forever on input $x$
    - then $Q$ runs forever on at least one input
- In fact, the $<Q>$ we create will act the same on all inputs
Creating \(<Q>\) from \(<P>,x\) 

- Given \(<P>,x\) modify code of \(P\) to:
  - Replace all input statements of \(P\) that read input \(x\), by assignment statements that ‘hard-code’ \(x\) in \(P\)
- This creates a new program text \(<Q>\)
- It would be easy to write a program \(T\) that changes \(<P>,x\) to \(<Q>\)

The transformation

\begin{verbatim}
int main()
{
    ...
    scanf("%d",&u);
    ...
    scanf("%d",&v);
    ...
    123 712
}
\end{verbatim}

\begin{verbatim}
int main()
{
    ...
    u = 123;
    ...
    v = 712;
    ...
}
\end{verbatim}

Program to solve Halting Problem if “always halts” were decidable

- Suppose “always halts” were solvable by program \(A\)
- On input \(<P>,x\)
  - execute the program \(T\) to transform \(<P>,x\) into \(<Q>\) as on last slide
  - call \(A\) with \(<Q>\) (the output of \(T\)) as its input and use \(A\)'s output as the answer.
- This would do the job of \(H\) which we know can’t exist so \(A\) can’t exist

Claim: Given \(<P>,x\) it is undecidable to determine whether or not \(P\) tries to divide by 0 when run on input \(x\)

Claim: Given \(<P>,x\) it is undecidable to determine whether or not \(P\) accesses an array out of bounds when run on input \(x\)

The “yes” problem

- Given: \(<R>\), the code of a program \(R\)
- Output: 1 if \(R\) outputs “yes” on every input 0 if not.

Claim: the “yes” problem is undecidable
Same kind of idea as “always halts”

- To solve the Halting Problem need to be able to handle inputs of the form \(<P, x>\)
- We’ll create a new program code \(<R>\) so that
  - If \(P\) halts on input \(x\)
    - then \(R\) always outputs “yes”
  - If \(P\) runs forever on input \(x\)
    - then \(R\) does something else on at least one input.

Creating \(<R>\) from \(<P, x>\)

- Given \(<P, x>\) modify code of \(P\) to:
  - Remove all output statements from \(P\)
  - Replace all input statements of \(P\) that read input \(x\), by assignment statements that hard-code \(x\) in \(P\)
  - Add a new last statement that prints “yes”
- This creates a new program text \(<R>\)
- It would be easy to write a program \(T\) that changes \(<P, x>\) to \(<R>\)

Program to solve Halting Problem if the “yes” problem were decidable

- Suppose the “yes” problem were solvable by program \(Y\)
- On input \(<P, x>\)
  - execute the code to transform \(<P, x>\) into \(<R>\) as on last slide
  - call \(Y\) with \(<R>\) (the output of \(T\)) as its input and use \(Y\)'s output as the answer.

Equivalent program problem

- Input: the codes of two programs, \(<P>\) and \(<Q>\)
- Output: 1 if \(P\) produces the same output as \(Q\) does on every input
  0 otherwise
- Claim: The equivalent program problem is undecidable

Claim: The equivalent program problem is undecidable

A general phenomenon: Can’t tell a book by its cover

- Suppose you have a problem \(C\) that asks, given program code \(<P>\), to determine some property of the input-output behavior of \(P\), answering 1 if \(P\) has the property and 0 if \(P\) doesn’t have the property.

**Rice’s Theorem:** If \(C\)'s answer isn’t always the same then there is no program deciding \(C\)
Even harder problems

• Recall that with the halting problem, we could always get at least one of the two answers correct
  – if it halted we could always answer 1 (and this would cover precisely all 1's we need to do) but we can't be sure about answering 0
• There are natural problems where you can't even do that!
  – The equivalent program problem is an example of this kind of even harder problem.

Quick lessons

• Don't rely on the idea of improved compilers and programming languages to eliminate major programming errors
  – truly safe languages can't possibly do general computation
• Document your code!!!!
  – there is no way you can expect someone else to figure out what your program does with just your code ....since.....in general it is provably impossible to do this!