Phil 320

Chapter 8: Turing computable functions are recursive

Closing the loop: \( R \subseteq A \subseteq T \subseteq R \Rightarrow R = A = T. \)

Restrict attention to functions \( f(x) \) of one argument, though it can easily be extended to the general case.
Suppose \( f \) is computed by \( M \). Want to show that \( f \) is a recursive function. Actually, we will do this for all Turing machines at once, using \( m \) to stand for the code number of a Turing machine. (This is why it was important that the coding by p.r.)

Outline of Proof:

1. *Wang coding.* Binary coding of contents of tape plus position of cursor. We do this with a *left number* \( p \) and a *right number* \( r \).
2. *Configuration.* A function \( \text{Conf}(m, x, t) \) tracks the progress of Turing machine \( M \) coded by \( m \) applied to input \( x \). Each \( t \) will represent a single step in the progress of the Turing machine calculation. \( \text{Conf}(m, x, t) \) will encode three numbers: \( p, r \) and the current state \( q \) of the machine. Conf will be defined by primitive recursion.
3. *Starting configuration.* \( \text{Conf}(m, x, 0) = \text{inpt}(m, x) \) codes the starting configuration.
4. *Next step.* To get \( \text{Conf}(m, x, t+1) \) from \( \text{Conf}(m, x, t) \) is just a matter of one of (essentially) eight simple operations, depending on whether the scanned symbol is 1 or B, and on whether the specified action is B, 1, L or R. So we’ll have a definition by cases which is p.r.
5. Finally, we can recover \( f(x) \) from the right number \( r \) in the final state \( t \) where \( M \) halts. We’ll define the halted state to be \( q = 0 \), and use minimization to find the least \( t \) such that the associated \( q \)-value of Conf is 0. (This will be the only place we use minimization.)

1. *Wang coding*

a. *Left and right numbers*
Consider a tape containing only 1’s and 0’s (or blanks), with at most finitely many 1’s. Then we can code the content of the tape and the position of the pointer using two numbers.

Ex 1: BBB11B11B1B1B1BB

↑

The *left number* \( p \) is the binary number to the left of the pointer, treating all blanks as 0. In this case, the binary number is 11011, so \( p = 27 \). (Here the ‘1’s’ position is just to the left of the pointer.)
The right number $r$ is the binary number written backwards from right to left, ending with the scanned square, treating all blanks as 0. In this case, 110101, so $r = 53$. (Here, the 1’s position is the scanned square.)

Ex. 2: Suppose the two numbers are 93 (left) and 42 (right). What is on the tape and where is the pointer?

$$93 = 1011101 \text{ and } 42 = 101010$$

BBB1B111B1B1B1BBBBB

↑

b. Effect of actions taken by the machine

i) What happens to the left and right numbers ($p$ and $r$) when M moves one square left?

In Ex. 1: $p = 11011$ or 27; $p' = 1101$ or $(27-1)/2$

$r = 110101$ or 53; $r' = 1101011$ or $107 = (53*2) + 1$.

In general, what happens if M moves left or right is summarized by this table:

<table>
<thead>
<tr>
<th></th>
<th>p odd</th>
<th>p even</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>M moves left</td>
<td>$p' = (l-1)/2$</td>
<td>$p' = p/2$</td>
<td>$p' = \text{quo}(p, 2)$</td>
</tr>
<tr>
<td>r odd</td>
<td>$r' = 2r + 1$</td>
<td>$r' = 2r$</td>
<td>$r' = 2r + \text{rem}(p, 2)$</td>
</tr>
<tr>
<td>r even</td>
<td>$r' = (r-1)/2$</td>
<td>$r' = r/2$</td>
<td>$r' = \text{quo}(r, 2)$</td>
</tr>
</tbody>
</table>

M moves right $p' = 2p + 1$ $p' = 2p$ $p' = 2p + \text{rem}(r, 2)$

$r' = (r-1)/2$ $r' = r/2$ $r' = \text{quo}(r, 2)$

ii) What happens if M prints 0 or 1?

In Ex. 1: $p = 11011$ or 27; $p' = 11011$, unchanged

[Print 0] $r = 110101$ or 53; $r' = 110100$ or 52

The left number is unaffected. The right number either increases by 1, decreases by 1 or stays the same. It depends what is currently scanned. The number being scanned is 0 if $r$ is even and 1 if $r$ is odd. Define

$$\text{scan}(r) = \text{rem}(r, 2) \quad -- \text{the number currently scanned}$$

Then the following table tells us what happens to $p$ and $r$.

<table>
<thead>
<tr>
<th></th>
<th>$\text{scan}(r) = 0$</th>
<th>$\text{scan}(r) = 1$</th>
<th>General</th>
</tr>
</thead>
<tbody>
<tr>
<td>M prints 0</td>
<td>$p' = p$</td>
<td>$p' = p$</td>
<td>$p' = p$</td>
</tr>
</tbody>
</table>
\[ r' = r \quad \quad \quad \quad r' = r - 1 \quad \quad \quad \quad r' = r - \text{scan}(r) \]

M prints 1 \[ p' = p \quad \quad \quad\quad \quad \quad \quad p' = p \quad\quad \quad \quad p' = p \]
\[ r' = r + 1 \quad \quad \quad\quad \quad r' = r \quad\quad \quad \quad r' = r + (1 - \text{scan}(r)) \]

iii) Summary. The book uses functions \textit{newleft} and \textit{newright} instead of \( p' \) and \( r' \), and indexes these functions by the four possible actions:

\[ \begin{align*}
0 & : \text{print 0} \\
1 & : \text{print 1} \\
2 & : \text{move left one square} \\
3 & : \text{move right one square}
\end{align*} \]

Using the above information, we have:

- **[Print 0]** \( \text{newleft}_0(p, r) = p \) \quad \quad \quad \text{newright}_0(p, r) = r - \text{scan}(r) \)
- **[Print 1]** \( \text{newleft}_1(p, r) = p \) \quad \quad \quad \text{newright}_1(p, r) = r + (1 - \text{scan}(r)) \)
- **[Move left]** \( \text{newleft}_2(p, r) = \text{quo}(p, 2) \) \quad \quad \quad \text{newright}_2(p, r) = 2r + \text{rem}(p, 2) \)
- **[Move right]** \( \text{newleft}_3(p, r) = 2p + \text{rem}(r, 2) \) \quad \quad \quad \text{newright}_3(p, r) = \text{quo}(r, 2) \)

Or more simply, letting \( a \) stand for the act number 0, 1, 2 or 3:

\[ \text{newleft}(p, r, a) = \begin{cases} 
\{ p \} & \text{if } a = 0 \text{ or } a = 1 \\
\{ \text{quo}(p, 2) \} & \text{if } a = 2 \\
\{ 2p + \text{rem}(r, 2) \} & \text{if } a = 3
\end{cases} \]

\[ \text{newright}(p, r, a) \text{ defined similarly by cases.} \]

Each of these functions is p.r., since the conditions and the functions used in the definition by cases are all p.r.

\section*{c. Codes for initial and final position}

**Initial**: BB11…1BBBB \((x+1 1's)\)

\[ \uparrow \]

Clearly, the left number \( p = 0 \).

To figure out \( r \), note that a string of \( k \) 1’s is the number \( 2^k - 1 \). Hence, \( r = 2^{(x+1)} - 1 \). In the text, this is defined by the function

\[ \text{start}(x) = 2^{(x+1)} - 1 \]

which is p.r.

**Final**: BB11…11BBB \((f(x) + 1 1's)\) \quad \text{if in standard position}

\[ \uparrow \]
\[ p = 0; r = 2^{f(x)+1} - 1. \]

To get from the final \( r \) to \( f(x) \), we basically want to take the log in base 2 of \( r \). Recall:

\[
\begin{align*}
\lg(1,2) &= 0 \\
\lg(2,2) &= \lg(3,2) = 1 \\
\lg(4,2) &= \ldots = \lg(7,2) = 3 \\
&\ldots \\
\lg(x,2) &= \text{greatest } w \leq x \text{ such that } 2^w \leq x.
\end{align*}
\]

This function is p.r. [max. with p.r. bound]. Then if \( r \) is the final right number on the tape, \( f(x) \) is given by the function

\[
\text{value}(r) = \lg(r,2), \text{ where } r \text{ is the final right number on the tape.}
\]

We also need a way to check whether the machine is in standard position when it halts. If either \( p \neq 0 \) (so there are 1’s to the left) or \( r \neq 2^{\lg(r,2)+1} - 1 \) (so that the right number is not just a solid block of 1’s), the machine is NOT in standard position. Define nonstandard(\( p, r \)) to be the characteristic function for the relation

\[
[p \neq 0 \lor r \neq 2^{\lg(r,2)+1} - 1],
\]

which is a p.r. relation. Then the machine is in standard position if and only if nonstandard(\( p, r \)) = 0.

2. Configuration

Define a function of triplets to \( \mathbb{N} \):

\[
\text{triple}(p, q, r) = 2^p \cdot 3^q \cdot 5^r
\]

And reverse function from \( \mathbb{N} \) to triplets. Given a code number \( n \):

\[
\begin{align*}
\text{left}(n) &= \text{lo}(n, 2) \\
\text{state}(n) &= \text{lo}(n, 3) \\
\text{right}(n) &= \text{lo}(n, 5)
\end{align*}
\]

These functions are all p.r.

Check: We can go back and forth between triplets and single numbers that are divisible by no primes except 2, 3, 5.

\[
\text{Triple}(5, 3, 4) = 2^5 \cdot 3^3 \cdot 4^4
\]

\[
\text{left}(450) = 1; \text{state}(450) = 2; \text{right}(450) = 2, \text{ since } 450 = 2 \times 3^2 \times 5^2
\]
Thus, we can encode complete information about the contents of the tape and position of
the cursor and the current state of the machine as a single number, triple\((p, q, r)\).

3. Codes for Turing machines

Recall the way we coded Turing machines in chapter 4 (with slight modification):

- States: Code \(q_i\) as \(i\), as in chapter 4. But now code the halted state as 0.
- Symbols: Code B as 0, 1 as 1
- Actions: B, 1, L, R as 0, 1, 2, 3

Each machine is coded as a finite sequence of length \(4k\), where \(k\) is number of states
(excluding halted state).

Example:  
\[
\begin{array}{c}
1:R \\
1 \\
B:1 \\
0
\end{array}
\]

\(1q_0, Rq_1 \Rightarrow 1, 0, 3, 1 \Rightarrow 2^1 3^0 5^3 7^1\)

It is convenient to count the first entry in such a sequence as the 0th entry. The even entries
give us acts; the odd entries give us the new state to go into.

In general, given such a sequence, if in state \(q\) and scanning symbol \(i\), the next act to
perform is given by the entry in position

\[4(q - 1) + 2i\]

and the new state is given by the entry in position

\[4(q - 1) + 2i + 1.
\]

Hence if \(m\) is the code for the \(m\)'th Turing machine, \(q\) is the current state and \(r\) is the current
right number (so that the currently scanned symbol is \(\text{scan}(r)\)), we can define the functions
that give us the next act and the new state by:

- action\((m, q, r) = \text{entry}(m, 4(q - 1) + 2 \text{ scan}(r))\)
- newstate\((m, q, r) = \text{entry}(m, 4(q - 1) + 2 \text{scan}(r) + 1)\)

Here, entry\((m, k)\) is the \(k\)'th entry in the code number \(m\) for the \(m\)th Turing machine. For
the above example, \(m = 2^1 \cdot 5^3 \cdot 7^1\), so entry\((m, 0) = 1\), entry\((m, 1) = 0\), entry\((m, 2) = 3\),
entry\((m, 3) = 1\).

These functions are all p.r.
4. Defining a p.r. function that tracks each Turing machine

Want to define Conf(m, x, t) where m codes the machine M that computes f and t is a stage in the computation of f(x), such that for each stage t up to and including halting, Conf(m, x, t) = triple(p, q, r) where p, q, and r are the left number, state and right number at stage t.

If we can show that this Configuration function is p.r., we’ll be essentially done. For the state q will be positive unless and until M halts, when it becomes 0. So we can find the value of f(x) by using Minimization: it is the value of $\text{lo}(r, 2)$ at the least t where $q = \text{lo}(\text{Conf}(m, x, t), 3) = 0$. We’ll define Conf by primitive recursion.

**Base case:** Conf(m, x, 0) = triple(0, 1, start(x)) [initially, $p = 0$, $q = 1$ and $r = \text{start}(x)$]

**Recursion clause:** We want to define Conf(m, x, $t'$) in terms of m, x and Conf(m, x, t).

Let $c = \text{Conf}(m, x, t)$. Then

i) $p = \text{left}(c)$, $q = \text{state}(c)$ and $r = \text{right}(c)$ are p.r.

ii) $a = \text{action}(m, q, r)$ and $q^* = \text{newstate}(m, q, r)$ are p.r.

iii) $p^* = \text{newleft}(p, r, a)$ and $r^* = \text{newright}(p, r, a)$ are p.r.

iv) Conf(m, x, $t'$) = triple($p^*$, $q^*$, $r^*$) is p.r.

So by composition, Conf(m, x, $t'$) is defined via a p.r. function from (m, x, Conf(m, x, t)).

5. Finally: f is recursive.

By definition, if $t$ is the stage at which M halts (no prescribed action), then M is in state 0 for the first time at $t$, and the centre value $q$ of Conf(m, x, t) is 0 for the first time. We need a function that tells us if M has halted in standard position at stage $t$. This happens $\leftrightarrow$ [state = 0 and nonstandard(p, r) = 0]. Define

$$\text{standardhalt}(m, x, t) = \min\{\text{state}(\text{Conf}(m, x, t)), \text{nonstandard}(p, r)\}$$

for $p = \text{left}(\text{Conf}(m, x, t))$ and $r = \text{right}(\text{Conf}(m, x, t))$.

Then standardhalt(m, x, t) is p.r., and is either never 0 if the computation of f(x) does not halt in standard position, or 0 for the first $t$ where the computation halts in standard position. Let

$$\text{halt}(m, x) = \begin{cases} \text{least } t \text{ such that standardhalt}(m, x, t) = 0 & \text{if there is such a } t \\ \text{undefined} & \text{otherwise.} \end{cases}$$

Then halt is recursive since standardhalt is p.r.

If the machine does halt in standard position at stage $t$, then its output is given by
value(Conf(m, x, t)).

So putting it all together:

\[ F(m, x) = \text{value}(\text{Conf}(m, x, \text{halt}(m, x))) \]

is, first of all, recursive by composition. Second, \( F(m, x) \) is exactly the value of the function computed by the \( m \)'th Turing machine applied to input \( x \) (where defined, and otherwise undefined). So for the particular \( m \) that codes \( f \), we have

\[ f(x) = F(m, x) \]

which shows that \( f \) is recursive and completes the argument.

We know have: \( T = A = R \)

8.2 Universal Turing Machines

We won't cover this in detail; just make a few remarks. Definitely skip Theorem 8.5.

1. Kleene Normal Form Theorem: In obtaining a recursive function from initial functions, the operation of minimization needs to be used only once.

   \textbf{Proof}: In the above process defining \( F(m, x) \), minimization was used only once, in the definition of \( \text{halt}(m, x) \). If \( f \) is recursive, we can find an abacus machine and then a Turing machine \( M \) that computes \( f \), and by the above proof, we have \( f(x) = F(m, x) \) where \( m \) is the code for \( M \).

2. Alternative definitions of Turing computability.

   We could have defined Turing-computable functions using one-way tape or machines that allow more than just two symbols. But the class of Turing-computable functions would be just the same.

   \textit{1-way tape}: The proof does depend upon the argument in chapter 6 which constructs a Turing machine that computes an abacus-computable function while never moving to the left of its starting square. (I did not bother doing it with this restriction, but you can.) If \( f \) is a Turing-computable function in our sense, then by the above, \( f \) is recursive, hence abacus-computable, hence computable by a 1-way Turing machine.

   \textit{More than two symbols}: Suppose \( f \) is Turing-computable by \( M \) with more than two symbols. With three symbols, the proof just given can be re-done to show that such a function is recursive: just use base 3 for the Wang coding and represent Turing machines by sequences of length \( 6k \) instead of \( 4k \). Similarly with four symbols, use base 4 and
sequences of length \(8k\). But then the function \(f\) is abacus-computable and thus computable by a two-symbol Turing machine.

3. Universal Turing machines.

We proved that there is a 2-place recursive \(F\) such that for every one-place recursive \(f\), there exists \(m\) such that

\[ f(x) = F(m, x). \]

The same thing holds for \(n\)-place functions. These functions are called \emph{universal} functions.

But now, \(F\) itself is recursive and so is computed by some Turing machine \(U\). \(U\) is called a \emph{universal Turing machine}. This means that for any Turing machine \(M\), if \(f(x)\) is the 1-place function computed by \(M\), there is some \(m\) such that \(f(x) = U(m, x)\). So a general-purpose computer exists that can mimic any special-purpose computer just by being given an extra input that codes instructions as to what machine it is to mimic.