# Church-Turing Thesis

1. All reasonable formulations of the intuitive notion of effective computability are equivalent.
2. Turing machine computability is a reasonable formalization of effective computability.

Recall that the WHILE language has the following grammar:

$E ::= x \mid d \mid \text{cons } E_1\ E_2 \mid \text{hd } E \mid \text{tl } E \mid =?\ E_1\ E_2$
$C ::= X := E \mid C_1; C_2 \mid \text{while } E \text{ do } C$
$P ::= \text{read } X; C\text{: write } Y$

Here is the GOTO language. It is like WHILE, but with simpler, more primitive operations:

$I ::= X := \text{nil} \mid X := Y \mid X := \text{hd } Y \mid X := \text{tl } Y \mid X := \text{cons } Y\ Z \mid$
$\quad \text{if } X \text{ goto } \ell \text{ else } \ell' \mid \text{goto } \ell$

where $\ell$ and $\ell'$ are numeric labels. Instructions all have labels:
$LI ::= I\ell : I \mid LI_1; LI_2$

The data is the same as for the WHILE language, with D built from nil and (x.y).

For GOTO language, the input and output are always in variable X, so we never bother with a read or write in the program.

We can write a compiling function from WHILE-programs to GOTO-programs. Recall that in section 3.7 we wrote a compiling function to compile a general WHILE-program to one in which each command includes at most one command. We illustrate how to compile from this kind of WHILE-program to GOTO programs via the example of the reverse program:

```
read X;
  Y := nil;                           I1: Y := nil;
  while X do                          I2: if X goto 3 else goto 7;
    Z := hd X;           □            I3: Z := hd X;
    Y := cons Z Y;                    I4: Y := cons Z Y;
    X := tl X;                        I5: X := tl X;
write Y                               I6: goto 2;
                                      I7: X := Y;
```

Based on the ideas shown in the example, we can write a compiling function from WHILE programs to GOTO programs. Details are left to the reader.

It is also easy to write a compiling function from GOTO to WHILE programs. Just add an extra variable, PC, to keep track of the program counter and use while and case statements (recall we earlier showed how to define case statements in the WHILE languages). Here is the translation of the GOTO version of reverse back into WHILE:

```
read X;
   PC := 1;
   while PC do
      case PC of
         1 □  { Y := nil; PC := 2 }
         2 □  { if X then PC := 3 else PC := 7 }
         3 □  { Z := hd X; PC := 4 }
         4 □  { Y := cons Z Y; PC := 5 }
         5 □  { X := tl X; PC := 6 }
         6 □  { PC := 2 }
         7 □  { X := Y; PC := nil }
write X
```

As before, the details of the compiling function are left to the reader.

We can simulate a Turing machine by a WHILE program in a similar way, but we first must deal with the problem that Turing machines and WHILE programs take different kinds of data. In particular, Turing machine tapes hold strings from $\{B,0,1\}^*$, while WHILE program variables hold elements of D. To simulate a Turing machine, we must encode its data as elements of D. We define $c_{01B}$: $\{B,0,1\}^*$ □ D by $c_{01B}(a_1 \ldots a_n) = (a_1^{\ddagger} \ldots a_n^{\ddagger})$ – a list of elements from D, where

$B^{\ddagger}$ = nil, $0^{\ddagger}$ = (nil.nil), and $1^{\ddagger}$ = (nil.(nil.nil)).

We will encode the current configuration $(q, L\underline{a}R)$ as follows. A new variable, State, will remember the current state (state $q_i$ will be held with State = i). The tape contents will be held in 3 variables, Lr, C, and Rt. When the current configuration has tape contents as $L\underline{a}R$, then variables will be set so that Rt = $c_{01B}(R)$, C = $c_{01B}(a)$, and Lr = $c_{01B}(L^{rev})$, where $L^{rev}$ is the reverse of L. We keep the reverse of L in Lr so that the character just to the left of the tape head will be the head of Lr. This will make it easier to shift right or left by modifying the contents of Rt, C, and Lr.

The actual simulation of a Turing machine by a WHILE-program is very similar to the way we simulate GOTO programs, except we use cases based on both the state and input. For example, suppose □$(q_1, B) = (q_3,$ □ $)$, □$(q_1, 0) = (q_5,$ □$)$, and □$(q_1, 1) = (q_8,$ □ $)$. Then if $q_1$ is the start state and x is the input, we translate the program as:

```
read Rt;            (* encoded input *)
  State := 1;
  Lr := nil;        (* empty list *)
  C := nil;         (* code for B *)
  while State do
    case State, C of
      1, nil            □ { State := 3; Lr := cons C Lr;
                            C := hd Rt; Rt := tl Rt; }

      1, (nil.nil)      □ { State := 5 }

      1, (nil.(nil.nil)) □ { State := 8; Rt := cons C Rt;
                            C := hd Lr; Lr := tl Lr; }
      …
  Rt := trim Rt;    (* throw away everything after 1st B *)
write Rt
```

We actually cheated here by not worrying about the cases where we move left when Lr = nil or move right when Rt = nil. However, that is easy enough to fix, and is left as an exercise for the reader.

Finally, we can complete our proofs of equivalence if we can show that we can simulate a GOTO program with a Turing machine program. As before, we have to encode the input as the data in a GOTO program is given as an element of D, while Turing machines deal with strings from $\{B,0,1,\ldots\}^*$. We will make our lives a bit easier by using a multi-tape Turing machine whose alphabet includes "B", "0", "( ", ".", and ")". We will encode elements of D as follows $c_D(\text{nil}) = 0$, and $c_D(\,(x.y)\,) = (\,c_D(x)\,.\,c_D(y)\,)$. We demonstrate how the compiling function would work by sketching the translation of the GOTO program for reverse, above, to a Turing machine program.

We will include a separate tape for every variable in the program as well as a scratch tape. Each tape (aside from possibly the scratch tape) will hold the encoding of a single element of D, bounded on both sides by "B"'s. If the contents of the tapes are $(x_1, \ldots, x_n)$ then the "standard configuration" for the tapes is $(\underline{B}x_1, \ldots, \underline{B}x_n)$ – that is with the tape head over the "B" just to the left of the non-"B" contents of each tape. When we write "std config" below, that means execute a Turing machine program to move each of the tape heads to the standard configuration. When we write "erase X" or "copy Y to X", that should be interpreted as commands operating on the tapes associated with X and Y.

The Turing machine will have a state $q_i$ for each instruction label i of the GOTO program, though it will also need a number of other states for intermediate steps. The program below is descriptive and would take a lot of tedious (but straightforward) work in order to convert it to a true Turing machine program. However, it should be sufficiently detailed for you to be able to understand how the simulation works.

Simulating the GOTO-language reverse program from above:

$q_1$: erase Y, write “0” on Y (representing nil), std config, go to $q_2$.

$q_2$: if X = 0, std config, go to $q_7$. Otherwise std config, go to $q_3$.

$q_3$: erase Z, copy hd X to Z, std config, go to $q_4$.

$q_4$: write “(“ to scratch, copy Z to scratch, write “.”, copy Y to scratch, write “)”, erase Y, copy scratch to Y, erase scratch, std config, go to $q_5$.

$q_5$: copy tl X to scratch, erase X, copy scratch to X, erase scratch, std config, go to $q_6$.

$q_6$: go to $q_2$.

$q_7$: erase X, copy Y to X, std config, halt.

Again, technically, we must show how to write Turing machine programs to do all of the things listed with each state, but it is not difficult to show that we can write the necessary Turing machine programs.

We have shown (or, more accurately, suggested) the following compilation functions:
WHILE-programs □ GOTO-programs
Turing-machines □ WHILE-programs
GOTO-programs □ Turing-machines
Thus all are equivalent.

Last time we also showed that TM’s with alphabet {B, O, 1} and one tape are equivalent to multi-tape/non-deterministic/rich alphabet TM’s. Other equivalent models of computation include the lambda calculus (a very simple functional language whose only operations are function definition and application – and no constants) and generalized grammars. The fact that all of these are equivalent provides strong evidence for the truth of the Church-Turing thesis.