

Unit B7 Supplement, 3/7/05

Part 1: Church-Turing Thesis

As we discussed in class, the Church-Turing thesis states that the informal notion of computation is captured by Turing machines. Although it is called a “thesis” (i.e., proposition), it involves an informal notion, which can never be “proven.” That is why, it should be considered more as a definition. However, due to the equivalence between Turing machines with a wide range of other formalisms, most TM variants, real computers (with qualifications), Unlimited Register Machines, recursive functions, lambda calculus, etc., this thesis indeed captures the essence of at least algorithmic part of computation (more about non-algorithmic computation later). That is, if there are so many different ways to represent the same notion, albeit informal, such a notion must have something deep. For example, through the connection between TMs and recursive functions, we can often translate results in computer science and mathematics back and forth, providing different perspectives useful in different contexts.

Here are examples of some other abstract models (see the definitions in Exercise B4 and Unit B5/B7). If you skim the examples below, you may realize why the TM is the most popular model of computation.

Unlimited Register Machine (URM)

- Example: Addition $r_1 + r_2$
 - Reset $r_3 = 0$, with *Zero*
 - Count up r_1 and r_3 until $r_3 = r_2$, iterating the basic operations including *Successor* and *Jump*
 - The result of the addition is in r_1 .

The simulation of a URM with a TM is fairly intuitive. The other direction requires a few tricks. For example, we will need to be able to handle all symbolic computation with URMs; this can be done by associating all the tape symbols with unique natural numbers (ASCII code?). A URM can then represent the TM tape within its register, encode the tape position in a special register, simulate the basic TM operations as subroutines, etc.

Recursive Functions

- Example: Addition $add(x, y)$
 - $add(x, 0) = x$
 - $add(x, y + 1) = Successor(add(x, y))$ [using the basic function *Successor*]
- Example: Multiplication $mult(x, y)$
 - $mult(x, 0) = 0$
 - $mult(x, y + 1) = add(mult(x, y), x)$ [using pre-defined functions]
- Example: Factorial $fac(x)$
 - $fac(0) = 1$
 - $fac(y + 1) = mult(fac(y), Successor(y))$ [using pre-defined functions]

The simulation of a recursive function with a TM is fairly intuitive. The other direction can go indirectly through the simulation of a URM with a recursive function. This simulation is still

involved. First, it is not very intuitive to simulate certain URM operations with a recursive function (e.g., Jump as a recursive function?). One way to cope with this situation is to limit the simulation to the cases where the given URMs that can compute a function (if not, we can implement a looping function). So, by this assumption, we know that the URM (program) P receives inputs $\mathbf{x} = x_1, x_2, \dots, x_k$ (boldface for a list or “vector”) in the respective registers and computes the result $P(\mathbf{x})$ to be left in R_1 . Let us define two functions. First, $result(\mathbf{x}, t) = r_1$ computing the result. Second, $next(\mathbf{x}, t)$ would return 0 if the computation terminates, or would return the next step (e.g., a function ID) if the computation still continues. If a function is defined, $P(\mathbf{x})$ must terminate at some step t_i (i.e., $t_i =$ the least t such that $next(\mathbf{x}, t) = 0$), and the result $result(\mathbf{x}, t_i)$ appears R_1 . Otherwise, $P(\mathbf{x})$ would loop (i.e., $next(\mathbf{x}, t) \neq 0$ for any t). In either case, the desired function (which is computed by P) is defined as a recursive function $result(\mathbf{x},$ the least t such that $next(\mathbf{x}, t) = 0$)

Lambda Calculus

- Example: Identity function
 - $\lambda x.x$
- Example: Church numeral
 - Representation of a natural number n : $\lambda f.\lambda x.f^n(x)$
 - $0 = \lambda f.\lambda x.x$ [Note: $\lambda f.\lambda x.f^0(x)$]
 - $Succ = \lambda n.\lambda f.\lambda x.(f((n f) x))$
 - $1 = Succ\ 0 = \lambda f.\lambda x.(f(((\lambda f.\lambda x.x) f) x)) = \lambda f.\lambda x.(f(\lambda x.x x)) = \lambda f.\lambda x.(f x)$
[Note: $\lambda f.\lambda x.f^1(x)$]
- Example: Conditional
 - $T = \lambda x\lambda y.x$ [i.e., picking the first argument]
 - $F = \lambda x\lambda y.y$ [i.e., picking the second argument]
 - $B\ M\ N$ (if B then M else N) where B is T or F

The simulation of lambda calculus with TMs would be intuitive, although tedious as usual. The other direction can go through recursive functions.

Part 2: Rice’s Theorem

Although you may not have noticed, many of the problems in Module B Review Exercise (Required Problem B, Options 1 through 6) can be answered very concisely by using Rice’s theorem. As the option problems indicate, many general problems in computer science refers to a non-trivial property of a certain general procedure. Thus, this theorem is widely applicable and useful to pinpoint the limitations of algorithmic computation. This is a review of this theorem with some insight into why this theorem holds. First, the definition again.

Rice’s Theorem: A non-trivial property of a TM is undecidable.

Note that a “non-trivial” property refers to any property of a TM that is not “constant” in the following sense. If the problem is to return “yes” or implement a constant function (e.g., *Zero*), the behavior is fixed and always decidable. Such a constant behavior is considered “trivial.” Everything else, including very simple problem such as telling if the input has at least one ‘\$’ (Comprehensive Exercise Required Problem A), is non-trivial. We also know that each TM is

associated with a language which the TM at least “recognizes” (but not necessarily decides). The operation of a TM can also be associated with an algorithm (if terminates), procedure, etc. So, we might consider the following corollary as a slightly more applicable form of Rice’s theorem.

Corollary: A non-constant property of a TM, language, algorithm, procedure, recursive function, etc. is undecidable.

Let us see why this is the case, using Comprehensive Exercise Required Problem B, defined as $TM_{DOLLAR} = \{M \mid \text{TM } M \text{ accepts a string that contains at least one ‘\$’}\}$. First this is a problem about TMs (cf. Required Problem A, which is a problem about strings). Although it is not explicitly indicated as in $ACCEPT_{TM} = \{\langle M, w \rangle \mid \text{TM } M \text{ accepts } w\}$, TM_{DOLLAR} still needs to pair the given TM with all sorts of string inputs. But the sequence of strings given to the TM, interpreted as a language, may be a non-TM-recognizable one. Then, regardless of the TM at work at that point, such an input cannot be recognized. So, this problem must be undecidable. In a similar fashion, we can intuitively understand that Rice’s theorem and the corollary above, referring to any nontrivial property of a TM.

For this particular problem TM_{DOLLAR} , we can narrow down the computability class further. Since it would be possible to create a modified UTM that simulates the given TM on an input. If the modified UTM evaluates that the given TM detects at least one ‘\$’, we can pick up all the positive cases. Thus, it is semi-decidable, as in $ACCEPT_{TM}$ and $HALT_{TM}$.

// End