Craig Silverstein
Gates 470
Phone: (72)3-1787
Office Hours: Wed 4:30–5:30, Fri 1–3, Gates 470
Section: Monday 2:15–3:15, 380-380W

Computer Science 154
School of Engineering
Stanford University

**Section Handout #6**

# The Universal Turing Machine

We tell you to think of a Turing Machine as a computer program, and this is helpful, but you must be careful. A TM is an *executable* program. The *source code* of a program is actually an *encoding* of a Turing Machine in an alphabet easier for people to understand.

The distinction is crucial to everything we do from here on out. A TM, like an executable, takes input and acts on it. A TM encoding, like source code, does not run by itself. However, by examining the source code, it is possible to figure out information about the program it represents.

In real life, only one executable is written from scratch: the compiler (or the interpreter). Let's concentrate on the interpreter case: we'll suppose we have an executable of a BASIC interpreter. Everything is actually written in BASIC source code. When you want to run a BASIC "program," you give the source code to the interpreter, along with any input, and the BASIC interpreter looks at the source code. The interpreter figures out what the source code wants to do on the input, and the interpreter does it. In no case was an executable of your source code produced; instead the interpreter *emulated* the program that your source code would compile to.

The BASIC interpreter is really a Universal Turing Machine. It's a single TM (that is, program) that takes a description of another TM, takes an input, and does what that other program would do without ever actually constructing the TM it was given. Just like the BASIC source code was never turned into machine language, the input TM is never actually constructed. Instead, by looking at the encoding, the Universal TM figures out what to do.

Now the Universal TM is just one example of a program that takes source code as input. Lots of programs you use every day work like this. Some of them do not treat the source code as special (eg a word counter), some recognize that it is source code but only in a shallow, syntactic way (an auto-formatter), and some analyze the source code and take action based on that (our BASIC interpreter).

Most of the Turing Machines we talk about from now on will take source code as at least part of their input. That is, the TM will take as input the encoding of another TM. When we talk about $M$, we are talking about a machine, but when we talk about $\langle M\rangle$ we are talking about its encoding. $M$ is a tuple; $\langle M\rangle$ is a string.

# Self-referential Turing Machines

The fact that TMs can work on encodings of other TMs means that TMs can be self-referential. And whenever you get into self-reference ("this sentence is false") you get into the realm of paradox. And indeed, it's easy to construct a paradox. Suppose we have a TM $M$ and we give it its own encoding as input: that is, we run $M$ on the string $\langle M\rangle$. Now, to $M$ the input isn't anything special, so it will do what it always does to strings: accept, reject, or go into an infinite loop. Consider the set $\{M : M \text{ does not accept} \langle M\rangle\}$. This set, consisting of machines instead of strings, is not a language. But $\{\langle M\rangle : M \text{ does not accept} \langle M\rangle\}$ *is* a language, call it $L$.

Now suppose we had a Turing Machine deciding $L$, call it $M_L$. That is, for all strings in $L$ it accepts, and for all strings not in $L$ it rejects. What does $M_L$ do on its own encoding $\langle M_L\rangle$? Well, if $M_L$ accepts, then $M_L$ has accepted its own encoding, and therefore $\langle M_L\rangle$ is not in $L$. Therefore

$M_L$ has just accepted a string not in $L$, which is bad news. On the other hand, if $M_L$ rejects, then $\langle M_L \rangle$ *is* in $L$. Then $M_L$ has rejected a string in $L$. Again, bad news. There's no way $M_L$ can do the right thing. When we can't do the right thing, this is proof (in mathematics, though not in real life) that we don't exist. The conclusion: there's no TM that decides $L$; that is, $L$ is not recursive. More commonly, in this situation we use a synonym for "recursive": $L$ is not *decidable.*

## Reductions

Once we've proven one language is undecidable, we can prove a lot more are as well. It goes by contrapositive: we prove "if our language were decidable, so would be that hard language over there. But we know that language isn't decidable. Therefore, we aren't either." How do we do this first if/then step? Remember: this step *assumes* our language is decidable, that is, there exists a TM deciding our language. Now we have to use this (imaginary) TM to prove the hard language is decidable.

There are two ways to go about this: **construction** and **oracle reference**. I will illustrate both these methods on the following reduction: Our language is $L_{\text{HALT}} = \{\langle M \rangle, w : M \text{ halts on input } w\}$. The known-to-be-hard language is $L_{\text{ACCEPT}} = \{\langle M \rangle, w : M \text{ accepts } w\}$.

**Consult the oracle:** Treat the decision process for our language as an oracle that you can query. That is, if you want to know if $w$ is in our language, you ask the oracle: "Is $w$ in our language?" The oracle then answers, correctly, "yes" or "no." You use this oracle to construct a decision procedure for the known-to-be-hard language.

Here's how we do the example. We assume an oracle for deciding $L_{\text{HALT}}$. To decide $L_{\text{ACCEPT}}$ we take the input $\langle M \rangle, w$ and ask the oracle if $M$ halts on $w$. If the oracle says "no," then we reject the input: $M$ does not accept $w$. If it says "yes," we simulate running $M$ on $w$ (we include the Universal TM as a subroutine) and decide whatever $M$ decides, safe in the knowledge it will decide something eventually.

**Construct the deciding TM:** Take the TM that decides our language, and massage it into a TM that decides the known-to-be-hard language.

Here's how we do the example. We pretend we have a TM that takes input $\langle M \rangle, w$ and accepts if $M$ halts on $w$ and rejects otherwise. We change the TM as follows: when before the TM would go to $q_a$ (accepting because $M$ halts), instead we have it go to $q_u$, the start state of a Universal TM. Our new TM will reject if $M$ does not halt on $w$ (which is good) and otherwise will have the UTM simulate $M$ on $w$, accepting if $M$ would accept $w$ (good) and rejecting if $M$ would reject $w$ (also good). We know it halts because the non-halting case was dealt with before we invoked the UTM.

## Hints on Problem Set 5

**(2a)** Remember, R.E. means (here) if $M$ accepts some word, we stop; but if it accepts no word, our TM is allowed to go on forever.

**(2b)** $L_{\text{NOT-EMPTY}}$ is "our" language and $L_{\text{HALT}}$ is the known-to-be-hard language.

**(2c)** This language is not the complement of the language in (2a), but it's close...

**(3)** I recommend $L_{\text{NOT-EMPTY}}$ for the known-to-be-hard problem. Your algorithm to decide $L_{\text{NOT-EMPTY}}$ should look like this: "Take the input machine $M$ and modify it in some way. Then ask the oracle if the new machine goes off the left end on any input. If it does, .... If not, ...." Tip for modifying $M$: make it so that $M$ never falls off the left end in the normal course of operation. Then add new states that force the head off the left end in situations we care about.