Lecture #12: Universal Turing Machines

Lecture Presentation

A First Problem To Be Solved

Function To Be Computed

Let $\Sigma_b = \{0, 1\}$, let $\Sigma_p = \{0, 1, \#\} = \Sigma_b \cup \{\#\}$, and let $L_{\text{pair}} \subseteq \Sigma_p^\star$ be the set of strings in $\Sigma_p^\star$ with the form

$$\mu\#\nu \tag{1}$$

where $\mu, \nu \in L_{\text{bin}}$, for the language $L_{\text{bin}} \subseteq \{0, 1\}^\star$ introduced in the lecture presentation for Lecture #10 —so that μ and ν are both ***unpadded binary representations*** of non-negative integers. Let $f_{\text{add}} : \Sigma_p^\star \to \Sigma_b^\star$ be defined as follows:

- If $\omega = \mu\#\nu \in L_{\text{pair}}$ so that, in particular, μ and ν are unpadded binary representations of non-negative integers n and m, then $f_{\text{add}}(\omega)$ is the unpadded binary representation of their sum, $n + m$.
- On the other hand, if $\omega \in \Sigma_p^\star$ and $\omega \notin L_{\text{pair}}$, then $f_{\text{add}}(\omega) = \lambda$.

High-Level Description

Consider an input string $\omega \in \Sigma_p^\star$. As noted above if $\omega \notin L_{\text{bin}}$ then $f_{\text{bin}}(\omega) = \lambda$, so that a Turing machine computing f_{add} when given μ as input.

Suppose, instead, that $\omega \in L_{\text{bin}}$ — so that $\omega = \mu\#\nu$ for a pair of strings μ and ν, which are unpadded representations of a pair of integers n and m, respectively. If we initialize a pair of integers k and ℓ to be n and m, respectively, and —while ℓ is positive — add one to k and subtract one from ℓ then, when this process ends, $k = n+m$, so $f_{\text{add}}(\omega)$ is the unpadded binary representation of ω — and this string should be returned as output when a Turing machine, computing f_{add}, is executed with input ω.

Another description of this algorithm — which refers directory to the unpadded binary representations of the integers named above — is shown in Figure 1, on the next page. This refers to the function $f_{+1} : \Sigma_1^\star \to \Sigma_2^\star$ (for $\Sigma_1 = \Sigma_2 = \{0, 1\}$) considered above, and a function $f_{-1} : \Sigma_1^\star \to \Sigma_2^\star$ which is as follows:

On input $\omega \in \Sigma_1^\star$:

1. `if` ($\omega \in L_{\text{pair}}$) `{`
   // Let $\mu, \nu \in \{0,1\}^\star$ such that $\omega = \mu\#\nu$
2. $\widehat{\mu}$:= μ
3. $\widehat{\nu}$:= ν
4. `while` ($\widehat{\nu} \neq 0$) `{`
5. $\widehat{\mu}$:= $f_{+1}(\widehat{\mu})$
6. $\widehat{\nu}$:= $f_{-1}(\widehat{\nu})$
   `}`
7. `return` $\widehat{\mu}$

   `} else {`
8. `return` λ

   `}`

Figure 1: A "High-Level" Description of a Multi-Tape Turing Machine to Compute f_{-1}

- If $\omega \in L_{\text{bin}}$ and ω is the unpadded binary representation of a ***positive*** integer n, then $f_{-1}(\omega)$ is the unpadded binary representation of $n-1$.
- If either $\omega = 0$ (so that $\omega \in L_{\text{bin}}$ and ω is the unpadded binary representation of 0) or $\omega \notin L_{\text{bin}}$ then $f_{-1}(\omega) = \lambda$.

Sketching a Proof of Correctness

Implementation-Level Algorithm

When designing a multi-Tape Turing machine, we should state how many tapes are used, and *how* each tape is used — including describing the information that will be stored on each tape, and how that information will be represented. It is ***not*** necessary (or, generally, helpful) to try to minimize the number of tapes used; you should probably focus on keeping things simple, instead.

For example, this high-level algorithm *could* be implemented, without too much trouble, using a two-tape Turing machine — but a three-tape Turing machine is easier to describe and understand — so that will be described instead.

It is also, generally, not necessary or helpful to try to minimize the number of steps used during an execution: If an implementation uses (a few) more steps than necessary, but this makes the implementation easier to describe and understand, then the "slower", but simpler implementation is probably the better choice.

- Tape #1 will be used to store the input string.
- If the test at line 1 is passed, so that the steps at lines 2 – 7 are reached and executed, then tape #2 will be used to store the string $\widehat{\nu}$.
- If the test at line 1 is passed, so that the steps at lines 2 – 7 are reached and executed, then tape #3 will be used to store the string $\widehat{\mu}$ — so that this tape will store $f_{\mathsf{add}}(\omega)$ when the execution, on input ω, ends.

Implementing the Test at Line 1

Implementing the Step at Line 2

Implementing the Step at Line 3

Implementing the Test at Line 4

Implementing the Step at Line 5

Implementing the Step at Line 6

Implementing Everything Else

The “Formal Description” is Too Big. What Can We Do?

What about “Proving Correctness”?

More About Universal Turing Machines

The preparatory reading for Lecture #12 introduced an alphabet, Σ_{TM}, that was used to define three languages:

- TM $\subseteq \Sigma_{\text{TM}}^{\star}$ is the language of valid encodings of Turing machines.[1]
- TM+I $\subseteq \Sigma_{\text{TM}}^{\star}$ is the language of encodings of Turing machines and input strings for those Turing machines.
- $A_{\text{TM}} \subseteq$ TM+I $\subseteq \Sigma_{\text{TM}}^{\star}$ is the language of encodings of Turing machines and input strings that are ***accepted*** by the encoded Turing machine.

As discussed in the readings, the languages TM and TM+I are both ***decidable***, and the language A_{TM} is ***recognizable***.

More About This

[1] We restricted attention, here, to Turing machines whose start states are different from both of their halting states. As discussed in the lecture notes, this restriction does not changes of languages that can be recognized, or decided, by Turing machines.

More about the Church-Turing Thesis

Concluding Remarks