Lecture 9: The Recursion Theorem

Michael Beeson

The problem of computing by recursion

- We still have not provided completely convincing evidence that every intuitively computable function is Turing computable, even though we have proved that the μ-recursive functions coincide with the Turing computable functions.
- For example, what about the Ackermann function?
- Another example: we previously showed how to assign indices to the μ-recursive functions, so that we can write φ_n for the μ-recursive function with index n. The question arises whether $\varphi_n(\mathbf{x})$ is a partial recursive function of both n and $\mathbf{x}$. If so then the partial recursive functions form a model of computation.
- In an earlier lecture, we had to postpone proving that, because we do not know a direct proof.
- Now, we see that $\varphi_n(\mathbf{x})$ is partial recursive if and only if it is Turing computable.
- But it is not trivial to write a Turing machine to compute $\varphi_n(\mathbf{x})$ (as a function of n as well as $\mathbf{x}$).

λ notation

We need a notation for "as a function of." Thus $f(x, y)$ can be considered as a function of x for each fixed y. We write this function as $\lambda x\, f(x, y)$. Then we have the basic law that

$$(\lambda x\, f(x, y))(u) = f(u, y)$$

Note that x is bound in the expression $\lambda x\, f(x, y)$.
We also use the λ-notation for several variables $\mathbf{x} = x_1, \ldots, x_n$.

$$\lambda \mathbf{x}\, f(\mathbf{x}, \mathbf{y})(\mathbf{u}) = f(\mathbf{u}, \mathbf{y}).$$

The process described by $\lambda \mathbf{x}$ is called, in English, "lambda abstraction" over the variables $\mathbf{x}$.

Example of λ-notation

This notation can bring more precision to the differential calculus, where the derivative operation really applies to a function. If we define, for example, $f(x) = x^2$, then the derivative Df is the function $\lambda x\, 2x$.

- We have $(Df)(u) = 2u$, something which is not written in calculus books.
- the expression (df/dx) is usually interpreted as what technically should be $(Df)(x)$ (which is $2x$).
- Functions are not the same as numbers. $\sin$ is a function, $\sin(x)$ is a number (if x is a number). Thus $\sin = \lambda x\ \sin(x)$.

The S_1^1 function

As a preliminary to the recursion theorem, we need to discuss the relationship between functions of n variables, and functions of $m < n$ variables obtained by fixing some of the n variables. For example, $g(x) = \lambda x\, f(x, y)$. The point is that code for computing g can be found by an algorithm, given code for computing f. This algorithm is a simple example of a program that treats programs as data.

- The traditional notation for this algorithm is S_1^1, which takes two arguments.
- The first argument is code for f.
- The second argument is x.
- The output is code for g.

The S_n^m functions

We treat this abstractly, using only the notion of a model of computation, given by a set X and some partial application functions $App(e, \mathbf{x})$ on X, where $\mathbf{x} = x_1, \ldots, x_n$.

We need computable functions $S_n^m(x)$ such that for $\mathbf{y} = y_1, \ldots, y_m$ we have

$$App(S_n^m(e, \mathbf{x}), y) = App(e, \mathbf{x}, \mathbf{y}) \tag{1}$$

We say e is an "index of f" if $f(\mathbf{x}) \cong App(e, \mathbf{x})$ for all $\mathbf{x}$.

The role of S_n^m can be described this way: $S^m, n(e, \mathbf{y})$ is an index of $\lambda \mathbf{x}\, f(\mathbf{y}, \mathbf{x})$ if e is an index of f.

See Kleene, p. 342, where the S_m^n functions are introduced for a model of computation that we have not studied, but you can see that the defining property is the same as here. The notation is due to Kleene.

Λ notation

- Kleene also introduced the Λ-notation, according to which $\Lambda\mathbf{x}\varphi(\mathbf{y},\mathbf{x})$ is an index of $\lambda\mathbf{x}\,\varphi(\mathbf{y},\mathbf{x})$.
- This works if we define $\Lambda\mathbf{x}\varphi_e(\mathbf{y},\mathbf{x})$ to be $S^m_n(e,\mathbf{y})$, where e is an index of φ. (See Kleene, p. 344.)
- Although the index e of $f=\varphi_e$ is not mentioned in the notation $\Lambda\mathbf{x}f(\mathbf{y},\mathbf{x})$, the notation without the index is mere human shorthand. You must have an index in mind.
- To clarify the notation: $\Lambda x\,\varphi_e(x,y)$ is an index of the function $\lambda x\,\varphi_e(x,y)$.
- φ_e refers either to a particular model of computation (e.g. Turing machines) or to an abstract model of computation, depending on the context.

The main property of Λ

$\Lambda\mathbf{x}\varphi(\mathbf{y},\mathbf{x})$ is an index of $\lambda\mathbf{x}\,\varphi(\mathbf{y},\mathbf{x})$.

Therefore

$$\phi_{\Lambda\mathbf{x}\varphi(\mathbf{y},\mathbf{x})}(u) = \lambda\mathbf{x}\,\varphi(\mathbf{y},\mathbf{x})(u) = \varphi(\mathbf{y},u)$$

S_n^m and λ in an abstract model

- The existence of computable S_n^m functions is the "detail" that we promised to supply when discussing the definition of "model of computation".
- A model of computation, by definition, has a set X and partial App functions for any number of variables, and the App functions for different numbers of arguments are connected by (1). The S_n^m functions must be computable, i.e. have indices.
- We also need to form indices of functions like $\lambda x, y F(y, x)$, or $\lambda y\, f(x, y, z)$, which cannot be directly defined using the S_n^m.
- It suffices to have the S_n^m plus "swap" functions $s_{i,j,n}$ that swap the i-th and j-th arguments of n arguments. Thus for example $s_{1,2,3}(e)$ is an index of $\lambda y, x\, \varphi_e(x, y, z)$.

The Turing model of computation has S_n^m functions

Proof. We describe a Turing machine that computes S_n^m. The input is a Turing machine e and a sequence $\mathbf{y}$. The output should be a Turing machine that takes input $\mathbf{x}$ and simulates the computation by e at input $\mathbf{x}\ \mathbf{y}$. We describe the desired output machine as a two-tape machine.

- Starting with $e\ \mathbf{y}$ on the main tape, and input $\mathbf{x}$ on the auxiliary tape, we insert $\mathbf{x}$ between e and $\mathbf{y}$, leaving a blank before and after $\mathbf{x}$.
- Then we return to the first square of $\mathbf{x}$ and run machine e. (We do not need to appeal to the universal machine.)
- What we need to output, however, is a one-tape machine that simulates this two-tape machine. But in the exercises, you have already shown how to transform a two-tape machine algorithmically into an equivalent one-tape machine.

That completes the proof of the existence of S_n^m functions.

The Turing model of computation has swap functions

- For notational simplicity, we give the proof of the existence of swap functions only for $s_{1,2,2}$, which swaps the order of arguments of a binary function.
- To compute $s_{1,2,2}(e)$, we first rename the start state of e.
- Our output will consist of a list of Turing machine instructions including those of e (with the renamed start state), plus additional instructions designed to interchange the order of the two inputs on the tape that e works on, for example by copying the first input to the right of the second, and then copying both inputs back to the original starting square.
- That completes the proof (or at least, that's as much as we are going to say about it).

The First Recursion Theorem

The first recursion theorem says that *any* recursion equation can be solved, in which the values of the function to be recursively defined are used in any way whatsoever.

Theorem (First recursion theorem)

Let H be Turing computable. Then there exists a Turing-computable g such that

$$g(\mathbf{x}) \cong H(\mathbf{x}, g(\mathbf{x}))$$

- Of course, the resulting function will generally be only partial; if it is total that will require an additional proof.
- For example, we can define $g(x) = g(x+1)$, which has one solution that is nowhere defined.
- This example also points up the fact that recursion equations generally do not have unique solutions, since any constant function also satisfies the equation.

The Second Recursion Theorem

We will prove the first recursion theorem as a corollary of an even more general theorem, which says that recursion equations are solvable even when F is allowed access to the code for the recursively defined function, not just to its values:

Theorem (Second recursion theorem)

Let F be Turing computable. Then there exists a Turing machine e computing a partial function g such that

$$g(\mathbf{x}) \cong F(\mathbf{x}, e)$$

For comparison, we repeat the equation from the First Recursion Theorem:

$$g(\mathbf{x}) \cong H(\mathbf{x}, g(\mathbf{x}))$$

The fixed-point theorem

The second recursion theorem in turn is a consequence of a theorem first stated and proved by Rogers in his 1967 book, *Recursion Theory*:

Theorem (Fixed-point theorem)

Let G be Turing computable and total. Then there exists an e such that for all x, $\varphi_e = \varphi_{G(e)}$.

The general setting for recursion theorems

Theorem

The three recursion theorems hold in any model of computation, i.e. set X with partial App functions and computable S_n^m and swap functions.

- Since the Turing-computable functions have S_n^m and swap functions, the result applies to that case.
- In the next slides, we will prove all four theorems in the abstract setting.

Proof of the first recursion theorem from the second

We can write the equation

$$g(\mathbf{x}) \cong H(\mathbf{x}, g(\mathbf{x}))$$

as

$$g(\mathbf{x}) \cong H(\mathbf{x}, App(e, \mathbf{x}))$$

and apply the second recursion theorem with

$$F(\mathbf{x}, e) = H(\mathbf{x}, App(e, \mathbf{x})).$$

Then

$$g(\mathbf{x}) = F(\mathbf{x}, e) = H(\mathbf{x}, App(e, \mathbf{x})).$$

Proof of the second recursion theorem from the fixed-point theorem

- In the abstract setting, the fixed point theorem says that if G is total computable, then there exists an e such that for all x, $App(e, x) \cong App(G(e), x)$.
- Suppose given the partial computable function F. Define $G(e) = \Lambda x\, F(\mathbf{x}, e)$.
- Note that G is a total function, since $\Lambda x\, F(\mathbf{x}, e)$ is always some Turing machine obtained from F, whether F is total or not.
- By the fixed-point theorem, there exists an e such that for all x, $App(e, x) \cong App(G(e), x)$.
- Define $g(\mathbf{x}) := App(e, \mathbf{x})$. Then we have

  $$g(\mathbf{x}) := App(e, \mathbf{x}) \cong App(G(e), \mathbf{x}) \cong App(\Lambda \mathbf{x}\, F(\mathbf{x}, e), x) \cong F(\mathbf{x}, e)$$

  as required for the second recursion theorem.

Notation

The calculations involved in the next proof are easier to follow if we stop writing App explicitly, and just write xy instead of $App(x, y)$. Thus

- we write xy for $App(x, y)$.
- $App(App(x, x), y)$ becomes $(xx)y$.
- That is not the same as $x(xy)$.
- We will try to avoid omitting parentheses but if we do omit them, xyz means $(xy)z$.

The fixed-point theorem

Given a total computable G, define

$$\omega := \Lambda x\, G(\Lambda y\, ((xx)y)).$$

Then define

$$e := \Lambda y\, ((\omega\omega)y)$$

A calculation then reveals that e is the desired fixed point:

$$\begin{array}{rcll} ex & \cong & (\Lambda y\, ((\omega\omega)y))x & \text{by definition of } e \\ & \cong & (\omega\omega)x & \text{reducing the } \Lambda \text{ term} \\ & \cong & ((\Lambda x\, G(\Lambda y\, ((xx)y)))\omega)x & \text{by definition of } \omega \\ & \cong & G(\Lambda y\, ((\omega\omega)y)))x & \\ & \cong & (G(e))x & \end{array}$$

In the unabbreviated notation, this is $App(e, x) \cong App(G(e), x)$, which is the abstract version of the fixed-point theorem. That completes the proof.

This wonderful, mysterious proof!

- The reader may be forgiven if he or she finds the proof mysterious, and has the feeling that although it is easy to check the proof line by line, it is still hard to understand.
- The origins of this proof will become clearer when we discuss the λ-calculus, but that will not entirely remove the mystery.
- Though the proof is mysterious, the theorem is basic and important. So *memorize the proof*.
- Write it out several times on a blank sheet of paper, copying at first, and eventually writing it out from memory.
- This is one of the secret, esoteric gems of knowledge passed on in the inner circles of mathematical logic.

Applications of the recursion theorem

Corollary

The μ-recursive functions form a model of computation. That is, if φ_n is the partial μ-function with index n and $App(n, x) := \varphi_n(x)$, then App is μ-recursive.

Proof. App is defined by recursion on n; that is, the value $\varphi_n(x)$ is defined in terms of some other values of φ_m for $m < n$. These recursion equations are quite complicated, but their exact form is no longer relevant. Supposing that e is a Turing machine that computes $g(n, x) = \varphi_n(x)$, for some F the recursion equations can be written as $g(n, x) = F(n, x, e)$. Therefore by the second recursion theorem, such an e actually exists. That completes the proof.

Remark. The proof has a certain magical quality: we get to *assume* that the desired Turing machine e exists; then we only have to check that the recursion equations we want to solve can be written in terms of e, and magically the Turing machine e appears.

No smaller model of computation on $\mathbb{N}$

The following theorem shows that there is no weaker notion of computation than Turing machines, that permits a universal machine.

Theorem (Minimality theorem)

Any model of computation on the natural numbers $\mathbb{N}$ must include all the Turing-computable functions; furthermore the computable functions in any model of computation on $\mathbb{N}$ are closed under the least-number operator.

Proof of the minimality theorem

Suppose given a model of computation on $\mathbb{N}$; let App be the application function(s) of the model, and temporarily let "computable" mean, computable in the model.

- It is easy to show, using the recursion theorem, that all primitive recursive functions must computable.
- If we show that the computable partial functions are closed under the μ operator, it will follow that all μ-recursive functions are computable (in the model), and hence all Turing computable functions also. Thus the only remaining thing to do is to show closure under the μ operator.
- This is essentially a piece of beginning computer science: search can be done recursively.
- Details on the next slide.

Search from recursion

Suppose $\chi(b, y)$ is (partial) computable. Then define

$$\phi(b, z) \cong \begin{cases} 0 & \text{if } \chi(b, z) = 0 \\ \phi(b, z')' & \text{if } \chi(b, z) \neq 0) \end{cases}$$

Then we claim

$$\mu y \left(\chi(b, y) = 0\right) \cong \phi(b, 0)$$

To prove this we prove the more general fact that $\phi(b, z)$ is the least k such that $y = z + k$ satisfies $\chi(b, y) = 0$, if there is a $y \geq z$ such that $\chi(b, y) = 0$. That is proved by induction on $y - z$, where y is the least $y > z$ such that $\chi(b, y) = 0$. (We expect to use induction someplace, because this is a theorem about $\mathbb{N}$.) The base case of the induction is the first line in the definition of ϕ, and the induction step follows from the second line, since when z increase to z', $y - z$ (which is not zero, or the first line would have applied) decreases by 1. That completes the proof.

No total App over $\mathbb{N}$

The question naturally arises: Could we have a model of computation in which App is a total function? In that case all functions would be total. It is clear that no such model of computation exists over $\mathbb{N}$, since we know that $\mu x\, x = x'$ is undefined, and by the minimality theorem, the μ operator can be defined in any model of computation over $\mathbb{N}$.