Lecture 12
Arithmetization of Syntax

Michael Beeson

Gödel numbers

- Gödel realized that the syntax of first-order logic can be coded as numbers.
- Today that does not seem like a deep realization, since formulas and terms are given by strings, strings are made of characters, characters have ascii codes that are made of bits, and long strings of bits are just large integers.
- If syntax is carefully defined, in fact the encoding of terms and formulas into numbers could be just the identity map!

Variables and constants

- Kleene says (p. 70) a variable is $a, b, c, \ldots$. The three dots need to be made more precise. Page 252, he says precisely that the variables are a with any number of subscript 1s.
- Modern books usually say $x_1, \ldots, x_n, \ldots$, which agrees with Kleene if we require n to be written in unary.
- We don't want to allow arbitrary strings for variable names, as there is the problem of distinguishing variables from constants, etc.
- Each theory T will have to declare its own constants, so for example in group theory we can declare e to be a constant if we wish. In that case we would have to modify the definition of variable so e doesn't qualify.
- We focus on $\mathbf{PA}$. The only constant is 0 so we have no conflicts.

Some remarks on variables

- Thus variables are strings of ascii characters.
- In typesetting terminology, they are not glyphs.
- An x is an x, whether it is in italics or roman or boldface, and regardless of the font size. These are matters of the *display* of the object.

Defining syntax

- Today the natural way to formalize syntax is to use grammar notation.
- We follow Kleene, p.70, where only informal notation is used.

```
char:- [a-z] | [A-Z].
digit:- [0-9].
digit_string:- digit | digit digit_string.
variable:- char | char _{digit_string}.
```

That way, the variable `x_27` is just the TEX code for x_{27}

Function symbols

- Again, each theory should declare its own.
- In the case of $\mathbf{PA}$, we don't have to worry about conflicts between variables and function names as the only function symbols are those for successor, multiplication, and addition.
- Conveniently, there are ascii symbols `+`, `*`, `'`.

```
function_symbol :-  + | ' | *.
```

Terms

```
constant :- 0.
term:- constant | variable | compound_term.
compound_term :-  ( term ' ) | ( term + term )
                | ( term * term ).
```

- Here we explicitly allow for infix and postfix notation.
- Note the many parentheses that are officially required.
- We never needed `function_symbol`, but it would in other theories be useful.

Logical symbols

- Unfortunately the logical symbols do not have ascii codes.
- Instead we use something like TEX codes.

```
propositional_connective:-
     \land | \lor | \implies | \neg.
quantifier:- \forall | \exists.
```

We regard, for example, $\supset$ as just another name for `\implies`.

Formulas

```
atomic_formula :-  term = term.
formula :- atomic_formula | compound_formula.
compound_formula :-
     ( formula propositional_connective formula ).
compound_formula :- (quantifier variable  formula ).
```

Use of grammar notation

- According to our definitions, terms and formulas are strings over the ascii alphabet.
- For example, the formula $\bar{2} + \bar{2} = \bar{4}$ is really `( ( ((0')') + ((0')')) = ((((0')')')')')`.
- $\forall x, y\,(x' + y = x + y')$ is really

```
(\forall x ( \forall y (((x') + y) = (x + (y')))))
```

- In Kleene, too, formulas are "sequences of symbols", which amounts to what we now call strings.

Parsing

- When we want to define something "by induction on terms", or "by induction on the complexity of a term t", we want to define, for example, the value on `(u + v)` in terms of the value at u and the value at v.
- This presumes that the relevant occurrence of + can be uniquely identified and computed.
- That is done by an algorithm called "parsing".
- Such algorithms are discussed in computer science and there are tools such as `yacc` and `lex` for generating them automatically from the grammar.
- Kleene and all other textbook authors do not treat this matter fully. See pp. 73-74, where Kleene discusses unambiguity of parentheses. That's as close as he comes to parsing.

Functor and Arity

- The upshot of parsing is that there are computable functions $functor$ and $arity$ that take a term or formula (as a string) and return the "main symbol" or functor, and the number of arguments, respectively.
- For example, on the input `((x+y)=z)`, the functor is the character $=$, and the arity is 2.
- On the input `(x+y)`, the functor is +, and the arity is 2.
- On the input `x`, the functor is `x` and the arity is 0.
- On the input

  ```
  (\forall x ( \neg (x' = 0)))
  ```

  the functor is `\forall` and the arity is 2. The first argument is the variable `x` and the second is the formula after the quantifier.
- Note that the arity is not 1. We need to consider the variable that is being quantified.

Extracting the arguments

Parsing also enables us to extract the indices of the parts of the input string that match the right side of a grammar rule. For example, consider the rule

```
term :- ( term + term)
```

Applied to the input `(x+y)` we can extract not only the functor and arity, but also the first and second arguments, namely `x` and `y`.

The parsing algorithm should provide computable functions arg_1 and arg_2 to do this. In $\mathbf{PA}$, the maximum arity is 2, so this is enough.

Parsing considered

There is, however, no good reason to ignore parsing. (I did promise to prove *everything* for you.) We consider how it is done for the case of $\mathbf{PA}$.

- If there's no initial paren, then it is a constant, or variable, or error.
- If there is an initial paren, and what follows is a quantifier and then a variable, then the last character should be right paren, and we delete it and parse recursively what follows the quantifier. It should be a formula, or we have an error. That formula is the second argument; the arity is 2, and the first argument is the variable. The functor is the quantifier at the beginning. If the quantifier is not followed by a variable that is also an error.
- Otherwise the input has to be a formula or a term.

Parsing continued

- The key is to keep a parentheses count while moving right over the input after the initial left paren.
- We start at 0 and increment for left paren, decrement for right paren. If we encounter an $=$, $+$, or * sign with paren count 0, that is the functor, and we can locate the two arguments and parse them recursively. If the paren count becomes negative, it is an error.
- If the paren count returns to zero and we see a ’ symbol just left of the final paren, then the arity is 1. Delete the first and last parens and parse the string between them to get the arg.

Parsing is primitive recursive

- Here we identify strings with the numbers whose 8-bit segments give the characters of the string.
- The recursive algorithm on the previous slide is defined by course-of-values recursion, since the arguments of the recursive call are always shorter than the top-level argument.
- The functions that extract the substrings are defined in terms of division by 2, to extract certain bits, so they are primitive recursive.
- The primitive recursive functions are closed under course-of-values recursion.
- Conclusion: $functor$, arg_1, arg_2, and $arity$ are primitive recursive.
- They are total functions, returning a specified value `err` (perhaps 255) if parsing fails.

Summary

- Each string is identified with a number whose bits are the bits of its characters.
- We worked out examples of this "coding" at the beginning of the course.
- Thus each string "is" a number.
- In particular each variable, term, formula is a number as well as a string.
- The functions $functor$, $arity$, arg_1, and arg_2 are primitive recursive.

An alternative to parsing

- We could just define a term to be a sequence (f, a, b) where f is a function symbol of arity 2 and a and b are terms, or a sequence (f, a), where f is successor and a is a term. Then the functions $arity$, $functor$, arg_1, and arg_2 would already exist, easily defined from $length(x)$ and $x[i]$, the unpacking functions for sequences.
- But using strings corresponds to the natural human use of formal systems.
- Also, using strings is necessary to implement logic on a computer.
- Therefore I discussed the parsing issue.

Gödel numbers

- Historically, this result was achieved differently, as Gödel worked two decades before the ascii code was defined, and strings were not as well-understood then as now.
- We therefore discuss the traditional method of reducing formulas, terms, etc. to numbers.
- Gödel assigned a number, since called the Gödel number, to each syntactic object.
- The Gödel number of a formula ϕ is written

$$\ulcorner \phi \urcorner.$$

- Here ϕ is a string and $\ulcorner \phi \urcorner$ is a number.
- See Kleene, §52, p. 254 for details.
- In these lectures, I continue to use $\ulcorner \phi \urcorner$ when I want to emphasize that string ϕ is being considered as a number, e.g. so it can be discussed in $\mathbf{PA}$.

Basic syntax is primitive recursive

The following predicates are primitive recursive. Here x ranges over numbers, thought of as strings.

- x is a term
- x is a constant
- x is a variable
- x is a formula

The characteristic functions of these predicates can be defined in terms of $functor$, $arity$, arg_1, and arg_2, following the recursive definitions given earlier.

Substitution

Recall that $t[x := s]$ is the result of substituting term s for each free occurrence of x in term t.

Now that we are regarding t and s as numbers, we want to show that $t[x := s]$ is a primitive recursive function $Subst(s, x, t)$ of s, x, and t.

- Here t can be a formula as well as a term.
- It is defined by recursion on t. The recursive call applies $Subst$ to the arguments of t.
- If t is a quantified formula, and the first argument of t is x, then we return the second argument unchanged. (Because we are only substituting for free occurrences of x.)
- In all other cases, we call $Subst$ on the arguments of t, and then combine the results to create a new term or formula with the same arity and functor as t.
- Since the arguments are smaller than the input, this is definable by course-of-values recursion, and hence is primitive recursive.

Free and free-for

- Compare Kleene p. 253.
- E contains x free if $Subst(0, x, E) \neq E$. This is a clever reversal of our intuitive idea that $Subst$ is defined in terms of "free".
- t is free for x in E means that you can substitute t for x in E without accidentally "capturing" other free variables of t by substituting inside scopes where they are quantified.
- If x is not a variable or t is not a term, then t is by definition not free for x in E.
- Otherwise "free-for" is defined recursively.
- The main clause is that t is free for x in $\forall y \phi$ if either ϕ does not contain x free, or t does not contain y free and t is free for x in ϕ.
- free-for is also primitive recursive since it's defined by course-of-values recursion.
- Similarly for $\exists$.

Propositional Axioms

There are infinitely many axioms. There are finitely many logical axiom schemata. The axioms are those formulas that "have a certain form." For example, all formulas of the form

$$(A \supset (B \supset A))$$

are axioms. A string is an axiom of this form if its functor is $\supset$, its first arg is a formula A, its second arg has functor $\supset$, and the second arg has a formula for its first arg and the second arg is exactly A. This is a primitive recursive definition.

Similarly, there are primitive recursive predicates defining what it means for a formula to match each of the (finitely many) propositional axiom schemata. Hence the predicate $PropAx(x)$ defining "x is a propositional axiom" is primitive recursive.

Quantifier Axioms

There are two quantifier axiom schemata, both of which assume t is free for x in A:

$$\forall x\, A \supset A[x := t]$$

$$A[x := t] \supset \exists x\, A$$

For each of these two, the predicate "u has the the form of this axiom schemata" is primitive recursive.

- The hard part of this is to extract t primitive recursively. Note that t is not given explicitly. Suppose for the moment we can do that. Then consider the first axiom schema.
- The predicate we need to define says that u is a formula, the functor of u is $\forall$, the first arg is a variable x, the second arg has functor $\supset$, and if A is its first arg then its second arg is equal to $Subst(t, x, A)$.
- Since $Subst$ is primitive recursive, and the primitive recursive predicates are closed under "and", we'll be done as soon as we can extract t primitive recursively.

Extracting t from $A[x := t]$ and A

- Kleene says nothing about this issue; perhaps the notation $A(x)$ and $A(t)$ caused him not to notice that it is an issue.
- Here's an algorithm to do it: compare the two inputs $A[x := t]$ and $A[x]$ character-by-character until they differ, say with your left index finger pointing into $A[x := t]$ and your right pointing into $A[x]$. When they first differ (at position i), you must have x on the right and the start of t on the left. So move your left finger farther along to the least j such that the substring between i and j is a term. That term is t.
- That function is primitive recursive, since $Term(u)$ is primitive recursive, and the primitive recursive functions are closed under bounded search. (We search first for the first mismatch, and then for j such that the substring from i to j is a term.)

Numerals

- The function $Num(x) = \ulcorner \bar{x} \urcorner$ is primitive recursive.
- $Num(0) = \ulcorner 0 \urcorner = 48$.
- $Num(x+1)$ is the number corresponding to the string formed by concatenating "(" with $Num(x)$ and ")".

One of your homework problems is to finish this proof.

D is an immediate consequence of E

This means E has the form $C \supset A$ and D has the form $C \supset \forall x\, A$, where A and C are formulas and x is not free in C.

- We first check that A and D are both formulas with functor $\supset$.
- Then we check that they both have the same first argument C.
- Let $Q = arg_2(D)$. If $functor(Q) \neq \forall$ then D is not an immediate consequence of E. If $functor(Q) = \forall$ then $arg_1(Q)$ must be a variable, and $arg_2(Q)$ must equal $arg_2(E)$ (which is A). If any of these conditions fails then D is not an immediate consequence of E.
- If they all hold, then D is an immediate consequence of E.
- Those are all primitive recursive functions and relations, so this is a primitive recursive relation.

D is an immediate consequence of E and F

That means that F has the form $E \supset D$.

$$functor(F) = \ulcorner \supset \urcorner \wedge arg_1(F) = E \wedge arg_2(F) = D$$

The proof predicate Prf

- Recall that we have primitive recursive functions $length(x)$ and $x[i]$ to extract the length and members of a coded sequence.
- It doesn't matter what sequence coding we use as long as the coding and uncoding are primitive recursive.
- A proof is a sequence Y such that:
- For each $i < length(Y)$, $Y[i]$ is a formula, and one of the following holds: Either
- $Y[i]$ is an axiom, or
- $\exists j < i\,(Y[i]$ is an immediate consequence of $Y[j])$, or
- $\exists j, k < i(Y[i]$ is an immed. consequence of $Y[j]$ and $Y[k])$.
- There is a primitive recursive predicate $\mathrm{Prf}(Y)$ defining "Y is a proof."

We say that a proof is a proof of its last formula. Thus:

$$\mathrm{Prf}(Y, A) := \mathrm{Prf}(Y) \wedge A = Y[length(Y) - 1]$$

Prf is primitive recursive

Proof: it is defined by bounded quantification over a primitive recursive predicate.

- Therefore it is Turing computable
- and representable
- That means we really can discuss the meta-theory of $\mathbf{PA}$ *within* $\mathbf{PA}$.

Fun with Gödel numbers

The result of substituting the numeral for m for x in ϕ:

$$\phi[x := \bar{m}]$$

is a formula whose Gödel number is given by a primitive recursive function of m, even though m is *not* a free variable of $\phi[x := \bar{m}]$. Namely,

$$Subst(Num(m), \ulcorner x \urcorner, \ulcorner \phi \urcorner).$$

Note that $Subst$ takes three Gödel numbers. $Num(m)$ is already a Gödel number. $Subst$ is not a formula, it is a primitive recursive function, so its arguments should be numbers, not terms.

More fun with Gödel numbers

Now we can substitute *that* number into another formula ψ:

$$\psi[z := \overline{\ulcorner \phi[x := \bar{m}] \urcorner}]$$

and miraculously this too is given by a formula of $\mathbf{PA}$ with free variable m:

$$\psi[z := Num(Subst(Num(m), \ulcorner x \urcorner, \ulcorner \phi \urcorner))]$$

Technically that is not a formula as it stands, as $Subst$ and Num are not symbols of $\mathbf{PA}$. But let $P(m, y)$ represent the function on the right of $:=$. Then what we mean is

$$\exists z \, P(m, z) \wedge \psi.$$

In sloppy short notation

If $\psi(z)$ and $\phi(x)$ are formulas with one free variable then $\psi(\phi(\bar{m}))$ is a formula with one free variable m.

- It is vital to understand exactly what this means, as spelled out precisely on the previous slides.
- Understanding this point is necessary to understand Gödel's proof of his incompleteness theorem.
- To study this, write down on a blank page, $\psi(\phi(\bar{m}))$, and without looking try to write out exactly what it means. Repeat until you can do it.
- This may take many attempts. Mathematical logic, as I told you the first day, is not a spectator sport.

Σ_1^0 relations and formulas

- A Σ_1^0 formula is a formula of the form

$$\exists z_1, \ldots, z_m A(\mathbf{x}, y, z_1, \ldots z_m)$$

  where A is a bounded arithmetic formula.
- A Σ_1^0 relation is one defined by a Σ_1^0 formula
- Example: the formula defining $x < y$, namely $\exists z\,(z' + x = y)$.
- Non-example: the formula $\forall x, y\,(x' + y = x + y')$, is not Σ_1^0, because it begins with $\forall$, not $\exists$.
- In an exercise you were asked to show that all μ-recursive functions have Σ_1^0 graphs (using the $\mathbf{T}$-predicate):

$$\exists k\,(\mathbf{T}(e, x, k) \wedge U(k) = y)$$

  is a Σ_1^0 formula, so the e-th Turing computable function has Σ_1^0 graph.

Conversely, if f has a Σ_1^0 graph, then f is μ-recursive

- Suppose $f(\mathbf{x}) = y$ if and only if $\mathbb{N} \models \exists \mathbf{z}\, A(\mathbf{x}, y, \mathbf{z})$ where A is a bounded arithmetic formula and $z = z_1, \ldots, z_m$.
- Recall $(k)_i$ extracts the i-th member of a coded sequence.
- Then let

$$g(\mathbf{x}) \quad = \quad \mu k \, (A(\mathbf{x}, (k)_0, (k)_1, \ldots (k)_{m+1})$$

- Then $f(x) = (g(x))_0$.
- This gives us another characterization of the computable functions: they are the ones with Σ_1^0 graphs.

True Σ^0_1 sentences are provable

- Note, sentences, i.e. no free variables.
- Let $\exists \mathbf{x}\, A(x)$ be a Σ^0_1 formula, so A is bounded.
- The idea is that if there is an x such that $A(\bar{x})$, we can just verify (within $\mathbf{PA}$) that indeed $A(\bar{x})$ holds.
- if $\exists \mathbf{x}\, A(x)$ is true, that means there is an m such that $A(\bar{m})$ is true, i.e.,

$$\langle \mathbb{N}, +, \cdot, ', 0 \rangle \models A[x := \bar{m}]$$

- Since A is bounded, the relation it defines is represented by A, so

$$\vdash A[x := \bar{m}]$$

- then by $\exists$-introduction,

$$\vdash \exists x\, A$$

Provable implies provably provable

- Suppose $\vdash \phi$.
- That is, $\exists k \, \mathrm{Prf}(k, \overline{\ulcorner \phi \urcorner})$ is true.
- Since that is a true Σ_1^0 sentence, it is provable.
- Explicitly,

$$\vdash \exists k \, \mathrm{Prf}(k, \overline{\ulcorner \phi \urcorner}).$$

- Note that ϕ is an arbitrary formula. It may have free variables and many unbounded quantifiers.