MAU11602
Lecture 19
2026-03-12

Encoding

We can encode lists of symbols as natural numbers.
For example, I view the text of these slides as a list of characters but my laptop views them as a list of bits, which is just the binary representation of an integer.
Kurt Gödel in 1931 didn't have a laptop but he had the same basic idea.
He didn't use binary, and I won't either. Instead I'll use an encoding strategy due to Raymond Smullyan.
Suppose we want to encode lists of symbols chosen from a set of p symbols, called the alphabet.
The obvious encoding is the base p representation of natural numbers, but we'll see it has some problems.
The basic idea is to associate each symbol in the alphabet with a digit from 0 to $p-1$, and then identify lists of symbols with lists of digits, and lists of digits with natural numbers via the base p representation.
So the list $(d_{l-1}, d_{l-2}, \ldots, d_1, d_0)$ is identified with the natural number $\sum_{i=0}^{l-1} d_i p^i$, which looks like $d_{l-1} d_{l-2} \cdots d_1 d_0$ when written in base p.

Problems, and a solution

The base p idea almost works, but it has some problems.
The list $(0, 0, 7)$ and the list (7) are different lists but 007 and 7 are the same natural number, in base p for $p > 7$.
Relatedly, we might want to refer to the empty list. It should be represented by the natural number

$$\sum_{i=0}^{0-1} d_i p^i = 0,$$

but this is usually written as "0" in base p rather than as "".
But "0" is the encoding of the list (0), of length 1.
Smullyan's idea is very simple. Keep representing the list $(d_{l-1}, d_{l-2}, \ldots, d_1, d_0)$ by the natural number $\sum_{i=0}^{l-1} d_i p^i$, but use the digits 1 to p instead of 0 to $p-1$.
I'll call the string of digits $d_{l-1} d_{l-2} \cdots d_1 d_0$ the modified base p representation of the natural number.
The crucial property of this representation is that it is unique. There are no ambiguities created by leading zeroes, because there are no zeroes!

Examples

Converting a list of digits to a natural number is mostly straightforward. In principle we use the sum $\sum_{i=0}^{l-1} d_i p^i$, but in practice it's more efficient to use the equivalent form

$$((\cdots((d_{l-1}p + d_{l-2})p + d_{l-3})p + \cdots)p + d_1)p + d_0.$$

As an algorithm: start with a value of 0 and then every time we read a digit we multiply the current value by p and add the digit. The final value is the number whose modified base p representation is the list of digits.
Example, what number does the modified base 7 number 57 represent?
We start with 0, multiply by 7 to get 0, add 5, multiply by 7 to get 35, then add 7 to get 42.
This is of course different from the ordinary base 7 representation, which would be 60, but the algorithm is the same.
You can check, by the same method, that 5623 is the modified base 7 representation of the number 2026. It's also the ordinary base 7 representation.
The modified base 7 representation of 0 is the empty list of digits .

Examples, continued

Conversion in the other direction, from a number to its modified base p representation, is the reverse process.

Start with the empty string. If your number is zero then you're done. Otherwise, find a digit congruent to it modulo p, prepend it to your string, substract it from your number, divide by p, and repeat.

For example, what is the modified base 3 representation of 2026?

The steps in the calculation are

```
         2026
1         675
31        224
231        74
2231       24
32231       7
132231      2
2132231
```

so the modified base 3 representation of 2026 is 2132231.

Back and forth

Since we've identified natural numbers with lists we can convert numerical operations to list operations and list operations to numerical operations.
For example, we can take two lists of symbols, convert them to natural numbers, add the numbers, and convert the result to a list.
With the ordinary base 10 representation this is what you probably think of as "adding numbers".
There are rules for doing this, involving carrying and such. *Not* formalising those as rules of inference is the reason we don't use decimal representation in Peano arithmetic and instead use the $SSS\cdots SS0$ notation.
Essentially the same rules work in modified base p.
This is not very interesting.
What's more interesting is converting list operations to numerical operations. The most interesting string operation is appending (concatenation). What is its numerical counterpart?
Is its numerical counterpart arithmetic, in the sense of yesterday's lecture?

Append

Suppose we have numbers a and b whose modified base p representations are $d_{l-1}d_{l-2}\cdots d_1 d_0$ and $e_{m-1}e_{m-2}\cdots e_1 e_0$.
Appending the list of digits gives a list $f_{n-1}f_{n-2}\cdots f_1 f_0$, where $n = l + m$ and $f_j = e_j$ if $j < m$ and $f_j = d_{j-m}$ if $j \geq m$.
This is the modified base p representation of some number c. How is c related to a and b?

$$a = \sum_{i=0}^{l-1} d_i p^i \quad b = \sum_{i=0}^{m-1} e_i p^i \quad c = \sum_{i=0}^{l+m-1} f_i p^i$$

$$p^m \cdot a + b = \sum_{i=0}^{l-1} d_i p^{m+i} + \sum_{i=0}^{m-1} e_i p^i = \sum_{i=m}^{l+m-1} d_{i-m} p^i + \sum_{i=0}^{m-1} e_i p^i = \sum_{i=0}^{l+m-1} f_i p^i$$

so

$$c = p^m \cdot a + b.$$

Arithmeticity

The equation $c = p^m \cdot a + b$ isn't part of the language of Peano arithmetic, which doesn't have exponentiation.
We know though that b is a natural number with a representation of length m, and the smallest such number is $\frac{p^m-1}{p-1}$, whose representation is just m 1's, while the largest such number is $p\frac{p^m-1}{p-1}$, whose representation is just m p's. So, if we write d for p^m,

$$\frac{d-1}{p-1} \leq b \leq p\frac{d-1}{p-1}.$$

There is only one power of p with this property.
We can rewrite these inequalities in a way which avoids subtraction and division.

$$d + b < b \cdot p \wedge b \cdot p + p \leq p \cdot d + b.$$

Assuming p is prime, d is therefore the unique natural number such that

$$(\forall e.\,((\exists f.d = e \cdot f) \rightarrow e = S0 \vee (\exists f.e = p \cdot f))) \wedge d + b < b \cdot p \wedge b \cdot p + p \leq p \cdot d + c.$$

Arithmeticity, continued

We want to say that if $d = p^m$ then $c = d \cdot a + b$, i.e.

$$((\forall e.((\exists f.d = e \cdot f) \to e = S0 \vee (\exists f.e = p \cdot f))) \wedge d + c < c \cdot p \wedge c \cdot p + p \leq p \cdot d + c) \\ \to c = d \cdot a + b.$$

This is a boolean expression with five free variables, p, a, b, c, and d.
We want to express, in Peano arithmetic, the fact that the modified base p representation of c is the concatenation of the representations of a and b. This should have four free variables p, a, b, and c.
In other words, we need to bind d. Which quantifier should we use $\forall$ or $\exists$?
Either is fine! There is exactly one d which makes the left side of the $\to$ true, so either choice will express the fact that the right hand side is true for that d.
Making either choice, we see that the function which takes p, a and b to c is an arithmetic function.
If we extend the notion of arithmetic sets and functions to lists via the modified base p encoding then append is an arithmetic function.
This observation is much more powerful than it might appear!

The language of balanced parentheses

Ultimately we want to convert statements about Peano arithmetic into statements within Peano arithmetic, but let's start with a simpler language.

The language of balanced parentheses, also called the Dyck language, has two symbols, `(` and `)`, in its alphabet and consists of those lists of symbols where we can pair off each `(` with a *later* `)`.

Clearly every such list has an equal number of `(`'s and `)`'s, but not every list with an equal number of `(`'s and `)`'s belongs to the language.

`)(` is not allowed, for example.

Does `(()(()))(()(()))` belong to the language of balanced parentheses?

No, but this probably isn't obvious unless know the trick for checking membership: keep a running total of the number of `(`'s minus the number of `)`'s.

```
   (  (  )  (  (  )  )  (  )  )  )  (  (  )  (  (  )  )
0  1  2  1  2  3  2  1  2  1  0 -1  0  1  0  1  2  1  0
```

This count should never be negative, but as long as it stays non-negative and ends with zero we can always pair off the `(`'s and `)`'s, and do so in way that makes the parentheses properly nested.