Universal Gödel Numberings or What Computability Theory Says About Programming Languages Lecture Notes for COMP 598 Summer 2020

Prakash Panangaden[*]

21st May 2020

One of the remarkable facts about algorithms is that they can be written in a formal language and enumerated. In fact they can be enumerated in a computable fashion. Even more, the enumerations admit *computable* ways of composing algorithms. Today these notions might seem passé, but these are the basic ideas that led to programming languages. We take this for granted when we first learn a programming language. We know that programs are just strings and that strings – even very long ones – can be enumerated. We take it for granted that programs can be composed. These fundamental ideas come from Turing's and Church's notions of universality and Gödel's notions of coding formulas as numbers.

1 Universal Functions

Definition 1. A binary function $U : \mathbf{N}^2 \longrightarrow \mathbf{N}$ is said to be **universal** for the class of *all computable* unary functions if:

(i) $\forall n,\ U_n : x \mapsto U(n,x)$ is computable; the functions U_n are called the **sections** of U;

(ii) $\forall$ computable f, $\exists n$ such that $U_n = f$, i.e.

$$\forall x \in \mathbf{N},\ U(n,x) = f(x).$$

[*]The treatment here follows the presentation in "Computable Functions" by Shen and Verasagin.

Neither U nor f need be total and there could be many different n's for a given f. Think of n as a "code" for the function f.

So far we have not said that U itself is computable.
Theorem 2. There is a binary **computable** function U, which is universal for the class of all unary computable functions.

Proof. Consider all legal programs of YFPL[1] written out as ascii strings. Order them lexicographically[2] to obtain a list of the programs: $p_0, p_1, p_2, p_3, \ldots$. Now we define

$$U(n, x) = p_n(x).$$

By $p_n(x)$ we mean the effect of running p_n on x. Clearly U is universal and from n we can effectively construct p_n, so U is computable. ∎

Of course we are relying on the fact that we already believe that all algorithms can be coded in YFPL. Turing had to show that he could design a universal Turing machine which would read the description of any Turing machine and simulate it.

In modern parlance, U is just an *interpreter* for YFPL and n is just a numerical encoding of the program text. If we took strings as our basic data type then n would *be* just the program text.

Can we make our interpreter always halt?
Theorem 3. There is no **total** computable function of two arguments which is universal for the class of total computable unary functions.

Proof. Let U be any total computable function of two arguments. Define $d(n) \overset{\text{def}}{=} U(n, n) + 1$. Clearly d is total. Consider any natural number n, $d(n) = U(n, n) + 1 \neq U_n(n)$. Thus, for any n, U_n and d will disagree on at least one value, namely on n itself. This means that for all n $U_n \neq d$; i.e. U is not universal. ∎

Why does this proof not apply to partial functions?[3]

If f and g are computable functions then $f \circ g$ is also computable. In fact, there should be an *algorithm* to figure out the code of $f \circ g$ given the codes of f and g. With programs represented as strings this is obvious; typically the

[1]Your Favourite Programming Language
[2]This is the order used in a dictionary.
[3]This is not a rhetorical question, please answer it.

programming language provides a composition construct, usually a semicolon. In terms of enumerations of Turing machines or some other primitive model we need to work more.

Definition 4. An onto function $\nu : \mathbf{N} \longrightarrow S$, where S is any (countable) set, is called a **numbering** of S. A value of n for which $\nu(n) = s$ is called a code number of s.

Note the use of the indefinite article; a given element may have many code numbers. This should not be a surprise: a given function has many programs (algorithms) to compute it.

Clearly, any universal function for the unary computable functions defines a numbering of all the unary computable functions: $n \mapsto U_n$. We often write U_n for the unary function denoted by the code n. We are using the word "code" to mean "the number associated with a computable function" in some enumeration. However, you should keep in mind the intuition that this is just a "program."

What we want is to show that for the "right kind" of universal function U, that there is a *total* computable function $c : \mathbf{N}^2 \longrightarrow \mathbf{N}$ such that

$$\forall p, q, x \; (U_p \circ U_q)(x) = U(p, U(q, x)) = U_{c(p,q)}(x) = U(c(p, q), x).$$

This will require some conditions on U. Of course, U has to be a universal computable function, but we will need more.

Definition 5. Let U be a binary universal computable function for the class of unary computable functions. It is called a **Gödel universal function** if, for any binary computable function V, there exists a total computable unary function σ such that $\forall m, x \; V(m, x) = U(\sigma(m), x)$.

What does this mean? Why does anyone care? Let us examine what it is saying. First, note that if you have any two-argument computable function $V(\cdot, \cdot)$ (two-input program) and you fix one of the arguments, say the first one is fixed to the value m, you get a one-argument computable function which we can *informally* write as $V(m, \cdot)$. So there should be a code (program) for this function. How easy is it to find this code? Well for a Gödel universal function we can assert that there is *some* total computable function σ which takes the fixed argument m and works out a code for $V(m, \cdot)$; this is what $\sigma(m)$ is. The equation above says if you fill in the second argument as well you should be computing $V(m, x)$ and the right hand side of the equation says that this is obtained by using the code $\sigma(m)$ (running the "program" $\sigma(m)$) with the universal function U (on the interpreter U). Note that you have to write a different σ for each V.

This is all very well, but how do we know that there are any such things?
Theorem 6. A Gödel universal computable function exists.

Proof. [4] This proof is a bit long and contains some interesting facts in their own right.

First we note that we can give a one-to-one and onto map between natural numbers and pairs of natural numbers. Think of writing the pairs in an infinite square table with the pairs of the form $(0, n)$ in the first row, the pairs of the form $(1, n)$ in the second row and so on. If we look at the diagonals from lower left rising towards the right we see that in every such row the numbers in each pair add up to the same value. We can thus enumerate the pairs by just going down these diagonals. We get

$$\tau(i,j) = \frac{1}{2}(i+j)(i+j+1) + i.$$

We can see the pattern of the enumeration in the picture below

$$\begin{tikzcd}
(0,3) \arrow[dr] & \dots & & \\
(0,2) \arrow[dr] & (1,2) \arrow[dr] & \dots & \\
(0,1) \arrow[dr] & (1,1) \arrow[dr] & (2,1) \arrow[dr] & \dots \\
(0,0) \arrow[u] & (1,0) \arrow[uul] & (2,0) \arrow[uuull] & (3,0)\dots
\end{tikzcd}$$

The function τ is clearly one-to-one and onto. We can invert τ by writing a simple program to find the elements of the pair. The following program inverts τ on the natural number N and puts the values in the variables X, Y[5].

```
X,Y = 0;
while (tau(x,y) != 0) do
```

[4] This simplified version of the proof is due to Georghe Comainici. The original, which I gave several years ago had an unnecessary twist.

[5] I have not tested the pseudo-code below so it may be buggy. I have written and tested a functional version that works correctly.

```
{ Y = Y + 1;
  if tau(X,Y) > N then {X = X + 1; Y = 0}}
```

Given a number n we write $\mathsf{fst}(n)$ and $\mathsf{snd}(n)$ for the numbers that you get by decoding n viewed as a pair. In other words $\tau(\mathsf{fst}(n), \mathsf{snd}(n)) = n$ and $\mathsf{fst}(\tau(p,q)) = p$ and $\mathsf{snd}(\tau(p,q)) = q$.

Now given any universal function U, for unary computable functions we define a Gödel universal function G by $G(n,x) = U(\mathsf{fst}(n), \tau(\mathsf{snd}(n), x))$. We have to show that this really is Gödel universal. Given any binary computable function V we define a unary function f by $f(k) = V(\mathsf{fst}(k), \mathsf{snd}(k))$. We need to exhibit a total computable unary function σ such that $V(m,x) = G(\sigma(m), x)$. Now since U is universal we have $\exists i$ such that $f(k) = U(i,k)$. We define σ by $\sigma(m) = \tau(i,m)$. Now note that $G(\sigma(m), x) = G(\tau(i,m), x) = U(\mathsf{fst}(\tau(i,m)), \tau(\mathsf{snd}(\tau(i,m)), x))$. If we use the basic relations between τ and fst and snd to simplify we get

$$G(\sigma(m), x) = U(i, \tau(m,x)) = f(\tau(m,x)) = V(m,x).$$

This is exactly what we want for a Gödel universal function. ∎

We took a lot of trouble to find a Gödel universal function, is it worthwhile? Yes! Now we can compose functions *effectively.*

Theorem 7. Let G be a Gödel universal function for the class of unary computable functions. Then there exists $c : \mathbf{N}^2 \to \mathbf{N}$, a total computable function, such that $\forall p,q,x\ (G_p \circ G_q)(x) = G(p, G(q,x)) = G(c(p,q), x)$.

Proof. Define a binary computable function V by the formula

$$V(m,x) = G(\mathsf{fst}(m), G(\mathsf{snd}(m), x)) \text{ or } V(\tau(p,q), x) = G(p, G(q,x)).$$

Since G is a Gödel universal function we can find a total computable unary function s such that $V(m,x) = G(s(m), x)$. Thus the binary total computable function c that we are looking for is given by $c(p,q) = s(\tau(p,q))$.

Let us verify that this works as advertised[6]:

$$G(c(p,q), x) = G(s(\tau(p,q)), x) = V(\tau(p,q), x) = G(p, G(q,x)).$$

∎

[6]A good test for you is to justify every equality.

What we have now is a programming system. We have a way of effectively encoding programs as data, we have a "universal" interpreter that can read programs (described by their code numbers) and inputs and execute the program on the desired input, and we can build new programs (and compute their codes) by composing old ones. We have restricted ourselves to unary and binary functions in order to not have the notational complexity overwhelm the ideas. If we defined everything for general arity functions, and appropriately generalized all the definitions, we would obtain exactly what is called an *acceptable programming system* in the computability theory literature.

The programming languages that we have today give us some simple notation for combining statements (just ; for basic statements but obviously something much more complicated for linking together independently compiled modules) to produce more complex statements and ultimately programs. The results described in this note show that this is possible. Behind the scences, some things have to be done to make all this work, but whatever it is, it will be effective and can thus be implemented. The essential idea of a programming language is that complicated programs can be build up by combining simpler programs. It is not at all clear how to do this with, for example, Turing machines and the early pioneers had to work hard to show that these things that we now take for granted are possible. This was the work of Church and his students, Kleene and Rosser and also others like Post.