Computability of Bass-Serre structures

Christian Perfect

February 28, 2011

1 Building blocks

1.1 Functions

Let $\mathbb{Z}_{\geq 0}$ denote the non-negative integers. A function $f(x_1, \ldots, x_n)$ is a map,

$$f : \mathbb{Z}_{\geq 0}^n \to \mathbb{Z}_{\geq 0},$$

taking $n \geq 0$ parameters and resulting in a single value.

1.2 Notation

Now I will describe the notation and constructions I will use to define functions. $E \equiv E' := D$ means I will write E or E' to mean D.

1.2.1 Arity

Functions in general can take any number of parameters, so it is clearly impractical to write out definitions of each operation for each possible arity. When the arity of a function f doesn't matter, I will write it as $f(\mathbf{x})$, where $\mathbf{x}$ represents an arbitrarily big vector. When a function has to take at least a specific number of parameters $y_1, \ldots, y_n$, and potentially more, I will write $f(\mathbf{x}, y_1, \ldots, y_n)$

1.2.2 Composition

$$f \circ g(x) \equiv f(g(\mathbf{x})) := C(f, g)(\mathbf{x})$$

or just

$$f \circ g := C(f, g).$$

1.2.3 Recursion

Definitions of recursive functions $f := R(g, h)$ will follow this format:

$$\begin{aligned} f(\mathbf{x}, 0) &:= g(\mathbf{x}), \\ f(\mathbf{x}, n+1) &:= h(n, f(\mathbf{x}, n), \mathbf{x}). \end{aligned}$$

Note that the parameter of recursion n can be made to be in any position by composition with the projection functions:

$$f(x_1, \ldots, x_m, n) := R(g, h)(x_1, \ldots, x_m, , n),$$

$$\begin{aligned} f'(x_1, \ldots, x_k, n, x_{k+1}, \ldots, x_m) &:= C(f, P^1_{m+1}, \ldots, P^k_{m+1}, P^{m+1}_{m+1}, P^{k+1}_{m+1}, \ldots, P^m_{m+1}) \\ &= f(x_1, \ldots, x_m, n). \end{aligned}$$

1.2.4 Iteration

Finite iteration of a function is achieved by repeatedly applying the composition operator.

$$f^{(n)} := \underbrace{C(f, C(f, C(f, \ldots C(f}_{n-1 \text{ compositions}}, f)).$$

1.2.5 Infix operators

An infix operator is simply a function of two variables, so for any operator $*$ some suitable function f_* could be defined:

$$x * y := f_*(x, y.)$$

1.2.6 Sets

A set $S \subseteq \mathbb{N}^d$ is defined by its characteristic function χ_S.

$$\chi_S(\mathbf{x}) := \begin{cases} 1 & \text{if } \mathbf{x} \in S, \\ 0 & \text{if } \mathbf{x} \notin S. \end{cases}$$

1.2.7 Relations

A relation is defined by the set of things which satisfy it. In fact, a d-ary relation can be thought of as a subset of $\mathbb{N}^d$, so the same notation as for sets can be

used. In addition, for binary relations R I will write,

$$xRy := \chi_R(x, y).$$

1.2.8 Logic

The concept of falsity will be represented by the number 0, and all other values will represent truth. With this in mind, predicates $P(\mathbf{x})$ can be written out as ordinary functions once the logical operations have been defined.

1.3 Constructing functions

We will begin by defining the *primitive functions*:

- *The zero function.* The zero function is a nullary function which returns the number 0. For brevity I will write 0 to mean the zero function.
- *The successor function.* The successor function $s(x) := x + 1$ is a unary function which adds 1 to its input.
- *The projection functions.* For every $n > 0$ and every $1 \leq i \leq n$ there is a projection function $P_n^i(x_1, \ldots, x_n) := x_i$. P_n^i is an n-ary function which returns its ith parameter.

There are two operations we shall use to create new functions from these primitives.

- *Composition*: $h := C(f, g_1, \ldots, g_m)$ means that $h(\mathbf{x}) := f(g_1(\mathbf{x}), \ldots, g_m(\mathbf{x}))$
- *Recursion*: $h := R(f, g)$ means that $h(\mathbf{x}, 0) := f(\mathbf{x})$, and $h(\mathbf{x}, n + 1) := g(n, h(\mathbf{x}, n), \mathbf{x})$

A *bound variable* is one which is used to define a function, not passed as a parameter into it. For instance, in P_n^i, n and i are bound variables.

Mixing functions of different arities can be dealt with systematically. For every n-ary function f, and every $m \geq n$, we can define an m-ary function f_m such that:

$$f_m(x_1, \ldots, x_n, \ldots, x_m) := C(f, P_m^1, \ldots, P_m^n) = f(x_1, \ldots, x_n).$$

Variable substitution can also be dealt with systematically:

$$f(x_1, \ldots, y, \ldots, x_n) := f(x_1, \ldots, y(x_1, \ldots, x_n), \ldots, x_n) = C(f, P_n^1, \ldots, y, \ldots, P_n^n).$$

2 Properties of classes of functions

A class $\mathcal{C}$ of functions is *closed under finite composition* if, whenever $f, g_1, \ldots, g_m$ all belong to $\mathcal{C}$, then $C(f, g_1, \ldots, g_m)$ also belongs to $\mathcal{C}$. In short, a function defined by finitely many compositions of functions in $\mathcal{C}$ is also a function in $\mathcal{C}$.

A class $\mathcal{C}$ is *closed under bounded recursion* if, whenever f, g, h belong to $\mathcal{C}$ and $\forall \mathbf{x},\ R(f,g)(\mathbf{x}) \leq h(\mathbf{x})$, then $R(f,g)$ also belongs to $\mathcal{C}$. In other words, if a recursion on two functions of $\mathcal{C}$ is bounded everywhere by another function in $\mathcal{C}$, then that recursion is also in $\mathcal{C}$.

3 The Grzegorczyk hierarchy

The *Grzegorczyk hierarchy* is an infinite nested hierarchy of classes $\mathcal{E}^n$ of functions which are each closed under finite composition and bounded recursion.

$\mathcal{E}^0$ is the smallest class of functions which contains the primitive functions and is closed under composition and bounded recursion. So $\mathcal{E}^0$ consists of the primitive functions, defined previously, and whatever else can be constructed by finitely many applications of the composition and bounded recursion operations.

$\mathcal{E}^{n+1}$, for $n \geq 0$, contains all of $\mathcal{E}^n$ and anything obtained by a single *unbounded* recursion on $\mathcal{E}^n$ functions, and is closed under finite composition and bounded recursion.

3.1 The stockpile of functions

Below I will give explicit definitions for the operations of elementary arithmetic, plus some more useful things, using the notation defined above. I have tried to use definitions which, though not the most obvious, place functions as low down on the hierarchy as possible.

3.2 $\mathcal{E}^0$ functions

Add a constant n to x by iterating the successor function n times on x:

$$+_n(x) \equiv x + n := s^{(n)}(x). \tag{1}$$

Note that n is a bound variable, that is, there is a different function $+_n$ for each n.

Construct a function c_n returning a constant number n by composing $+_n$ with z:

$$c_n := +_n(0). \tag{2}$$

From now on I will just write the number n to mean the application of the function $c_n()$, when appropriate.

Decrement by 1:

$$\begin{aligned} \mathrm{dec}(0) &:= 0, \\ \mathrm{dec}(n+1) &:= n. \end{aligned} \tag{3}$$

Proper subtraction:

$$\begin{aligned} x \dot{-} 0 &:= x, \\ x \dot{-} (y+1) &:= \mathrm{dec}(x \dot{-} y). \end{aligned} \tag{4}$$

$x \dot{-} y$ is bounded by $P_2^1 \equiv x$, so belongs to $\mathcal{E}^0$.

Signature:

$$\begin{aligned} \mathrm{sg}(0) &:= 0, \\ \mathrm{sg}(x+1) &:= 1. \end{aligned} \tag{5}$$

And reverse signature:

$$\begin{aligned} \overline{\mathrm{sg}}(0) &:= 1, \\ \overline{\mathrm{sg}}(x+1) &:= 0. \end{aligned} \tag{6}$$

Equality:

$$\begin{aligned} x = y &:= \mathrm{eq'}(x, y, \overline{\mathrm{sg}}(x \dot{-} y)). \\ \mathrm{eq'}(x, y, 0) &:= \overline{\mathrm{sg}}(y \dot{-} x), \\ \mathrm{eq'}(x, y, n+1) &:= 1. \end{aligned} \tag{7}$$

eq' is bounded by c_1, so belongs to $\mathcal{E}^0$.

Logical AND:

$$\begin{aligned} x \wedge 0 &:= 0, \\ x \wedge (n+1) &:= x. \end{aligned} \tag{8}$$

$\wedge$ is bounded by $P_2^1 \equiv x$, so is in $\mathcal{E}^0$.

Logical OR:

$$\begin{aligned} x \vee 0 &:= x, \\ x \vee (n+1) &:= 1. \end{aligned} \tag{9}$$

$\vee$ is bounded by $s \circ P_2^1 \equiv x + 1$, so is in $\mathcal{E}^0$.

Logical NOT:

$$\neg x := \overline{\mathrm{sg}}(x). \tag{10}$$

Ordering:

$$\begin{aligned} x > y &:= \mathrm{sg}(x \dot{-} y), \\ x \geq y &:= (x > y) \vee (x = y), \\ x \leq y &:= \neg(x > y), \\ x < y &:= (x \leq y) \wedge \neg(x = y). \end{aligned} \tag{11}$$

The smallest of two numbers:

$$\min(x, y) := x \dot{-} (x \dot{-} y). \tag{12}$$

Remainder when dividing x by y:

$$\begin{aligned} 0 \bmod y &:= 0, \\ (x+1) \bmod y &:= \mathrm{rm'}\left(y \dot{-} \left((x \bmod y) + 1\right), x \bmod y\right). \end{aligned} \tag{13}$$

$$\begin{aligned} \mathrm{rm'}(0, y) &:= 0, \\ \mathrm{rm'}(x+1, y) &:= y + 1. \end{aligned}$$

$\mathrm{rm'}(x, y)$ is bounded by $y + 1$ so belongs to $\mathcal{E}^0$. $x \bmod y$ is defined by recursion on $\mathcal{E}^0$ functions and is bounded by $y - 1$, so belongs to $\mathcal{E}^0$.

3.3 $\mathcal{E}^1$ functions

Adding two numbers together is achieved by an unbounded recursion on the successor function.

$$\begin{aligned} x + 0 &:= x, \\ x + (y + 1) &:= (x + y) + 1. \end{aligned} \tag{14}$$

Hence addition must belong to $\mathcal{E}^1$ and not $\mathcal{E}^0$.

We can define another version of $+_x$ where x is unbound:

$$+_x(y) := x + y. \tag{15}$$

. This version is in $\mathcal{E}^1$. This is really an abuse of notation, but it will be useful in a little while.

The biggest of two numbers:

$$\max(x, y) := x + (y \dot{-} x). \tag{16}$$

Note that $max(x, y)$ is bounded by one of P_2^1 or P_2^2, but I don't think there's a way of defining it without using addition, so it has to be in $\mathcal{E}^1$.

Absolute difference:

$$|x - y| := (x \dot{-} y) + (y \dot{-} x). \tag{17}$$

Any constant multiple $n \cdot x$ is produced by n compositions of addition:

$$n \cdot x := +_x^{(n)}(0). \tag{18}$$

So we can also get any positive- and constant-coefficient linear polynomial $a_1x_1 + a_2x_2 + \ldots a_nx_n$ by finite compositions. For polynomials with negative coefficients, computing all the positive terms then subtracting the negative terms will give the closest match, because we can't use negative numbers.

Quotient:

$$\begin{aligned} \left\lfloor \frac{0}{y} \right\rfloor &:= 0, \\ \left\lfloor \frac{x+1}{y} \right\rfloor &:= \left\lfloor \frac{x}{y} \right\rfloor + \overline{\mathrm{sg}}\left((x+1) \bmod y\right). \end{aligned} \tag{19}$$

Does x divide y:

$$x \mid y := \overline{\mathrm{sg}}(x \bmod y). \tag{20}$$

Bounded minimisation:

$$\begin{aligned} \min_{i<1} P(\mathbf{x}, i) &:= \overline{\mathrm{sg}}(P(\mathbf{x}, 0)), \\ \min_{i<y+1} P(\mathbf{x}, i) &:= \left(\min_{i<y} P(\mathbf{x}, i)\right) + \mathrm{sg}\left(\left(\min_{i<y} P(\mathbf{x}, i)\right) = y\right). \end{aligned} \tag{21}$$

Because we always have to return a value, if there is no value under the bound such that the predicate holds, then we return the bound.

Bounded maximisation:

$$\max_{i<y} P(\mathbf{x}, i) := y \dot{-} \min_{i<y} P(\mathbf{x}, y \dot{-} i). \tag{22}$$

If P is a $\mathcal{E}^1$ function then so are bounded minimisations and maximisations over P, because they are defined by a recursion over $\mathcal{E}^1$ functions, bounded by y.

3.4 $\mathcal{E}^2$ functions

Multiplication is an unbounded recursion on the addition operation:

$$\begin{aligned} x \cdot 0 &:= 0, \\ x \cdot (y+1) &:= (x \cdot y) + x. \end{aligned} \tag{23}$$

We can get x^n, when n is constant, by n compositions of multiplication by x, and hence any (positive-coefficient) polynomial by finite compositions of addition and multiplication. So polynomials belong to $\mathcal{E}^2$.

Series sum:

$$\begin{aligned} \sum_{i<0} f(\mathbf{x}, i) &:= 0, \\ \sum_{i<y+1} f(\mathbf{x}, i) &:= f(\mathbf{x}, y+1) + \sum_{i<y} f(\mathbf{x}, i). \end{aligned} \tag{24}$$

Note that the level of the sum on the hierarchy really depends on the level of $f(\mathbf{x}, i)$. However, we can say that if f is in $\mathcal{E}^2$ then $\sum_{i<y} f(\mathbf{x}, i)$ is also in $\mathcal{E}^2$ because it entails y additions. If y is a bound variable and f is in $\mathcal{E}^1$, then we could even place the sum in $\mathcal{E}^1$ by defining the sum as y compositions of addition.

Bounded existential quantifier:

$$\exists_{i<y} P(\mathbf{x}, i) := \sum_{i<y} P(\mathbf{x}, i). \tag{25}$$

Definition by cases: if $P_i(\mathbf{x})$, $1 \leq i \leq k$ is a finite collection of mutually exclusive, exhaustive predicates, then we can define

$$f(x) := \begin{cases} g_1(\mathbf{x}) & \text{if } P_1(\mathbf{x}), \\ \vdots & \vdots \\ g_k(\mathbf{x}) & \text{if } P_k(\mathbf{x}). \end{cases}$$

In our language, this is:

$$f(x) := \sum_{i<k+1} g_i(x) \cdot sg(P_i(x)), \tag{26}$$

where k is a bound variable. If all the P_i and g_i are in $\mathcal{E}^2$, then so is f, because we are just doing k multiplications and additions.

3.5 $\mathcal{E}^3$ functions

$\mathcal{E}^3$ is where just about all the really useful operations are.

First of all, we can get exponentiation by unbounded recursion on the multiplication operation:

$$\begin{aligned} x^0 &:= 1, \\ x^{y+1} &:= x \cdot x^y. \end{aligned} \tag{27}$$

Factorial:

$$\begin{aligned} 0! &:= 1, \\ (x+1)! &:= (x+1) \cdot x!. \end{aligned} \tag{28}$$

Series product:

$$\begin{aligned} \prod_{i<0} f(\mathbf{x}, i) &:= 1, \\ \prod_{i<y+1} f(\mathbf{x}, i) &:= f(\mathbf{x}, y+1) \cdot \prod_{i<y} f(\mathbf{x}, i). \end{aligned} \tag{29}$$

If f is in $\mathcal{E}^3$ then so is the product $\prod_{i<y} f(\mathbf{x}, i)$, as it consists of y multiplications. If y is a bound variable, then we can define the product using y compositions of multiplication instead of the recursion, so if f is in $\mathcal{E}^2$ then the series product is too.

Universal quantifier:

$$\forall_{i<y} P(\mathbf{x}, i) := \prod_{i<y} P(\mathbf{x}, i). \tag{30}$$

Primality:

$$x \text{ prime} := (x > 1) \wedge \forall_{z<x} \left((z < 2) \vee \neg(z \mid x) \right). \tag{31}$$

The first prime after x:

$$\text{nextprime}(x) := \min_{i \leq x!+1} (i \text{ prime}) \wedge (i > x). \tag{32}$$

The bound on this function comes from Euclid or some such bearded Greek. Note that this is an exceptionally inefficient way of finding primes.

The nth prime:

$$\begin{aligned} p_0 \equiv p(0) &:= 2, \\ p_{n+1} \equiv p(n+1) &:= \text{nextprime}(p_n). \end{aligned} \tag{33}$$

Exponent of x in the decomposition of y:

$$[y]_x := \max_{i<y} \left(x^i \mid y \right). \tag{34}$$

3.6 Lists

We will be working with n-tuples, or lists, extensively. All of the list operations can be defined as $\mathcal{E}^3$ functions on numbers by way of the Gödel encoding.

Gödel encoding of an n-tuple:

$$\mathbf{x} \equiv [x_0, \ldots, x_n] := \prod_{0 \leq i \leq n} p_i^{x_i+1}. \tag{35}$$

The $+1$ in the exponent is so we can reliably determine the length of a list. The bold $\mathbf{x}$ is shorthand so I don't have to write out the brackets formation. It shouldn't be confused with the use of $\mathbf{x}$ previously to mean an arbitrary number of parameters in a function application. An encoded list $\mathbf{x}$ is just a single number.

Value of nth position in an encoded list:

$$(\mathbf{x})_i := [\mathbf{x}]_{p_i} - 1. \tag{36}$$

Length of an encoded list:

$$|\mathbf{x}| := \min_{i<\mathbf{x}} \left([\mathbf{x}]_{p_i} = 0\right). \tag{37}$$

The index of the first occurrence of a given value in an encoded list:

$$\text{find}(\mathbf{x}, n) := \min_{i<|\mathbf{x}|} (\mathbf{x})_i = y. \tag{38}$$

Whether an encoded list contains a given value:

$$y \in \mathbf{x} := \text{find}(\mathbf{x}, y) < |\mathbf{x}|. \tag{39}$$

The largest element of an encoded list:

$$\text{biggest}(\mathbf{x}) := \text{biggest}'(\mathbf{x}, |\mathbf{x}| - 1), \tag{40}$$

$$\begin{aligned} \text{biggest}'(\mathbf{x}, 0) &:= (\mathbf{x})_0, \\ \text{biggest}'(\mathbf{x}, n+1) &:= \max\left((\mathbf{x})_{n+1}, \text{biggest}'(\mathbf{x}, n).\right) \end{aligned}$$

To obtain the sub-list of $\mathbf{x}$ starting at position s and ending at position e.:

$$\mathbf{x}[s \ldots e] \equiv \text{slice}(\mathbf{x}, s, e) := \prod_{i=s'}^{e'} (p_{i-s'})^{(\mathbf{x})_i+1}, \tag{41}$$

$$\begin{aligned} e' &:= \max(0, \min(e, |\mathbf{x}| - 1)), \\ s' &:= \max(0, \min(s, e')). \end{aligned}$$

To append a single value to the end of an encoded list:

$$\text{splice}(\mathbf{x}, n) := \mathbf{x} \cdot p_{|\mathbf{x}|}^{n+1}. \tag{42}$$

To concatenate two encoded lists:

$$\mathbf{x} +\!\!+ \mathbf{y} := \text{splice}(\mathbf{x}, (\mathbf{y})_0) +\!\!+ \mathbf{y}[1 \ldots |\mathbf{y}| - 1] \tag{43}$$

$x +\!\!+ y$ is bounded by $p_{|\mathbf{x}|+|\mathbf{y}|}^{(|\mathbf{x}|+|\mathbf{y}|)(\text{biggest}(\mathbf{x})+\text{biggest}(\mathbf{y}))}$, so is $\mathcal{E}^3$-computable.

4 Graphs

Definition 4.1. A graph is a set of vertices V and a set of directed edges $E = E^+ \cup E^-$, with two associated maps. The first,

$$e \to (\iota(e), \tau(e)),$$

associates each edge with its initial and terminal vertices. The second,

$$e \to \bar{e},$$

is an involution on the edges, which must satisfy the conditions that $\bar{\bar{e}} = e$ and $\iota(\bar{e}) = \tau(e)$, $\forall e \in E$. We will write e_{ij} to mean the edge with $\iota(e_{ij}) = i$ and $\tau(e_{ij}) = j$. For any pair of edges e and $\bar{e}$, one is 'positive' and belonging to E^+, and one 'negative' and belonging to E^-.

Definition 4.2. A *path* in a graph is a sequence of edges $(e_1, e_2, \ldots, e_n)$ such that $\tau(e_i) = \iota(e_{i+1})$, $1 \leq i < n$. A path is *reduced* if it does not contain an edge followed by its inverse, that is, if $e_{i+1} \neq \bar{e}_i$, $1 \leq i < n$.

Definition 4.3. A *circuit* is a path $(e_1, \ldots, e_n)$ with $\iota(e_1) = \tau(e_n)$.

Definition 4.4. A graph is *connected* if there exists a path between each pair of its vertices.

5 Computable Trees

Definition 5.1. A tree is a connected, non-empty graph with no reduced circuits.

A *rooted* tree has a particular vertex v_0 said to be at the 'top'. In an *ordered* tree, the set of children of each vertex is ordered, so there is a leftmost child, and so on.

Definition 5.2. A *plane recursive tree* is a rooted, ordered tree obtained by a process which begins with the root node and recursively adds vertices below existing ones.

We can assign a labelling to a plane recursive tree by giving vertices labels corresponding to the order in which they were added to the tree, beginning with 0 for the root. This way, the sequence of labels encountered on a path heading downwards is always increasing.

If we say that the positive edges point downwards, every positive edge can also be given a unique labelling by saying that the edge e_{ij} has label $max(i, j)$. For the purposes of the computation, negative edges will have the same label as their positive counterpart.

Definition 5.3. Suppose we have a tree $T = (V, E)$ with root vertex v_0. T is $\mathcal{E}^n$-computable if there is an $\mathcal{E}^n$-decidable labelling

$$i : V \to \mathbb{N},$$

(we can assume $i(v_0) = 0$) and an $\mathcal{E}^n$-computable 'parent function'

$$\phi_T : i(V) \to i(V),$$

which gives the label of the parent of a vertex. For completeness, we also require that $\phi_T(v_0) = i(v_0)$.

Definition 5.4. A finite plane recursive tree can be entirely described in a canonical way by a *depth-first walk*. Note that every node has a single parent and a finite ordered set of children. The process on 'visiting' a vertex v_i goes as follows: for each child node v_j, travel along e_{ij} and 'visit' v_j, then travel backwards along $\overline{e_{ij}}$. The sequence of labelled edges travelled along by beginning this process at v_0 is the depth-first walk of the tree.

Note that the root vertex's label will not appear in this sequence because it has no positive edge leading into it.

Because an edge and its inverse have the same label, the first occurrence of a label in the sequence represents the positive edge, and the second occurrence represents the trip back up the negative edge.

The depth-first walk can be encoded as a Gödel number by the method in Section 3.6. We will define a function $\phi_{\mathbf{t}}(n)$ which takes an encoded depth-first walk $\mathbf{t}$ of a tree and gives the label of the parent of the vertex with label n.

5.1 Some functions to work with encoded trees

Let $\mathbf{t}$ be an encoded depth-first walk of a tree.

To find the first and second occurrences of the edge with label n in the walk:

$$\begin{aligned} \mathrm{fst}(\mathbf{t}, n) &:= \mathrm{find}(\mathbf{t}, n), \\ \mathrm{snd}(\mathbf{t}, n) &:= \mathrm{find}\left(\mathbf{t}[\mathrm{fst}(\mathbf{t}, n) + 1 \ldots |\mathbf{t}| - 1], n\right). \end{aligned} \tag{44}$$

find is $\mathcal{E}^3$-computable, and snd is bounded by $|\mathbf{t}|$ so is also $\mathcal{E}^3$-computable.

To get the label of the nth child of the root:

$$\begin{aligned} \mathrm{child}(\mathbf{t}, 0) &:= (\mathbf{t})_0, \\ \mathrm{child}(\mathbf{t}, n+1) &:= (\mathbf{t})_{\mathrm{snd}(\mathbf{t}, \mathrm{child}(\mathbf{t}, n))+1}. \end{aligned} \tag{45}$$

child is constructed from $\mathcal{E}^3$ functions and is bounded by the number of vertices in the tree. As the tree is finite, child is $\mathcal{E}^3$-computable.

The depth-first walk of the subtree descended from a vertex v_n is the subsequence found between the two occurrences of n in $\mathbf{t}$.

To find the subtree descended from the nth child node of the root:

$$\text{subtree}(\mathbf{t}, n) := \mathbf{t}[\text{fst}(\text{child}(\mathbf{t}, n)) + 1 \ldots \text{snd}(\text{child}(\mathbf{t}, n)) - 1]. \tag{46}$$

subtree is constructed from $\mathcal{E}^3$ operations so is $\mathcal{E}^3$-computable.

The number of children of the root node:

$$\text{kids}(\mathbf{t}) := \min_{i<\mathbf{t}}(\text{child}(\mathbf{t}, i) = 0). \tag{47}$$

kids is constructed by bounded minimisation on $\mathcal{E}^3$ operations so is $\mathcal{E}^3$-computable.

Whether the root has a child with label n:

$$\text{haschild}(\mathbf{t}, n) := \left(\mu_{i<\text{kids}(\mathbf{t})} (\text{child}(\mathbf{t}, i) = n) \right) < \text{kids}(\mathbf{t}). \tag{48}$$

Finally, the parent function $\phi_{\mathbf{t}}$. The idea is basically to do the depth-first walk, remembering which vertex you just came from. If you reach a subtree which doesn't contain n then turn back, or if you reach the required vertex then return the label of the vertex you just came from.

$$\begin{aligned} \phi_{\mathbf{t}}(0) &:= 0, \\ \phi_{\mathbf{t}}(n) &:= \text{p}'(x, n, 0). \end{aligned} \tag{49}$$

$$\text{p}'(x, n, p) := \begin{cases} 0 & n \notin \mathbf{t}, \\ p & \text{haschild}(\mathbf{t}, n), \\ \text{p}'(\text{subtree}(\mathbf{t}, z), n, \text{child}(\mathbf{t}, z)) & \text{otherwise.} \end{cases}$$

$$z := \min_{i<kids(\mathbf{t})} (n \in subtree(\mathbf{t}, i)).$$

We will also need to know if the tree contains an edge e_{ij}:

$$e_{ij} \in \mathbf{t} \Leftrightarrow \phi_{\mathbf{t}}(i) = j \vee \phi_{\mathbf{t}}(j) = i \tag{50}$$

Lemma 5.5. A finite ordered tree is $\mathcal{E}^3$ computable.

Proof. As the tree is finite, of course its labelling is an $\mathcal{E}^3$-decidable subset of $\mathbb{N}$, and $\phi_{\mathbf{t}}$ as defined above is $\mathcal{E}^3$-computable. □

6 Computability of Bass-Serre groups

Let $\Gamma = (V, E)$ be a connected graph.

Definition 6.1. Associate with each vertex v a vertex group $G_v = \langle X_v \mid R_v \rangle$, with the X_v all pairwise disjoint. Call the set of all vertex groups $\mathbf{G}$.

For each edge $e \in E$ associate two isomorphic groups $A_e \leq G_{\iota(e)}$ and $B_e \leq G_{\tau(e)}$, with $A_e = B_{\bar{e}}$, and an isomorphism $\phi_e : A_e \to B_e$.

$(\mathbf{G}, \Gamma)$ is called a *graph of groups.*

Definition 6.2. Let T be a spanning tree of Γ (a subtree of Γ containing every vertex), with a root vertex v_0. Then $G = \pi_1(\mathbf{G}, \Gamma, T, v_0)$ is called the *fundamental group* of $(\mathbf{G}, \Gamma)$, defined by

$$G = \pi_1(\mathbf{G}, \Gamma, T, v_0) = (*_{v \in V} G_v) \underset{\phi_e}{*} F(E) \tag{51}$$

The generators of G are

$$X = \left(\bigcup_{v \in V} X_v \right) \cup \left(\bigcup_{e \in E} \{e\} \right),$$

and the relations are

$$R = \left(\bigcup_{v \in V} R_v \right) \cup \{e^{-1} A_e e = B_e, \forall e \in E\} \cup \{e^{-1} = \bar{e}, \forall e \in E\} \cup \{e = 1, \forall e \in T\}.$$

(this looks horrible, please help!)

Definition 6.3. A word $w \in X^*$ is *admissible* if it is in the form

$$w = a_0 e_1 a_1 e_2 \dots a_{n-1} e_n a_n$$

where $(e_1, \dots, e_n)$ is a circuit in Γ with vertex sequence $(v_0, \dots, v_n)$, such that $v_0 = v_n$, $v_i = \iota(e_{i+1}) = \tau(e_i)$, $0 < i < n$, and $a_i \in G_{v_i}$, $i = 0, \dots, n$.

It will be more convenient from now on to refer to G_{v_i} by G_i.

Lemma 6.4. (Higgins?) Let $w \in X^*$ be an admissible word, and suppose $w =_G 1$. Then either:

- $n = 0$ and $a_0 = 1$ in G_{v_0}, or
- $n > 0$ and there exists $0 < i < n$ such that $e_{i+1} = \bar{e}_i$ and $a_i \in B_{e_i}$.

Definition 6.5. Let $\phi_T(n)$ be the parent function for T. The path in T from

v_i to v_j can be found by freely reducing the path $\psi(i,j)$, defined by:

$$\begin{aligned}
\psi(0,0) &:= \epsilon, \\
\psi(i,0) &:= (e_{i,p_T(i)}) \cdot \psi(p_T(i)), \\
\psi(0,i) &:= \psi(i,0)^{-1}, \\
\psi(i,j) &:= \psi(i,0)\psi(0,j).
\end{aligned} \tag{52}$$

The free reduction of $\psi(i,j)$ will be implicit from now on.

What we really want is a function $\pi(g) : G \to X^*$ that gives an admissible word beginning and ending at v_0 and containing a representation of g as a contiguous subword.

If g belongs to some G_{v_i}, then $\pi(g) = \psi(0,i) \cdot g \cdot \psi(i,0)$. Otherwise, if $g = e_{ij}$, some edge letter, then $\pi(e_{ij}) = \psi(0,i) \cdot e_{ij} \cdot \psi(j,0)$.

Lemma 6.6. Let w be a word on G. Then by replacing each generator g by $\pi(g)$ and freely reducing the resulting word, we obtain an admissible word starting at v_0 which is equivalent to w. Call this word $\pi(w)$, abusing notation a bit.

Proof. The new word is equivalent to w because the only letters we are adding are edge letters from the spanning tree, which are all trivial in the presentation of G. □

Theorem 6.7. *Suppose we have $G = \pi_1(\mathbf{G}, \Gamma, T, v_0)$, where Γ has ν vertices. Suppose all the G_i are f.g. $\mathcal{E}^n$-computable groups for $n \geq 3$. Assume all the identified subgroups are $\mathcal{E}^n$-decidable, and all the isomorphisms ϕ_e are $\mathcal{E}^n$-computable. Then G is $\mathcal{E}^{n+1}$-computable.*

Proof. We will show that the word problem of G is $\mathcal{E}^{n+1}$-decidable. We can assume that the generating sets of all the G_v are disjoint.

Encode the depth-first walk of the spanning tree T as a Gödel number $\mathbf{t}$ per Section 5.1. Then the parent function $\phi_{\mathbf{t}}$ is $\mathcal{E}^3$-computable.

To define $i(G)$, we will begin by using the first ν^2 numbers to represent potential edges. For any $i = a * \nu + b$ and $i \leq \nu^2$, $i \in i(G) \Leftrightarrow e_{ab} \in T$.

We can assume the generators of the vertex groups have indices greater than ν^2. The rest of $i(G)$ works the same way as the standard free group index in Cannonito-Gatterdam. ((Not explained well: C-G assign a number to each generator, and then words are encoded as sequences of those numbers paired with a power, and the index of an element is the least index of an equivalent word. I want the numbers assigned to the group generators to be greater than ν^2.))

Given a word w, compute $w' = \pi(w)$. We now need to split w' into an admissible sequence $(a_0, e_1, \ldots, e_n, a_n)$.

We must decide for each letter in w' either which vertex group it is from or if it is an edge letter. Assign to each letter w_i of w' a code as follows: If w_i belongs

to G_i, then its code is i. If it is an edge letter, then its code is $\text{biggest}(\mathbf{t}) + 1$. We can then define a $\mathcal{E}^n$-decidable equivalence relation $\approx$ on the indices of the elements of G, whose equivalence classes correspond to the vertex groups plus one more for each edge letter. For completeness, say that $g \not\approx n$ whenever $n \notin i(G)$.

As there are finitely many equivalence classes of $\approx$, and they are all $\mathcal{E}^n$-decidable, $\approx$ is $\mathcal{E}^n$-decidable.

The next task is to split w' into 'syllables', or contiguous subwords. A syllable is either a single edge letter or a word from one of the vertex groups.

From now on, let $\mathbf{w}$ be an encoded admissible word.

Define a function which gives the position of the start of the syllable to which $(\mathbf{w})_i$ belongs:

$$\begin{aligned} \text{backtrack}(\mathbf{w}, 0) &:= 0, \\ \text{backtrack}(\mathbf{w}, i+1) &:= \begin{cases} i+1 & (\mathbf{w})_{i+1} \leq \nu^2, \\ i+1 & (\mathbf{w})_i \not\approx (\mathbf{w})_{i+1}, \\ \text{backtrack}(\mathbf{w}, i) & (\mathbf{w})_i \approx (\mathbf{w})_{i+1}. \end{cases} \end{aligned} \tag{53}$$

backtrack is constructed from $\mathcal{E}^n$ operations and is bounded by $|\mathbf{w}|$, so is $\mathcal{E}^n$-computable.

Now, the start of the nth syllable is given by:

$$\begin{aligned} \text{start}(\mathbf{w}, 0) &:= 0, \\ \text{start}(\mathbf{w}, n+1) &:= \min_{\text{start}(\mathbf{w},n)+1 \leq i < |\mathbf{w}|} (\text{backtrack}(\mathbf{w}, i) \neq \text{start}(\mathbf{w}, n)). \end{aligned} \tag{54}$$

The number of syllables can be computed like so:

$$\text{numsyllables}(\mathbf{w}) = \min_{i < |\mathbf{w}|} (\text{start}(\mathbf{w}, i) = |\mathbf{w}|. \tag{55}$$

And now the nth syllable itself can be found:

$$\text{syllable}(\mathbf{w}, n) = \mathbf{w}[\text{start}(\mathbf{w}, n) \ldots \text{start}(\mathbf{w}, n+1) - 1]. \tag{56}$$

syllable is constructed from $\mathcal{E}^n$-computable functions and is bounded by $\mathbf{w}$ so is $\mathcal{E}^n$-computable.

We can then use the conditions of Lemma 6.4 to determine if $w' = (a_0, e_1, \ldots, e_n, a_n)$, encoded as $\mathbf{w}'$, is trivial.

If $n = 0$, then $w' = 1 \Leftrightarrow a_0 = 1$, which is an $\mathcal{E}^n$-decidable question.

If $n > 0$, then we need to find a sequence of the form $e^{-1}A_e e$ or $eB_e e^{-1}$. The

first of these is given by

$$\begin{aligned}\min_{i<\text{numsyllables}(\mathbf{w}')-2} & ((\text{syllable}(\mathbf{w}', i) = e \in E)\wedge \\ & (\text{syllable}(\mathbf{w}', i+1) \in B_e)\wedge \\ & (\text{syllable}(\mathbf{w}', i+2) = \text{syllable}(\mathbf{w}', i)^{-1}))\end{aligned} \tag{57}$$

(and the same the other way round for A_e.) Note that since the first syllable must be an edge letter, we can say the last syllable is the inverse of the first by checking its length is 1, then computing its inverse. We don't need to know how to compute the whole multiplication table to do this.

If the result of that calculation is numsyllables$(\mathbf{w}') - 2$ then w is not trivial. Otherwise, we can replace the found sequence eB_ee^{-1} with $\phi_e(\text{syllable}(\mathbf{w}', i+1)$, and try again. The new word is still admissible and has fewer syllables than the original one, so repeated applications of this process will eventually lead to a word of one syllable or a negative answer. Because ϕ_e might increase the index of the word, this recursion raises us up a level on the Grzegorczyk hierarchy.

So the word problem of G is $\mathcal{E}^{n+1}$(A) decidable, and hence G is $\mathcal{E}^{n+1}$-computable. □

7 Pregroups

A pregroup (P, D) is $\mathcal{E}^n$*-computable* if:

- there is an $\mathcal{E}^n$-computable indexing $i : P \to \mathbb{N}$,
- D is $\mathcal{E}^n$-decidable as a subset of $i(P) \times i(P)$ (from now on we will consider D as consisting of pairs of indices instead of pairs of elements),
- $m : D \to i(P)$ is $\mathcal{E}^n$-computable.

Lemma 7.1. Let (P, D) be a countable, $\mathcal{E}^n$-computable pregroup. The universal group $\mathbf{U}(P)$ is $\mathcal{E}^{n+1}$-computable.

Proof. The index (i', m') of of $\mathbf{U}(P)$ will be on reduced P-words, encoded as Gödel numbers. So it is only necessary to define the multiplication function.

Because of Theorem 4 in Rimlinger, any P-words X and Y represent the same element of $\mathbf{U}(P)$ if and only if $X \sim Y$, that is, they are the same length and there is an interleaving of X equal to Y. The upshot is, when we have a reduced P-word X, we can find the $Y \sim X$ of minimal index by enumerating all the P-words of length $l_P(x)$. This step needs to be performed after any multiplication, so take that as read from here on.

Say we have two reduced P-words, $x = x_1 \dots x_m$ and $y = y_1 \dots y_n$. If $(x_m, y_1) \notin D$, then the product $m'(\mathbf{x}, \mathbf{y})$ is the concatenation $\mathbf{x} +\!\!+ \mathbf{y}$. (There can't be any other P-word of the same length equivalent to xy and with lower index, can there?)

If $x_m = y_1^{-1}$, then $m'(\mathbf{x}, \mathbf{y}) = m'([x_1 \dots x_{m-1}], [y_2 \dots y_n])$.

If $xy = z \in P$, then $m'(\mathbf{x}, \mathbf{y}) = m'(m'([x_1 \dots x_{n-1}], [z]), [y_2 \dots y_n])$. As z may have an index as big as any $\mathcal{E}^n$ function, the recursion becomes unbounded at this point, so m' is $\mathcal{E}^{n+1}$ computable. $\square$