The Primitive Recursive Functions are Recursively Enumerable

Stefan Kahrs

University of Kent at Canterbury
Department of Computer Science
Canterbury CT2 7NF
smk@kent.ac.uk

Abstract. Meta-operations on primitive recursive functions sit at the brink of what is computationally possible: the semantic equality of primitive recursive programs is undecidable, and yet this paper shows that the whole class of p.r. functions can be enumerated without semantic duplicates. More generally, the construction shows that for any equivalence relation $\approx$ on natural numbers, $\mathbb{N}/\approx$ is r.e. if $\approx$ is co-semi-decidable.

Keywords: Computability, Recursion Theory

1 Introduction

Starting point for this rather theoretical work were practical considerations in the areas of program testing, program generation [6] and genetic programming [4].

Primitive Recursive programs are in these areas an attractive tool, because (i) they always terminate, (ii) terminating programs that are not expressible through primitive recursion are of little practical interest, and (iii) Primitive Recursive programs can be expressed with a relatively sparse abstract syntax that keeps semantic redundancies at bay.

The latter is important when we want to search through the space of primitive recursive functions, e.g. to test higher-order functions. The bigger programs get, the rarer are the ones that could not be expressed more concisely. The fundamental question this paper addresses is: can we list all the primitive recursive functions without semantic duplicates, and perhaps in such a way that only the shortest ones of each equivalence class appears in the list?

Surprisingly, the answer to this question is "yes!". It surprised me, because the equality relation between primitive recursive functions is not recursive — it is undecidable whether two p.r. functions have the same input/output behaviour. Section 3 goes through the fundamental arguments why that is the case. This has implications *on the kind of* enumerations of p.r. functions that are possible. In particular, although it is possible to list only the shortest p.r. program of each equivalence class, it is *impossible* to overall list the different programs by size.

Viewing the natural numbers as Gödel numbers of p.r. functions, the semantic equality between p.r. functions can be interpreted as an equivalence relation $\approx$ on natural numbers, and our question can be expressed as: is the set $\{\min[n]_{\approx} \mid n \in \mathbb{N}\}$ recursively enumerable, i.e. the set of the smallest Gödel-numbers of all equivalence classes. It turns out that this question can be answered generically, without reference to primitive recursion: to construct an enumeration of $\{\min[n]_{\bowtie} \mid n \in \mathbb{N}\}$ it suffices that the equivalence relation $\bowtie$ is co-semi-decidable.

2 Basics: What is Primitive Recursion?

Primitive Recursion was first introduced as a method by Richard Dedekind in 1888 [3], and its expressiveness was explored in much more detail by Skolem, Ackermann and Péter in the 1920s and 1930s [1, 9, 8]; Rósza Péter's article was apparently the first to use the term "Primitive Recursion". Primitive Recursion is a scheme that permits to define a function by descending recursion in one variable, or to phrase it in the imperative paradigm: it has for-loops as the only iterative control structure. One very special property of Primitive Recursive programs is that they always terminate.

This paper uses a fairly natural representation of Primitive Recursion taken from [2]. This is based on two connectives, (generalised) composition `Cn` and the Primitive Recursion operator `Pr`, and as primitives the constant-0 function `Z`, the successor function `S`, and the argument selectors $\mathtt{id}_n^m$ which select the n-th argument out of m. The primitive recursive functions are all those expressible through these combinators. For example, the addition operation would be represented by the expression $\mathtt{Pr[id}_1^1\mathtt{,Cn[S,id}_1^3\mathtt{]]}$.

The semantics of generalised composition is as follows:

$$\mathtt{Cn}[f, g_1, \ldots, g_n](\boldsymbol{x}) = f(g_1(\boldsymbol{x}), \ldots, g_n(\boldsymbol{x}))$$

where $\boldsymbol{x}$ is an argument tuple of the length expected by the g_i functions. In category theory the construction $\mathrm{Cn}[f,g_1,\ldots,g_n]$ is usually expressed by separating composition from tuple formation, as $f \circ \langle g_1, \ldots, g_n \rangle$.

The semantics of the primitive recursion operator $\mathrm{Pr}[f,g]$ can be given as:

$$\begin{aligned} \mathtt{Pr}[f, g](0, \boldsymbol{x}) &= f(\boldsymbol{x}) \\ \mathtt{Pr}[f, g](n+1, \boldsymbol{x}) &= g(\mathtt{Pr}[f, g](n, \boldsymbol{x}), n, \boldsymbol{x}) \end{aligned}$$

It follows from the definition that if f has arity k then g must have arity $k+2$, and the newly defined $\mathtt{Pr}[f, g]$ has arity $k+1$, though it is fairly straightforward to relax these conditions. The reason this form of recursion corresponds to for-loops is that we can compute $\mathtt{Pr}[f, g](n, x_1, \ldots, x_k)$ as follows:

```
int s=f(x1,...,xk);
for(int i=0; i<n, i++) s=g(s,i,x1,...,xk);
return s;
```

The next section shows that the Primitive Recursive functions on their own are already very expressive. We can express all partially recursive functions if we combine them with the minimisation operator $\mathtt{Mn}$: $\mathtt{Mn}[f](\boldsymbol{x})$ is defined to be the smallest n such that $f(n, \boldsymbol{x}) = 0$ if such an n exists, and undefined otherwise.

3 Primitive Recursion and the Halting Problem

One of the most fundamental results in Computability is Kleene'e normal form theorem [5]. This states that any computable function can be expressed with just a single use of the minimisation operator, using otherwise only the Primitive Recursive primitives and connectives. Moreover this is constructive in the sense that we can construct such a normal form from the description of the function. This works because the configuration of a Turing machine can be encoded as a number, the state transition operation on the encoding is Primitive Recursive and we only need one unbounded loop to iterate state transition to simulate a running Turing machine.

This theorem has several consequences for Primitive Recursive functions. Notice first that we can also normalise the loop bodies, because minimisation only looks for the first non-zero value. In the following, let $\equiv$ denote observational equivalence between computable functions.

Lemma 1. *There is a normalisation operation* $\mathtt{norm}$ *on the Primitive Recursive functions with the following properties:*

$$\mathtt{Mn}[f] \equiv \mathtt{Mn}[\mathtt{norm}[f]]$$
$$\mathtt{Mn}[\mathtt{norm}[f]] \equiv \mathtt{Mn}[\mathtt{norm}[g]] \Rightarrow \mathtt{norm}[f] \equiv \mathtt{norm}[g]$$

Proof. We can specify $\mathtt{norm}[f](x_1, \ldots, x_n)$ to return 0 if $f(m, x_2, \ldots, x_n) = 0$ for any $m \leq x_1$, and to return 1 otherwise. This matches the requirements and is clearly expressible through primitive recursion. For example, for binary f, we can define $\mathtt{norm}[f]$ as follows:

$$\mathtt{norm}[f] = \mathtt{Pr}[\mathtt{Cn}[f, \mathtt{Z}, \mathtt{id}^1_1], \mathtt{Cn}[\mathtt{Pr}[\mathtt{Z}, \mathtt{id}^3_3], \mathtt{Cn}[f, \mathtt{Cn}[\mathtt{S}, \mathtt{id}^3_2], \mathtt{id}^3_3], \mathtt{id}^3_1]$$

□

Proposition 1. *Semantic equality between Primitive Recursive functions is undecidable.*

Proof. We can reduce the halting problem of any program p to the halting problem of its Kleene normal form, which in turn reduces to the halting problem for $\mathtt{Mn}[f]$, the sole application of minimisation in the Kleene normal form of p. Lemma 1 reduces this further, first to the halting problem for $\mathtt{Mn}[\mathtt{norm}[f]]$. For $k1 = \mathtt{Cn}[\mathtt{S}, \mathtt{Z}]$, the function that constantly returns 1, we have that $\mathtt{Mn}[k1]$ is not halting. Thus the halting problem for $\mathtt{Mn}[\mathtt{norm}[f]]$ can be reduced to the problem whether $\mathtt{norm}[f]$ and $\mathtt{norm}[k1]$ are equivalent. Those two functions are both Primitive Recursive, and hence deciding semantic equality of Primitive Recursive functions would provide a way to deciding the halting problem. □

Another consequence of Kleene's normal form theorem tells us something about the runtime complexity of functions not expressible through Primitive Recursion. If there is a Primitive Recursive function that computes an upper bound for the number of times the body of the sole unbounded loop of the Kleene normal form needs to be performed then we can replace the unbounded loop by a bounded one and have overall a Primitive Recursive construction. Hence, the runtime complexity for any function not expressible through Primitive Recursion is outside $O(f)$ for any Primitive Recursive function f — which is dramatically worse than the standard "infeasibility" criterion in complexity theory.

This explains why one should not aim to go beyond the universe of Primitive Recursive functions in approaches for program testing or genetic programming.

4 Limits to Enumerability of Equivalence Classes

Let $\bowtie$ be an equivalence relation on $\mathbb{N}$. A function $f : \mathbb{N} \to \mathbb{N}$ is called a *canonical enumeration* of $\mathbb{N}/\bowtie$ if the following properties hold:

- $\forall n, m.\ n \bowtie f(m) \Rightarrow f(m) \leq n$ (i.e. f returns only canonical representatives of equivalence classes);
- $\forall n.\ \exists m.\ f(m) \bowtie n$ (f finds all equivalence classes)

The function f is called a *strong* canonical enumeration if it is in addition monotonic, i.e. if it enumerates the equivalence class representatives in order. In addition, one could require that f should be injective, but this exclude the case that $\bowtie$ only has finitely many equivalence classes.

Co-semi-decidabilty of a relation means that there is a computable procedure which will terminate with the answer "false" for all non-members of the relation, and fail to terminate otherwise. This also means that one can stratify such a relation as follows.

Proposition 2. *If R is an n-ary co-semi-decidable predicate then there is a computable sequence of recursive n-ary predicates R_i such that*

$$\forall x_1, \ldots, x_n.\ (R(x_1, \ldots, x_n) \iff \forall i.\ R_i(x_1, \ldots, x_n))$$

Proof. We can turn the co-semi-decision procedure for R into a decision procedure for each R_i by limiting its number of computation steps to i, interpreting the state "I have not finished yet" of R as "true" for R_i. □

Clearly, the semantic equivalence of Primitive Recursive functions (of any given arity) is co-semi-decidable, because we can enumerate all possible inputs and apply the functions to them until a difference emerges. Because Primitive Recursive functions can only differ by return values (not by termination behaviour) different functions will be found out.

Proposition 3. *If $\bowtie$ is a co-semi-decidable equivalence relation and f a computable strong canonical enumeration for $\mathbb{N}/\bowtie$ then $\bowtie$ is decidable.*

Proof. Take any $n \in \mathbb{N}$. Because f is monotonic the finite set $A = \{f(k) \mid f(k) \leq n\}$ can effectively be computed by finding the first k with $f(k) > n$. Because f enumerates all equivalence classes n must belong to $[a]_{\bowtie}$ for some $a \in A$, and it cannot be in any of the other classes. We can now compute a sequence of sets A_i as follows: $A_0 = A$, $A_{k+1} = \{a \mid a \in A_k, a \bowtie_k n\}$, where the relations $\bowtie_k$ are recursive relations approximating $\bowtie$ as given by proposition 2. For some p, A_p must be a singleton set $\{q\}$ and we must have that q is the minimum element of $[n]_{\bowtie}$. Thus, to decide whether $m \bowtie n$ holds we can use this procedure to compute the minimum elements of $[m]_{\bowtie}$ and $[n]_{\bowtie}$ and compare them for equality. □

Corollary 1. *There is no computable strong enumeration for* $\mathbb{N}/\approx$.

Thus to enumerate the Primitive Recursive functions we have to aim for something weaker, either a non-canonical or a non-strong enumeration.

5 How to construct the enumeration

For our application domain of Primitive Recursive functions (technically, this is used here for p.r. functions in one argument only — but the construction is not substantially different for other arities), we can define suitable approximants $\approx_k$ of $\approx$ as follows:

$$n \approx_k m \iff \langle n \rangle(k) = \langle m \rangle(k)$$

The notation $\langle m \rangle$ stands here for the m-th unary Primitive Recursive function w.r.t. an enumeration that is allowed to contain semantic duplicates, e.g. a straightforward interpretation of numbers as encodings of syntax trees. Thus, $\langle n \rangle(k)$ stands for applying the n-th p.r. function (relative to that encoding) to the number k. Clearly, the conjunction of all $\approx_k$ gives the full $\approx$ relation.

We can approximate the quotient $\mathbb{N}/\approx$ through the sequence $A_0 = \mathbb{N}/\mathit{true}$, $A_{n+1} = A_n\star \approx_n$, where the operation $\star$ is refinement of equivalence classes: $(A/R) \star S = A/(R \cap S)$. Operationally, this can be realised by quotienting each equivalence class of A/R by S and then flattening the result.

One fundamental observation is that this process only splits equivalence classes, it never amalgamates them. This means each minimum of an equivalence class at stage A_n will remain the minimum of a class throughout and thus be a minimum of an equivalence class of $\mathbb{N}/\approx$. Therefore, all such minima will necessarily be in the range of the enumeration; slight caution is necessary, because A_1 is already infinite for p.r. functions and enumerating all these classes first would not constitute an exhaustive enumeration of all p.r. functions. Although for our application domain it is relatively straightforward to diagonalise through these infinite sets, in the *general case* it becomes an issue whether the quotient by $\bowtie_i$ splits each equivalence class of A_i into infinitely many new ones — if it does not the diagonalisation could deadlock. In any case, this problem can be avoided:

Theorem 1. *There is a computable canonical enumeration of* $\mathbb{N}/\approx$.

Proof. We can compute finite and finitely sized sets of equivalence classes $B_k = \{0, \ldots, k\}/\equiv_k$ where $t \equiv_k u \iff t \approx_0 u \wedge \cdots \wedge t \approx_k u$. Again, the minimum of each class in each B_k is a minimum of a class in $\mathbb{N}/\approx$.

Using this, the n-th primitive recursive function can be found as follows:

- find the smallest k such that B_k has at least n equivalence classes;
- let p be the number of equivalence classes in B_{k-1};
- the result is the $(n-p)$-th smallest equivalence representative of B_k that was not already an equivalence representative in B_{k-1}.

This construction is guaranteed to find all equivalence classes, because (i) $\mathbb{N}/\approx$ is infinite and thus there is always a B_k of sufficiently large cardinality; (ii) each $\approx$-equivalence class has a minimum, say p; (iii) that value is introduced to the process at B_p, though at that stage not necessarily as the minimum of a class; (iv) there is a minimum value q such that $\neg(n \equiv_q p)$ for all $n < p$; (v) thus p becomes an equivalence class representative at $B_{\max(p,q)}$, and therefore (vi) the number of equivalence classes of $B_{\max(p,q)}$ is an upper bound for when the enumeration would list p, and thus its class. □

The argument also generalises beyond the equivalence between p.r. functions:

Theorem 2. *Any co-semi-decidable equivalence relation $\bowtie$ on $\mathbb{N}$ gives rise to a computable canonical enumeration of $\mathbb{N}/\bowtie$.*

Proof. The construction is similar as for $\approx$ except for two points: (i) the $\bowtie_k$ relations taking the place of $\approx_k$ can be constructed by proposition 2; (ii) in general, there may be no smallest k such that B_k has at least n equivalence classes, because $\mathbb{N}/\bowtie$ may be finite. However, this can be mended by allowing the enumeration f to repeat values: let

$$C_k = \{\min(s) \mid s \in B_k\} \setminus \{f(0), \ldots, f(k-1)\}.$$

Thus C_k is the set of all equivalence representatives in B_k that have not been listed by the enumeration f at smaller inputs. If C_n is non-empty, we define $f(n)$ to be its minimum, otherwise $f(n) = f(n-1)$. □

Remark: what fails to work with the original construction when the relation $\bowtie$ has finitely many equivalence classes, is that the algorithm cannot produce arbitrarily many numbers. In this case, if we know the number of classes then the construction can again be turned into a decision procedure; but if this exact number is not known then at some point the process will have produced all classes without knowing that it has. An example of an equivalence relation of that kind is semantic equality (at sufficiently complex types) in finitary PCF [7].

6 Conclusion

The paper proves the theorem that any co-semi-decidable equivalence relation gives rise to a construction that exhaustively enumerates canonical representatives of all equivalence classes of the relation — in particular allowing to enumerate the Primitive Recursive functions without semantic duplicates.

However, it is impossible to enumerate these representatives in order — unless the equivalence relation in question is decidable.

From a pragmatic point of view the result is not hugely significant, because the enumeration procedure is itself hugely inefficient; still, a pre-computation of some finite segment of the enumeration could be used to compactify the search space for all p.r. functions, for instance for a testing tool. This inefficiency of the enumeration is ultimately inevitable, but some improvements can still be made to accelerate the search for equivalence class representatives, essentially by decelerating the equivalence class refinement; for example, B_k in the proof of theorem 1 could be defined as $\{0, \ldots, 2^k\}/\equiv_k$.

References

1. Wilhelm Ackermann. Zum Hilbertschen Aufbau der reellen Zahlen. *Mathematische Annalen*, 99:118–133, 1928.
2. George Boolos and Richard Jeffrey. *Computability and Logic*. Cambridge University Press, 1974.
3. Richard Dedekind. *Was sind und sollen die Zahlen?* Vieweg, 1888.
4. Stefan Kahrs. Genetic programming with primitive recursion. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 941–942, New York, NY, USA, 2006. ACM Press.
5. Stephen Cole Kleene. Recursive predicates and quantifiers. *Transactions of the American Mathematical Society*, 53:41–73, 1943.
6. Pieter Koopman and Rinus Plasmeijer. Systematic synthesis of functions. In H. Nilsson, editor, *7th Symposion on Trends in Functional Programming*, 2006.
7. Ralph Loader. Finitary PCF is not decidable. *Theoretical Computer Science*, 266(1-2):341–364, 2001.
8. Rózsa Péter. Über den Zusammenhang der verschiedenen Begriffe der rekursiven Funktion. *Mathematische Annalen*, 110:612–632, 1934.
9. Thoralf Skolem. Begründung der elementaren Arithmetik durch die rekurriende Denkweise ohne Anwendung scheinbarer Veränderlichen mit unendlichem Ausdehnungsbereich. In Jean van Heijenoort, editor, *From Frege to Gödel*, pages 302–333. Harvard University Press, 1923.

A Haskell Code

Here is the Haskell code that generates the list of all non-equivalent p.r. programs. This code is not meant to be optimal, just a proof of concept. First, the data type and its semantic interpretation:

```
data Prog = Proj Integer | Z | S |
            Rec Prog Prog | Comp Prog Prog [Prog]
```

The type is used for p.r. functions of all arities. The GADT feature of the GHC compiler would allow to specify the programs with their arity, but such a type is not easy to work with when it comes to enumerate its values. Instead, the interpretation permits to use any values of type `Prog` for any arity, taking a list of integers as input for an interpreted program.

```
interpret :: Prog -> [Integer] -> Integer
interpret Z _ = 0
interpret S [] = 1
interpret S (x:_) = x+1
interpret (Proj n) [] = 0
interpret (Proj 0) (x:_) = x
interpret (Proj n) (_:y) = interpret (Proj (n-1)) y
interpret (Rec f g) (n:xs) =
    | n==0 = interpret f xs
    | otherwise = interpret g (interpret (Rec f g) ys:ys)
      where ys=(n-1):xs
interpret (Rec f g) [] = interpret f []
interpret (Comp f g1 gs) arg =
    interpret f [interpret g arg | g<-g1:gs]
```

For the purpose of defining all p.r. functions of arity 1 it would suffice to limit projections and parallel compositions up to certain arities (3 and 2, respectively), because that would suffice to define Gödelisation functions which can encode (and decode) tuples of numbers as single numbers. Such a restriction would also help to search through the space of all functions more effectively.

However, the priority is here to show the principle. Thus that part of the code is kept simple, using a straightforward 1-to-1 mapping between unbounded integers and the type of unary p.r. functions:

```
enumerate :: Integer -> Prog
enumerate 0 = Z
enumerate 1 = S
enumerate n =
    let m=n-2; k=div m 3; (a,b)=twosplit k; (b1,b2)=twosplit b
    in  case mod m 3 of
            0 -> Proj k
            1 -> Rec (enumerate a)(enumerate b)
            2 -> Comp (enumerate a)(enumerate b1)(enumerateL b2)

enumerateL :: Integer -> [Prog]
enumerateL 0 = []
enumerateL m = enumerate a : enumerateL b
    where (a,b)=twosplit (m-1)

twosplit :: Integer -> (Integer,Integer)
twosplit 0 = (0,0)
twosplit n =
    let (a,b)=twosplit (div n 4); l=mod n 2; r=div (mod n 4) 2
    in  (2*a+l,2*b+r)
```

Instead of operating on equivalence classes of integers, the code operates on equivalence classes of programs, avoiding the indirection through the function

`enumerate`. The approximation equivalence classes are given through the functions `equivalence` and `fullequivalence`, where an expression of the form `equivalence(enumerate k)(enumerate j) n` would correspond to the notation $k \approx_n j$ used in the paper, and `fullequivalence` has a similar relationship to $\equiv_n$.

```
equivalence :: Prog -> Prog -> Integer -> Bool
equivalence m n p = interpret m [p] == interpret n [p]

fullequivalence :: Prog -> Prog -> Integer -> Bool
fullequivalence m n p = all (equivalence m n) [0..p]
```

Now, the inifinite list of all p.r. programs is assigned to the constant `allProg`. The function `develop` has two parameters, the first is the list of equivalence classes B_{n-1} (or rather, their images under `enumerate`), the second the number n. The function operates in two stages: first, `putin` places the program corresponding to n in these equivalence classes w.r.t. equivalence relation $\equiv_{n-1}$; this can at most find one new equivalence class, $[n]$ itself. Secondly, `testfor` refines the so-obtained classes with $\approx_n$; this has two results: the list of all newly-found equivalence class representatives and B_n.

```
allProg :: [Prog]
allProg = develop [] 0

develop :: [[Prog]] -> Integer -> [Prog]
develop classes n =
    [ p | b ] ++ newclasses ++ develop nclasses (n+1)
    where
    p = enumerate n
    (b,iclasses) = putin classes p (n-1)
    (newclasses,nclasses) = testfor n iclasses

putin :: [[Prog]] -> Prog -> Integer -> (Bool,[[Prog]])
putin [] p _ = (True,[[p]])
putin ((xs@(c:_)):xss) p n =
    if fullequivalence p c n then (False,(xs++[p]):xss)
    else let (b,cs)=putin xss p n in (b,xs:cs)

testfor :: Integer -> [[Prog]] -> ([Prog],[[Prog]])
testfor n xss =
    let aux=map (testfor1 n) xss
    in  (concat (map fst aux),concat (map snd aux))

testfor1 :: Integer -> [Prog] -> ([Prog],[[Prog]])
testfor1 n xs =
    let zs = quotient [ (p,interpret p [n]) | p<-xs ]
    in  (map head (drop 1 zs),zs)
```

```
quotient :: Eq a => [(b,a)] -> [[b]]
quotient [] = []
quotient ((x,y):rest) =
    let (a,b)=partition ((==y).snd) rest
    in  (x:map fst a):quotient b
```