CS 311: Computational Structures

James Hook and Tim Sheard

November 26, 2013

8 Exploring Computability

This note presents evendence for the Church-Turing thesis by developing in a little more detail the results showing the the partial recursive functions, as defined in Section 7, are Turing complete.

Informally, the basic building blocks of computability are

1. A syntactic notion of program, where each program can be described as a number, and all programs can be written down as list of numbers.
2. The ability to write down the trace of a computation that can be verified by a series of simple (terminating) steps.
3. Having a large enough set of programs, in particular there needs to be a universal program (also known as an interpreter) that can read a program and its input and generate its output.

For Turing machines we had Turing machine descriptions and computation histories. Note that this is exactly the pattern Sipser uses in all definitions of acceptance for the machine models he presents. We asserted the existance of universal Turing machines, but did not describe in detail how one might be constructed.

For the partial recursive functions we will give a few more details. We have already shown an interpreter written in Haskell for the primitive recursive functions (this is an analogue to a universal program, or interpreter). The extension of this interpreter to partial recursive functions is given as an exercise.

These notes develop all of the mechanisms necessary to encode partial recursive functions as numbers, and exploits this encoding in building programs illustrating key results of computability. We have structured the notes into a main narrative, which is sometimes incomplete, and an appendix, which is sometimes distractingly detailed.

8.1 Pairing Functions

Pairing functions take a pair of natural numbers and encode them as a single natural number. The pairs can be recovered from the original number.

Cantor developed a pairing function that is one-to-one and onto. There is a reasonable article on these pairing functions on Wikipedia (`http://en.wikipedia.org/wiki/Pairing_function`).

The code fragment below implements Cantor's pairing function:

```
pair :: Integer -> Integer -> Integer
pair k1 k2 = ((k1 + k2) * (k1 + k2 +1) `div` 2) + k2
```

The pairs can be deconstructed by this code fragment:

```
unpair :: Integer -> (Integer,Integer)
unpair z = let w = (squareRoot (8*z + 1) - 1) `div` 2
               t = (w * w + w) `div` 2
               y = z - t
               x = w - y
           in (x, y)
```

Some sample values:

	0	1	2	3	4
0	0	2	5	9	14
1	1	4	8	13	19
2	3	7	12	18	25
3	6	11	17	24	32
4	10	16	23	31	40

Note how the numbers *flow* along the diagonals, from lower left to upper right.

Using pairing functions, it is possible to encode other datatypes. For example, the following fragment of Haskell encodes and decodes lists of Integers:

```
eList :: [Integer] -> Integer
eList [] = pair 0 0
eList (x:xs) = pair 1 (pair x (eList xs))

dList :: Integer -> [Integer]
dList l = let (t,c) = unpair l
              (h, tl) = unpair c
          in case t of
             0 -> []
             1 -> h:(dList tl)
             _ -> []  -- make it total; nonsense is nil
```

In this encoding each list element is represented by a pair. If the first element is a `0` then it represents the empty list (nil). If the first element is a `1` then it represents a non-empty list (cons). In that case the second element can be decoded into the head of the list and the tail of the list (which is itself a pair).

For example:

$[]$	$(0,0)$	0
$[2]$	$(1,(2,(0,0)))$	13
$[2,3]$	$(1,(2,(1,(3,(0,0)))))$	246751
$[2,3,4]$	$(1,(2,(1,(3,(1,(4,(0,0)))))))$	$945239141275481237930403376$

In a similar manner the encoding can be extended to any datatype. Consider the datatype `PrimRec` to describe the primitive recursive programs described in Appendix A.1.

```
data PrimRec
        = Z
        | S
        | P Int
        | C PrimRec [PrimRec]
        | PR PrimRec PrimRec
```

The primitive recursive functions can be encoded and decoded with the following fragment. Note there is one rule for each of the 5 ways we can form an element of `PrimRec`.

```
ePR :: PrimRec -> Integer
ePR Z = pair 0 0
ePR S = pair 1 0
ePR (P i) = pair 2 (toInteger i)
ePR (C f gs) = pair 3 (pair (ePR f) (eList (map ePR gs)))
ePR (PR g h) = pair 4 (pair (ePR g) (ePR h))

dPR x = let (t,b) = unpair x
            (b1,b2) = unpair b -- note:  Lazy
        in case t of
           0 -> Z
           1 -> S
           2 -> P (fromInteger b)
           3 -> C (dPR b1) (map dPR (dList b2))
           4 -> PR (dPR b1) (dPR b2)
           _ -> Z
```

Using this encoding, called a Gödel numbering, we can see that the Gödel number of the plus function (PR P_1 (C S [P_2])) is

```
*Goedel> ePR $ PR (P 1) (C S [P 2])
4511739842654672905730185440573223378237806974280320
*Goedel> dPR $ 4511739842654672905730185440573223378237806974280320
PR (P 1) (C S [P 2])
```

This example uses the $ notation from Haskell, which stands for an open parenthesis that extends as far to the right as possible, Thus `f $ x + 1` means the same thing as `f (x + 1)`. Contrast this with `f x + 1`, which means `(f (x)) + 1`. While this idiom at first seems unusual, in many cases it will remove quite a bit of syntactic clutter from programs.

8.2 Why we need Partial Functions

How hard is it to define a function that is not in the set of Primitive Recursive Functions?

In this section we will write a Haskell function that is total, computable, but not primitive recursive. We will build it by using a technique called diagonalization. The concept of diagonalization was introduced by Cantor to show that the reals and the natural numbers have different cardinalities, that is, there is not a one-to-one correspondence between them.

In Cantor's construction, a countable list of decimal expansions of real numbers is assumed. From that list a procedure is defined to build a number that is different from all in the list. This number is built by taking the diagonal of the list—building a number that differs from the first in the first position, from the second in the second position, and so on.

How do we diagonalize over the primitive recursive functions? We start by building a table that lists all such functions and their values:

		The program (`dPr` x) applied to										
x	`dPR` x	0	1	2	3	4	5	6	7	8	9	10	$\cdots$
0	Z	0	0	0	0	0	0	0	0	0	0	0	
1	S	1	2	3	4	5	6	7	8	9	10	11	
2	Z	0	0	0	0	0	0	0	0	0	0	0	
3	P_1	0	1	2	3	4	5	6	7	8	9	10	
4	S	1	2	3	4	5	6	7	8	9	10	11	
5	Z	0	0	0	0	0	0	0	0	0	0	0	
6	$C\ Z$ []	0	0	0	0	0	0	0	0	0	0	0	
7	P_1	0	1	2	3	4	5	6	7	8	9	10	
8	S	1	2	3	4	5	6	7	8	9	10	11	
9	Z	0	0	0	0	0	0	0	0	0	0	0	
10	PR $Z\ Z$	0	0	0	0	0	0	0	0	0	0	0	

We want to compute a function that differs from all functions in the list, so we construct a function that adds one to the result found on the diagonal above. That is, where the diagonal above has as its first 10 values:

$$0, 2, 0, 3, 5, 0, 0, 7, 9, 0, 0$$

We wish to define a function that instead returns:

$$1, 3, 1, 4, 6, 1, 1, 8, 10, 1, 1$$

Such a function can be expressed in Haskell as

```
diagonal x = (eval p (ncopies (arity p) x))
   where p = dPR x

notdiagonal x = 1 + diagonal x
```

The `diagonal` function takes one natural number argument, x. It interprets it as a primitive recursive function, p. It then evaluates that function (applied to its own encoding x, repeated as many times as necessary to match the arity of the function). The function `notdiagonal` differs from diagonal at every point, since it is always defined to be one greater.

The Haskell function `notdiagonal` is a total computable function. It is defined on all natural numbers.

The function `notdiagonal` is different from any primitive recursive function. Why? Suppose it was primitive recursive. Then there would be some program, p in the language of primitive recursive functions, and that function would have a number, n, such that $n =$ `ePR` p. But $p(n) = z$ for some number z, and `notdiagonal` $n = 1 + z$. Thus `notdiagonal` $\neq p$. Having obtained a contradiction we conclude that `notdiagonal` is not a primitive recursive function.

What facts about primitive recursive functions did we use in this argument? Not very many. We relied on the property that primitive recursive functions are total. We relied on the existence of the `eval` function. And we relied on the existence of a function from numbers to programs. This situation is called an *effective enumeration.*

Definition 8.1 *An* effective enumeration *of a set of functions is a mapping of the natural numbers onto the set of functions, $\phi_0, \phi_1, \ldots$, together with a computable function* `eval` *such that* `eval` $i\ x = \phi_i(x)$.

Fact 8.2 *The primitive recursive functions can be effectively enumerated. The function* `evalGoedel` *is an effective enumeration of the class of primitive recursive functions respect to the* `ePR`/`dPR` *encoding.*

```
evalGoedel i x = eval p (take (arity p) (repeat x))
    where p = dPR i
```

Theorem 8.3 *Any effective enumeration of a set of total functions is incomplete. That is, there are total computable functions that are not included in the enumeration.*

The proof is simply a generalization of the argument given above that there is a total computable function that is not primitive recursive.

Corollary 8.4 *There is no effective enumeration of the set of total computable functions.*

Thus any theory of the computable functions that includes all total computable functions must also include partial functions.

Problem 8.1 *How does the possibility that* `eval` *might not terminate change the argument that effective enumerations are incomplete? (This is probably too unstructured to assign as a problem, but is an important question to consider.)*

8.3 Pairing is Primitive Recursive

Recall the definitions for describing primitive recursive functions given in Appendix A.1. To implement pairing with the Primitive Recursive functions, we need to come up with descriptions in `PrimRec` sufficient to encode the definitions from Section 8.1. The basic arithmetic and logical operations you implemented in homework are a good start, but need to be extended. Some of the operations from the homework are given here:

```
-- Elementary arithmetic
mkconst 0 = Z
mkconst n = C S [mkconst (n-1)]

one    = mkconst 1
plus   = PR (P 1) (C S [P 2])
times  = PR Z (C plus [P 2, P 3])
pred   = PR Z (P 1)
monus' = PR (P 1) (C pred [P 2])
monus  = C monus' [P 2, P 1]


-- Elementary logic
false = Z
true  = one
and   = PR false (P 3)
or    = PR (P 1) true
not   = PR true false
ifte  = PR (P 2) (P 3)

-- Predicates
iszero  = PR true false
nonzero = PR false true
equal   = C and [C iszero [monus],
                 C iszero [monus']]
lteq    = C iszero [monus]
lt      = C nonzero [monus']
```

To extend this list, it is very handy to develop a programming idiom that supports *bounded* search for values that satisfy a predicate. This allows us, for example, to write a naive algorithms for division and square root that seek values satisfying the specifications of these functions. For example

```
div x y = {find the smallest z | (z == x) || ((y*z <= x) && (x < y*(z+1)))}

sqrt x = {find the smallest z | (z == x) || ((z*z <= x) && (x < (z+1)*(z+1)))}
```

Note that the search is always bounded by some number (here it is x in both definitions) that is an input to the the search.

The Appendix A.2 contains a development of this bounded search technique, and applies it to define the functions `div` and `sqrt`. All the gory details are fully explained in Appendix A.2.

With these functions defined, we can now give primitive recursive definitions that implement Cantor's pairing function. The function `pair` constructs pairs.

Here are our definitions from Section 8.1 as Haskell functions.

```
pair :: Integer -> Integer -> Integer
pair k1 k2 = ((k1 + k2) * (k1 + k2 +1) `div` 2) + k2
```

The pairs can be deconstructed by this code fragment:

```
unpair :: Integer -> (Integer,Integer)
unpair z = let w = (squareRoot (8*z + 1) - 1) `div` 2
               t = (w * w + w) `div` 2
               y = z - t
               x = w - y
           in (x, y)
```

Here are the same definitions described as primitive recursive functions.

```
-- Cantor Pairing Function and Projections
pair = C plus [C div [C times [C plus [P 1, P 2],
                                C S [C plus [P 1, P 2]]],
                      mkconst 2],
               P 2]

w = C div [C pred [C sqrt [C S [C times [mkconst 8, P 1]]]],
           mkconst 2]
t = C div [C plus [C times [w,w],w], mkconst 2]
pi2 = C monus [P 1,t]
pi1 = C monus [w,pi2]
```

The function `pi1` is the first (left) projection. The function `pi2` is the second (right) projection and together they describe the `unpair` Haskell function.

At this point we have demonstrated that primitive recursive functions are sufficiently powerful to express functions that allow data structures like lists, trees, and program abstract syntax, to be encoded in the natural numbers.

In particular, the primitive recursive pairing functions that can encode the descriptions of Turing machines, inputs to Turing machines, configurations, and computation histories.

We will assert without demonstration that, using similar techniques, we can define primitive recursive functions that test properties of lists, trees, and programs. However, henceforth we will demonstrate such ideas with programming languages, such as Haskell, Scheme, or C.

8.4 Partial Recursive Functions are Turing Complete

We will briefly return to a point made at the end of Section 7. The partial recursive functions are Turing complete.

We review the evidence supporting this claim.

1. We can represent Turing machine descriptions as numbers using primitive recursive pairing functions.
2. We can represent Turing machine configurations and computation histories as numbers using pairing functions.
3. We can write a primitive recursive predicate, known as the Kleene T predicate, that takes the description of a Turing machine, e, the input to the Turing machine, x, and a computation history of the Turing machine, y, and returns 1 if y is a halting computation history of machine e on input x, and 0 otherwise. NOTE, the function T tests if the given computation history y actually describes a run of e on input x. It does not compute the computation history y. Such a function would be much harder to write.
4. We can write a primitive recursive function, U, that takes a halting computation history y and returns the final output of the computation.
5. Given a Turing machine e and input x, we can use unbounded search to define a partial recursive function of e and x that returns the least y such that $T(e, x, y)$ holds. That function can be composed with U to yield a partial function θ such that $\theta(e, x) = z$ if and only if the Turing machine e on input x computes value z. Because the search is *unbounded* is the reason we need the *partial* recursive functions and not the *primitive* recursive functions.

The last three steps of this argument are asserted without proof.

8.5 Trace Summary

Consider the following descriptions of traces for various systems.

- **DFA.** A DFA $M = (Q, \Sigma, \delta, q_0, F)$ accepts a string $w_1 w_2 \ldots w_n$ if there exists a sequence of states $r_0 r_1 \ldots q_n$ if

  1. $r_0 = q_0$
  2. $r_{i+1} = \delta(r_i, x_{i+1})$
  3. $r_n \in F$

  Given such a DFA, a string, and a sequence of states, it is easy to verify whether or not the string is accepted.

- **PDA.** A PDA is defined as a 6-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$. PDA M accepts a string $w = w_1 w_2 \ldots w_n$ by final state if there exists a sequence of states and stacks, $r_0, r_1, \ldots, r_n$ and $s_0, s_1, \ldots, s_n$ such that:

1. $r_0 = q_0$ and $s_0 = \epsilon$
2. $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$ and there exists a t such that $s_{i+1} = bt$ and $s_i = at$.
3. $r_n \in F$

- **Turing Machine.** A Turing machine is defined as a 7-tuple: $M = (Q, \Sigma, \Gamma, \delta, q_0, q_a, q_r)$. A configuration for a TM is a string of the form $\alpha q \beta$ where

  1. $\alpha \in \Gamma^*$
  2. $\beta \in \Gamma^*$
  3. $q \in Q$
  4. The string α represents the non-blank tape contents to the left of the head.
  5. The string β represents the non-blank tape contents to the right of the head, including the currently scanned cell.

  Two configurations are related by $\vdash$ if:

  1. if $\delta(q, a) = (p, b, \mathbf{R})$. then $uqav \vdash ubpv$
  2. if $\delta(q, a) = (p, b, \mathbf{L})$. then $ucqav \vdash upcbv$

  A turing machine accepts a string w if there exists a sequence of configurations $c_0, c_m, \ldots, c_n$ such that.

  1. $c_0 = q_0\ w$
  2. $c_i \vdash c_{i+1}$
  3. $c_n = \alpha\ q_a\ \beta$

  Given a chain of related configurations, it is easy to verify that a string is accepted.

- **Primitive Recursive Functions.** A primitive recursive function p applied to arguments $(x_1, \ldots, x_n)$ computes a value n, if there exists a trace tree t that corresponds to the computation that p applied to $(x_1, \ldots, x_n)$ computes n. A trace tree is a tree of nodes where each node is labelled with a triple (program, input, result), called its *tag*. Subtrees of a trace tree coorespond to subcomputations for the compound nodes labeled with `C` and `PR` programs (the only frm of `PrimRec` that have sub-programs). A well formed tree has subtress which correspond to sub-computations. Just as we can write a program that checks if configurations for Turing machines are related by the its transition function, we can check that parent and child nodes in trace trees are related by the rules of computation for `PrimRec` given by `eval`. Given a trace tree where all the parent-child nodes are related we can write a total function that checks if a program, its input, and a given trace-tree are consistent. (Trace trees are presented in the appendix, Section A.3.)

Conclusion: All the computational systems we have studied can be verified by simple terminating programs, given the appropriate trace object. It may be very expensive to find a trace, but given a trace it is easy to check that if the trace really describes the computation. In addition the strategy we showed for the primitive recursive functions easily extends to traces for the partial recursive functions.

Problem 8.2 *Consider the context free grammars. Outline what a trace for a context free grammar might look like. Define a simple CFG, and provide a trace in the form you described above for a short string.*

8.6 Limits of Computability

The first step in exploring the limits of computability is to understand how non-termination can be handled. We start by taking two concepts we have developed above, evaluation and verification, and observing that even when evaluation is partial we can make the corresponding verification total, provided we have a certificate justifying the calculation.

In appendix Section A.3 we show explicitly how to construct a verifier for the primitive recursive functions. We left as an exercise the verification of the partial recursive functions (Problems A.4, A.5). We have described in Section 8.4 how to verify computation histories for Turing machines.

In this section we assume that the partial recursive functions have been implemented and are represented by the datatype `MuR` (for mu-recursive, or μ-recursive).

```
data MuR = Z
         | S
         | P Int
         | C MuR [MuR]
         | PR MuR MuR
         | Mu MuR

eval :: MuR -> [Integer] -> Integer
eval Z _ = 0
eval S (x:_) = x+1
eval S _  = 0 -- relaxed
eval (P n) xs = nth n xs
eval (C f gs) xs = eval f (map (\g -> eval g xs) gs)
eval (PR g h) (0:xs) = eval g xs
eval (PR g h) (x:xs) = eval h ((x-1) : eval (PR g h) ((x-1):xs) : xs)
eval (PR _ _) [] = 0  -- relaxed
eval (Mu f) xs = try_from f xs 0

try_from f xs n = if eval f (n:xs) == 0 then n else try_from f xs (n+1)
```

The `MuR` type has the same constructors as `PR`, plus the additional constructor `Mu`. As before we have a function `eval` that takes a `MuR` and a list of `Integer` arguments and returns an `Integer`. However, now `eval` is a partial function. For example, the `MuR` program `Mu S` is undefined on all values. The `eval` function goes into an infinite loop when given this program. We can see the possibility of non-termination from the function `try_from` which is recursive and clearly does not have an argument that decreases on every recursive call.

Furthermore, we assume that we have developed a tracing evaluator, `trace` which given a program and its input might return a trace (it might also run forever) for the `MuR` functions, and a trace verifier, `verify`, (which given a program, its input, and a trace) determines if the trace is valid. They are very similar to the analogous functions on `PR` given in the Appendix A.3, except that, like `eval`, the trace generator can run forever. Of course the verifier can only verify finite traces, but the verifier halts on every program, input, and trace with an answer.

It is convenient to repackage `verify` as the following function `valid`, which captures the goal that it is verifying:

```
valid program input result trace =
      (verify program input trace) &&
      (last trace == result)
```

The first observation we make is that if the k-ary `MuR` function f applied to the inputs $(n_1, \ldots, n_k)$ is defined and equal to w, then `eval` of $f(n_1, \ldots, n_k)$ returns w and `trace` applied to f and the inputs $(n_1, \ldots, n_k)$ will return a trace c which we will call a certificate, and `verify f` $(n_1, \ldots, n_k)$ c will return `True`. The construction of the functions `trace` and `valid` can be seen as a proof of this statement.

The next observation is that if there is a certificate that validates $f(n_1, \ldots, n_k) = w$ then $f(n_1, \ldots, n_k)$ is indeed defined and equal to w. These observations are combined in the theorem:

Theorem 8.5 (`eval` f $(n_1, \ldots, n_k)) = w$ *if and only if there exists a certificate* c *such that* (`valid` f $(n_1, \ldots, n_k)$ w c)*.*

Note that this theorem relates the partial function `eval` to the total function `valid`.

8.6.1 Halting

In Section 8.2, we used the technique of diagonalization to construct a total function that was not primitive recursive. In this section we will use a similar technique to prove that there is not a total `MuR` function `halt` with the specification that `halt` f $(n_1, \ldots, n_k)$ if and only if $f(n_1, \ldots, n_k)$ is defined.

Suppose there were such a function. Then we could construct the function:

```
notdiagonal x = if (halt (dMuR x) (repeat x)) then loop else 0

loop = loop    -- a value that loops
```

If we apply `notdiagonal` to the code for a value that loops when given its own description as input, such as `Mu S`, we should get a function that terminates and returns 0. On the other hand, if we apply `notdiagonal` to a program that returns a value when applied to its own description, then the diagonal function will loop.

Since by supposition `halt` is a partial recursive function, the function `notdiagonal` is also partial recursive. Since it is a partial recursive function, there is a formal description of it in the type `MuR`. Call that description p. Furthermore, p can be encoded by a natural number, n using the `eMuR` function. That is there is an n such that $n = \texttt{eMuR}\, p$.

Consider `notdiagonal` n.

Elaborating the definition gives:

```
if (halt (dMuR n) (repeat n)) then loop else 0
```

Since we know n to represent p, and know p is a one place function, this can be rewritten as:

```
if (halt p [n]) then loop else 0
```

We proceed by cases on `halt p [n]`:

- `halt p [n] = true`

  In this case `notdiagonal n` simplifies to `loop`. But that contradicts that $p =$ `notdiagonal` since p halts on n while `notdiagonal` loops.

- `halt p [n] = false`

  In this case `notdiagonal n` returns 0. This also contradicts $p =$ `notdiagonal` since p does not halt on n while `notdiagonal` returns the value 0.

Having obtained contradictions in both cases, we conclude that there is no partial recursive function equivalent to `halt`, as we had assumed.

This proves that the halting problem is not computable:

Theorem 8.6 *There is no total computable function that computes* `halt`*.*

The halting problem can be formulated for all reasonable Turing complete models of computation.

9 Consequences of the Halting Problem

So what do we know exactly about the Halting problem?

We know that if a partial recursive function p is defined on input x then the `eval` function will terminate and find the answer. That means we can algorithmically say "yes" to the question "does p halt on x?" in all cases where the answer is "yes".

However, we know that there is no total function that decides the halting problem.

Are these statements in conflict? Not exactly.

We will classify how hard a problem is by how well a computable function can answer a language membership question. There are three cases we look at today: (1) we can definitively say yes or no to any question is $x \in A$ with a total computable function, (2) we can definitively say yes if $x \in A$, but in some cases where $x \notin A$ the function is undefined, and (3) none of the above: there are both $x \in A$ and $y \notin A$ on which the function in question is undefined.

In the first case we will say that A is *decidable.* There is a total function p that says yes or no to all questions about membership in A. ($p\ x \leftrightarrow x \in A$)

In the second case we will say that A is *recognizable.* That is, there is a partial recursive function that accepts all elements of A. ($x \in A \rightarrow p\ x$)

The third case we will refer to as *not recognizable.*

In the context of this vocabulary, we will say that the set of pairs (p, x) such that p halts on input x is recognizable but not decidable. Note this is exactly analogous to Siper's acceptance problem A_{TM}. The definitions used throughout this section correspond to those of similarly named Turing machine concepts in Sipser.

In practical terms, decidability is a strongly desirable property. If a problem is not decidable then any total algorithmic solution will be approximate. That is, if it always generates an answer it will get some answers wrong!

Do undecidable problems arise in practice? The answer to this is yes, particularly in the context of properties of programs. There is a general theorem, known as Rice's theorem, that says that any non-trivial property of program behaviors is undecidable. That is, if ϕ is a nontrivial property of programs, and we want $\phi(p)$ to imply $\phi(q)$ whenever p and q behave the same on all inputs (i.e. $\forall x.p(x) = q(x)$), then ϕ is undecidable.

A Appendix: Details of the Construction

A.1 Primitive Recursive Functions as Haskell Datatype

In the homework and in class we have described the primitive recursive functions as a language. Below we give an extended[1] Context Free Grammar for such a language.

```
Term  -> Z
       | S
       | P Index                       nth projection
       | C Term  [ Term (, Term)* ]    composition
       | PR  Term  Term                primitive recursion
       | ( Term )                      grouping
Index -> 1 | 2 | 3 | ...
```

[1]We extend the notation of context free grammar using the Kleene-star operator and the ... notion to denote the infinite set if strings denoting the postive numbers.

Well-formed strings in this language denote primitive recursive functions. In this note we describe how we might compute over such representations. Our notation of choice is the Haskell programming language. In this section we give a small Haskell program that introduces data structures to describe terms that denote primitive recursive functions, and several functions over these data structures. First the description of data.

```
data PrimRec
        = Z
        | S
        | P Int
        | C PrimRec [PrimRec]
        | PR PrimRec PrimRec
```

This Haskell declaration introduces a new datatype called `PrimRec`. In essence it defines an inductively defined set, which we should be familiar with. Elements in this set include `Z`, `(PR S Z)`, etc. Note that there are 5 rules for defining elements of `PrimRec`. As in most inductively defined sets, some elements incorporate smaller elements inside. In `PrimRec` those elements starting with `C` and `PR` incorporate other elements of the set.

Note that the grammar to describe strings was *designed* so that strings in that grammar would be well-formed Haskell programs as well as descriptions of the primitive recursive programs.

Functions over elements of `PrimRec` will have 5 rules, one for each of the ways to form an element of `PrimRec`. The denotation of the elements of `PrimRec` are total functions of arity n that consume and produce integers. We can write a function that computes these denotations. We call this function `eval`. It takes a `PrimRec` and a list of integers (the functions inputs), and produces an integer.

```
eval :: PrimRec -> [Integer] -> Integer
eval Z _     = 0
eval S (x:_) = x+1
eval S _     = 0 -- default value for erroneous case
eval (P n) xs | n <= length xs = nth n xs
eval (P n) xs = 0 -- default value for erroneous case
eval (C f gs) xs = eval f (map (\g -> eval g xs) gs)
eval (PR g h) (x:xs) =
   if x==0 then eval g xs
           else eval h ((x-1) : eval (PR g h) ((x-1):xs) : xs)
eval (PR _ _) [] = 0  -- default value for erroneous case

nth _ []    = 0 -- default value for erroneous case
nth 0 _     = 0 -- default value for erroneous case
nth 1 (x:_) = x
nth (n) (_:xs) = nth (n-1) xs
```

The function `eval` is total. It returns some integer for every `PrimRec` and every list of inputs. Some of these these are not compatible, on these cases `eval` returns 0. There are 5 compatible cases (one for each of the ways a PrimRec can be formed.

Here is a sample execution from my Haskell session.

```
*Examples> plus
PR (P 1) (C S [P 2])
*Examples> eval plus [3,4]
7
```

Note: This is a slight variation on the version of the code presented in the previous note. The key differences are:

1. The datatype PR has been renamed PrimRec to avoid confusing the datatype (now PrimRec) with the constructor (still PR).
2. Several cases that previously generated run-time errors are now defined. This makes eval total on PrimRec. The previous version of eval was a partial function; it would throw an exception in some cases.

A.2 Bounded minimization is Primitive Recursive

Mathematically, if $P(x_1, x_2, \ldots, x_k)$ is a predicate (returns 0 if false, 1 if true), then

$$\mu x < n.P(x, x_2, \ldots, x_k)$$

returns the least x such that $P(x, x_2, \ldots, x_k) \neq 0$, provided there is such an x less than n. If no such element exists it returns a value $x \geq n$. Note that this is very similar to the unbounded search capability in rule VI, in Section 7.6. Adding the bound significantly restricts its expressive power, and makes it possible to define search within the Primitive Recursive functions.

Bounded minimization gets implemented as a k-place primitive recursive function that takes the bound as its first argument. That function is generated by the Haskell function bmin, which takes the predicate P and its arity as arguments. For example, the following interaction reports the least integer less than 2 that satisfies the true predicate and the least integer less than 2 that satisfies the false predicate:

```
*PRCantor> eval (bmin true 1) [2]
0
*PRCantor> eval (bmin false 1) [2]
2
```

The Haskell function that generates instances of bmin is given below. First, the "model" code of a Haskell function that behaves like the primitive recursive function is given. Then the code that actually generates the primitive recursive predicate is given.

```
-- Bounded minimization (search)
bmin_model p (0:xs) = 1
bmin_model p (x:xs) = let r = (bmin_model p (x-1:xs))
                      in if r < x then r
                         else if p (x-1:xs) then x-1 else x

bmin p n = PR one
              (C ifte [C lt [P 2, P 1],
                       P 2,
                       C ifte [C p (P 1: take (n-1) (map (\j -> P j) [3..])),
                               P 1,
                               C S [P 1]]])
```

In this code, the expression `[3..]` indicates the infinite list of numbers starting at 3. These numbers are turned into instances of the projection function by the map function. Thus this expression:

```
map (\j -> P j) [3..])
```

stands for the infinite list of projection functions, `[P 3, P 4, ... ]`. The function `take` only requires the first `n-1` elements of this list. Haskell uses a technique called lazy evaluation that makes it possible to write functions that manipulate potentially infinite objects, as long as they ultimately only demand finitely many values.

For example, here is a hand formatted version of the output of a simple predicate generate by `bmin`:

```
*PRCantor> bmin false 1
PR (C S [Z])
   (C (PR (P 2) (P 3))
      [C (C (PR Z (C S [Z]))
            [PR (P 1) (C (PR Z (P 1)) [P 2])])
         [P 2,P 1],
       P 2,
       C (PR (P 2) (P 3))
         [C Z [P 1],P 1,C S [P 1]]])
```

Which can be simplified to:

```
*PRCantor> bmin false 1
PR one
   (C ifte
      [C lt
         [P 2,P 1],
       P 2,
       C ifte
         [C false [P 1],
          P 1,
          C S [P 1]]])
```

Once bounded minimization is available, it is possible to write naive search-based algorithms to find values easily characterized by predicates. These algorithms tend to be inefficient, but they do demonstrate that the functions can be computed by functions in the class.

For example, we can implement division of natural number a by natural number b as a search for the least q such that $q * b \leq a$ and $a < (q+1) * b$. This condition is coded up as the three place predicate `isdiv` that expects arguments q, a and b:

```
isdiv = C and [ C lteq [C times [P 1, P 3], P 2],
                C lt   [P 2, C times [C S [P 1], P 3]]]
```

Using this predicate and bounded minimization we can implement natural number division as follows:

```
div = C (bmin isdiv 3) [C S [P 1], P 1, P 2]
```

In a similar manner, we can implement a very inefficient square root algorithm:

```
issqrt = C and [ C lteq [C times [P 1, P 1],P 2],
                 C lt   [P 2, C times [C S [P 1], C S [P 1]]]]
sqrt = C (bmin issqrt 2) [P 1, P 1]
```

With these functions defined, we can now give primitive recursive definitions that implement Cantor's pairing function. The function `pair` constructs pairs. The function `pi1` is the first (left) projection. The function `pi2` is the second (right) projection.

```
-- Cantor Pairing Function and Projections
pair = C plus [C div [C times [C plus [P 1, P 2],
                               C S [C plus [P 1, P 2]]],
                      mkconst 2],
               P 2]

w = C div [C pred [C sqrt [C S [C times [mkconst 8, P 1]]]],
           mkconst 2]
t = C div [C plus [C times [w,w],w], mkconst 2]
pi2 = C monus [P 1,t]
pi1 = C monus [w,pi2]
```

At this point we have demonstrated that primitive recursive functions are sufficiently powerful to express functions that allow data structures like lists, trees, and program abstract syntax, to be encoded in the natural numbers.

In particular, the primitive recursive pairing functions that can encode the descriptions of Turing machines, inputs to Turing machines, configurations, and computation histories.

We will assert without demonstration that, using similar techniques, we can define primitive recursive functions that test properties of lists, trees, and programs. However, henceforth we will demonstrate such ideas with programming languages, such as Haskell, Scheme, or C.

Problem A.1 *Given a 3-place primitive recursive function f, write an equivalent 1-place function f' that takes as input a single number representing the triple as nested pairs and behaves as f. That is, $f'(\texttt{pair}\ x_1\ (\texttt{pair}\ x_2\ x_3)) = f(x_1, x_2, x_3)$.*

Discussion The diagonalization presented here generalizes well to all effective enumerations of total functions, but it does not reflect much of the essence of the class of primitive recursive functions. Ackermann developed a function that diagonalizes over the primitive recursive functions in a different way. He constructs a function that grows faster than any primitive recursive functions.[2]

Ackermann's diagonalization is in some ways more satisfying than the one presented above, but is specific to the primitive recursive functions.

A.3 Certifying Computation with Traces

One of the characteristics of a computational system is that it proceeds by a series of easily verifiable steps. For a given system, a set of these verifiable steps is called a trace. For the finite automata we only needed to check that the state transitions were appropriate for the input. For push down automata we had to additionally manage the stack contents. For Turing machines we introduced configurations and verified that one followed from another according to the definition of the machine.

How can we certify a computation for the primitive recursive or partial recursive functions? Each step is a very simple calculation. In this section we present a simple record of that calculation as a tree. The root node of the tree is at the bottom of the picture. The children of each node are those lines above that node that are indented exactly 3 spaces. For example, consider the adding 2 and 3 (e.g. $plus(2,3) = \mathrm{PR}(P\ 1)(C\ S\ [P\ 2])\ (2,3))$. That calculation is described by the trace tree in the picture below.

[2]Hein discusses Ackermann's function in Section 13.2.3. He presents a proof sketch. If you wish to elaborate the proof sketch, consider calculating the number n based on the nesting of PR operations in the primitive recursive function.

```
10             S (4) = 5
9              P 2 (1,4,3) = 4
8           C S [P 2] (1,4,3) = 5
7                 S (3) = 4
6                 P 2 (0,3,3) = 3
5              C S [P 2] (0,3,3) = 4
4                 P 1 (3) = 3
3              PR (P 1) (C S [P 2]) (0,3) = 3
2           PR (P 1) (C S [P 2]) (1,3) = 4
1        PR (P 1) (C S [P 2]) (2,3) = 5
```

Each line represents an easily verified calculation step. Line 1 asserts that the root of the computation ($add(2,3)$) is equal to 5. To certify this we must verify the assertions on lines 2 and 8, which are the immediate sub-trees of the root (note the indentation). These two subtrees follow directly from the way a `PR` node is evaluated. Study the case for primitive recursion on non-zero inputs (the second line below) from Section **??**

$$
\begin{aligned}
f(0, x_1, \ldots, x_k) &= h(x_1, \ldots, x_k) \\
f(n+1, x_1, \ldots, x_k) &= g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k)
\end{aligned}
$$

The first subtree `PR (P 1) (C S [P 2]) (1,3) = 4` (line 2) follows from the computation $f(n, x_1, \ldots, x_k)$, and the second subtree `C S [P 2] (1,4,3) = 5` (line 8) follows from the computation $g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k)$ where the sub term $f(n, x_1, \ldots, x_k)$ has been replaced by its value (4) to get $g(n, 4, x_1, \ldots, x_k)$.

The subtree on line 2 is also a `PR` term so it has subtrees on lines 3 and 5. This pattern repeats until a node without subterms (`Z`, `S`, or `(P n)`) is reached, in which case we have a simple assertion (lines 4, 6, 7, 9, and 10).

What rules are we using to verify the calculations? For the atomic primitive recursive functions Z, S and P_i, we directly verify that the result is obtained from the arguments correctly.

For the composition form $(C\ f\ [g_1, \ldots, g_k])(x_1, \ldots, x_n) = z$, we need to verify that the tree has the shape:

$$
\begin{array}{rl}
 & f(y_1, \ldots, y_k) \ = \ z \\
 & g_k(x_1, \ldots, x_n) \ = \ y_1 \\
 & \vdots \\
 & g_1(x_1, \ldots, x_n) \ = \ y_k \\
C\ f & [g_1, \ldots, g_k])(x_1, \ldots, x_n) \ = \ z
\end{array}
$$

Where each `C` node has $k+1$ subtrees Note that the subtrees beginning on lines 5 and 8 conform to this pattern.

For the primitive recursion form we have two cases: one for when the first argument is zero and one for when the first argument is nonzero. We consider them in order.

In the zero case we have the form (PR f g)$(0, x_2 \ldots, x_k) = z$. In this case we verify we have the tree has shape

$$\begin{array}{rcl} f(x_2, \ldots, x_n) & = & z \\ (\text{PR} \quad f\ g)\ (0, x_2 \ldots, x_k) & = & z \end{array}$$

Where the `PR` node has one subtree. This occurs in line 3 of the example.

For the nonzero case we have the form (PR f g)$(x+1, x_2 \ldots, x_k) = z$. In this case we need to verify that the tree has the shape

$$\begin{array}{rcl} g(x, r, x_2, \ldots, x_n) & = & z \\ (\text{PR}\ f\ g)(x, x_2 \ldots, x_k) & = & r \\ (\text{PR} \quad f\ g)(x+1, x_2 \ldots, x_k) & = & z \end{array}$$

Where the `PR` node has exactly two subtrees. This pattern occurs in line 1 and 2 of the example.

Exercise A.2 *Test your understanding of the example by showing the explicit correspondence between these rules and the example. Label each part of the example with the corresponding variables in the three patterns presented above.*

Exercise A.3 *Develop trace trees for the following computations:*

1. `(PR (C S [Z]) Z) [0]`
2. `(PR (C S [Z]) Z) [1]`
3. `(PR Z (C S [Z])) [0]`
4. `(PR Z (C S [Z])) [1]`
5. `(PR (P 2) (P 3)) [1,2,3]`

Problem A.4 *How can we verify a step of unbounded search? Develop verification rules for rule VI.*

A.3.1 Haskell code to generate and verify traces (optional)

To demonstrate that this is all directly computable we implement a version of trace trees in Haskell. We represent a step as a triple consisting of the function, the arguments, and the result. We represent this by the Haskell type `(PR,[Integer],Integer)`. A trace tree will store such a triple at every node in the tree. A trace tree node is *good* if every stored triple is consistent with the sub-trees of that node. A trace tree root justifies the final calculation in the trace. So, the example justifies that `plus` of 2 and 3 is 5.

First we give the Haskell code to define a trace tree.

```
data TraceTree x = Ztree x
         | Stree x
         | Ptree Int x
         | Ctree (TraceTree x) [TraceTree x] x
         | PRtree0 (TraceTree x) x
         | PRtreeN (TraceTree x) (TraceTree x) x
         | Stuck x
         deriving (Show, Eq)

tag (Ztree x) = x
tag (Stree x) = x
tag (Ptree n x) = x
tag (Ctree x xs y) = y
tag (PRtree0 y z) = z
tag (PRtreeN x y z) = z
tag (Stuck z) = z

res x = z where (_,_,z) = tag x
```

Note that each node type is tagged with a polymorphic component of type `x`. We can select this tag using the `tag` function. In our code the tag is always a triple `(PR,[Integer],Integer)`, so the function `res` selects the final result from a node.

The checking function is similar in its structure to the `eval` function, in that it analyzes the structure of the primitive recursive function.

```
trace2:: PR -> [Integer] -> TraceTree (PR,[Integer],Integer)
trace2 Z xs = Ztree (Z,xs,0)
trace2 S (x:xs) = Stree (S,(x:xs),x+1)
trace2 S [] = Stuck (S,[],0)
trace2 (P n) xs | n <= length xs = Ptree n (P n,xs,nth n xs)
trace2 (P n) xs = Stuck (P n,xs,0)
trace2 (C f gs) xs = let gtraces = map (\g -> trace2 g xs) gs
                         ys = map res gtraces
                         ftrace = trace2 f ys
                     in Ctree ftrace gtraces (C f gs,xs,res ftrace)
trace2 (PR g h) (0:xs) =
   let trace = trace2 g xs
   in PRtree0 trace (PR g h,0:xs,res trace)
trace2 (PR g h) (x:xs) =
   let rectrace = trace2 (PR g h) ((x-1):xs)
       r = res rectrace
       htrace = trace2 h ((x-1):r:xs)
       r' = res htrace
   in PRtreeN rectrace htrace (PR g h,x:xs,r')
trace2 (PR g h) [] = Stuck (PR g h,[],0)
```

The trace in the example was generated by this function, although it was pretty printed by a function (not shown) which indicates the tree structure by indenting.

In addition to generating trace trees it is possible to check them to see if they are good traces. The following code verifies that a trace is good. The `do` construct propagates the failure behavior of Maybe using a mechanism called a Monad.

```
check:: TraceTree (PR,[Integer],Integer) -> PR -> Maybe Integer
check (Ztree (Z,xs,0)) Z = Just 0
check (Stree (S,x:xs,y)) S =
   if y == (x+1) then Just y else Nothing
check (Ptree n (P m,xs,i)) (P k) =
   if (n==m) && (k==m) && (m <= length xs) && (nth m xs == i)
      then Just i
      else Nothing
check (Ctree t ts (C f gs,xs,i)) (C h ks) =
  do { ys <- zipM check ts gs
     ; j <- check t f
     ; let (_,zs,k) = tag t
     ; if (zs == ys) && (j == i) && (k == i) && (h==f) && (gs==ks)
          then (Just i)
          else Nothing }
check (PRtree0 t (PR f g,0:xs,i)) (PR h j) =
  do { z <- check t f
     ; if (z==i) && (f==h) && (g==j)
          then Just i
          else Nothing }
check (PRtreeN s t (PR f g,n:xs,i)) (PR h j) =
  do { r <- check s (PR f g)
     ; k <- check t g
     ; let (_,ws,_) = tag s
           (_,zs,_) = tag t
     ; if (f==h) && (g==j) && (k==i) && (ws==(n-1):xs) && (zs==(n-1):r:xs)
          then (Just i)
          else Nothing }
check x y = Nothing
```

Problem A.5 *Extend to the partial recursive functions. That is, build a verifier and tracing evaluator to verify and generate verifiable traces for the* `MuR` *functions.*

As you study this program look at the structure of the recursion. The check function is defined by induction on `TraceTree`. It is a total function.

Exercise A.6 *Define a function* `verify` *that takes a* `TraceTree` *for a computation and returns a Boolean.*