Computability via Recursive Functions

Church’s Thesis

- All effective computational systems are equivalent!
- To illustrate this point we will present the material from chapter 4 using the partial recursive functions, rather than Turing machines.

- We believe the arguments we will make are easier to follow than the Turing machine arguments.

Building blocks of Computability Theory

- 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.
- The ability to write down the trace of a computation that can be verified by a series of simple (terminating) steps.
- Having a large enough set of programs, in particular there needs to be a universal program that can read a program and its input and generate its output.

“Implementing” the Primitive Recursive Programs

- We have argued that the Primitive Recursive programs are simple yet very expressive
- Expressive enough to supply (almost) all the building blocks of computability theory

- We demonstrate this by giving each block an exact implementation
- We implement these in Haskell so we can run them.

Describing the PR functions as Haskell data

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

Our grammar

Term	$\rightarrow$ Z	
	| S	
	| P n	nth projection
	| C Term [$Term_1$, … ,$Term_n$]	composition
	| PR Term Term	primitive recursion
	| (Term)	grouping

- By design, this is similar to our context free grammar describing the primitive recursive functions
- This Haskell datatype exactly describes an inductively defined set.

An interpreter

Defined for every PrimRec every input of any length, returns 0 for ill-formed terms where aritys don't match

```
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
```

Pairing functions

- Assign a unique integer to every pair of integers.
- Recover the pair from the result

	0	1	2	3	4	5	6
0	1	3	6	10	15	21	28
1	2	5	9	14	20	27	
2	4	8	13	19	26		
3	7	12	18	25			
4	11	17	24				
5	16	23					
6	22						

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)
```

Pairing to encode Lists

- [] (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))))))) 94523914127548123793040376
- Rules
  - Nil is the pair (0,0)
  - (x:xs) is the nested pair (1,(x, encoding of xs))
  - Recall [1,3,5] is (1 : (3 : (5 : [])))

Haskell code

[2,3]
(1, (2, (1, (3, (0, 0)))))

```
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 (but nonsense)
```

Extending to other data

- We can use pairing to encode any inductively defined data set

- In particular we can use paring to endode the PrimRec datatype of Haskell

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

Example

- Plus = PR (P 1) (C S [P 2])

A cons cell (x:xs)

(4,((2,1),(3,((1,0),(1,((2,2),(0,0)))))))

The empty list []

4511739842654672905730185440573223378237806974280320

dPR 4511739842654672905730185440573223378237806974280320

PR (P 1) (C S [P 2])

Are there non-Primitive Recursive Functions?

dPR x applied to the number on top

x	dPR x	0	1	2	3	4	5	6	7	8	9	10
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**

The **red** numbers on the diagonal show the result of applying i^{th} function to i.

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

notdiagonal x = 1 + diagonal x
```

- Argue why notdiagonal is not primitive recursive

Argument

- Proof by contradiction
- Assume notdiagonal was primitive recursive
- Then there is some j such that
  - ePr j = notdiagonal

`eval (ePr i) i`

x	ePr x	0	1	…	j	…
0	Z	0	0	…	0	…
1	S	1	2	…	J+1	…
…						
J	notdiagonal				w	
…						

We see

diagonal j = w

notdiagonal j = w

But we defined

notdiagonal x = diagonal x + 1

So we have a contradiction

What facts did we assume?

- Primitive recursive functions are total
- There exists an eval function
  - Given a PrimRec and arguments returns the result
- There is a function from numbers to programs
  - ePR

- An *effective enumeration* of a set of total-functions is a mapping from the natural numbers onto the set of funtions; $f_1, f_2, \ldots f_n$, together with a computable function `eval` such that
  - `(eval i x =` f_i`(x))`

Theorem

- Every effective enumeration is incomplete. That is there exist some total computable functions which are not included in the enumeration.

- Corrollaries
  - There is no effective enumeration of the computable functions
  - Any enumeration of the computable functions must include some partial functions!

Pairing is primitive recursive

- There are functions in PrimRec that denote the pairing functions.

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

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)
```

- We know from the homework that most of the parts of pair and unpair are in PrimRec. What ones are missing?

Bounded search

```
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)))}
```

A search that is bounded by a known value.

This operation, which we call bmin is primtive recursive. In fact a definition for it is given in Appendix   A.2

Thus we can define div and sqrt as primitive recursive

pair

```
pair k1 k2 = ((k1 + k2) * (k1 + k2 +1) `div` 2) + k2

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]
```

unpair

```
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)

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]

unpair x = (pi1 x, pi2 x)
```

Building Blocks

- 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.
- We can now provide the first building block using the primitive recursive functions
- Here are the first 11 functions
- [Z, S, Z, P 1, S, Z, C Z [], P 1, S, Z, PR Z Z, …]
- Why do some functions appear twice?

Partial Recursive programs

```
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)
```

Properties

- Like PrimRec with one additional operator Mu
- Unlike for PrimRec, eval is not total

```
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)
```

Partial Recursive programs are Turing Complete

- Partial recursive functions can simulate TM
  - We can represent TM using numbers using partial recursive pairing
  - We can represent TM configurations and computation histories using pairing
  - We can write a total predicate, T, (i.e. it doesn't use Mu) such that
    - T machine input history = 1 if the machine history is a halting history
    - T machine input history = 0 If the machine history is not a halting history
  - We can write a total function, U, that given a machine, a halting history, that returns the final output
  - Given a TM: e, an input: x, we can use unbounded search that return the least y such that T(e,x,y) holds. Note that like a TM, this might not halt because it does use Mu operator

Traces

- For every computation system we defined acceptance by the existence of a trace

- Acceptance by DFA by a sequence of states
- Acceptance of CFG by a sequence of derivations
- Acceptance by PDA
- Acceptance by TM

DFA trace

- A DFA = $(\mathbf{Q}, \Sigma, \delta, \mathbf{q_0}, \mathbf{F})$, *accepts* a string
- $\mathbf{w}$ = "$w_1 w_2 \ldots w_n$" iff

  – There exists a sequence of states $[r_0, r_1, \ldots r_n]$ with 3 conditions

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

  Page 40 in Sisper

Acceptance is about finding a sequence.

How do we find such a sequence?

CFG Trace

- The single-step derivation relation $\Rightarrow$ on $(V\cup T)^*$ is defined by:

1. $\alpha \Rightarrow \beta$ iff β is obtained from α by replacing an occurrence of the lhs of a production with its rhs. That is, $\alpha' A \alpha'' \Rightarrow \alpha' \gamma \alpha''$ is true iff $A \rightarrow \gamma$ is a production. We say $\alpha' A \alpha''$ yields $\alpha' \gamma \alpha''$

2. We write $\alpha \Rightarrow^* \beta$ when β can be obtained from α through a sequence of several (possibly zero) derivation steps.

3. The *language of the CFG* , G, is the set

- $L(G) = \{w \in T^* \mid S \Rightarrow^* w\}$ (where S is the start symbol of G)

$S \Rightarrow^* w$ means there exists a sequence
$S \Rightarrow W_1 \Rightarrow W_2 \Rightarrow \ \dots \ \Rightarrow W$

PDA trace

- Suppose a string w can be written: $w_1\ w_2 \ldots w_m$
  - $W_i \in \Sigma_\varepsilon$ Some of the w_i are allowed to be ε
  - I.e. One may write "abc" as $a\ \varepsilon\ b\ c\ \varepsilon$
- If there exist two sequences
  - $r_0\ r_1 \ldots r_m \in Q$
  - $s_0\ s_1 \ldots s_m \in \Gamma^*$ (The s_i represent the stack contents at step i)

1. $r_0 = q_0$ and $s_0 = \varepsilon$
2. $(r_{i+1}, \alpha) \in \delta(r_i, w_{i+1}, A)$
   - $s_i = A\beta$ $s_{i+1} = \alpha\beta$
3. $r_m \in F$

The initial state and stack

Corresponding elements in the sequences are related to the next via the transition function.

The last state in the sequence is in the Final states.

TM Trace

- Recall a configuration (ID) has the form $\alpha\ q\ \beta$
  - where $\alpha, \beta \in \Gamma^*$ and $q \in Q$.
  - The string α represents the tape contents to the left of the head.
  - The string β represents the non-blank tape contents to the right of the head, including the currently scanned cell.
  - q represents the current state
- Recall configurations c_1, c_2 are related by
  - c_1 |- c_2
  - If the TM can legally move from c_1 to c_2
- A computation history (c_1, … , c_n) is a sequence of |- related configurations (each c_i |- c_{i+1})

Accepting (rejecting) Histories

- A computation history (c_1, … , c_n) is called an *accepting* history if c_1 is a start configuration and c_n is an accepting configuration

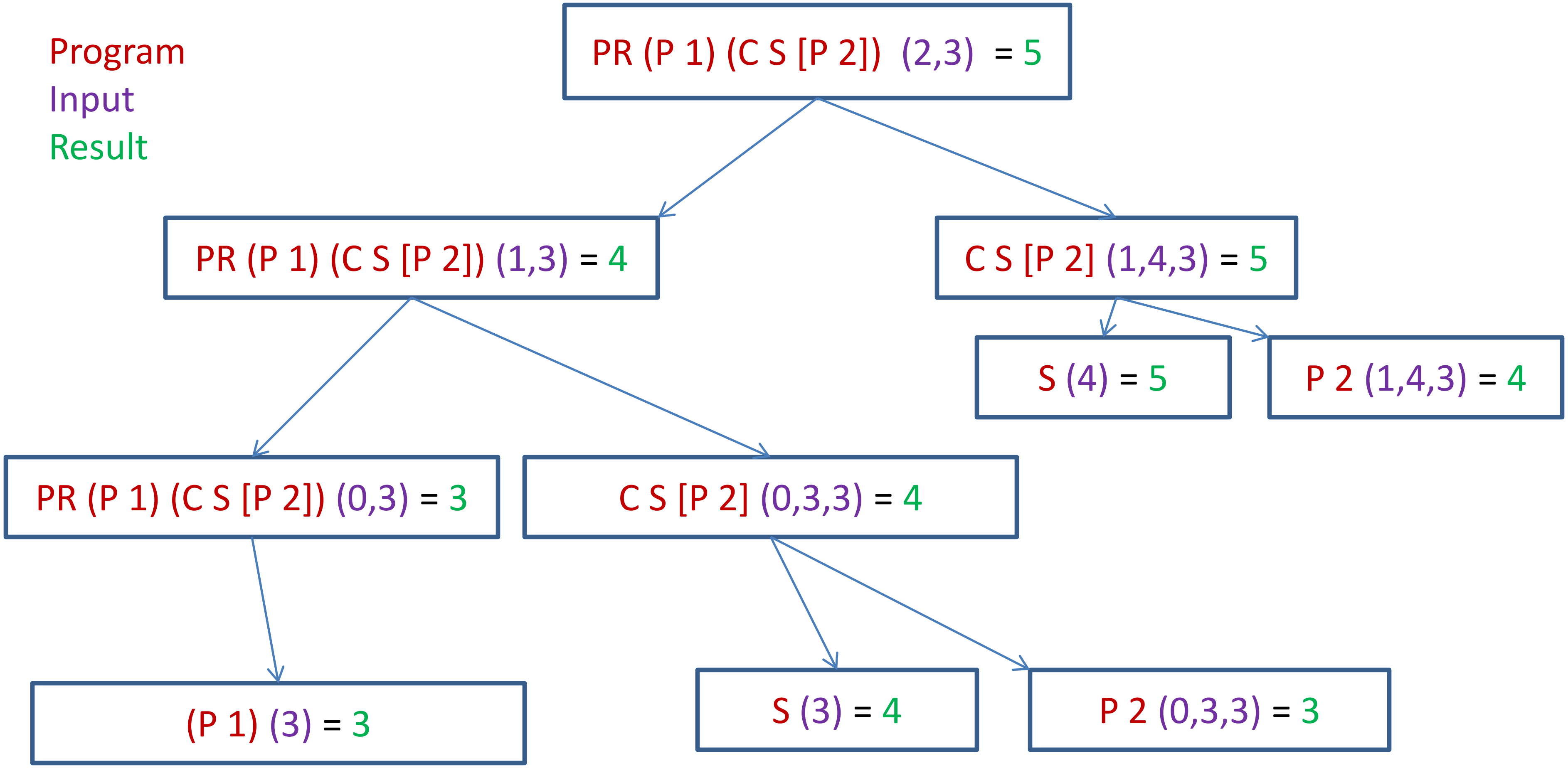
- A computation history (c_1, … , c_n) is called an *rejecting* history if c_1 is a start configuration and c_n is an rejecting configuration

If a TM does not halt on a given input, there does not exist an accepting (rejecting) history.

Traces for recursive functions

- A trace for primitive (partial) recursive functions is not a sequence but a Tree.
- Each node in the tree is labeled with a triple
  - (program, input,result)
- Compound programs (C, PR, Mu) have subtrees.
- In a Trace-tree, the subtrees are related by the computation rules.

Program
Input
Result



f(0,x1, ..., xk) = h(x1,...,xk)
f(Succ(n),x1, ..., xk)= g(n, f(n,x1,...,xk**),** x1,...,xk**)**

$f(x_1,\ldots x_n) = h(\ g_1(x_1,\ldots,x_n),\ \ldots\ ,g_m(x_1,\ldots,x_n)\ )$

Well formedness of trace trees is computable by a total function

- We encode trace trees by using pairing
- We use the rules of computation to relate a node and its subtrees.

- Construction of trace trees is computable by a partial function.
  - If a computation halts we can compute its trace tree
  - If it doesn’t the computation of the trace tree will also loop

Big result

- eval prog (input) = result --- Partial
- trace prog (input) = trace-tree --- Partial
- verify prog input trace-tree = boolean -- Total

- valid program input result trace =
  (verify prog input trace) &&
  (last trace = result) --- Total

- Theorem for n-ary function f
  - eval f (n_1,…, n_k) = w
  - If and only if
  - There exists a trace-tree c, such that (valid f (n_1,…, n_k) w c)

The halting problem

- Use diagonalization to show that there does not exist a total partial recursive program, halt, such that `halt (dMuR f) n` is True if and only if `eval f n` is defined.

- Suppose halt exists, then use it to define

```
Opposite(x,n) =
   if halt(x,n)
      then loop
      else 0

notdiagonal x = opposite (dMuR x) x
```

How halt(p,i) and opposite(p,i) are related.

halt(p,i)	True	False (looping)	
opposite(p,i)	Loop	0	

How How halt(p,n) and opposite(p,n) and notdiagonal(n) are related.

halt(p,n)	True	False (looping)	
opposite(p,n)	Loop	0	
notdiagonal(n)	Loop	0	Where p = dMuR n

The curious case when all are applied to notdiagonal, whose index is k

halt(notdiagonal,k)	True	False (looping)	
opposite(notdiagonal,k)	Loop	0	
notdiagonal(k) =	Loop	0	