Lessons touched by this meeting according to schedule

* 6. 28/10/2024
  + Generation of computable functions
    - Primitive recursion and examples [§2.4]
    - Definition by cases. Algebra of Decidability. Bounded sums and products.
    - Bounded quantification [§2.4.6, §2.4.7, §2.4.10]
    - Bounded minimalisation [§2.4.12, §2.4.13, §2.4.14, §2.4.15]
* 7. 29/10/2024
  + Generation of computable functions
    - Unbounded minimalisation [§2.5]
    - Computability of the inverse function. Finite functions and their computability.
  + Partial recursive functions [§3.1, §3.2, §3.7]
    - Definition
    - The class of partial recursive functions coincide with the class of URM-computable functions [statement and some ideas]

Immagine che contiene testo, Carattere, diagramma, calligrafia

Descrizione generata automaticamente

What we learn as important concepts here:

* If a function is defined by cases, then every part can be also seen as decidable predicate, so everything is computable
  + Meaning of vectors defined functions
  + Meaning of characteristic functions
* Algebra of decidable predicates (important when you will see Rice-Shapiro)

Based on the notation shown in the image, this appears to be representing:

1. Q(x̄,y) - A predicate or relation Q taking a vector x̄ (representing multiple inputs x1,...,xk) and y as arguments
2. Nk+1 → N - A function mapping from k+1-dimensional natural numbers to natural numbers

Immagine che contiene testo, calligrafia, carta, Prodotto di carta

Descrizione generata automaticamente

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamenteExercises on the predicates become important ahead:

Immagine che contiene testo, Carattere, schermata, documento

Descrizione generata automaticamente

Sum and product need to be bounded, which means it has to hold for all subpredicates. Clarification on notation being used:

(1) For g(x⃗,y) = Σz<y f(x⃗,y): The function can be defined by primitive recursion as:

g(x⃗,0) = 0  
g(x⃗,y+1) = g(x⃗,y) + f(x⃗,y)

Both the base case and step case use computable functions:

* The constant 0 function is computable (basic function)
* f is computable by hypothesis
* Addition is computable
* The composition of computable functions is computable

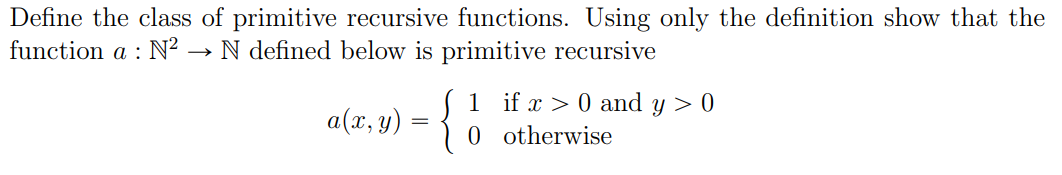
Therefore g is computable by primitive recursion.

(2) For h(x⃗,y) = Πz<y f(x⃗,y): Similarly, this function can be defined by primitive recursion as:

h(x⃗,1) = 1  
h(x⃗,y+1) = h(x⃗,y) · f(x⃗,y)

The components are computable:

* The constant 1 function is computable (basic function)
* f is computable by hypothesis
* Multiplication is computable
* The composition of computable functions is computable

Exercise (2024-02-16)

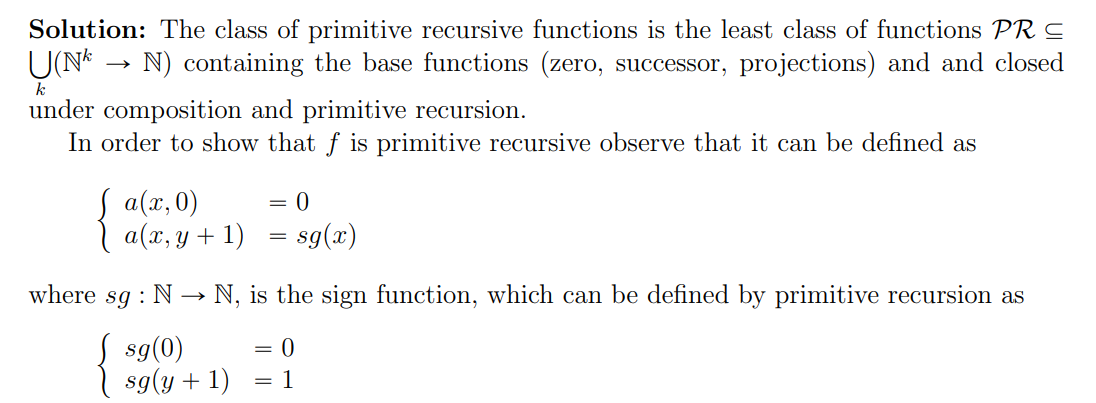


Immagine che contiene testo, Carattere, documento

Descrizione generata automaticamente

The class of primitive recursive functions (PR) is the least class of functions containing:

(a) The zero function: z : N → N, z(x) = 0

(b) The successor function: s : N → N, s(x) = x + 1

(c) The projection functions: U\_i^k : N^k → N, U\_i^k(x₁,...,xₖ) = xᵢ

and closed under:

1. Composition
2. Primitive recursion

Let's define it using composition of primitive recursive functions:

1. First define double(y) = 2y by primitive recursion:

double(0) = 0

double(y+1) = double(y) + 2 = (double(y) + 1) + 1

1. This function is primitive recursive as it uses only composition and primitive recursion from basic functions.
2. Then we can express f(y) = 2y + 1 as: f(y) = s(double(y)) Where s is the successor function (basic function) and double(y) is primitive recursive as shown above.

Since f is obtained by composition of primitive recursive functions (s and double), and composition preserves primitive recursion, we conclude that f(y) = 2y + 1 is primitive recursive.

Immagine che contiene testo, Carattere, schermata, algebra

Descrizione generata automaticamente

Solution:

1. First, let's prove CB ⊆ C: This is straightforward since URM-Back is a restricted version of URM. Any URM-Back program is already a valid URM program with the same semantics, therefore every URM-Back computable function is URM computable.
2. Now, let's prove C ⊆ CB: We need to show that any URM program can be transformed into an equivalent URM-Back program. The key is to simulate forward jumps using backward jumps.

Given a URM program P of length n, we can construct a URM-Back program P' as follows:

a) Transform each forward jump instruction J(m,n,t) into:

* Initialize a counter register Rc to n-t (where n is program length)
* Use a loop with backward jumps that:
  + Decrements Rc
  + Jumps back to decrement again if Rc > 0
  + When Rc = 0, we've effectively moved forward t steps

b) Specifically, for a forward jump J(m,n,t), replace it with:

T(m,k) // Save comparison values

T(n,k+1)

Z(k+2) // Initialize counter

T(1,k+3) // Save original contents

...

J(k,k+1,p) // Compare original values

J(1,1,q) // Backward jump to decrement counter

where p points to the start of the counter decrement sequence and q points to the corresponding target instruction.

This transformation preserves the semantics of the original program, uses only backward jumps and is clearly effective (can be done algorithmically). Therefore C = CB.

Immagine che contiene testo, Carattere, schermata, linea

Descrizione generata automaticamenteNow, for an important concept: bounded minimalization.

In plain English, this means we are only using bounded operations (sums, products):

1. We're looking for the smallest value of z that is less than y where f(x⃗,z) = 0
2. If no such z exists (i.e., f(x⃗,z) ≠ 0 for all z < y), then h returns y

The key differences from unbounded minimalization are:

* The search is bounded by y
* The function h is always total because:
  + Either we find a z < y where f(x⃗,z) = 0
  + Or we hit the bound y and return it
* Immagine che contiene testo, Carattere, linea, bianco

  Descrizione generata automaticamenteImmagine che contiene testo, Carattere, schermata, bianco

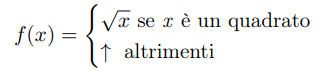
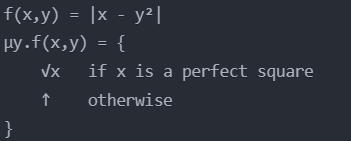
  Descrizione generata automaticamenteWe don't have the issue of divergence that exists with unbounded minimalization

Fibonacci function introduces the concept of 🡪 pair encoding, basically at an inductive step defining recursion on a couple of previously defined values

Immagine che contiene testo, Carattere, calligrafia, design

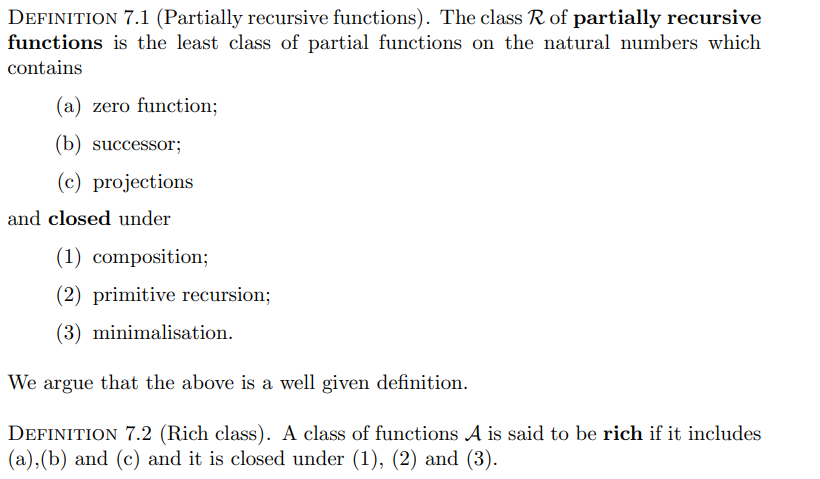
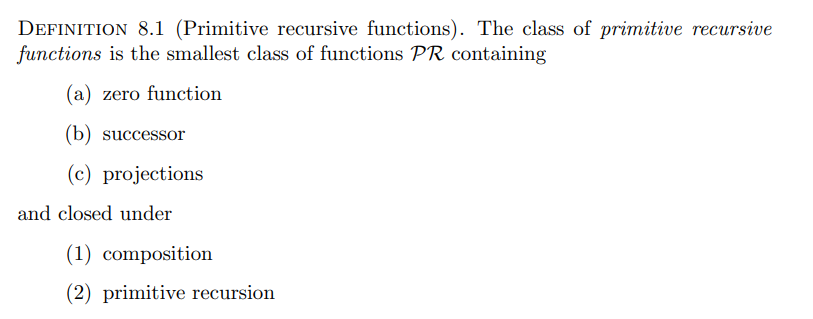
Descrizione generata automaticamenteUnbounded minimalization (μy.f(x̄,y)) searches for the smallest y where f(x̄,y) = 0, but unlike bounded minimalization, this search has no upper limit. This is equivalent to a while loop:

The key difference from bounded minimalization is that this search might:

* Never find a solution (no y makes f(x̄,y) = 0)
* Never terminate (hit a value where f(x̄,y) is undefined)

This works because:

* If x is a perfect square (like 16), it will find y (4) where y² = x
* If x is not a perfect square (like 5), it will never find such a y
* The function continues searching indefinitely until it either finds a solution or determines none exists - it captures the idea of "searching until a condition is met" without a known upper bound

Comment on:

The closure is important because of diagonalization, which introduces us to the concept of the “search” of unbounded functions (total/not computable – you will see that in diagonalization):

LOSURE AND INVERSE FUNCTIONS IN COMPUTABILITY THEORY

1. Closure Properties

A class of functions is said to be "closed" under an operation if applying that operation to functions in the class yields another function that is also in the class. In computability theory, we are particularly interested in three key closure properties:

a) Composition:

* If f and g are computable functions, then their composition h(x) = f(g(x)) is also computable
* This allows building complex computations from simpler ones

b) Primitive Recursion:

* Allows defining functions through recursion with a base case and recursive step
* Example: factorial function
  + Base case: f(0) = 1
  + Recursive step: f(n+1) = (n+1)×f(n)

c) Minimalization:

* Given computable f(x,y), allows finding least y such that f(x,y) = 0
* Written as μy[f(x,y) = 0]
* May result in partial functions, unlike composition and primitive recursion

1. Inverse Functions and Diagonalization

For a function f: N → N that is:

* Computable
* Injective
* Not necessarily total

Its inverse f⁻¹ defined as:

f⁻¹(y) = x if f(x) = y

f⁻¹(y) = ↑ if ∄x. f(x) = y

We see here the following properties:

a) Computability:

* Can be computed using minimalization: f⁻¹(y) = μx[|f(x) - y|]
* Works even when f is partial because minimalization handles undefined values

b) Diagonalization Proof Sketch (it will become more useful later):

1. Start with injective computable f
2. For any y, search systematically for x where f(x) = y
3. The search termination is guaranteed by injectivity
4. The search process is effective using minimalization
5. Therefore f⁻¹ is computable

We might draw some conclusions here:

* Not all computable functions have computable inverses
* The function must be injective for this result
* Diagonalization is crucial for proving properties about computability
* The result extends to partial functions through careful use of minimalization

Taken from an Italian exam (2018-11-20-parziale):

*Consider a variant of the URM machine that includes jump instructions, transfer instructions, and two new instructions:*

*1. A(m,n) - writes in register m the sum of registers m and n (i.e., rm ← rm + rn)*

*2. C(n) - writes in register n the value of its complemented sign (i.e., rn ← sg̅(rn))*

Determine the relationship between the set C' of functions computable with the new machine and the set C of functions computable with the standard URM machine. Is one contained in the other? Is the inclusion strict? Justify your answers.

Solution

Let us denote the modified machine as URM\*. We observe that the instructions of URM\* can be encoded in the standard URM machine.

1. Encoding A(m,n):

An instruction Ij: A(m,n) can be replaced with a jump to the following routine, where q is the index of the first unused register by the program (thus initially at 0):

SUB : J(n,q,j+1)

S(m)

S(q)

J(1,1,SUB)

2. Encoding C(m):

Similarly, indicating with q an unused register index, an instruction Ij: C(m) can be replaced with a jump to the subroutine:

SUB : J(n,q,ZERO)

Z(n)

J(1,1,j+1)

ZERO: S(n)

J(1,1,j+1)

More formally, we prove that C\* ⊆ C by showing that, for any number of arguments k and for any program P using both sets of instructions, we can obtain a URM program P' that computes the same function, i.e., such that f(k)P' = f(k)P.

The proof proceeds by induction on the number h of A and C instructions in the program:

* Base case (h = 0): Trivial, since P with 0 instructions A and C is already a URM program.
* Inductive step (h → h + 1): Assuming the result holds for h, let's prove it for h + 1. Program P contains at least one A or C instruction. Let j be its index:

1 : I1

...

j : A(m,n)

...

ℓ(P): Iℓ(P)

We construct a program P2 using a register not referenced in P, q = max{ρ(P), k} + 1:

1 : I1

...

j : J(1,1,SUB)

...

ℓ(P): Iℓ(P)

J(1,1,END)

SUB : J(n,q,ZERO)

Z(n)

J(1,1,j+1)

ZERO: S(n)

J(1,1,j+1)

END :

P2 computes the same function as P and contains h instructions of type A or C. By inductive hypothesis, there exists a URM program P' such that which is the desired program.

For the opposite inclusion C ⊆ C\*, we proceed analogously, observing that instructions Z(n) and S(n) are codifiable in the modified machine.

Given a program P and indices q1 and q2 of unused registers (thus initially at 0), consider the program:

C(q1) // makes q1 contain 1

P'

where P' is obtained from P by replacing each instruction Z(m) with T(q2,m) and each instruction S(m) with A(m,q1).

Therefore, we conclude that C = C\*.

Immagine che contiene testo, Carattere, bianco

Descrizione generata automaticamenteFirst recall that PR is the class of primitive recursive functions containing:

* Zero function z(x) = 0
* Successor function s(x) = x + 1
* Projection functions U\_i^k(x₁,...,xₖ) = xᵢ and closed under composition and primitive recursion.

We'll proceed by induction on k:

Base case (k = 2): Define sum₂(x₁,x₂) by primitive recursion on x₂:

sum₂(x₁,0) = x₁ = U\_1^2(x₁,0)

sum₂(x₁,y+1) = s(sum₂(x₁,y))

This is primitive recursive as it uses only:

* Base case: projection function (basic)
* Recursive step: composition of successor (basic) with recursive call

Inductive step (k > 2): Assume sumₖ₋₁ is primitive recursive. Define:

sumₖ(x₁,...,xₖ) = sum₂(sumₖ₋₁(x₁,...,xₖ₋₁), xₖ)

This is primitive recursive because:

* sumₖ₋₁ is primitive recursive (inductive hypothesis)
* sum₂ is primitive recursive (base case)
* Their composition is primitive recursive

Therefore, by induction, sumₖ is primitive recursive for all k ≥ 2.

(Extended given a question in class)

It can hold for all k up to n. In fact, when we prove by induction that sumₖ is primitive recursive for all k ≥ 2, we're actually proving:

1. Base case (k = 2): sum₂ is primitive recursive
2. Inductive step: If sumₘ is primitive recursive for all m where 2 ≤ m < k, then sumₖ is also primitive recursive

This means that at each step k, we have already established that:

* sum₂ is primitive recursive
* sum₃ is primitive recursive
* ...
* sumₖ₋₁ is primitive recursive

This is known as strong induction or complete induction, where we use the fact that the property holds for all previous values, not just the immediately preceding one. However, in this specific case, we only need the immediately preceding case sumₖ₋₁ to construct sumₖ, so simple induction suffices.

For example, to show sum₄ is primitive recursive:

1. We know sum₃ is primitive recursive (from previous step)
2. Define sum₄(x₁,x₂,x₃,x₄) = sum₂(sum₃(x₁,x₂,x₃), x₄)
3. This is primitive recursive as composition of primitive recursive functions

The process continues for any finite k.