6. The Recursive Functions and the Equivalence Theorem

This Sect. has three parts:

(a) Introduction of the **partial recursive functions**.

- Third model of computation.
- Partial rec. functions is the main model used for proving theorems in computability theory.
- It extends the primitive-recursive functions so that we obtain a full model of computation.

(b) Equivalence of the models of computation.

- Proof that sets of URM-computable functions, of TM-computable functions and of partial recursive functions coincide.

(c) The Church Turing Thesis.

- Evidence why these models of computation define the computable functions.

(a) The Part. Rec. Functions

Ackermann Function

- At the end of Subsection 5 (a) we have started to introduce a series of functions

  $$\mathsf{add} \longrightarrow \mathsf{mult} \longrightarrow \mathsf{exp} \longrightarrow \mathsf{superexp} \longrightarrow \mathsf{supersuperexp} \longrightarrow \cdots$$

  where each function in this sequence is defined by primitive recursion using the previous function.
- Such a sequence will eventually exhaust the primitive recursive function.

The Ackermann Function

- Traditionally, instead of defining a sequence of binary functions one defines a sequence of unary functions with similar growth rate, the **Ackermann function**.
- The Ackermann function exhausts all primitive recursive functions.
- The uniform version of the Ackermann function will therefore no longer be primitive recursive.
- In order to obtain a complete model of computation, we will need to extend the primitive recursive function by closure under μ.
  - The resulting functions will be called the **partial recursive functions**.

The Ackermann Function

- Let $n \in \mathbb{N}$.
  The **n-th branch of the Ackermann function** $\mathrm{Ack}_n : \mathbb{N} \to \mathbb{N}$, is defined by

$$\begin{aligned}
\mathrm{Ack}_0(y) &= y + 1 \ , \\
\mathrm{Ack}_{n+1}(y) &= (\mathrm{Ack}_n)^{y+1}(1) := \underbrace{\mathrm{Ack}_n(\mathrm{Ack}_n(\cdots \mathrm{Ack}_n(}_{y+1 \text{ times}} 1))) \ .
\end{aligned}$$

  Ack_n is prim. rec. **for fixed n**.

Examples

$$
\begin{aligned}
\mathsf{Ack}_0(n) &= n+1 \quad . \\
\mathsf{Ack}_1(n) &= \mathsf{Ack}_0^{n+1}(1) \\
&= 1 \underbrace{+1+\cdots+1}_{n+1\ \text{times}} \\
&= 1+n+1 = n+2 \quad . \\
\mathsf{Ack}_2(n) &= \mathsf{Ack}_1^{n+1}(1) \\
&= 1 \underbrace{+2+\cdots+2}_{n+1\ \text{times}} \\
&= 1+2(n+1) \\
&= 2n+3 > 2n \quad .
\end{aligned}
$$

Examples

$$\mathsf{Ack}_2(n) > 2n \quad .$$

$$\begin{aligned}
\mathsf{Ack}_3(n) &= \mathsf{Ack}_2^{n+1}(1) \\
&> \underbrace{2 \cdot 2 \cdot \cdots \cdot 2}_{n+1 \text{ times}} \cdot 1 \\
&= 2^{n+1} > 2^n \quad . \\
\mathsf{Ack}_4(n) &= \mathsf{Ack}_3^{n+1}(1) \\
&> \underbrace{2^{\cdot^{\cdot^{\cdot^{2^1}}}}}_{n+1 \text{ times}} \quad .
\end{aligned}$$

Examples

$$\mathrm{Ack}_4(n) > \underbrace{2^{\cdot^{\cdot^{\cdot^{2^1}}}}}_{n+1 \text{ times}} .$$

- $\mathrm{Ack}_5(n)$ will iterate Ack_4 $n+1$ times, etc.
- So even for very small n, $\mathrm{Ack}_5(n)$ will exceed the number of particles in the universe, and Ack_5 is therefore not realistically computable.

The Ackerm. Funct. is Prim. Rec.

$$\begin{aligned} \mathsf{Ack}_0(y) &= y+1 \ , \\ \mathsf{Ack}_{n+1}(y) &= (\mathsf{Ack}_n)^{y+1}(1) \ . \end{aligned}$$

Proof that Ack_n is prim. rec. by Induction(n):

- **Base-case:**
  $\mathsf{Ack}_0 = \mathsf{succ}$ is prim. rec.

The Ackerm. Funct. is Prim. Rec.

$$\begin{aligned}\mathsf{Ack}_0(y) &= y+1 \ ,\\ \mathsf{Ack}_{n+1}(y) &= (\mathsf{Ack}_n)^{y+1}(1) \ .\end{aligned}$$

- **Induction step:**
  Assume Ack_n is primitive recursive.
  Show Ack_{n+1} is primitive recursive. We have:

$$\begin{aligned}\mathsf{Ack}_{n+1}(0) &= (\mathsf{Ack}_n)^1(1)\\ &= \mathsf{Ack}_n(1) \ ,\\ \mathsf{Ack}_{n+1}(y+1) &= (\mathsf{Ack}_n)^{y+2}(1)\\ &= \mathsf{Ack}_n(\mathsf{Ack}_n^{y+1}(1))\\ &= \mathsf{Ack}_n(\mathsf{Ack}_{n+1}(y)) \ .\end{aligned}$$

The Ackerm. Funct. is Prim. Rec.

- So:

  $$\begin{aligned} \mathrm{Ack}_{n+1}(0) &= \mathrm{Ack}_n(1)\ , \\ \mathrm{Ack}_{n+1}(y+1) &= \mathrm{Ack}_n(\mathrm{Ack}_{n+1}(y))\ . \end{aligned}$$

  which shows that Ack_{n+1} is primitive recursive, using the assumption (induction hypothesis) that Ack_n is primitive recursive.
- Therefore Ack_n is primitive recursive for all $n \in \mathbb{N}$.
  End of proof.
- **Remark:**
  - Ack_n for **fixed** n is **primitive recursive**.
  - However a **uniform** version of the Ackermann function is **not primitive recursive**.

The Uniform Ackermann Function

The uniform version of the **Ackermann function** $\mathsf{Ack} : \mathbb{N}^2 \to \mathbb{N}$ is defined as

$$\mathsf{Ack}(n,m) := \mathsf{Ack}_n(m) \quad .$$

Therefore we have the equations

$$\begin{array}{rcl} \mathsf{Ack}(0,y) & = & y+1 \quad , \\ \mathsf{Ack}(x+1,0) & = & \mathsf{Ack}(x,1) \quad , \\ \mathsf{Ack}(x+1,y+1) & = & \mathsf{Ack}(x,\mathsf{Ack}(x+1,y)) \quad . \end{array}$$

In order to show that Ack is not prim. rec., we show first the following technical lemma:

Lemma 6.1

For each n, m, the following holds:

(a) $\text{Ack}(m, n) > n$.

(b) Ack_m is strictly monotone,
i.e. $\text{Ack}(m, n + 1) > \text{Ack}(m, n)$.

(c) $\text{Ack}(m + 1, n) > \text{Ack}(m, n)$.

(d) $\text{Ack}(m, \text{Ack}(m, n)) < \text{Ack}(m + 2, n)$.

(e) $\text{Ack}(m, 2n) < \text{Ack}(m + 2, n)$.

(f) $\text{Ack}(m, 2^k \cdot n) < \text{Ack}(m + 2k, n)$.

The proof will be omitted in the lecture.

Jump over proof.

Proof of Lemma 6.1 (a)

Induction on m.

$m = 0$:

$$\mathsf{Ack}(0, n) = n + 1 > n \quad .$$

$m \to m + 1$: Side-induction on n

$n = 0$:

$$\mathsf{Ack}(m + 1, 0) = \mathsf{Ack}(m, 1) \overset{\mathsf{IH}}{>} 1 > 0 \quad .$$

Proof of Lemma 6.1 (a)

$n \to n+1$:

$$\begin{array}{rcl} \mathsf{Ack}(m+1, n+1) & = & \mathsf{Ack}(m, \mathsf{Ack}(m+1, n)) \\ & \overset{\text{Main IH}}{>} & \mathsf{Ack}(m+1, n) \\ & \overset{\text{Side IH}}{>} & n \quad , \end{array}$$

therefore

$$\mathsf{Ack}(m+1, n+1) > n+1 \quad .$$

Proof of Lemma 6.1 (b)

Case $m = 0$:

$$\mathsf{Ack}(0, n + 1) = n + 2 > n + 1 = \mathsf{Ack}(0, n) \ .$$

Case $m = m' + 1$:

$$\begin{aligned} \mathsf{Ack}(m' + 1, n + 1) &= \mathsf{Ack}(m', \mathsf{Ack}(m' + 1, n)) \\ &\overset{\text{(a)}}{>} \mathsf{Ack}(m' + 1, n) \ . \end{aligned}$$

Proof of Lemma 6.1 (c)

Induction on m.
$m = 0$:

$$\mathsf{Ack}(1, n) = n + 2 > n + 1 = \mathsf{Ack}(0, n)$$

Proof of Lemma 6.1 (c)

$m \to m+1$: Side-induction on n:

$n = 0$:

$$\mathsf{Ack}(m+2,0) = \mathsf{Ack}(m+1,1) \overset{\text{(b)}}{>} \mathsf{Ack}(m+1,0) \enspace .$$

$n \to n+1$:

$$\begin{aligned} &\mathsf{Ack}(m+2,n+1) \\ &\qquad = \quad \mathsf{Ack}(m+1,\mathsf{Ack}(m+2,n)) \\ &\qquad \overset{\text{main-IH}}{>} \quad \mathsf{Ack}(m,\mathsf{Ack}(m+2,n)) \\ &\qquad \overset{\text{side-IH + (b)}}{>} \quad \mathsf{Ack}(m,\mathsf{Ack}(m+1,n)) = \mathsf{Ack}(m+1,n+1) \enspace . \end{aligned}$$

Proof of Lemma 6.1 (d)

Case $m = 0$:

$$\mathsf{Ack}(0, \mathsf{Ack}(0, n)) = n + 2 < 2n + 3 = \mathsf{Ack}(2, n) \quad .$$

Assume now $m > 0$.
Proof of the assertion by induction on n:
$n = 0$:

$$\begin{array}{rcl} \mathsf{Ack}(m + 2, 0) & = & \mathsf{Ack}(m + 1, 1) \\ & = & \mathsf{Ack}(m, (\mathsf{Ack}(m + 1, 0)) \\ & > & \mathsf{Ack}(m, \mathsf{Ack}(m, 0)) \quad . \end{array}$$

Proof of Lemma 6.1 (d)

$n \to n+1$:

$$\begin{array}{rcl}
\mathsf{Ack}(m+2, n+1) & = & \mathsf{Ack}(m+1, \mathsf{Ack}(m+2, n)) \\
& \overset{\text{IH,(b)}}{>} & \mathsf{Ack}(m+1, \mathsf{Ack}(m, \mathsf{Ack}(m, n))) \\
& \overset{\text{(b),(c)}}{>} & \mathsf{Ack}(m, \mathsf{Ack}(m-1, \mathsf{Ack}(m, n))) \\
& = & \mathsf{Ack}(m, \mathsf{Ack}(m, n+1)) \quad .
\end{array}$$

Proof of Lemma 6.1 (e)

Case $m = 0$:

$$\begin{array}{rcl} \mathsf{Ack}(m, 2n) & = & \mathsf{Ack}(0, 2n) \\ & = & 2n + 1 \\ & < & 2n + 3 \\ & = & \mathsf{Ack}(2, n) = \mathsf{Ack}(m + 2, n) \quad . \end{array}$$

Proof of Lemma 6.1 (e)

Case $m = m' + 1$:
Induction on n:
$n = 0$:

$$\text{Ack}(m'+1, 2n) = \text{Ack}(m'+1, 0) < \text{Ack}(m'+3, 0) = \text{Ack}(m'+3, n) \ .$$

$n \to n + 1$:

$$\begin{array}{rcl} \text{Ack}(m' + 1, 2n + 2) & = & \text{Ack}(m', \text{Ack}(m', \text{Ack}(m' + 1, 2n)) \\ & \overset{\text{(d)}}{<} & \text{Ack}(m' + 2, \text{Ack}(m' + 1, 2n)) \\ & \overset{\text{IH}}{<} & \text{Ack}(m' + 2, \text{Ack}(m' + 3, n)) \\ & = & \text{Ack}(m' + 3, n + 1) \end{array}$$

Proof of Lemma 6.1 (f)

Induction on k:
$k = 0$: trivial.
$k \to k+1$:

$$\begin{array}{rcl}
\mathsf{Ack}(m, 2^{k+1} \cdot n) & = & \mathsf{Ack}(m, 2 \cdot 2^k \cdot n) \\
& \overset{\text{(e)}}{<} & \mathsf{Ack}(m+2, 2^k \cdot n) \\
& \overset{\text{IH}}{<} & \mathsf{Ack}(m+2+2k, n) \\
& = & \mathsf{Ack}(m+2(k+1), n) \quad .
\end{array}$$

Lemma 6.2

Every primitive recursive function $f : \mathbb{N}^n \to \mathbb{N}$ can be majorised by one branch of the Ackermann function:

There exists N s.t.

$$f(x_0, \ldots, x_{n-1}) < \mathsf{Ack}_N(x_0 + \cdots + x_{n-1})$$

for all $x_0, \ldots, x_{n-1} \in \mathbb{N}$.

Especially, if $f : \mathbb{N} \to \mathbb{N}$ is prim. rec., then there exists an N s.t.

$$\forall x \in \mathbb{N}. f(x) < \mathsf{Ack}_N(x)$$

The proof will be omitted in the lecture.

Jump over proof.

Proof of Lemma 6.2

We write, if $\vec{x} = x_0, \ldots, x_{n-1}$

$$\sum(\vec{x}) := x_0 + \cdots x_{n-1} \quad .$$

We proof the assertion by induction on the definition of primitive recursive functions.

Proof of Lemma 6.2

Basic functions:

zero:

$$\mathrm{zero}(x) = 0 < x + 1 = \mathrm{Ack}_0(x) \quad .$$

succ:

$$\mathrm{succ}(x) = \mathrm{Ack}(0, x) < \mathrm{Ack}(1, x) = \mathrm{Ack}_1(x) \quad .$$

proj_i^n:

$$\begin{array}{rcl} \mathrm{proj}(x_0, \ldots, x_{n-1}) & = & x_i \\ & < & x_0 + \cdots + x_{n-1} + 1 \\ & = & \mathrm{Ack}_0(x_0 + \cdots + x_{n-1}) \quad . \end{array}$$

Proof of Lemma 6.2

Composition: Assume assertion holds for $f : \mathbb{N}^k \to \mathbb{N}$ and $g_i : \mathbb{N}^n \to \mathbb{N}$. Show assertion for $h := f \circ (g_0, \ldots, g_{k-1})$.
Assume

$$f(\vec{y}) < \mathrm{Ack}_l(\sum(\vec{y})) \quad ,$$

and

$$g_i(\vec{x}) < \mathrm{Ack}_{m_i}(\sum(\vec{x})) \quad .$$

Let $N := \max\{l, m_0, \ldots, m_{k-1}\}$. By Lemma 6.1 (c) it follows

$$f(\vec{y}) < \mathrm{Ack}_N(\sum(\vec{y})) \quad ,$$

$$g_i(\vec{x}) < \mathrm{Ack}_N(\sum(\vec{x})) \quad .$$

Proof of Lemma 6.2

Then, with M s.t. $l < 2^M$, we have

$$\begin{array}{rcl}
h(\vec{x}) & = & f(g_0(\vec{x}), \ldots, g_{k-1}(\vec{x})) \\
 & < & \mathsf{Ack}_N(g_0(\vec{x}) + \cdots + g_{k-1}(\vec{x})) \\
 & \overset{\text{(b)}}{<} & \mathsf{Ack}_N(\mathsf{Ack}_N(\sum(\vec{x})) + \cdots + \mathsf{Ack}_N(\sum(\vec{x}))) \\
 & = & \mathsf{Ack}_N(\mathsf{Ack}_N(\sum(\vec{x})) \cdot k) \\
 & < & \mathsf{Ack}_N(\mathsf{Ack}_N(\sum(\vec{x})) \cdot 2^M) \\
 & < & \mathsf{Ack}_{N+2M}(\mathsf{Ack}_N(\sum(\vec{x}))) \\
 & \leq & \mathsf{Ack}_{N+2M}(\mathsf{Ack}_{N+2M}(\sum(\vec{x}))) \\
 & < & \mathsf{Ack}_{N+2M+2}(\sum(\vec{x})) \ .
\end{array}$$

Proof of Lemma 6.2

Primitive recursion, $n > 1$:
Assume assertion holds for $f : \mathbb{N}^n \to \mathbb{N}$ and $g : \mathbb{N}^{n+2} \to \mathbb{N}$.
Show assertion for $h := \text{primrec}(f, g) : \mathbb{N}^{n+1} \to \mathbb{N}$.
Assume

$$f(\vec{x}) < \text{Ack}_l(\sum(\vec{x})) \quad ,$$

$$g(\vec{x}, y, z) < \text{Ack}_r(\sum(\vec{x}) + y + z) \quad .$$

Let $N := \max\{l, r\}$, $k < 2^M$. Then

$$f(\vec{x}) < \text{Ack}_N(\sum(\vec{x})) \quad ,$$

$$g(\vec{x}, y, z) < \text{Ack}_N(\sum(\vec{x}) + y + z) \quad .$$

Proof of Lemma 6.2

We show

$$h(\vec{x}, y) < \mathsf{Ack}_{N+3}(\sum(\vec{x}) + y)$$

by induction on y:

$y = 0$:

$$\begin{array}{rcl} h(\vec{x}, 0) & = & g(\vec{x}) \\ & < & \mathsf{Ack}_N(\sum(\vec{x})) \\ & < & \mathsf{Ack}_{N+3}(\sum(\vec{x}) + 0) \quad . \end{array}$$

Proof of Lemma 6.2

$y \to y+1$:

$$
\begin{array}{rcl}
h(\vec{x}, y+1) & = & g(\vec{x}, y, h(\vec{x}, y)) \\
& < & \mathsf{Ack}_N(\sum(\vec{x}) + y + h(\vec{x}, y)) \\
& \overset{\text{IH}}{<} & \mathsf{Ack}_N(\sum(\vec{x}) + y + \mathsf{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
& < & \mathsf{Ack}_N(\mathsf{Ack}_{N+3}(\sum(\vec{x}) + y) + \mathsf{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
& = & \mathsf{Ack}_N(2 \cdot \mathsf{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
& < & \mathsf{Ack}_{N+2}(\mathsf{Ack}_{N+3}(\sum(\vec{x}) + y)) \\
& = & \mathsf{Ack}_{N+3}(\sum(\vec{x}) + y + 1)
\end{array}
$$

Proof of Lemma 6.2

Primitive recursion $n = 0$:
Assume $l \in \mathbb{N}$, $g : \mathbb{N}^2 \to \mathbb{N}$. Show assertion for $h := \text{primrec}(l, g) : \mathbb{N}^1 \to \mathbb{N}$.
Define

$$\begin{array}{lcll} f' & : & \mathbb{N} \to \mathbb{N}, & f'(x) = l \ , \\ g' & : & \mathbb{N}^3 \to \mathbb{N}, & g'(x, y, z) = g(y, z) \ , \\ h' & : & \mathbb{N}^2 \to \mathbb{N}, & h := \text{primrec}(f', g') \ . \end{array}$$

Using the constructions already shown and IH it follows that

$$h'(x, y) < \text{Ack}_N(x + y)$$

for some N. Therefore

$$h(y) = h'(0, y) < \text{Ack}_N(y) \ .$$

Theorem 6.3

The Ackermann function is **not primitive recursive**.

Proof:
Assume Ack were primitive recursive.
Then

$$f : \mathbb{N} \to \mathbb{N}, \qquad f(n) := \mathsf{Ack}(n, n)$$

is prim. rec.
Then there exists an $N \in \mathbb{N}$, s.t. $f(n) < \mathsf{Ack}(N, n)$ for all n.
Especially

$$\mathsf{Ack}(N, N) = f(N) < \mathsf{Ack}(N, N) \quad ,$$

a contradiction.

Omit direct argument for existence of computable but not primitive recursive functions

Remark

Direct argument for the existence of non-primitive recursive computable functions:
Assume all computable functions are prim. rec..
Define $h : \mathbb{N}^2 \rightarrow \mathbb{N}$,

$$h(e,n) = \begin{cases} f(n), & \text{if } e \text{ encodes a string in ASCII} \\ & \text{which is a term denoting a unary} \\ & \text{primitive recursive function } f, \\ 0, & \text{otherwise.} \end{cases}$$

Remark: $h(e,n)$ is similar to $\{e\}(n)$ as defined in Section 4 (c), but referring to primitive recursive functions instead of TM computable functions.

Remark

If e code for f, then $h(e, n) = f(n)$.
h is computable, therefore primitive recursive.

h can be considered as an **interpreter** for the language of primitive recursive functions.

Remark

If e code for f, then $h(e,n) = f(n)$.
h is prim. rec.
For every primitive recursive function f there exists a code e of f, therefore

$$\forall n.h(e,n) = f(n) \quad ,$$

$$f = \lambda n.h(e,n) \quad .$$

Define

$$f : \mathbb{N} \to \mathbb{N}, \qquad f(n) := h(n,n) + 1 \quad .$$

f is defined in such a way, that it cannot be of the form $\lambda n.h(e,n)$: If it were, we would get

$$h(e,e) + 1 = f(e) = (\lambda n.h(e,n))(e) = h(e,e) \quad .$$

Remark

If e code for f, then $h(e,n) = f(n)$.
h is prim. rec.
$f(n) := h(n,n) + 1$.
$f \neq \lambda n.h(e,n)$.
h is primitive recursive, therefore as well f.
Therefore $h = \lambda n.h(e,n)$ for some e, which cannot be the case.
Therefore we obtain a contradiction.

Need for Partial Functions

- The argument generalises to the following:
- **Remark 6.4** If we have any collection of computable functions such that
  - all functions can be encoded as natural numbers
    - (this means that all functions computable have a finite description)
  - and the equivalent of the function h as defined above is computable,

  then this collection of functions does not contain all computable functions.
- This remark has the same proof as the direct argument for existence of computable functions which are not primitive recursive.

Extension of the above

- By Remark 6.4 it follows that by adding the Ackermann function as basic function to the definition of primitive recursive functions, or any other computable functions, we still won't obtain all computable functions
- **Proof:** If we add any such function, we obtain a collection of functions, for which we can define an encoding of all functions computed by it as natural numbers, s.t. the conditions of the remark are fulfilled.

Extension of the above

- We can read Remark 6.4 as follows:
  - There is no programming language, which computes all computable functions and such that all functions definable in this language are total.
  - **Argument:** If we had such a language, then we could define codes e for programs in this language, and therefore we could define a computable $h : \mathbb{N}^2 \to \mathbb{N}$, s.t.

$$h(e,n) = \begin{cases} f(n), & \text{if } e \text{ is a program in this language} \\ & \text{for a unary function } f, \\ 0, & \text{otherwise.} \end{cases}$$

- By Remark 6.4 not all computable functions are of the form $\lambda n.h(e,n)$ for some e.

Limitations of Prim. Rec. Functs.

We can define prim. rec. functions which

- compute the result of running n steps of a URM or TM,
- check whether a URM or TM has stopped,
- obtain, depending on n arguments the initial configuration for computing $\mathrm{U}^{(n)}$ for a URM U or $\mathrm{T}^{(n)}$ for a TM T,
- extract from a configuration of U and T, in which this machine has stopped, the result obtained by $\mathrm{U}^{(n)}$, $\mathrm{T}^{(n)}$.

Will be done formally for TMs in Subsection 6 (b).

Limitations of Prim. Rec. Functs.

- However, TMs and URMs don't allow to compute an n, s.t. after n steps the URM or TM has stopped.
  - That's the Turing halting problem.
- Extension of prim. rec. functions by adding to the principles of forming functions closure under μ.
- Resulting set called the set of **partial recursive functions.**
  - See Remark 6.4above (omitted in this lecture) for an argument, why partial functions cannot be avoided.
- Using the μ-operator, we will be able to find the least n s.t. a TM or URM stops (and return $\perp$, if no such n exists.)
  - Using this fact we will be able to show that all TM- and URM-computable functions are partial recursive.

Limitations of Prim. Rec. Functs.

- $\mu(f)$ might be partial, even if f is total.
  $\Rightarrow$ Resulting set is set of **partial** functions, therefore name **partial** recursive functions.
- The **recursive** functions will be the partial recursive functions, which are total.

Partial Rec. Functions

Inductive definition of the set of **partial recursive functions** f
together with their **arity**,
i.e. together with the k s.t. $f : \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N}$.
We write "$f : \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N}$ is partial recursive" for "f is partial recursive with arity k", and $\mathbb{N}$ for $\mathbb{N}^1$.

- The following **basic functions** are partial recursive:
  - zero : $\mathbb{N} \overset{\sim}{\rightarrow} \mathbb{N}$,
  - succ : $\mathbb{N} \overset{\sim}{\rightarrow} \mathbb{N}$,
  - $\mathrm{proj}_i^k : \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N}$ $(0 \leq i \leq k)$.

Partial Rec. Functions

- If
  - $g : \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N}$ is partial recursive,
  - for $i = 0, \ldots, k-1$ we have $h_i : \mathbb{N}^n \overset{\sim}{\rightarrow} \mathbb{N}$ is partial recursive,

  then $g \circ (h_0, \ldots, h_{k-1}) : \mathbb{N}^n \overset{\sim}{\rightarrow} \mathbb{N}$ is partial recursive as well.

Partial Rec. Functions

- If
  - $g : \mathbb{N}^n \stackrel{\sim}{\to} \mathbb{N}$,
  - $h : \mathbb{N}^{n+2} \stackrel{\sim}{\to} \mathbb{N}$ are partial recursive,

  then $\mathsf{primrec}(g, h) : \mathbb{N}^{n+1} \stackrel{\sim}{\to} \mathbb{N}$ is partial recursive as well.
- If
  - $k \in \mathbb{N}$,
  - $h : \mathbb{N}^2 \stackrel{\sim}{\to} \mathbb{N}$ is partial recursive,

  then $\mathsf{primrec}(k, h) : \mathbb{N} \stackrel{\sim}{\to} \mathbb{N}$ is partial recursive as well.

Partial Rec. Functions

- If $g : \mathbb{N}^{k+2} \overset{\sim}{\to} \mathbb{N}$ is partial recursive, then $\mu(g) : \mathbb{N}^{k+1} \overset{\sim}{\to} \mathbb{N}$ is partial recursive as well. (Remember that $f := \mu(g)$ has defining equation

$$
\begin{array}{rcl}
f(\vec{x}) & \simeq & \mu y.(g(\vec{x}, y) \simeq 0) \\
& \simeq & \begin{cases} \min\{k \in \mathbb{N} \mid \\ \qquad g(\vec{x}, k) \simeq 0 \\ \qquad \wedge \forall l < k.g(\vec{x}, l) \downarrow\}, & \text{if such a } k \text{ exists,} \\ \bot, & \text{otherwise.)} \end{cases}
\end{array}
$$

Remark

- One can see that if we omit the closure of the partial recursive functions under primrec, but add closure under addition, multiplication, and the characteristic function of equality $\chi_=$ we obtain the set of partial recursive functions. Here

$$\chi_= : \mathbb{N}^2 \longrightarrow \mathbb{N}$$
$$\chi_=(m,n) = \begin{cases} 1 & \text{if } m = n \\ 0 & \text{if } m \neq n \end{cases}$$

- So the partial recursive functions can be defined as the least set of partial functions containing those basic functions, which is closed under composition and μ.
- The proof that this is the case is however rather complicated, see for instance Daniel E. Cohen's book, Sect. 9.5.2, p. 124.

Comparison with Prim. Rec. Functs

- Note that the partial recursive definitions are defined as the primitive recursive functions, but
  - we close them as well under μ;
  - partial recursive functions can be (because of the closure under μ) be partial; therefore all operations under which the partial recursive functions are closed refer to operations on partial functions rather than total functions.
    - Remember that the primitive recursive functions are **total**.

Recursive Functions

(a) A **recursive function** is a partial recursive function, which is total.

(b) A **recursive relation** is a relation $R \subseteq \mathbb{N}^n$ s.t. χ_R is recursive.

Example:

Ack is recursive, but not primitive recursive.

Closure of Part. Rec. Func.

- Every prim. rec. function (relation) is recursive.
- The recursive functions and relations have the same closure properties as those discussed for the prim. rec. functions and relations.

Closure of Part. Rec. Func.

Let for a predicate $U \subseteq \mathbb{N}^{n+1}$

$$\mu z.U(\vec{n}, z) := \begin{cases} \min\{z \mid U(\vec{n}, z)\}, & \text{if such a } z \text{ exists,} \\ \bot, & \text{otherwise.} \end{cases}$$

Then we have that, if U is recursive, then the function

$$f(\vec{n}) :\simeq \mu z.U(\vec{n}, z)$$

is partial recursive as well.

Proof:

$$f(\vec{n}) \simeq \mu z.(\chi_{\neg U}(\vec{n}, z) \simeq 0) \ .$$

(b) Equivalence of the Models of Computatio

- We are going to show Theorem 6.9 (which will be stated below) which expresses that
  - the set of URM-computable functions,
  - the set of TM-computable functions, and
  - the set of partial recursive functions

  coincide.
- The proof will use the round robin method:
  We will show that
  - URM-computable functions are TM-computable,
  - TM-computable functions are partial recursive,
  - partial recursive functions are URM-computable.

Equivalence Models of Computatio

- Show
  - URM-computable functions are TM-computable,
  - TM-computable functions are partial recursive,
  - partial recursive functions are URM-computable.
- Two directions have already been shown:
- By Theorem 4.3 all **URM-computable** functions are **TM-computable**.
- Furthermore, by Lemma 3.3 we obtain immediately
  - **Lemma 6.5** All **partial recursive** functions are **URM computable**.
- Therefore, all we need to show that all **TM-computable** functions are **partial recursive** (will be Corollary 6.8).

TM-computable Implies Part. Rec.

- We will define for TMs T a code $\mathrm{encode}(\mathrm{T}) \in \mathbb{N}$
  - This will be a more explicit encoding than the one given in Sect. 4 (c).
  - Reason for giving the more explicit encoding is that we want to show exlicitly that we can compute from a code for a Turing machine T the function T^n partial recursively.
    - Most auxiliary functions will be defined primitive recursively.
  - We could have used the more complex encoding used here in Sect. 4 (c) as well.
- Essentially we will encode each instruction of a TM (which is a 5 tuple) as a natural number and encode a TM by the code for the sequence of the codes for its instructions.

TM-Computable Implies Part. Rec.

- The functions $\mathrm{encode}(\mathrm{T})$, $\{\cdot\}^n$ will be redefined in this Subsection (and the rest of the lecture) referring to this more explicit encoding.
- This will be done in such a way that every natural number can be interpreted as a code for a Turing machine.

TM-Computable Implies Part. Rec.

- So,we will define for TMs T a (new) code $\mathrm{encode}(\mathrm{T}) \in \mathbb{N}$ and show:
  - For $n \in \mathbb{N}$ there exist partial rec. func.

$$f_n : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N}$$

  s.t. for every TM T we have

$$\forall \vec{m} \in \mathbb{N}^n . f_n(\mathrm{encode}(\mathrm{T}), \vec{m}) \simeq \mathrm{T}^{(n)}(\vec{m}) \quad .$$

- f_n will be called **universal partial recursive functions**. f_n can be seen as an interpreter for TM.
- We will later write $\{e\}^n(\vec{m})$ for $f_n(e, \vec{m})$.

Encoding of TMs

- The definition of f_n will be done using the following steps:
  - We define an encoding of **configurations** of TMs as natural numbers c.
    - The configuration will determine the content of the tape, the position of the head, and the state of the head, at a given time.
  - We define a primitive recursive function

$$\text{next} : \mathbb{N}^2 \to \mathbb{N}$$

    such that,
    - if e is a code for a TM, c a configuration,
    - then $\text{next}(e, c)$ is the configuration obtained after one step of the TM is carried out.
    - If the TM has stopped then $\text{next}(e, c) = c$.

Encoding of TMs

- We define a function

$$\underset{\sim}{\mathsf{init}}^n : \mathbb{N}^n \to \mathbb{N} \enspace ,$$

  such that $\mathsf{init}^n(\vec{m})$ is the configuration of the TM, at the beginning when it is started with arguments $\vec{m}$ initially written on the tape and the head in initial state on the left most bit.
- We define a function

$$\underset{\sim}{\mathsf{extract}} : \mathbb{N} \to \mathbb{N} \enspace ,$$

  such that
  - if c is a configuration in which the TM stops,
  - $\mathsf{extract}(c)$ is the natural number corresponding to the result returned by the TM.

Encoding of TMs

- We define a function

$$\text{iterate} : \mathbb{N}^3 \to \mathbb{N} \ \ ,$$

s.t. $\text{iterate}(e, c, n)$ is the result of iterating TM e with initial configuration c for n steps.

- We define a predicate

$$\text{terminate} \subseteq \mathbb{N}^2$$

s.t. $\text{terminate}(e, c)$ holds, iff the TM e with configuration c stops.

Encoding of TMs

- We define a predicate

$$\underset{\sim\sim\sim}{\mathsf{Terminate}^n}(e, \vec{m}, k)$$

s.t. $\mathsf{Terminate}^n(e, \vec{m}, k)$ holds, iff the TM with code e started with initial arguments $\vec{m}$ terminates after exactly k steps.

Encoding of TMs

- We obtain

$$\begin{array}{ll} & \mathrm{T}^{(n)}(\vec{m}) \\ \simeq & \mathsf{extract}(\mathsf{iterate}(e, \mathsf{init}^{n}(\vec{m})), \mu k.\mathsf{Terminate}^{n}(e, \vec{m}, k)) \\ \simeq & \mathsf{U}'(e, \vec{m}, \mu k.\mathsf{Terminate}^{n}(e, \vec{m}, k)) \end{array}$$

  for some primitive recursive U'.

- We jump over most of the definitions, and give directly the definition of $\mathsf{iterate}$, $\mathsf{Terminate}^{n}$.
  Jump to Definition of $\mathsf{iterate}$

Encoding of TMs

- Several Steps needed for this.
- Use of **Gödel-brackets** in order to denote the code of an object:
  - E.g. $\ulcorner x \urcorner$ for the code of x.
  - Then $\ulcorner x \urcorner$ is called the **Gödel-number** of x.
- Assume encoding $\ulcorner x \urcorner$ for symbols x of the alphabet of a TM s.t.
  - $\ulcorner 0 \urcorner = 0$,
  - $\ulcorner 1 \urcorner = 1$,
  - $\ulcorner \sqcup \urcorner = 2$.

Kurt Gödel

Kurt Gödel (1906 – 1978)
Most important logician of the 20th century.
The use of Gödel numbers in order to encode complex data structures into natural numbers was one of his techniques for proving the famous Gödel's Incompleteness Theorems.
Introduced the recursive functions in his Princeton 1933 – 34 lectures.

Encoding of TM

- We assume an encoding $\ulcorner s \urcorner$ for states s s.t. $\ulcorner s_0 \urcorner = 0$ for the initial state s_0 of the TM.
- We encode the directions in the instructions by
  - $\ulcorner \mathrm{L} \urcorner = 0$,
  - $\ulcorner \mathrm{R} \urcorner = 1$.
- We assume the alphabet consists of $\{0, 1, \sqcup\}$ and symbols occurring in the instructions.
  - $\Rightarrow$ no need to explicitly mention the alphabet in the code for a TM.
- We assume the states consists of $\{s_0\}$ and the states occurring in the instructions.
  - $\Rightarrow$ no need to explicitly mention the set of states in the code for a TM.

Encoding of TM

- No need to mention s_0, $\sqcup$.
  $\Rightarrow$ TM can be identified with its instructions.
- An instruction $\mathrm{I} = (s, a, s', a', D)$ will be encoded as
  $$\mathrm{encode}(\mathrm{I}) = \pi^5(\lceil s \rceil, \lceil a \rceil, \lceil s' \rceil, \lceil a' \rceil, \lceil D \rceil) \enspace .$$
- A set of instructions $\{\mathrm{I}_0, \ldots, \mathrm{I}_{k-1}\}$ will be encoded as
  $$\langle \mathrm{encode}(\mathrm{I}_0), \ldots, \mathrm{encode}(\mathrm{I}_{k-1}) \rangle \enspace .$$
  This is as well $\mathrm{encode}(\mathrm{T})$ of the corresponding TM.

Encoding of a Configuration

Configuration of a TM given by:

- The code $\ulcorner s \urcorner$ of its state s.
- A segment (i.e. a consecutive sequence of cells) of the tape, which includes
  - the cell at head position,
  - all cells which are not blank.

  A segment $a_0, \ldots, a_{n-1}$ is encoded as

  $$\text{encode}(a_0, \ldots, a_{n-1}) := \langle \ulcorner a_0 \urcorner, \ldots, \ulcorner a_{n-1} \urcorner \rangle \ .$$

- The position of the head on this tape.
  Given as a number i s.t. $0 \leq i < n$, if the segment represented is $a_0, \ldots, a_{n-1}$.

Encoding of a Configuration

- A configuration of a TM, given by
  - a state s,
  - a segment $a_0, \ldots, a_{n-1}$,
  - head position i,

  will be encoded as

$$\pi^3(\ulcorner s \urcorner, \mathrm{encode}(a_0, \ldots, a_{n-1}), i) \quad .$$

- As we have seen in Sect. 4, at any time during the computation of a TM, only finitely many cells of the tape are not blank.
  - Therefore at any intermediate step the state of a TM can be encoded as a configuration.

Simulation of TM – symbol, state

$a = \pi^3(\ulcorner s \urcorner, \langle \ulcorner a_0 \urcorner, \ldots, \ulcorner a_{n-1} \urcorner \rangle, i)$.

- We define primitive recursive functions symbol, $\mathsf{tapeseg}$, $\mathsf{position}$, $\mathsf{state} : \mathbb{N} \to \mathbb{N}$,
  which extract
  - the state of the TM $\mathsf{state}(a)$,
  - the segment of the tape $\mathsf{tapeseg}(a)$,
  - the position of the head in that segment $\mathsf{position}(a)$,
  - and the symbol at the head $\mathsf{symbol}(a)$,

  from the current configuration a:

$$\begin{array}{rcl} \mathsf{state}(a) & := & \pi^3_0(a) \\ \mathsf{tapeseg}(a) & := & \pi^3_1(a) \\ \mathsf{position}(a) & := & \pi^3_2(a) \\ \mathsf{symbol}(a) & := & (\mathsf{tapeseg}(a))_{\mathsf{position}(a)} \end{array}$$

Simulation of TM – lookup

- We define a prim. rec. function

$$\text{lookup} : \mathbb{N}^3 \to \mathbb{N} \quad ,$$

s.t. if
  - e is the code for a TM,
  - s state,
  - a a symbol,

then $\text{lookup}(e, s, a)$ is defined as

$$\pi^3(\lceil s' \rceil, \lceil a' \rceil, \lceil D \rceil)$$

where s', a', D are s.t. (s, a, s', a', D) is the first instruction e of the TM corresponding to state s and symbol a.

Simulation of TM – lookup

- If no such instruction exists, the result will be $0 (= \pi^3(0, 0, 0))$.
  lookup is defined as follows:
  - First find using bounded search the index of the first instruction starting with $\ulcorner s \urcorner, \ulcorner a' \urcorner$.
  - Then extract the corresponding values from this instruction.

  The details will be omitted in the lecture.
  Jump over details.

Definition of lookup

Formally the definition is as follows:

- Define an auxiliary primitive recursive function $g : \mathbb{N}^3 \to \mathbb{N}$,

$$g(e, q, a) = \mu i \leq \mathsf{lh}(e).\pi^5_0((e)_i) = q \wedge \pi^5_1((e)_i) = a \quad ,$$

  which finds the index of the first instruction starting with q, a.

- Now define

$$\mathsf{lookup}(e, q, a) := \pi^3(\pi^5_2((e)_{g(e,q,a)}), \pi^5_3((e)_{g(e,q,a)}), \pi^5_4((e)_{g(e,q,a)}))$$

Simulation of TM – hasinstruction

- There exists a primitive recursive relation $\mathsf{hasinstruction} \subseteq \mathbb{N}^3$, s.t.
  - $\mathsf{hasinstruction}(e, \lceil s \rceil, \lceil a \rceil)$ holds, iff the TM corresponding to e has a instruction (s, a, s', a', D) for some s', a', D.
  - $\mathsf{hasinstruction}$ is defined as a byproduct of defining lookup.
  - The details will be omitted in the lecture. Jump over details.

Definition of $\mathsf{hasinstruction}$

$\mathsf{hasinstruction}$ is defined, using the function g from the previous item, as

$$\chi_{\mathsf{hasinstruction}}(e,q,a) = \mathsf{sig}(\mathsf{lh}(e) \dot{-} g(e,q,a)) \ .$$

If there is such an instruction,

$$g(e,q,a) < \mathsf{lh}(e) \ ,$$

therefore

$$\mathsf{lh}(e) \dot{-} g(e,q,a) > 0 \ .$$

Otherwise

$$\begin{aligned} g(e,q,a) &= \mathsf{lh}(e) \ , \\ \mathsf{lh}(e) \dot{-} g(e,q,a) &= 0 \ . \end{aligned}$$

Simulation of TM – next

- There exists prim rec. function

$$\mathsf{next} : \mathbb{N}^2 \to \mathbb{N}$$

  s.t.
  - if e encodes a TM
  - and c is a code for a configuration,

  then
  - $\mathsf{next}(e, c)$ is a code for the configuration after one step of the TM is executed,
  - or equal to c, if the TM has halted.

Jump over details.

Simulation of TM – next

- Informal description of next:
  - Assume the configuration is

    $$c = \pi^3(s, \langle a_0, \ldots, a_{n-1} \rangle, i) \quad .$$

  - Check using hasinstruction, whether the TM has stopped.
    If it has stopped, return c.
  - Otherwise, use lookup to obtain the codes for
    - the next state s',
    - next symbol a',
    - direction D.
  - Replace in $\langle a_0, \ldots, a_{n-1} \rangle$, the ith element by a'.
    Let the result be x.

Simulation of TM – next

Next state is s', next Symbol is a', direction is D.
x = result of substituting the symbol in original segment.

- Might be that head will leave segment.
  - Then extend segment.
- Case $D = \ulcorner L \urcorner$ and $i = 0$:
  Extend the segment to the left by one blank.
  Result of next is

$$\pi^3(s', \langle \ulcorner \sqcup \urcorner \rangle * x, 0) \ .$$

- Case $D \neq \ulcorner L \urcorner, i \geq n - 1$.
  Result of next is

$$\pi^3(s', x * \langle \ulcorner \sqcup \urcorner \rangle, n) \ .$$

Simulation of TM – next

Next state is s', next Symbol is a', direction is D.
x = result of substituting the symbol in original segment.

- Otherwise let
  - $i' := i - 1$, if $D = \lceil L \rceil$,
  - $i' := i + 1$, if $D \neq \lceil L \rceil$.

  Result of next is

$$\pi^3(s', x, i') \ .$$

Simulation of TM – terminate

- There exists a prim. rec. predicate $\mathsf{terminate} \subseteq \mathbb{N}^2$, s.t. $\mathsf{terminate}(e, c)$ holds, iff the TM e with configuration c stops:

$$\mathsf{terminate}(e, c) :\Leftrightarrow \neg\mathsf{hasinstruction}(e, \mathsf{state}(c), \mathsf{symbol}(c)) \quad .$$

Simulation of TM – init

- There exists a primitive recursive function $\text{init}^n : \mathbb{N}^n \to \mathbb{N}$,
s.t. for any TM T containing alphabet $\{0, 1, \sqcup\}$
$\text{init}^n(\vec{m})$ is the initial configuration for computing $\mathrm{T}^{(n)}(\vec{m})$
for any TM T, the alphabet of which contains $\{0, 1, \sqcup\}$:

$$\begin{aligned}\text{init}^n&(m_0, \ldots, m_{n-1})\\ &= \pi^3(\ulcorner s_0 \urcorner, \text{bin}(m_0) * \langle \ulcorner \sqcup \urcorner \rangle * \text{bin}(m_1) * \langle \ulcorner \sqcup \urcorner \rangle * \cdots\\ &\qquad * \langle \ulcorner \sqcup \urcorner \rangle * \text{bin}(m_{n-1}), 0) \quad .\end{aligned}$$

Simulation of TM – extract

- There exists a primitive recursive function $\mathsf{extract} : \mathbb{N} \to \mathbb{N}$, s.t.
  - if c is a configuration in which the TM stops,
  - $\mathsf{extract}(c)$ is the natural number corresponding to the result returned by the TM.
- Assume $c = \pi^3(s, x, i)$.
- $\mathsf{extract}(c)$ obtained by applying bin^{-1} to the largest subsequence of x starting at i consisting of 0 and 1 only.

Simulation of TM – iterate

- There exists a primitive recursive function
  $\mathsf{iterate} : \mathbb{N}^3 \to \mathbb{N}$,
  s.t. $\mathsf{iterate}(e, c, n)$ is the result of iterating TM e with initial configuration c for n steps.
  The definition is as follows

$$\begin{aligned}\mathsf{iterate}(e, c, 0) &= c \quad ,\\ \mathsf{iterate}(e, c, n+1) &= \mathsf{next}(e, \mathsf{iterate}(e, c, n)) \quad .\end{aligned}$$

Simulation of TM – Terminate

- Let

$$\begin{aligned}&\mathsf{Terminate}^n(e,\vec{m},k) :\\ &\Leftrightarrow \mathsf{terminate}(e,\mathsf{iterate}(e,\mathsf{init}^n(\vec{m}),k))\wedge\\ &\quad\forall k' < k.\neg\mathsf{terminate}(e,\mathsf{iterate}(e,\mathsf{init}^n(\vec{m}),k'))\end{aligned}$$

  which is primitive recursive.

- Then, if e encodes a TM T,
  - then $\mathsf{Terminate}^n(e,\vec{m},k)$ holds if the computation of $\mathrm{T}^{(n)}(\vec{m})$ stops for the first time after k steps.
  - Note that there exists at most one k s.t. $\mathsf{Terminate}^n(e,\vec{m},k)$ holds.

Simulation of TM – Terminate

- As mentioned before, we obtain

$$\begin{aligned} & \mathrm{T}^{(n)}(\vec{m}) \\ \simeq\quad & \mathsf{extract}(\mathsf{iterate}(e, \mathsf{init}^n(\vec{m})), \mu k.\mathsf{Terminate}^n(e, \vec{m}, k)) \\ \simeq\quad & \mathsf{U}'(e, \vec{m}, \mu k.\mathsf{Terminate}^n(e, \vec{m}, k)) \end{aligned}$$

for some primitive recursive U'.

Definition of $\mathsf{T}^n_{\mathsf{Kleene}}$

$$\mathrm{T}^{(n)}(\vec{m}) \simeq \mathsf{U}'(e, \vec{m}, \mu k.\mathsf{Terminate}^n(e, \vec{m}, k))$$

- Kleene wanted to replace U' by some primitive recursive $\mathsf{U}_{\mathsf{Kleene}}$ which has only one argument.
- This can be achieved by searching instead of for the minimal k s.t.

  $$\mathsf{Terminate}^n(e, \vec{m}, k)$$

  for the minimal k s.t. the following predicate holds:

  $$\begin{aligned}&\mathsf{T}^n_{\mathsf{Kleene}}(e, \vec{m}, k)\\ &:= \mathsf{Terminate}^n(e, \vec{m}, \pi_0(k))\\ &\quad \wedge \mathsf{extract}(\mathsf{iterate}(e, \mathsf{init}^n(\vec{m})), \pi_0(k)) = \pi_1(k)\end{aligned}$$

Definition of $\mathsf{T}^{n}_{\mathsf{Kleene}}$

$$
\begin{aligned}
&\mathsf{T}^{n}_{\mathsf{Kleene}}(e, \vec{m}, k) \\
&:= \mathsf{Terminate}^{n}(e, \vec{m}, \pi_0(k)) \\
&\quad \wedge \mathsf{extract}(\mathsf{iterate}(e, \mathsf{init}^{n}(\vec{m})), \pi_0(k)) = \pi_1(k)
\end{aligned}
$$

- We obtain that if $\mathrm{encode}(\mathrm{T}) = e$, then

$$
\begin{aligned}
\mathrm{T}^{(n)}(\vec{m}) &\simeq \pi_1(\mu k.\mathsf{T}^{n}(e, \vec{m}, k)) \\
&\simeq \mathsf{U}_{\mathsf{Kleene}}(\mu k.\mathsf{T}^{n}(e, \vec{m}, k))
\end{aligned}
$$

where

$$\mathsf{U}_{\mathsf{Kleene}}(k) := \pi_1(k)$$

Kleene's NF Theorem

Kleene's Normal Form Theorem 6.6:

(a) There exists partial recursive functions $f_n : \mathbb{N}^{n+1} \xrightarrow{\sim} \mathbb{N}$ s.t.
if e is the code of TM T then

$$f_n(e, \vec{n}) \simeq \mathrm{T}^{(n)}(\vec{n}) \quad .$$

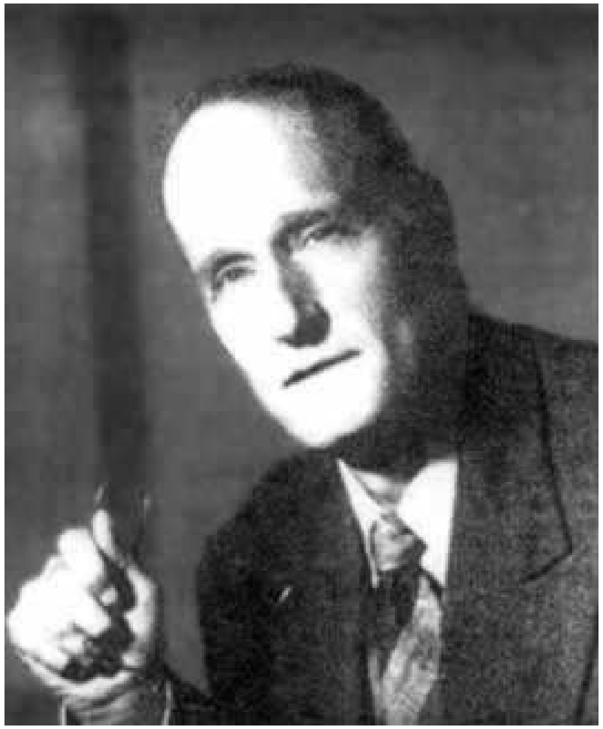
(b) There exist

- a primitive recursive function $\mathsf{U}_{\mathsf{Kleene}} : \mathbb{N} \to \mathbb{N}$,
- primitive recursive predicate $\mathsf{T}^n_{\mathsf{Kleene}} \subseteq \mathbb{N}^{n+2}$,

s.t. for the function f_n introduced in (a) we have

$$f_n(e, \vec{n}) \simeq \mathsf{U}_{\mathsf{Kleene}}(\mu z.\mathsf{T}^n_{\mathsf{Kleene}}(e, \vec{m}, z)) \quad .$$

Stephen Cole Kleene



Stephen Cole Kleene
(1909 – 1994)
Probably the most influential computability theorist up to now.
Introduced the partial recursive functions.

Definition 6.7

- Let $\mathsf{U}_{\mathsf{Kleene}}$, $\mathsf{T}^n_{\mathsf{Kleene}}$ as in Kleene's Normal Form Theorem 6.6.
  Then we define

$$\{e\}^n(\vec{m}) \simeq \mathsf{U}_{\mathsf{Kleene}}(\mu y.\mathsf{T}^n_{\mathsf{Kleene}}(e,\vec{m},y)) \quad .$$

  (This defines essentially the same function as the one defined in Section 4 (b), except that we refer now to the new encoding of Turing Machines.)

- So by Kleene's NF theorem, if $\mathrm{encode}(\mathrm{T}) = e$,then

$$\mathrm{T}^{(n)}(\vec{m}) \simeq \{e\}^n(\vec{m}) \quad .$$

Notation

- We will often omit the superscript n in $\{e\}^n(m_0, \ldots, m_{n-1})$
  - i.e. write $\{e\}(m_0, \ldots, m_{n-1})$ instead of $\{e\}^n(m_0, \ldots, m_{n-1})$.
- Further $\{e\}$ not applied to arguments and without superscript means usually $\{e\}^1$.

Remark

- The operations for extracting instructions from a TM above were primitive recursive and therefore total.
- Therefore, even if e is not a code for a TM, e will be treated as if were a code for a TM, namely the one with the instructions computed by the above operations.
- Omit details

Details of the Prev. Remark

- A code for such a valid TM is obtained by
  - treating e as a sequence of instructions,
  - deleting from it any $\mathrm{encode}(q, a, q', a', D)$ s.t. there exists an instruction $\mathrm{encode}(q, a, q'', a'', D')$ occurring before it,
  - and by replacing all directions > 1 by $\ulcorner \mathrm{R} \urcorner = 1$.

Corollary 6.8

(a) For every TM-computable function $f : \mathbb{N}^n \overset{\sim}{\to} \mathbb{N}$ there exists $e \in \mathbb{N}$ s.t.

$$f = \{e\}^n \quad .$$

(b) Especially, all TM-computable functions are partial recursive.

Theorem 6.9

The following sets coincide:

- the set of URM-computable functions,
- the set of Turing-computable functions,
- the set of partial recursive functions.

Proof:

- URM-computable implies TM-computable by Theorem 4.3.
- TM-computable implies partial recursive by Corollary 6.8.
- Partial recursive implies URM computable by Lemma 6.5.

Theorem 6.10

> The partial recursive functions $g : \mathbb{N}^n \overset{\sim}{\to} \mathbb{N}$ are exactly the functions $\{e\}^n$ for $e \in \mathbb{N}$.

Proof:

- By Kleene's Normal Form Theorem every TM-computable function $g : \mathbb{N}^n \overset{\sim}{\to} \mathbb{N}$ is of the form $\{e\}^n$.
- The TM-computable functions are the partial rec. functions.
- Therefore the assertion follows.

Remark

- Theorem 6.10 means:
  - $\{e\}^n$ is partial recursive for every $e \in \mathbb{N}$.
  - For every partial recursive function $g : \mathbb{N}^n \overset{\sim}{\to} \mathbb{N}$ there exists an e s.t. $g = \{e\}^n$.
- The e s.t. $g = \{e\}^n$ for $g : \mathbb{N}^n \overset{\sim}{\to} \mathbb{N}$ is called the **Kleene-index** of g.

Remark

- Therefore

$$f_n : \mathbb{N}^{n+1} \overset{\sim}{\to} \mathbb{N} \ , \qquad f_n(e, \vec{x}) \simeq \{e\}^n(\vec{x})$$

  forms a **universal n-ary partial recursive function**:
  It encodes all n-ary partial recursive function.

- So we can assign to each partial recursive function g a number, namely an e s.t. $g = \{e\}^n$.
  - Each number e denotes one partial recursive function $\{e\}^n$.
  - However, several numbers denote the same partial recursive function:
    There are e, e' s.t. $e \neq e'$ but $\{e\}^n = \{e'\}^n$.

Proof of Last Fact

Proof, that different e compute the same function:

- There are several algorithms for computing the same function.
- Therefore there are several Turing machines which compute the same function.
- These Turing machines have different codes e.

Lemma 6.11

The set F of partial recursive functions

(and therefore as well of Turing-computable and of URM-computable functions),

i.e.

$$F := \{f : \mathbb{N}^k \overset{\sim}{\rightarrow} \mathbb{N} \mid k \in \mathbb{N} \wedge f \text{ partial recursive}\}$$

is countable.

Proof of Lemma 6.11

Every partial recursive function is of the form $\{e\}^n$ for some $e, n \in \mathbb{N}$.

Therefore

$$f : \mathbb{N}^2 \to F \;, \qquad f(e, n) := \{e\}^n$$

is surjective.

$\mathbb{N}^2$ is countable.
Therefore by Corollary 2.15 (a), F is countable as well.
In fact

$$F = \{\{e\}^k \mid e, k \in \mathbb{N}\}$$

(c) The Church-Turing Thesis

We have introduced three models of computations:

- The URM-computable functions.
- The Turing-computable functions.
- The partial recursive functions.

Further we have shown that all three models compute the same partial functions.

The Church-Turing Thesis

Lots of other models of computation have been studied:

- The while programs.
- Symbol manipulation systems by Post and by Markov.
- Equational calculi by Kleene and by Gödel.
- The λ-definable functions.
- Any of the programming languages Pascal, C, C++, Java, Prolog, Haskell, ML (and many more).
- Lots of other models of computation.

The Church-Turing Thesis

- One can show that the partial functions computable in these models of computation are again exactly the partial recursive functions.
- So all these attempts to define a complete model of computation result in the same set of partial recursive functions.
- Therefore we arrive at the Church Turing Thesis

The Church-Turing Thesis

Church-Turing Thesis: *The (in an intuitive sense) computable partial functions are exactly the partial recursive functions (or equivalently the URM-computable or Turing-computable functions).*

Philosophical Thesis

- This thesis is **not a mathematical theorem**.
- It is a **philosophical thesis**.
- Therefore the Church-Turing thesis **cannot be proven**.
- We can only provide **philosophical evidence** for it.
- This evidence comes from the following **considerations and empirical facts**:

Empirical Facts

- All complete models of computation suggested by researchers define the same set of partial functions.
- Many of these models were carefully designed in order to capture intuitive notions of computability:
  - The Turing machine model captures the intuitive notion of **computation on a piece of paper** in a general sense.
  - The URM machine model captures the general notion of **computability by a computer**.
  - Symbolic manipulation systems capture the general notion of computability by **manipulation of symbolic strings**.

Empirical Facts

- No intuitively computable partial function, which is not partial recursive, has been found, despite lots of researchers trying it.
- A strong intuition has been developed that in principal programs in any programming language can be simulated by Turing machines and URMs.

Because of this, only few researchers doubt the correctness of the Church-Turing thesis.

Decidable Sets

- A predicate A is URM-/Turing-decidable iff χ_A is URM-/Turing-computable.
- A predicate A is decidable iff χ_A is computable.
- By Church's thesis to be computable is the same as to be URM-computable or to be Turing-computable.
- So the decidable predicates are exactly the URM-decidable and exactly the Turing-decidable predicates.

Halting Problem

- Because of the equivalence of the 3 models of computation, the halting problem for any of the above mentioned models of computation is undecidable.
- Especially it is undecidable, whether a program in one of the programming languages mentioned terminates:
  - Assume we had a decision procedure for deciding whether or not say a Pascal program terminates for given input.
  - Then we could, using a translation of URMs into Pascal programs, decide the halting problem for URMs, which is impossible.