Theoretical Computer Science

Computability

Technische Hochschule Rosenheim
Sommer 2022
Prof. Dr. Jochen Schmidt

Overview

- Decision problems and Church-Turing thesis
- Halting problem
- LOOP/WHILE/GOTO computability
- Primitive recursive functions
- μ-recursive functions and Ackermann function
- Busy-Beaver function

From a Problem to Program Execution

This chapter:
Decidability

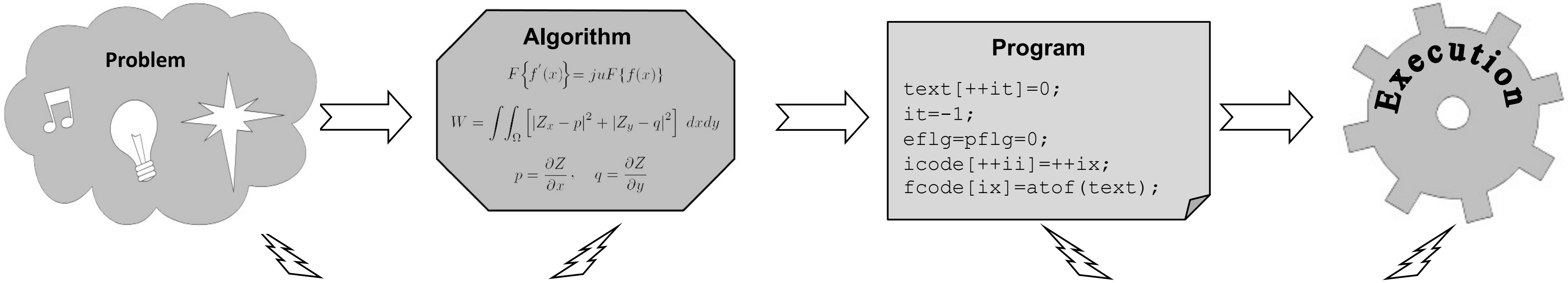
Can we construct an algorithm for any given problem?
At least in principle?

This chapter:
LOOP/WHILE, primitive/μ recursion

Can we write a computer program for any algorithm?
What are the minimum requirements for programming languages?

Can we translate any program to machine language?

Previous chapter:
Word problem/parsing



Computability
Turing Machines
Church-Turing Thesis
Halting problem
Formal Languages
LOOP/WHILE programs
Complexity Theory

What are the time or memory requirements? How do these grow depending on the amount of data processed?

Next chapter:
Complexity theory

Decision Problems and Church-Turing Thesis

Reminder: Universal Turing Machine

- Universal Turing Machine (UTM) = TM that can simulate any other TM
  - A computer is basically a universal TM
  - Construction described by Alan Turing in 1936
- Therefore, any algorithm can be described as a TM and be executed by a universal TM
- A system that can simulate any TM is called Turing-complete (*Turing-vollständig*)

- Is there anything more powerful than a Turing Machine?
  - No one has found any concept that is more powerful (= can solve more problems)
  - And not for lack of trying – researchers have looked at many very different concepts

Church-Turing Thesis

- Any algorithm can be represented as
  - Turing Machine („Turing-computable“, *Turing-berechenbar*)
  - Formal Language (Type 0)
  - Register machine (including random-access machines)
  - µ-recursive function
  - WHILE or GOTO program
  - …
- **all** these representations were proven to be equivalent!

Church-Turing Thesis

The class of functions captured by the formal definition of Turing computability exactly matches the class of intuitively computable functions.

Church-Turing Thesis

- **Thesis**: not provable, but generally accepted
- Indications of correctness
  - no one has been able to find a more comprehensive concept of computability than that of TM.
  - the equivalence of many different formalisms is a strong indication that with the TM we have actually found the concept of computability itself.

Consequence:
If a function is proven not to be Turing-computable, it is not computable at all.

Gödel's Incompleteness Theorems

Technische Hochschule Rosenheim

- Kurt Gödel (1906 – 1978)
- Opinion before Gödel: every mathematical statement is algorithmically decidable
  - i.e., in principle, one can prove whether it is true or false

- Incompleteness Theorems (Gödel 1931)
  1. Any sufficiently powerful, recursively enumerable formal system is either contradictory or incomplete.
  2. Any sufficiently powerful consistent formal system cannot prove its own consistency.

  - Originally: Proof that all consistent axiomatic formulations of number theory contain undecidable statements. There are statements that can neither be proved not disproved
  - So, not every statement is algorithmically decidable
  - Therefore, there exist problems that in principle cannot be solved by computers

Computability/Decidability

- Computable functions
  - Function f: $\mathbb{N}^k \rightarrow \mathbb{N}$ is said to be computable (*berechenbar*) if there exists an algorithm that calculates f($\boldsymbol{x}$) for an input $\boldsymbol{x} \in \mathbb{N}^k$.
  - Note: The term “algorithm” implies that the computations stops after a finite number of steps. The function value is then, e.g., the output (= contents of the tape) of a Turing Machine.

- Decidability
  - A set M is said to be decidable (*entscheidbar*), if its characteristic function $\chi(m)$ is computable.
  - $\chi(m)$ calculates whether an element m is contained in the set M or not:

$$\chi(m) = \begin{cases} 1 & \text{if } m \in M \\ 0 & \text{otherwise} \end{cases}$$

Non-computable Functions

- An algorithm
  - is represented using an alphabet Σ with a finite number of symbols
  - in the case of a TM, it can be proven that the binary alphabet is sufficient $\Sigma = \{0, 1\}$
  - has finite length
- So, an algorithm is a string built of symbols from Σ
- The set Σ^* contains all such strings and is countable (enumerable, *abzählbar*)

- Therefore, there are only countable many algorithms, i.e., all algorithms could in principle be numbered consecutively using the natural numbers
  - there is only a finite number of sources code of a given length in any programming language
  - in principle, you could write them all down

Non-computable Functions – Proof

- There are non-computable functions
- Consider the set of arithmetic functions f(n): $\mathbb{N} \to \mathbb{N}$
  it is already uncountable (*überabzählbar*)
- Proof
  - Proposition: The set f(n), $n \in \mathbb{N}$ is countable (and thus completely computable)
  - Then we can order the functions as in the following table:

	1	**2**	**3**	**4**	**…**
$\mathbf{f_1}$	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	…
$\mathbf{f_2}$	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	…
$\mathbf{f_3}$	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	…
$\mathbf{f_4}$	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	…
…	…	…	…	…	…

Non-computable Functions – Proof

- Construct a function g as follows
  - $g(1) = f_1(1) + 1$ thus, g differs from f_1
  - $g(2) = f_2(2) + 1$ thus, g differs from f_2
  - $g(3) = f_3(3) + 1$ thus, g differs from f_3
  - …
- g differs from all functions f_i
- g is obviously computable
- so g should be included in the table
- but this is not the case

- Conclusion: Contradiction – The assumption that we the table contains all functions $f(n)\colon \mathbb{N} \rightarrow \mathbb{N}$ is wrong. The arithmetic functions are uncountable.

Non-computable Functions – Conclusion

- Non-computable functions do exist
- There are uncountably many arithmetic functions
  - Out of these, only countably many are computable at most
  - The set is infinitely larger than the set of computable functions and therefore algorithms
- Compared to what a computer can **not** do, what it can do is negligibly small…

- Note:
  - Non-computable does not mean that there are problems for which simply no algorithm has been found yet.

  - It means: There are problems for which there can in principle exist no algorithm to solve them,
  - independent of the future development of computer hardware.

Halting Problem

Halting Problem (*Halteproblem*)

- Most important example of an undecidable problem in computer science
- Question:
  Is there a Turing Machine (or a program) HALT that can determine for any program P and its input whether it will ever stop running or not?

- Calling HALT(P) would give:
  - P halts eventually
  - P does halt

  without having to run P itself.
- HALT could therefore check whether a program P will get into an infinite loop.

Halting Problem: The Philosopher's Stone

- The halting problem is of paramount significance
- If it were decidable, we would have a philosopher's stone with which we could immediately solve all problems that can be formulated as a program.

- Example – Goldbach's conjecture:
  Any even integer $g > 2$ can be represented as the sum of two prime numbers.
  - so far unproven – has been shown to be correct for all $g < 2 \cdot 10^{18}$
  - write program that tests all even numbers g by trying whether g is the sum of two prime numbers
  - the program stops if this does not apply to a particular g
  - if Goldbach’s conjecture is true: The program will never stop
  - but this could be tested in advance by HALT(GOLDBACH) – we don’t have to run our program
  - this would enable us to prove or refute Goldbach's conjecture

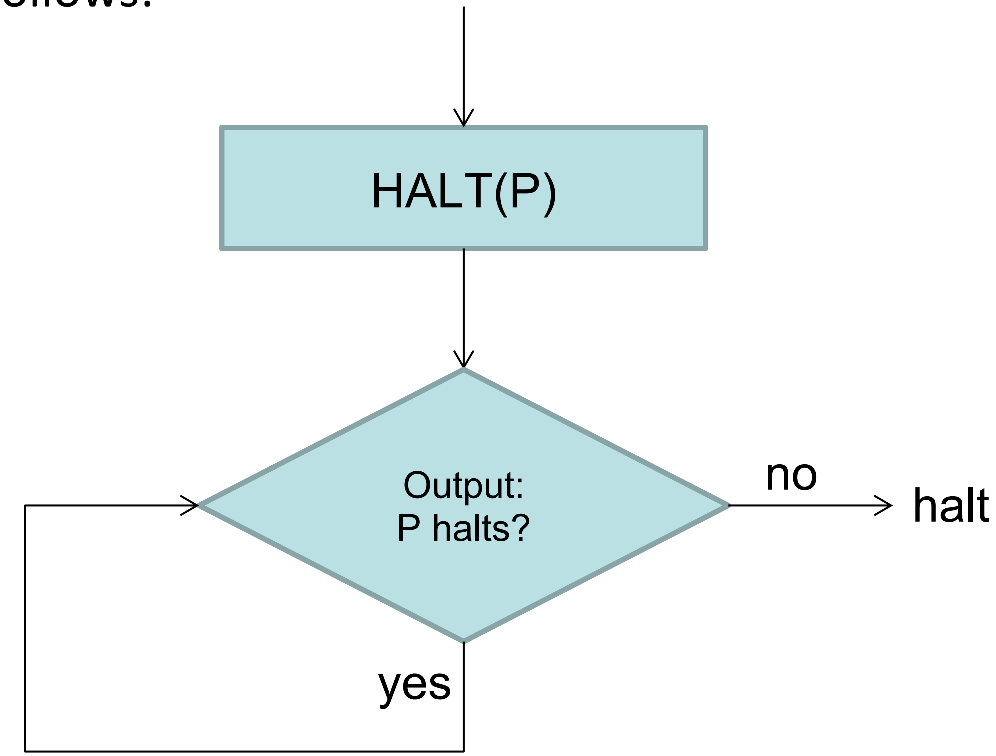
Proof – Special Halting Problem

- Proposition: There exists an algorithm to solve the halting problem
- So there is a program HALT
  - Input:
    - any program P to be tested
    - including its input data
  - Output: P „halts“ or „does not halt“

- For any input data for P: General halting problem (*allgemeines Halteproblem*)
- P uses its own code as input: Special halting problem (*spezielles Halteproblem*)

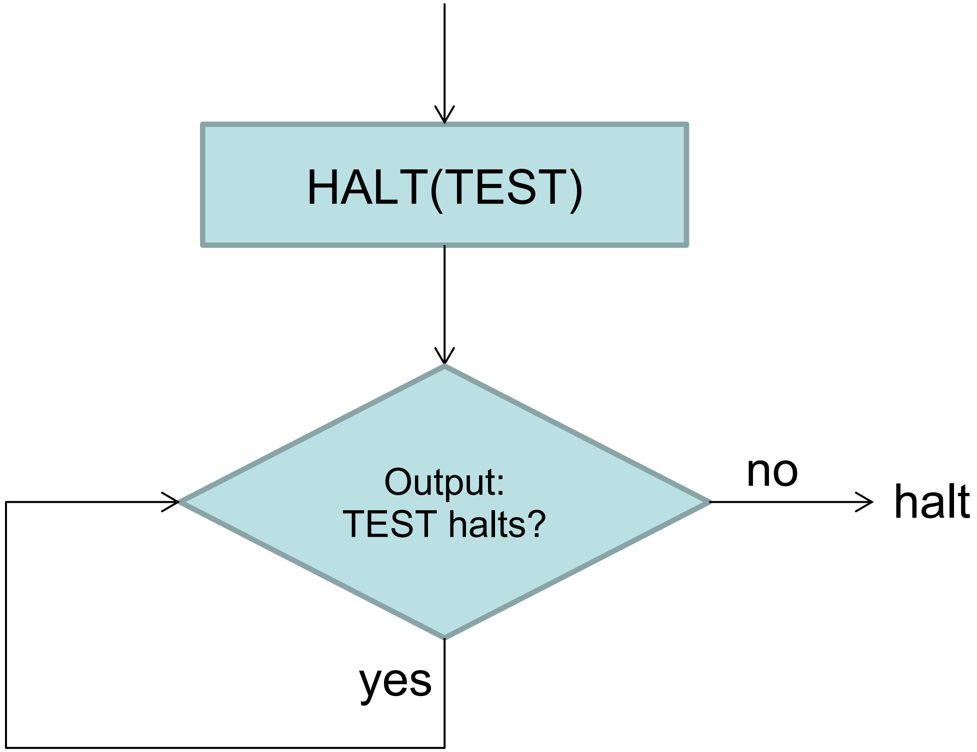
Proof – Special Halting Problem

Construct a program TEST as follows:



Proof – Special Halting Problem

Now: P = TEST (TEST uses itself as an input):



- 2 Cases
  - TEST(TEST) halts
    - Output of HALT(TEST):
      TEST does not halt
  - TEST(TEST) does not halt
    - Output of HALT(TEST):
      TEST halts
- Contradiction!
- Conclusion:
  - **HALT does not exist!**
  - **The special halting problem is undecidable**

General Halting Problem

- Proof of many other undecidable problems is possible by reduction to the special holding problem
  - i.e., embedding the special holding problem as a special case in the new problem
  - then the more general problem must be alle the more undecidable

- General halting problem
  - Decide whether P halts with any input
  - Reduction obvious: already undecidable with special case P as input
  - **The general halting problem is undecidable**

Blank Tape Halting Problem

- Decide whether P halts when TM is started on an empty tape (i.e., with no input)
- Reduction:
  - after starting, first step: write code of P to tape
  - then behavior as in case of special halting problem
- **The blank tape halting problem is undecidable**

Other undecidable problems & Rice's Theorem

- Do two TM/programs calculate the same function?
  - Equivalence problem
  - cannot be reduced to the halting problem: even more undecidable, requires separate proof
- Does TM/program calculate a constant function?

- Rice's theorem (1953):
  It is hopeless to try to algorithmically determine any aspect of the functional behavior of a TM – **all non-trivial properties of a TM/algorithm/program are undecidable**.
  - this can actually be proven!
  - and of course it extends to all other equivalent representations, like type 0 grammars.

Selected Problems on Formal Languages

- Emptiness problem (*Leerheitsproblem*)
  - Given: Grammar G (or equivalent automaton)
  - Question: is $L(G) = \emptyset$?

- Intersection non-emptiness problem (*Schnittproblem*)
  - Given: two grammars G_1 and G_2 (or equivalent automata)
  - Question: is $L(G_1) \cap L(G_2) = \emptyset$?

- Equivalence problem (*Äquivalenzproblem*)
  - Given: two grammars G_1 and G_2 (or equivalent automata)
  - Question: do G_1 and G_2 denote the same formal language, i.e., is $L(G_1) = L(G_2)$?

Word/Emptiness/Intersection/Equivalence Problems

- For which language classes/automata models is the problem decidable (solvable)?
- Table entries
  - yes: there is an algorithm that solves the problem
  - no: the problem is undecidable – there exists no general algorithm to solve it (and **never** will)

Language	Word problem	Emptiness problem	Equivalence problem	Intersection problem
Type 3	yes	yes	yes	yes
det.cf.	yes	yes	yes	no
Type 2	yes	yes	no	no
Type 1	yes	no	no	no
Type 0	no	no	no	no

Selected Problems on Formal Languages

Technische Hochschule **Rosenheim**

G, G_1 and G_2: context-free grammars

Undecidable are, e.g.,

- is $\overline{L(G)}$ context-free?
- is $L(G)$ deterministic context-free?
- is $L(G)$ regular?
- is $L(G_1) \cap L(G_2) = \emptyset$?
- is $L(G_1) \cap L(G_2)$ context-free?
- is $| L(G_1) \cap L(G_2) | = \infty$?
- is $L(G_1) \subseteq L(G_2)$?
- is $L(G_1) = L(G_2)$?

G_1, G_2: deterministic context-free grammars

Undecidable are, e.g.,

- is $L(G_1) \cap L(G_2) = \emptyset$?
- is $L(G_1) \cap L(G_2)$ context-free?
- is $| L(G_1) \cap L(G_2) | = \infty$?
- is $L(G_1) \subseteq L(G_2)$?

Undecidable Problems – Pitfalls

- Undecidable problems cannot be solved in principle – this is not a matter of hardware

- Undecidable means: There is no single, general algorithm to solve the problem
  - There may be many special cases where we can determine, e.g., whether a program halts, e.g.,
    - any program not containing any loops, jumps, or recursion will always halt.
    - any accepting finite automaton will always halt.
  - We may also be able to develop algorithms that can decide for a specific program at hand – this will never be a general algorithm, however, but work only for this case or a class of cases.

LOOP/WHILE/GOTO Computability

LOOP Programs

- Simple programming language
- Components
  - Variables: $x_0, x_1, x_2, x_3, \ldots$
  - Constants: 0, 1, 2, …
  - Separators: ; :=
  - Operators: + –
  - Keywords: LOOP DO END
- Syntax
  - $\mathbf{x_i := x_j + c}$ and $\mathbf{x_i := x_j - c}$
    are LOOP programs (where c is a constant)
  - if P_1 and P_2 are LOOP programs then so is
    $\mathbf{P_1 ; P_2}$
  - if P is a LOOP program and x_i a variable then
    LOOP $\mathbf{x_i}$ DO P END
    is also a LOOP program

- Semantics
  - program starts with parameters in the variables $x_1, \ldots, x_n$
  - all others are initialized with 0
  - only natural numbers are allowed
  - x_0 contains calculation result at the end
  - Assignments
    - + as usual
    - –: if result would become less than zero the value is set to zero
  - $P_1 ; P_2$ means: execute P_1, then P_2
  - LOOP x_i DO P END means
    - P is exexuted x_i times
    - Changing the variable in the loop has no effect

Notes on Natural Numbers in LOOP

- the sole use of natural numbers is not a restriction
- any alphabet can be mapped to the natural numbers
- as well as any rational number
- not: real numbers – but a TM/computer cannot process these anyway

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

LOOP Programs – Properties

- all LOOP-computable functions are total functions
  - the reverse does not apply: Ackermann function
- any LOOP program always halts in finite time
- Extension of assignments
  - $x_i := c$ is possible by $x_i := x_j + c$ if we choose an x_j that still has the values zero
  - $x_i := x_j$ by choosing $c = 0$
- IF-THEN
  - IF $x_i = 0$ THEN P END can be implemented by
  - $x_j := 1$;
LOOP x_i DO $x_j := 0$ END;
LOOP x_j DO P END

LOOP Programs – Example: Addition

Addition is LOOP-computable: $x_0 := x_1 + x_2$

$x_0 := x_1;$
LOOP x_2 DO $x_0 := x_0 + 1$ END

WHILE Programs

- Extension of the LOOP syntax by
  - if P is a WHILE program and x_i a variable then
    WHILE $x_i \neq 0$ DO P END
    is also a WHILE program
- Semantics:
  Execute P as long as the variable value is not zero

- Note: Strictly speaking, LOOP is no longer required (but we keep it)
  - LOOP x DO P END can be implemented as (variables renamed)
  - y := x;
    WHILE $y \neq 0$ DO y := y – 1; P END

WHILE Programs

- Partial functions can now also be described
  - Infinite loops are possible

- Any WHILE-computable function is also Turing-computable
  - TM can simulate WHILE programs
  - the reverse is also true

- For any program, a single WHILE loop is sufficient – proof follows

GOTO Programs

- Sequence of statements S with labels M
  $L_1 : S_1 ; L_2 : S_2 ; \ldots; L_n : S_n$

- Allowed statements:
  - Assignment: $x_i := x_j + c$ or $x_i := x_j - c$
  - Unconditional branch: GOTO L_i
  - Branch on condition: IF $x_i = c$ THEN GOTO L_i
  - Stop program: HALT

- Any GOTO-computable function is also WHILE-computable and vice versa

WHILE Described by GOTO

WHILE $x_i \neq 0$ DO P END

is equivalent to

L_1 : IF $x_i = 0$ THEN GOTO L_2;
P;
GOTO L_1 ;
L_2 : …

GOTO Described by WHILE

- GOTO Program: $L_1 : S_1 ; L_2 : S_2 ; \ldots; L_n : S_n$

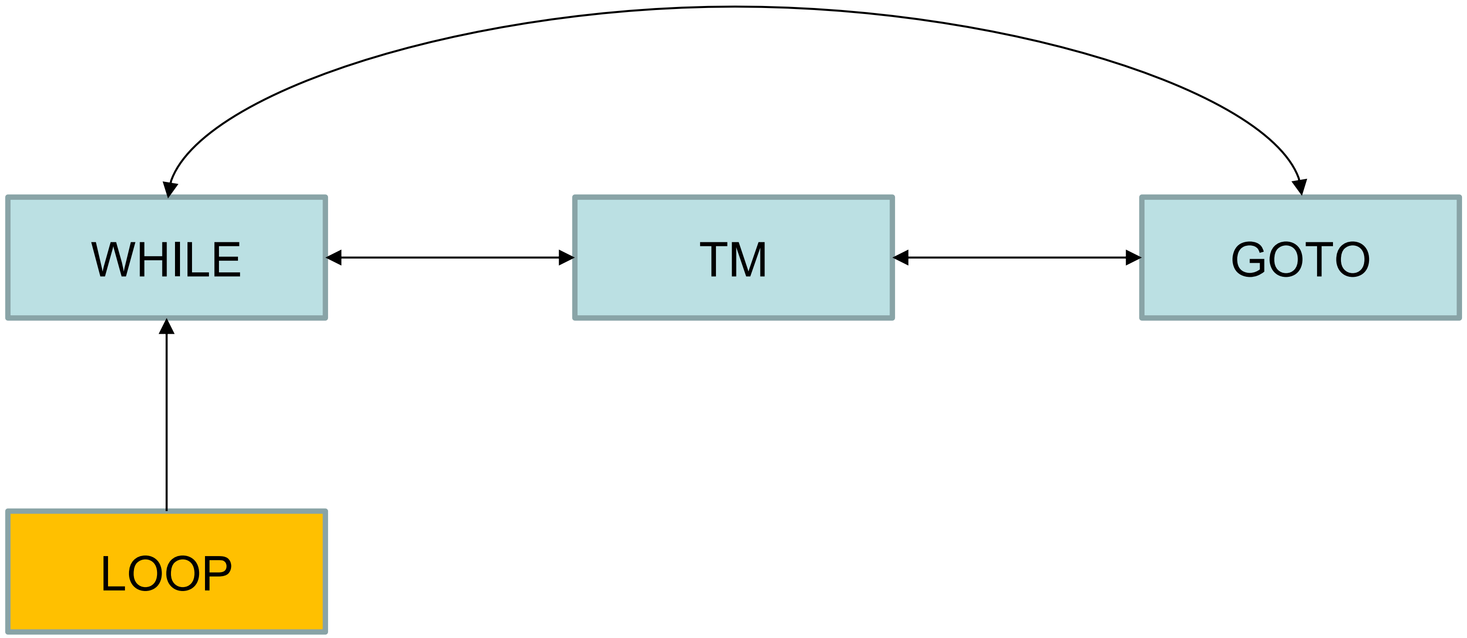
- Reformulated using WHILE:

```
z := 1;
WHILE z ≠ 0 DO
        IF z = 1 THEN S'1 END;
        IF z = 2 THEN S'2 END;
        …
        IF z = n THEN S'n END;
END
```

- Where S'_i =
  - $x_j := x_k \pm c$; $z := z + 1$ — if S_i = $x_j := x_k \pm c$
  - $z := k$ — if S_i = GOTO L_k
  - IF $x_i = c$ THEN $z := k$ ELSE $z := z + 1$ END (IF-THEN-ELSE can be represented by LOOP) — if S_i = IF $x_i = c$ THEN GOTO L_k
  - $z := 0$ — if S_i = HALT

- There is only one WHILE loop!

LOOP-GOTO-WHILE-TM



Primitive Recursive Functions

Primitive Recursion – Definition

- The following basic functions are primitive recursive:
  - all **constant functions** $f: \mathbb{N}_0^n \to \mathbb{N}_0, f(\boldsymbol{x}) = c \;, c \in \mathbb{N}_0 \,, \forall \boldsymbol{x} \in \mathbb{N}_0^n$
  - **Successor function** $s: \mathbb{N}_0 \to \mathbb{N}_0, s(x) = x + 1$
  - **Projection function** $p_i^n: \mathbb{N}_0^n \to \mathbb{N}_0, p_i^n(x_1, x_2, \dots, x_n) = x_i, \quad 1 \le i \le n$

- The functions constructed as follows are primitive recursive:
  - **Function composition**
    Let $g: \mathbb{N}_0^n \to \mathbb{N}_0$ and $h_1, h_2, \dots, h_n: \mathbb{N}_0^n \to \mathbb{N}_0$ be primitive recursive
    then $f(\boldsymbol{x}) = g(h_1(\boldsymbol{x}), \dots, h_n(\boldsymbol{x}))$ is also primitive recursive

  - **Primitive Recursion**
    Let $g: \mathbb{N}_0^n \to \mathbb{N}_0$ and $h: \mathbb{N}_0^{n+2} \to \mathbb{N}_0$ be primitive recursive
    then $f: \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ is also primitive recursive, where

$$f(0, \boldsymbol{y}) = g(\boldsymbol{y}), \qquad \boldsymbol{y} \in \mathbb{N}_0^n$$
$$f(x+1, \boldsymbol{y}) = h\big(x, \boldsymbol{y}, f(x, \boldsymbol{y})\big), \qquad x \in \mathbb{N}_0 \,, \boldsymbol{y} \in \mathbb{N}_0^n$$

Primitive Recursion – Examples

- Addition

  add(0, y) = g(y) = p_1^1(y) = y

  add(x + 1, y) = h(x, y, add(x, y))
  = s(p_3^3(x, y, add(x, y)))
  = add(x, y) + 1

  h = composition of successor & projection function

- Multiplication

  mult(0, y) = g(y) = 0

  mult(x + 1, y) = h(x, y, mult(x, y))
  = add(p_2^3(x, y, mult(x, y)),
  p_3^3(x, y, mult(x, y)))
  = add(y, mult(x, y))

  h = composition of addition & two projection functions

Primitive Recursion

- all primitive recursive functions are
  - computable
  - total
- the reverse is not true
- the class of primitive recursive functions exactly matches the class of LOOP-computable functions
- Conclusion: Any for-loop* can be replaced by a recursion and vice versa

* a for-loop in the sense of a counting loop as in the LOOP-language; note that a for-loop in some programming languages like C is much more general and actually has the power of a WHILE.

μ-recursive Functions

μ-Recursion

- Extension of the concept of primitive recursion
- We add the μ-operator:

  Let $f: \mathbb{N}_0^{n+1} \to \mathbb{N}_0$ be a μ-recursive function, then $\mu f: \mathbb{N}_0^n \to \mathbb{N}_0$ is also μ-recursive, with

$$\mu f(x_1, \dots, x_n) = \begin{cases} \min M & \text{if } M \neq \emptyset \\ \text{undefined} & \text{if } M = \emptyset \end{cases}$$

  where

$$\mathrm{M} = \{\mathrm{k} \mid f(k, x_1, \dots, x_n) = 0 \quad \text{and} \quad f(l, x_1, \dots, x_n) \text{ is defined } \forall l < k\}$$

- Now, partial functions can also be represented

Ackermann Function

- Are there total and computable functions that are not primitive recursive (and therefore not LOOP-computable)?

- Yes, for example the Ackermann function
  - discovered by ***Wilhelm Ackermann*** 1928
- Simplest known function that grows faster than any primitive recursive function
  - Implication: Faster than the factorial (*Fakultät*) and any exponential function

- Definition
  $a(0, y) = y + 1$
  $a(x + 1, 0) = a(x, 1)$
  $a(x + 1, y + 1) = a(x, a(x + 1, y))$

Ackermann Function – Example

$a(0, y) = y + 1$
$a(x + 1, 0) = a(x, 1)$
$a(x + 1, y + 1) = a(x, a(x + 1, y))$

- Calculation of a(1, 2)

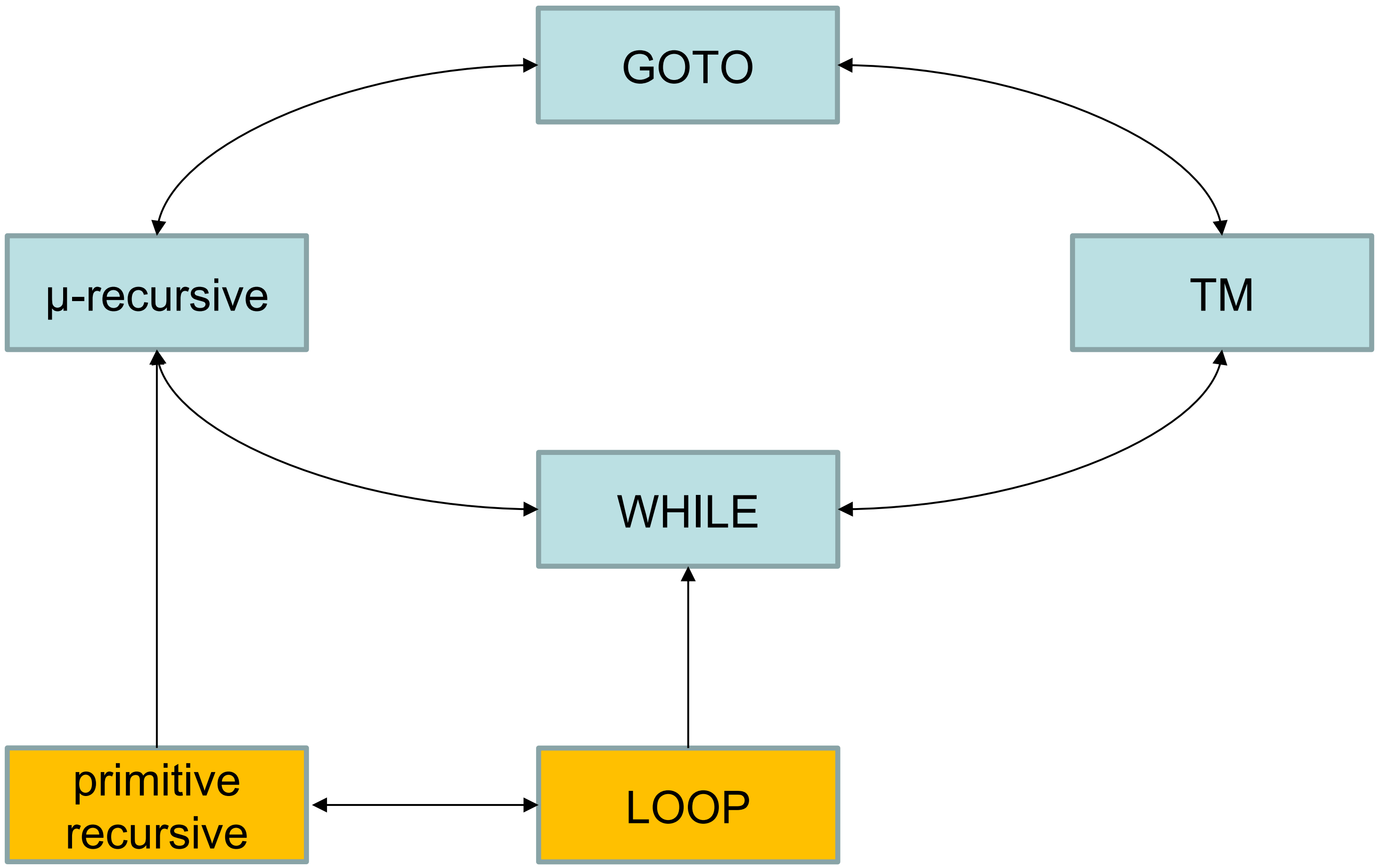
  $$\begin{aligned} a(1, 2) &= a(0, a(1, 1)) \\ &= a(0, a(0, a(1, 0))) \\ &= a(0, a(0, a(0, 1))) \\ &= a(0, a(0, 2)) \\ &= a(0, 3) \\ &= 4 \end{aligned}$$

- Growth of a(x, y)

  $a(1, 1) = 3$
  $a(1, 2) = 4$
  $a(2, 2) = 7$
  $a(3, 3) = 61$
  $a(4, 4) > 10^{10^{10^{2100}}}$

- Estimated number of atoms in the universe: ca. 10^{80}

Summary



Turing Completeness

- A system or programming language is said to be Turing-complete if it can do everything a Turing Machine can

- Not Turing-complete are, e.g.,
  - LOOP and primitive recursion
  - regular expressions
  - many widely used neural network architectures, like
    - multi-layer perceptron (MLP)
    - standard Recurrent Neural Networks (RNN) with a single hidden layer

- Turing-complete are, e.g.,
  - WHILE, GOTO and μ-recursion
  - all common procedural, object-oriented or functional programming languages
  - general recurrent neural networks
    - proof by Siegelmann and Sontag 1992
    - however: we do not have a training algorithm for these networks
  - probably (neural network) Transformer architectures

Busy Beaver

Definition

- Are there functions that are not μ-recursive (and therefore not WHILE-computable)?
- ***Tibor Radó*** 1962: Busy Beaver
- grows faster than any μ-recursive function
  - and thus faster than any computable function
  - can therefore not be represented by WHILE-/GOTO-programs or TM
  - Busy Beaver is not computable – there is no general algorithm to calculate the function

- **Definition**
  bb(0) = 0
  bb(n) = the maximum number of ones that a Turing Machine with n states (instructions, excluding HALT) and alphabet {0, 1} can write on an empty tape and halt. (0 = blank)

How to Obtain bb(n) – General Idea

1. List all Turing Machines with alphabet = {0,1} and n instructions
   - each instruction consists of two parts: total of 2n parts
   - for each part there are two options for the symbol to be written (0 or 1)
   - and two options for head movement (L, R)
   - and n+1 possible instructions (including HALT) for the following step

   - Total number of Turing Machines in existence with n instructions: $[4(n+1)]^{2n}$

   - For n=5: approx. $6.3 \cdot 10^{13}$

How to Obtain bb(n) – General Idea

2. Select all halting Turing Machines that write ones on an tape filled with zeros
   - these do exist for each n (will not be proven here)
   - although the halting problem that occurs is an indication of the non-computability of bb(n), it is not sufficient as a proof:
   - we do not presume that a general algorithm must exist to solve the halting problem
   - the Busy Beaver problem is very special
   - we could develop different adapted algorithms for each TM to be tested – the number is finite for each n

3. For each of the Turing Machines selected in this way, check how many ones it writes on the tape before it halts. The largest number of ones written is bb(n).

How to Obtain bb(n) – General Idea

- the proof that bb(n) is in fact not computable shall not be given here
- this does not mean that bb(n) cannot be determined for single values of n
- there is “just” no single, general algorithm that could calculate bb(n) for any given n

n	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**>6**
bb(n)	0	1	4	6	13	≥ 4098	$\geq 3.5 \cdot 10^{18267}$	?

Summary

- Computability
  - There exists an algorithm to calculate a function
  - that halts after a finite number of steps
- Decidability:
  - Computability of the characteristic function (YES/NO answer)
- Problem undecidable
  - a general algorithm that solves the problem cannot exist
  - however, there may well be solutions for some cases or with algorithms specially adapted to certain case
- There are infinitely more non-computable functions than computable functions

Sources

- H. Ernst, J. Schmidt und G. Beneken: *Grundkurs Informatik*. Springer Vieweg, 7. Aufl., 2020.
- Schöning, U.: *Theoretische Informatik – kurz gefasst*. Spektrum Akad. Verlag (2008)
- Hopcroft, J.E., Motwani, R. und Ullmann, J.D.: *Einführung in die Automatentheorie, formalen Sprachen und Komplexitätstheorie*. Pearson Studium (2002)