Theory of Computer Science

F2. WHILE-Computability

Gabriele Röger

University of Basel

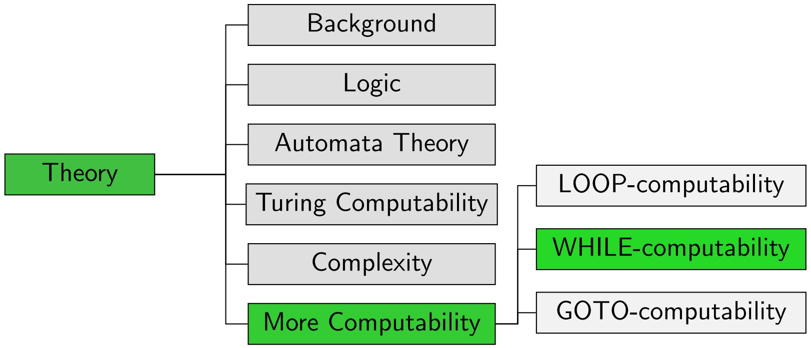
May 20, 2019

Theory of Computer Science

May 20, 2019 — F2. WHILE-Computability

F2.1 Introduction

Course Overview



LOOP, WHILE and GOTO Programs: Basic Concepts

Reminder:

- LOOP, WHILE and GOTO programs are structured like programs in (simple) "traditional" programming languages
- use finitely many variables from the set $\{x_0, x_1, x_2, \dots\}$ that can take on values in $\mathbb{N}_0$
- differ from each other in the allowed "statements"

F2.2 WHILE Programs

WHILE Programs: Syntax

Definition (WHILE Program)

WHILE programs are inductively defined as follows:

- $x_i := x_j + c$ is a WHILE program for every $i, j, c \in \mathbb{N}_0$ (addition)
- $x_i := x_j - c$ is a WHILE program for every $i, j, c \in \mathbb{N}_0$ (modified subtraction)
- If P_1 and P_2 are WHILE programs, then so is $P_1;P_2$ (composition)
- If P is a WHILE program, then so is WHILE $x_i \neq 0$ DO P END for every $i \in \mathbb{N}_0$ (WHILE loop)

German: WHILE-Programm, WHILE-Schleife

WHILE Programs: Semantics

Definition (Semantics of WHILE Programs)

The semantics of WHILE programs is defined exactly as for LOOP programs.

effect of WHILE $x_i \neq 0$ DO P END:

- If x_i holds the value 0, program execution finishes.
- Otherwise execute P.
- Repeat these steps until execution finishes (potentially infinitely often).

WHILE-Computable Functions

Definition (WHILE-Computable)

A function $f : \mathbb{N}_0^k \rightarrow_p \mathbb{N}_0$ is called WHILE-computable if a WHILE program that computes f exists.

German: f ist WHILE-berechenbar

WHILE-Program: Example

Example

```
WHILE $x_1 \neq 0$ DO
  $x_1 := x_1 - x_2$;
  $x_0 := x_0 + 1$
END
```

What function does this program compute?

F2.3 WHILE vs. LOOP

WHILE-Computability vs. LOOP-Computability

Theorem

Every LOOP-computable function is WHILE-computable.
The converse is not true.

WHILE programs are therefore strictly more powerful than LOOP programs.

German: echt mächtiger

WHILE-Computability vs. LOOP-Computability

Proof.

Part 1: Every LOOP-computable function is WHILE-computable.

Given any LOOP program, we construct an equivalent WHILE program, i. e., one computing the same function.

To do so, replace each occurrence of LOOP x_i DO P END with

```
$x_j := x_i$;
WHILE $x_j \neq 0$ DO
  $x_j := x_j - 1$;
  $P$
END
```

where x_j is a fresh variable. . . .

WHILE-Computability vs. LOOP-Computability

Proof (continued).

Part 2: Not all WHILE-computable functions are LOOP-computable.

The WHILE program

```
$x_1 := 1$;
WHILE $x_1 \neq 0$ DO
  $x_1 := 1$
END
```

computes the function $\Omega : \mathbb{N}_0 \to_p \mathbb{N}_0$ that is undefined everywhere.

Ω is hence WHILE-computable, but not LOOP-computable (because LOOP-computable functions are always total). □

Syntactic Sugar

As we can simulate LOOP loops from LOOP programs with WHILE programs, we can use all syntactic sugar we have seen for LOOP programs in WHILE programs e.g.

- $x_i := x_j$ for $i, j \in \mathbb{N}_0$
- $x_i := c$ for $i, c \in \mathbb{N}_0$
- $x_i := x_j + x_k$ for $i, j, k \in \mathbb{N}_0$
- IF $x_i \neq 0$ THEN P END for $i \in \mathbb{N}_0$
- IF $x_i = c$ THEN P END for $i, c \in \mathbb{N}_0$
- Additional syntactic sugar from the exercises

LOOP vs. WHILE: Is There a Practical Difference?

- We have shown that WHILE programs are strictly more powerful than LOOP programs.
- The example we used is not very relevant in practice because our argument only relied on the fact that LOOP-computable functions are always total.
- To terminate for every input is not much of a problem in practice. (Quite the opposite.)
- Are there any total functions that are WHILE-computable, but not LOOP-computable?

Ackermann Function: History

- David Hilbert conjectured that all computable total functions are primitive recursive (1926).
- Wilhelm Ackermann refuted the conjecture by supplying a counterexample (1928).
- The counterexample was simplified by Rózsa Péter (1935).

⇝ here: simplified version

Ackermann Function

Definition (Ackermann function)

The Ackermann function $a : \mathbb{N}_0^2 \to \mathbb{N}_0$ is defined as follows:

$$\begin{aligned} a(0,y) &= y+1 && \text{for all } y \geq 0 \\ a(x,0) &= a(x-1,1) && \text{for all } x > 0 \\ a(x,y) &= a(x-1,a(x,y-1)) && \text{for all } x,y > 0 \end{aligned}$$

German: Ackermannfunktion

Note: the recursion in the definition is bounded, so this defines a total function.

Table of Values

	$y=0$	$y=1$	$y=2$	$y=3$	$y=k$
$a(0,y)$	1	2	3	4	$k+1$
$a(1,y)$	2	3	4	5	$k+2$
$a(2,y)$	3	5	7	9	$2k+3$
$a(3,y)$	5	13	29	61	$2^{k+3}-3$
$a(4,y)$	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{k+3}-3$

Computability of the Ackermann Function

Theorem

The Ackermann function is WHILE-computable, but not LOOP-computable.

(Without proof.)

Computability of the Ackermann Function: Proof Idea

proof idea:

- WHILE-computability:
  - show how WHILE programs can simulate a stack
  - dual recursion by using a stack
  - $\rightsquigarrow$ WHILE program is easy to specify
- no LOOP-computability:
  - show that there is a number k for every LOOP program such that the computed function value is smaller than $a(k, n)$, if n is the largest input value
  - proof by structural induction; use $k =$ "program length"
  - $\rightsquigarrow$ Ackermann function grows faster than every LOOP-computable function

---

F2.4 WHILE vs. Turing

WHILE-Computability vs. Turing-Computability

Theorem

Every WHILE-computable function is Turing-computable.

(We will discuss the converse statement later.)

WHILE-Computability vs. Turing-Computability

Proof sketch.

Given any WHILE program, we construct an equivalent deterministic Turing machine.

Let $x_1, \dots, x_k$ be the input variables of the WHILE program, and let $x_0, \dots, x_m$ be all used variables.

General ideas:

- The DTM simulates the individual execution steps of the WHILE program.
- Before and after each WHILE program step the tape contains the word $bin(n_0)\#bin(n_1)\#\dots\#bin(n_m)$, where n_i is the value of WHILE program variable x_i.
- It is enough to simulate "minimalistic" WHILE programs ($x_i := x_i + 1$, $x_i := x_i - 1$, composition, WHILE loop).

...

WHILE-Computability vs. Turing-Computability

Proof sketch (continued).

The DTM consists of three sequential parts:

- initialization:
  - Write `0#` in front of the used part of the tape.
  - $(m - k)$ times, write `#0` behind the used part of the tape.
- execution:
  Simulate the WHILE program (see next slide).
- clean-up:
  - Replace all symbols starting from the first # with □, then move to the first symbol that is not □.

. . .

WHILE-Computability vs. Turing-Computability

Proof sketch (continued).

Simulation of $x_i := x_i + 1$:

1. Move left until a blank is reached, then one step to the right.
2. $(i + 1)$ times: move right until # or □ is reached.
3. Move one step to the left.

⇝ We are now on the last digit of the encoding of x_i.

4. Execute DTM for increment by 1. (Most difficult part: "make room" if the number of binary digits increases.)

. . .

WHILE-Computability vs. Turing-Computability

Proof sketch (continued).

Simulation of $x_i := x_i - 1$:

1. Move to the last digit of x_i (see previous slide).
2. Test if the digit is a `0` and the symbol to its left is # or □. If so: done.
3. Otherwise: execute DTM for decrement by 1. (Most difficult part: "contract" the tape if the decrement reduces the number of digits.)

. . .

WHILE-Computability vs. Turing-Computability

Proof sketch (continued).

Simulation of $P_1; P_2$:

1. Recursively build DTMs M_1 for P_1 and M_2 for P_2.
2. Combine these to a DTM for $P_1; P_2$ by letting all transitions to end states of M_1 instead go to the start state of M_2.

. . .

WHILE-Computability vs. Turing-Computability

Proof sketch (continued).
Simulation of WHILE $x_i \neq 0$ DO P END:

1. Recursively build DTM M for P.
2. Build a DTM M' for WHILE $x_i \neq 0$ DO P END that works as follows:
   1. Move to the last digit of x_i.
   2. Test if that symbol is 0 and the symbol to its left is # or □. If so: done.
   3. Otherwise execute M, where all transitions to end states of M are replaced by transitions to the start state of M'.

□

F2.5 Summary

Summary

- another new model of computation: WHILE programs
- strictly more powerful than LOOP programs.
- WHILE-, but not LOOP-computable functions:
  - simple example: function that is undefined everywhere
  - more interesting example (total function): Ackermann function, which grows too fast to be LOOP-computable
- Turing machines are at least as powerful as WHILE programs.