Theory of Computer Science

E2. GOTO Computability & Comparsion to Turing Computability

Gabriele Röger

University of Basel

May 24, 2023

E2.1 GOTO Programs

Motivation

We already know: WHILE programs are strictly more powerful than LOOP programs.

How do DTMs relate to LOOP and WHILE programs?

To answer this question, we make a detour over one more programming formalism, GOTO programs.

We will establish:

- WHILE programs are at least as powerful as GOTO programs.
- DTMs are at least as powerful as WHILE programs.
- GOTO programs are at least as powerful as DTMs.

$\Rightarrow$ Turing-computable = WHILE-computable = GOTO-computable

GOTO Programs: Syntax

Definition (GOTO Program)

A GOTO program is given by a finite sequence
$L_1 : A_1, L_2 : A_2, \ldots, L_n : A_n$
of labels and statements.

Statements are of the following form:

- $x_i := x_j + c$ for every $i, j, c \in \mathbb{N}_0$ (addition)
- $x_i := x_j - c$ for every $i, j, c \in \mathbb{N}_0$ (modified subtraction)
- HALT (end of program)
- GOTO L_j for $1 \leq j \leq n$ (jump)
- IF $x_i = c$ THEN GOTO L_j for $i, c \in \mathbb{N}_0$, $1 \leq j \leq n$ (conditional jump)

GOTO Programs: Semantics

Definition (Semantics of GOTO Programs)

- Input, output and variables work exactly as in LOOP and WHILE programs.
- Addition and modified subtraction work exactly as in LOOP and WHILE programs.
- Execution begins with the statement A_1.
- After executing A_i, the statement A_{i+1} is executed. (If $i = n$, execution finishes.)
- exceptions to the previous rule:
  - HALT stops the execution of the program.
  - After GOTO L_j execution continues with statement A_j.
  - After IF $x_i = c$ THEN GOTO L_j execution continues with A_j if variable x_i currently holds the value c.

GOTO-Computable Functions

Definition (GOTO-Computable)

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

E2.2 GOTO vs. WHILE

GOTO-Computability vs. WHILE-Computability

Theorem

Every GOTO-computable function is WHILE-computable.

If we allow IF statements, a single WHILE loop is sufficient for this.

(We will discuss the converse statement later.)

GOTO-Computability vs. WHILE-Computability

Proof sketch.

Given any GOTO program, we construct an equivalent WHILE program with a single WHILE loop (and IF statements).

Ideas:

- Use a fresh variable to store the number of the statement to be executed next.
  - ⇝ The variable of course has the form x_i, but for readability we write it as pc for "program counter".
- GOTO is simulated as an assignment to pc.
- If pc has the value 0, the program terminates.

...

GOTO-Computability vs. WHILE-Computability

Proof sketch (continued).

Let $\mathsf{L}_1 : A_1, \mathsf{L}_2 : A_2, \ldots, \mathsf{L}_n : A_n$ be the given GOTO program.

basic structure of the WHILE program:

```
pc := 1;
WHILE pc ≠ 0 DO
  IF pc = 1 THEN (translation of A_1) END;
  ...
  IF pc = n THEN (translation of A_n) END;
  IF pc = n + 1 THEN pc := 0 END
END
```

...

GOTO-Computability vs. WHILE-Computability

Proof sketch (continued).

Translation of the individual statements:

- $x_i := x_j + c$
  $\rightsquigarrow x_i := x_j + c;\ pc := pc + 1$
- $x_i := x_j - c$
  $\rightsquigarrow x_i := x_j - c;\ pc := pc + 1$
- HALT
  $\rightsquigarrow pc := 0$
- GOTO L_j
  $\rightsquigarrow pc := j$
- IF $x_i = c$ THEN GOTO L_j
  $\rightsquigarrow pc := pc + 1$; IF $x_i = c$ THEN $pc := j$ END

□

E2.3 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 (move existing content 2 positions to the right).
  - $(m - k)$ 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 tape cell.

...

WHILE-Computability vs. Turing-Computability

Proof sketch (continued).

Simulation of $x_i := x_i + 1$:

1. Move to the first tape cell.
2. $(i + 1)$ times: move right until # or $\square$ is reached.
3. Move one step to the left.

$\rightsquigarrow$ 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 $\square$. 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'.

$\square$

E2.4 Turing vs. GOTO

Turing-Computability vs. GOTO-Computability

Theorem (Turing-Computability vs. GOTO-Computability)

Every Turing-computable numerical function is GOTO-computable.

Proof sketch.

- Represent TM configuration (x, q, y) with three numbers, one for x, one for q and one for y.
- The tape content can be accessed and modified using DIV and MOD operations, which are GOTO-computable.
- For each transition, implement the corresponding modification of the configuration in terms of the three numbers.
- Use "IF . . . GOTO" statements for each tape symbol and state to jump to the implementation of the corresponding transition.

□

Final Result

Corollary

Let $f : \mathbb{N}_0^k \rightarrow_p \mathbb{N}_0$ be a function.

The following statements are equivalent:

- *f is Turing-computable.*
- *f is WHILE-computable.*
- *f is GOTO-computable.*

Moreover:

- *Every LOOP-computable function is Turing-/WHILE-/GOTO-computable.*
- *The converse is not true in general.*

Summary

results of the investigation:

- another new model of computation: GOTO programs
- Turing machines, WHILE and GOTO programs are equally powerful.
  - Whenever we said “Turing-computable” or “computable” in parts C or D, we could equally have said “WHILE-computable” or “GOTO-computable”.
- LOOP programs are strictly less powerful.