Berechenbarkeit und Komplexitätstheorie

Wintersemester 2021/2022

Aufgabenblatt 3

Abgabe: 26. November 2021 um 12 Uhr

Definition(en)

- Die k-stellige Cantor'sche Bijektion[1] $\langle\cdot\rangle : \mathbb{N}^k \to \mathbb{N}$ ist definiert als:

$$\langle x_1, \ldots, x_{k-1}, x_k\rangle = \langle x_1, \langle \ldots \langle x_{k-1}, x_k\rangle \ldots\rangle \text{ mit } \langle x, y\rangle = y + \left(\sum_{i=1}^{x+y} i\right)$$

  Weiter bezeichnen p_1, …, p_k die Umkehrfunktion(en) der Cantor'schen Bijektion, sodass $\langle p_1(z), \ldots, p_k(z)\rangle = z$ gilt. (Die tatsächliche Berechnung der Komponenten ist im Skript auf Seite 17 beschrieben.)

- Die Ackermannfunktion $A : \mathbb{N}_0 \times \mathbb{N}_0 \to \mathbb{N}$ ist wie folgt definiert:

$$\begin{aligned} A(0, y) &= y + 1 \\ A(x + 1, 0) &= A(x, 1) \\ A(x + 1, y + 1) &= A(x, A(x + 1, y)) \end{aligned}$$

Aufgabe 3.1 *(2 + 2 Punkte)*

Sei $f : \mathbb{N} \to \mathbb{N}$ eine WHILE-berechenbare, injektive, totale Funktion.

a) Zeigen Sie, dass die Umkehrfunktion f^{-1} von f ebenfalls WHILE-berechenbar ist.

b) Gilt das gleiche auch für LOOP-berechenbare Funktionen? (Begründen Sie Ihre Antwort.)

Lösung

a) WHILE-berechenbare Funktionen müssen nicht total sein; Also darf die gesuchte Umkehrfunktion f^{-1} partiell sein. Da f injektiv ist, ist die folgende Umkehrfunktion wohldefiniert:

$$f^{-1}(y) = \begin{cases} x, \text{ falls } f(x) = y \\ \text{undefiniert, sonst} \end{cases}$$

Um nun f^{-1} zu berechnen, kann man einen Algorithmus schreiben, der alle x ausprobiert, bis er dasjenige findet, für das gilt $f(x) = y$.

Beispielsweise:

```
1: x_0 := 0
2: x_t := 1
3: while x_t ≠ 0 do
4:     x_0 := x_0 + 1
5:     if f(x_0) = y then
6:         x_t := 0
```

[1] Vgl. Definition in Kapitel 3 auf Seite 16 im Skript.

```
7:     fi
8: od
```

b) Nein. LOOP-berechenbare Funktionen müssen total sein, aber eine Umkehrfunktion ist nicht zwingend total. (Bspw. ist $f\colon \mathbb{N} \to \mathbb{N}$, $x \mapsto 2 \cdot x$ total, aber die Umkehrfunktion $f^{-1}\colon \mathbb{N} \to \mathbb{N}$ ist nur für gerade Zahlen definiert.) Prinzipiell kann man in den meisten Fällen den Definitionsbereich einschränken um die undefinierten Fälle auszuschließen. Also statt $f^{-1}\colon \mathbb{N} \to \mathbb{N}$ kann man $f^{-1}\colon f(\mathbb{N}) \to \mathbb{N}$ setzen. (Im obigen Beispiel wäre dann $f^{-1}\colon \{x \in \mathbb{N} \colon \exists x' \in \mathbb{N}\ 2 \cdot x' = x\} \to \mathbb{N}$ eine totale Funktion.) Allerdings müsste dann die Definition der LOOP-Programme aus der Vorlesung, die beliebige natürliche Zahlen als Eingabe erlaubt, etwas verändert werden.

Aufgabe 3.2 *(2 + 2 Punkte)*

a) Gilt $\langle 1, 2, 3\rangle \leq \langle 2, 2, 2\rangle$? (Begründen Sie kurz.)

b) Bestimmen Sie $a, b, c \in \mathbb{N}$, sodass $\langle a, b, c\rangle = 102$.

Lösung

a) Nein, denn:

$$\begin{aligned}\langle 1, 2, 3\rangle &= \langle 1, \langle 2, 3\rangle\rangle \\ &= \langle 1, 18\rangle \\ &= 208 \\ \langle 2, 2, 2\rangle &= \langle 2, \langle 2, 2\rangle\rangle \\ &= \langle 2, 12\rangle \\ &= 117\end{aligned}$$

Und $208 \not\leq 117$.

b) Es gilt $\langle 2, 3, 1\rangle = \langle 2, 11\rangle = 102$

Aufgabe 3.3 *(2 + 3 + 2 Punkte)*

a) Implementieren Sie die Ackermann-Funktion in einer Programmiersprache Ihrer Wahl! (Die gewählte Sprache sollte "lesbar" sein – BRAINFUCK, MALBOLGE oder sonstige esoterische Programmiersprachen werden wahrscheinlich nicht korrigiert.)

b) Modifizieren Sie Ihre Implementierung aus dem vorigen Teil so, dass diese nicht rekursiv arbeitet, sondern die Rekursion durch einen Stack auflöst.

c) Schreiben Sie ein WHILE-Programm, welches die Ackermann-Funktion berechnet.

(*Hinweis:* Implementieren Sie den Stack im WHILE-Programm mithilfe der Cantor'schen Bijektion. Sie dürfen $\langle x, y\rangle$, p_1 und p_2 im WHILE-Programm nutzen, ohne diese selbst zu implementieren.)

Lösung

a) Die rekursive Variante der Ackermann-Funktion als Lua-Code:

```lua
function AckermannRek(x,y)
  if x == 0 then
    return y + 1
  elseif y == 0 then -- hier: x > 0
    return AckermannRek(x-1,1)
  else -- x,y > 0
    return AckermannRek(x-1,AckermannRek(x,y-1))
  end
end
```

b) Die stackbasierte Variante der Ackermann-Funktion als Lua-Code:

```lua
function AckermannStack(x,y)
  s = {x} -- Stack mit x als Element initialisiert
  while #s > 0 do -- #s: Anzahl der Stackelemente
    x = table.remove(s) -- entspricht s.pop()
    if x == 0 then
      y = y + 1
    elseif y == 0 then
      table.insert(s, x-1) -- entspricht s.push(x-1)
      y = 1
    else
      table.insert(s,x-1) -- entspricht s.push(x-1)
      table.insert(s,x) -- entspricht s.push(x)
      y = y-1
    end
  end
  return y
end
```

c) Das zugehörige While-Programm:

```
1: function Ackermann(x,y)
2:     s := ⟨x, 0⟩                          ▷ Initialisiere s und Push(x)
3:     sz = 1
4:     while sz > 0 do                      ▷ Solange der Stack nicht leer ist...
5:         x := p1(s)                       ▷ Pop()
6:         s := p2(s)
7:         sz := sz − 1
8:         if x = 0 then                    ▷ Verankerung
9:             y := y + 1
10:        else if y = 0 then               ▷ zweiter Fall
11:            s := ⟨x − 1, s⟩              ▷ Rekursiver Aufruf A(x − 1, 1)
12:            sz := sz + 1
13:            y := 1
14:        else
15:            s := ⟨x − 1, s⟩              ▷ Rekursive Aufrufe im dritten Fall
16:            s := ⟨x, s⟩
17:            sz := sz + 2
18:            y := y − 1
19:        fi
20:    od
21:    return y
22: end
```