Prof. Dr. Titus Dose
Fakultät für Informatik und Mathematik
OTH Regensburg

Theoretische Informatik
Wintersemester 2024/2025
23.10.2024

# Übungsblatt 3

(Besprechung am 07.11.2024)

1. **Converting between number representations**
   Complete the following table!

| decimal | binary | dyadic | 3-adic |
|---|---|---|---|
| 35 | | | |
| | 110001 | | |
| | | 121211 | |
| | | | 11131 |

2. **Simulation of a RAM by a Python program** 
   Apply the construction from Theorem 2.35 to the following RAM program, which computes a function $\mathbb{N} \times \mathbb{N} \to \mathbb{N}$ here.

```
0 R3 <- RR2                4 R0 <- R0 - R1
1 R3 <- R3 + R2            5 IF R0 > 0 GOTO 0
2 R2 <- R2 + R1            6 R0 <- RR2
3 RR2 <- R3                7 STOP
```

3. **Elimination of function calls** 
   The aim of this task is to apply the elimination of function calls (cf. step 4 of the proof of Theorem 2.37) using an example to understand the *general* procedure.

   The following While program computes a function which, in its definition, is similar to the Ackermann function.

```
def a(n,m):
    r = 0
    if(n == 0):
        r = (m + 1)
    if((n > 0) and (m == 0)):
        r = a((n-1),1)
    if((n > 0) and (m > 0)):
        r = a((n-1), a(n,(m-1)))
    if((r > ((n + m) + 1))):
        r = (r - 1)
    return r
```

   Convert this function into a Python program *in the same way as described in step 4 of Theorem 2.37*. The Python program should

   - contain exactly one function,
   - not contain any function calls and
   - compute the same function.

   Observe the hints for this task.

4. **TM program**

(a) Let $l\colon \mathbb{N} \to \mathbb{N}$ with $l(x) = 2x + 2$. Specify the program of a 1-tape TM that computes $l$ in dyadic representation.

(b) Let $l'\colon \mathbb{N} \to \mathbb{N}$ with $l'(x) = 2x + 3$. Specify the program of a 1-tape TM that computes $l'$ in dyadic representation.

# Hints

**Exercise 1:**
No justification is required here. Use Properties 2.6 and 2.14.

**Exercise 2:**
Implement the Python function `phi` specified in the proof (page 132).

**Exercise 3:**
In the elo course room you will find a similar task including a detailed sample solution, which you should use as a guide. The aim is to understand the general construction. Therefore, it makes sense to proceed according to step 4 of the proof of Theorem 2.37 and according to the sample solution in elo. In particular, you should not implement the specified function using a dynamic program.

**Exercise 4:**
Think about how the dyadic representations of the numbers $x$ and $2x + 2$ (resp., $2x + 2$ and $2x + 3$) are related. For the latter, remember your math lessons in elementary school. You can use the TM interpreter in the elo course room to test your program.

# Solutions

**Solution for exercise 1:**

| decimal | binary | dyadic | 3-adic |
|---|---|---|---|
| 35 | 100011 | 11211 | 322 |
| 49 | 110001 | 21121 | 1211 |
| 83 | 1010011 | 121211 | 2232 |
| 127 | 1111111 | 1111111 | 11131 |

**Solution for exercise 2:**
The specified RAM program does not compute a specific function, but is only intended to illustrate the simulation of a RAM using a Python program.

```
def read(u,v,a):            # liefert den Inhalt von Ra
  i = 0
  while (i < len(u) and u[i] != a): # Index a suchen
    i = i + 1
  if (i == len(u)):                 # Listen erweitern
    u += [a]
    v += [0]
  return v[i]                       # Inhalt von Ra zurï¿½ck

def write(u,v,a,b):         # schreibt b in Ra
  i = 0
  while (i < len(u) and u[i] != a): # Index a suchen
    i = i + 1
  if (i == len(u)):                 # Listen erweitern
    u += [a]
    v += [0]
  v[i] = b                          # schreibt b in Ra

def phi(x1,x2):
  u = [0,1]
  v = [x1,x2]
  ir = 0
  while (ir < 7):
    if (ir == 0):       # 0 R3 <- RR2
      i = read(u,v,2)
      j = read(u,v,i)
      write(u,v,3,j)
      ir = ir + 1
    if (ir == 1):       # 1 R3 <- R3 + R2
      i = read(u,v,3) + read(u,v,2)
      write(u,v,3,i)
      ir = ir + 1
    if (ir == 2):       # 2 R2 <- R2 + R1
      i = read(u,v,2) + read(u,v,1)
      write(u,v,2,i)
      ir = ir + 1
    if (ir == 3):       # 3 RR2 <- R3
      i = read(u,v,3)
      j = read(u,v,2)
      write(u,v,j,i)
      ir = ir + 1
    if (ir == 4):       # 4 R0 <- R0 - R1
      i = read(u,v,0) - read(u,v,1)
      if (i < 0):
        i = 0
      write(u,v,0,i)
      ir = ir + 1
    if (ir == 5):       # 5 IF R0 > 0 GOTO 0
      if (read(u,v,0) > 0):
        ir = 0
      else:
        ir = ir + 1
    if (ir == 6):       # 6 R0 <- RR2
      i = read(u,v,2)
```

```
      j = read(u,v,i)
      write(u,v,0,j)
      ir = ir + 1
  return read(u,v,0)
```

**Solution for exercise 3:**
Wir verwenden die folgenden Listen als Stack:

- argn speichert die ersten Parameter der jeweiligen Funktionsaufrufe
- argm speichert die zweiten Parameter der jeweiligen Funktionsaufrufe
- var speichert Werte der Variable r in den jeweiligen Funktionsaufrufen
- pos speichert die Positionen der jeweiligen Funktionsaufrufe, ist also eine Art Rücksprungadresse. Die Positionen sind im untenstehenden Programm markiert. An den jeweiligen Positionen muss Folgendes ausgeführt werden.

  0 Man befindet sich in der jeweiligen Ebene am Anfang der Funktion.

  1 Der rekursive Aufruf mit a(n-1,1) wurde getätigt und der Wert befindet sich in ret.

  2 Der erste rekursive Aufruf mit a(n,m-1) wurde getätigt und der Wert befindet sich in ret.

  3 Der zweite rekursive Aufruf mit a(n-1, a(n,m-1)) wurde getätigt und der Wert befindet sich in ret.

  4 Das letzte Element der Liste var ist der Rückgabewert innherhalb der entsprechenden Ebene.

```
def a(n,m):
    r = 0                              # Position 0
    if(n == 0):
        r = (m + 1)
    if((n > 0) and (m == 0)):
        r = a((n-1),1)                 # Position 1
    if((n > 0) and (m > 0)):
        r = a((n-1), a(n,(m-1)))       # Positionen 2 und 3
    if(r > ((n + m) + 1)):             # Position 4
        r = (r - 1)
  return r

def a_(n,m):
    argn = [n]                          # Initialisierung
    argm = [m]
    var = [0]
    pos = [0]
    ret = 0
    while len(pos) > 0:
        if pos[-1] == 0:                # Beginn einer Rekursion
            if argn[-1] == 0:
                var[-1] = argm[-1] + 1
                pos[-1] = 4
            elif argn[-1] > 0 and argm[-1] == 0:
                pos[-1] = 1             # Position merken
                pos += [0]              # wegen Funktionsaufruf Stack vergroessern
                argn += [argn[-1] - 1]
                argm += [1]
                var += [0]
            elif argn[-1] > 0 and argm[-1] > 0:
                pos[-1] = 2             # Position merken
                pos += [0]              # wegen Funktionsaufruf Stack vergroessern
                argn += [argn[-1]]
                argm += [argm[-1] - 1]
                var += [0]
            else:
                pos[-1] = 4
        elif pos[-1] == 1:
            var[-1] = ret               # Rueckgabewert der aufgerufenen Funktion
            pos[-1] = 4                 # weitergeben an aufrufende Funktion
        elif pos[-1] == 2:
            pos[-1] = 3                 # Position merken
            pos += [0]                  # wegen Funktionsaufruf Stack vergroessern
            argn += [argn[-1] - 1]      # Rueckgabewert der aufrufenden Funktion
```

```
            argm += [ret]               # ist Argument im naechsten Aufruf
            var += [0]
        elif pos[-1] == 3:
            var[-1] = ret               # Rueckgabewert der aufgerufenen Funktion
            pos[-1] = 4                 # weitergeben an aufrufende Funktion
        else:
            if var[-1] > argn[-1] + argm[-1] + 1:
                var[-1] = var[-1] - 1
            ret = var[-1]               # Rueckgabewert in ret speichern
            pos = pos[0:-1]             # und abgearbeitete Funktion vom Stack nehmen
            var = var[0:-1]
            argn = argn[0:-1]
            argm = argm[0:-1]
    return ret
```

**Solution for exercise 4:**

(a) Beobachtung: Bei dyadischer Zahlendarstellung entspricht die Operation $2x + 2$ dem Anhängen einer 2 an die Eingabe.

Arbeitsweise der TM:

- laufe im Zustand $z_0$ nach rechts bis zum ersten $\square$
- schreibe eine 2 und gehen in den Stoppzustand $z_1$

Programm der TM:

```
(z0,1)->(z0,1,R)    // erstes Leerzeichen suchen
(z0,2)->(z0,2,R)
(z0,_)->(z1,2,O)    // 1 schreiben und stoppen
```

(b) Nun müssen wir nach der Berechnung von $2x + 2$ noch 1 draufaddieren.

```
(z0,1)->(z0,1,R)    // erstes Leerzeichen suchen
(z0,2)->(z0,2,R)
(z0,_)->(z2,2,O)
(z2,1)->(z1,2,O)
(z2,2)->(z2,1,L)
(z2,_)->(z1,1,O)
```