13 Berechenbarkeit und Aufwandsabschätzung

13.1 Berechenbarkeit

Frage: Gibt es für jede Funktion, die mathematisch spezifiziert werden kann, ein Programm, das diese Funktion berechnet?

Antwort: Nein! [Turing 1937]

Hier: Informelle Argumentation

Definition: Eine Funktion, die programmiert werden kann, heißt *berechenbar*.

Zwei Funktionen

Betrachte

```
; return-seven : -> number
(define return-seven
  (lambda ()
    7))

; loop-forever : -> X
(define loop-forever
  (lambda ()
    (loop-forever)))
```

Für die Berechnungsprozesse gilt

- `(return-seven) => 7`
  terminiert und liefert das Ergebnis 7
- `(loop-forever) => (loop-forever) => ...`
  terminiert nicht

Das Halteproblem

Definition Sei `halts? : (-> X) -> boolean` definiert durch

- `(halts? f) => #t`, falls der Berechnungsprozesse `(f)` terminiert, und
- `(halts? f) => #f`, falls `(f)` *nicht* terminiert.

Beispiel

- `(halts? return-seven)` liefert `#t`
- `(halts? loop-forever)` liefert `#f`

Die Goldbach-Vermutung

- Vermutung: Jede gerade Zahl $n > 4$ ist Summe zweier Primzahlen.
  Goldbach 1742, bisher unbewiesen
- Funktion `goldbach` hält an, falls die Goldbach-Vermutung falsch ist.

```
; goldbach : -> string
(define goldbach
  (lambda ()
    (letrec ((test
              (lambda (n)
                (if (sum-of-two-primes? n)
                    (test (+ n 2))
                    "Goldbach-Vermutung ist falsch."))))
      (test 6))))
```

- `(halts? goldbach)` würde die Goldbach-Vermutung beweisen oder verwerfen.

`halts?` ist nicht berechenbar

- Angenommen, `halts?` kann (z.B. in Scheme) programmiert werden. Dann ist auch die folgende Funktion definierbar:

```
; debunk-halts? : -> number
(define debunk-halts?
  (lambda ()
    (if (halts? debunk-halts?)
        (loop-forever)
        42)))
```

- Was ist der Wert von `(halts? debunk-halts?)`?
  - Angenommen, der Wert ist `#t`. Dann ist

```
   (debunk-halts?)
=>
   (if (halts? debunk-halts?)
       (loop-forever)
       42)
=> nach Annahme
   (if #t
       (loop-forever)
       42)
=>
   (loop-forever)
```

    Da der Berechnungsprozess nicht terminiert, ergibt sich ein Widerspruch zur Definition von `halts?`.

- Was ist der Wert von `(halts? debunk-halts?)`?
  - Angenommen, der Wert ist `#f`. Dann ist

```
   (debunk-halts?)
=>
   (if (halts? debunk-halts?)
       (loop-forever)
       42)
=> nach Annahme
   (if #f
       (loop-forever)
       42)
=>
   42
```

    Jetzt terminiert der Berechnungsprozess und liefert einen Widerspruch zur Definition von `halts?`.

- Also muss die Annahme, dass `halts?` programmierbar ist, falsch sein!
- Mehr dazu in Informatik III

13.2 Aufwandsabschätzung

Motivation

Oftmals ist man nicht nur am Ergebnis einer Berechnung interessiert, sondern möchte, dass die Laufzeit möglichst kurz ist.

- Wieviele Berechnungsschritte werden ausgeführt?
- Wann ist eine Implementierung eines Algorithmus' besser als eine alternative Implementierung?

Bei der Aufwandsabschätzung unterscheidet man drei Fälle:

- Laufzeit im besten Fall (*best case*)
- Laufzeit im durchschnittlichen Fall (*average case*)
- Laufzeit im schlimmsten Fall (*worst case*)

Hier: Worst case.

Ein Beispiel

Frage: Wie hoch ist der Aufwand, um mittels `search-tree-member?` festzustellen, ob ein Element in einem Suchbaum vorhanden ist?

Maß für den Aufwand: Anzahl der rekursiven Aufrufe.

Antwort: Um festzustellen, ob ein Element in einem Suchbaum mit Tiefe n vorhanden ist, benötigt man höchstens $n + 1$ rekursive Aufrufe.

Zur Erinnerung:

```
; Feststellen, ob Element im Suchbaum vorhanden ist
; search-tree-member? : A search-tree(A) -> boolean
(define search-tree-member?
  (lambda (elem st)
    (let ((eq? (search-tree-label-equal-proc st))
          (le? (search-tree-label-leq-proc st)))
      (letrec ((member?
                (lambda (t)
                  (cond
                   ((empty-tree? t)
                    #f)
                   ((node? t)
                    (cond
                     ((eq? elem (node-label t))
                      #t)
                     ((le? elem (node-label t))
                      (member? (node-left t)))
                     (else
                      (member? (node-right t))))))))))
        (member? (search-tree-tree st))))))
```

Zur Erinnerung:

```
; Berechnet die Tiefe eines Binärbaums
; btree-depth: btree(A) -> nat
(define btree-depth
  (lambda (t)
    (cond
      ((empty-tree? t)
       0)
      ((node? t)
       (+ 1
          (max (btree-depth (node-left t))
               (btree-depth (node-right t))))))))
```

Behauptung: Im Berechnungsprozess für (`member?` t) ist die Anzahl n der Aufrufe von `member?` höchstens (`btree-depth` t) $+ 1$.

Beweis durch Terminduktion.

[Induktionsbasis] Nullstellige Symbole: $t =$ `the-empty-tree`

(`member? the-empty-tree`) `=>` `#f`, also $n = 1$.

Da (`btree-depth the-empty-tree`) `=>` `0` und $n = 1 \leq 1 = 0 + 1$ gilt die Behauptung.

[Induktionsschritt] Angenommen, die Induktionsbehauptung gilt für t_1 und t_2.

Zeige: Die Behauptung gilt auch für (`make-node` t_1 `y` t_2) mit beliebigem `y`.

Für den ersten Aufruf von `member?` gilt:

```
   (member? (make-node t1 y t2))
=>
   (cond
    ((eq? elem y) #t)
    ((le? elem y) (member? (node-left (make-node t1 y t2))))
    (else (member? (node-right (make-node t1 y t2)))))
```

Drei Fälle: `(eq? elem y)`, `(le? elem y)`, `else`

1. Falls `(eq? elem y)`, so ist

$$
\begin{aligned}
& n \\
= \;& 1 \\
\leq \;& 1 + \texttt{(max (btree-depth } t_1\texttt{) (btree-depth } t_1\texttt{))} \\
= \;& \texttt{(btree-depth (make-node } t_1 \texttt{ y } t_2\texttt{))}
\end{aligned}
$$

2. Falls `(le? elem y)`, so setzt sich der Berechnungsprozess wie folgt fort:

   `(member? (node-left (make-node` t_1 `y` t_2`)))`

   `=>`

   `(member?` t_1`)`

   Setze nun $n' =$ Anzahl der Aufrufe von `member?` im Berechnungsprozess von `(member?` t_1`)`. Dann ist

$$
\begin{aligned}
& n \\
= \quad & 1 + n' \\
\leq \quad & \quad \{ \textbf{ nach Induktionsvoraussetzung } \} \\
& 1 + \texttt{(btree-depth } t_1\texttt{)} + 1 \\
\leq \quad & \quad \{ \textbf{ Definition von Maximum } \} \\
& 1 + 1 + \texttt{(max (btree-depth } t_1\texttt{) (btree-depth } t_2\texttt{))} \\
= \quad & 1 + \texttt{(btree-depth (make-node } t_1 \texttt{ y } t_2\texttt{))}
\end{aligned}
$$

3. Der dritte Fall ist analog zu beweisen.

q.e.d.

13.3 Sortieren von Listen

Signatur: `list-sort : (X X -> bool) list(X) -> list(X)`

Erklärung: `(list-sort leq l)` liefert eine Liste mit den gleichen Elementen wie `l`, aber aufsteigend gemäß der kleiner–gleich Relation `leq` sortiert.

Beispiele:

```
(list-sort < (list 32 16 8))     ; == (list 8 16 32)
(list-sort string<? empty)       ; == empty
```

Muster:

```
(define list-sort
  (lambda (leq l)
    (cond
     ((empty? l)
      ...)
     ((pair? l)
      (... (first l) ... (list-sort leq (rest l)) ...)))))
```

Bemerkung:

- Es ist nicht sofort klar, wie aus dem Listenkopf und der sortierten Restliste eine sortierte Liste konstruiert werden kann.

$\Rightarrow$ Überlasse dies einer Hilfsdefinition `list-insert`.

- Anforderungen an `(list-insert leq x l)`: konstruiert aus einer kleiner–gleich Relation `leq`, einer Zahl und einer aufsteigend sortierten Liste eine aufsteigend sortierte Liste, die sowohl `x` als auch sämtliche Elemente von `l` enthält. Zum Vergleich der Listenelement wird `leq` benutzt.

Definition:

```
(define list-sort
  (lambda (leq l)
    (cond
     ((empty? l)
      empty)
     ((pair? l)
      (list-insert leq (first l) (list-sort leq (rest l)))))))
```

Hilfsdefinition: `list-insert`

Signatur: `list-insert : (X X -> boolean) X list(X) -> list(X)`

Erklärung: `(list-insert leq x l)` konstruiert aus einer kleiner–gleich Relation `leq`, einer Zahl `x` und einer aufsteigend sortierten Liste `l` eine aufsteigend sortierte Liste, die sowohl `x` als auch sämtliche Elemente von `l` enthält. Zum Vergleich der Listenelement wird `leq` benutzt.

Beispiele:

```
(list-insert < 17 (list 8 16 32)) ; == (list 8 16 17 32)
(list-insert < 4711 empty)        ; == (list 4711)
```

Muster:

```
(define list-insert
  (lambda (leq x l)
    (cond
     ((empty? l) ...)
     ((pair? l)
      (... (first l) ... (list-insert leq x (rest l)) ...)))))
```

Definition:

```
(define list-insert
  (lambda (leq x l)
    (cond
     ((empty? l)
      (list x))
     ((pair? l)
      (let ((head (first l)))
        (if (leq x head)
            (make-pair x l)
            (make-pair head (list-insert leq x (rest l))))))))))
```

Bemerkung: *Sortieren durch Einfügen*, insertion sort

Beispiel für `list-sort`

```
   (list-sort (list 32 16 8))
=> (list-insert (first (list 32 16 8)) (list-sort (rest (list 32 16 8))))
=> (list-insert 32 (list-sort (list 16 8)))
=> (list-insert 32 (list-insert (first (list 16 8)) (list-sort (rest (list 16 8)))))
=> (list-insert 32 (list-insert 16 (list-sort (list 8))))
=> (list-insert 32 (list-insert 16 (list-insert (first (list 8)) (list-sort (rest (list 8))))))
=> (list-insert 32 (list-insert 16 (list-insert 8 (list-sort empty))))
=> (list-insert 32 (list-insert 16 (list-insert 8 empty)))
=> (list-insert 32 (list-insert 16 (list 8)))
=> (list-insert 32 (make-pair 8 (list-insert 16 empty)))
=> (list-insert 32 (make-pair 8 (list 16)))
=> (list-insert 32 (list 8 16))
=> (make-pair 8 (list-insert 32 (list 16)))
=> (make-pair 8 (make-pair 16 (list-insert 32 empty)))
=> (make-pair 8 (make-pair 16 (list 32)))
=> (list 8 16 32)
```

Aufwandsabschätzung für `list-sort`

Sinnvolles Maß: Anzahl der ausgeführten Vergleichsoperationen.

Vorgehen:

- Ermittle obere Schranke für die Anzahl der in `list-insert` ausgeführten Vergleichsoperationen.
- Berechne damit eine obere Schranke für die Anzahl der in `list-sort` ausgeführten Vergleichsoperationen.

Definition: Sei $\mathcal{V}_i(l)$ die Anzahl der Vergleichsoperationen, die bei einem Aufruf `(list-insert leq x` l`)` ausgeführten werden. Dabei sind `leq` und `x` beliebig aber fix.

Behauptung: Es gilt: $\mathcal{V}_i(l) \leq$ `(length` l`)`.

Beweis durch Terminduktion.

[Induktionsbasis] Nullstellige Symbole: $l =$ `empty`.
`(list-insert leq x empty) => (list x)` und es wird keine Vergleich ausgeführt. Also $\mathcal{V}_i(l) = 0 \leq$ `(length empty)`.

[Induktionsschritt] Angenommen, die Induktionsbehauptung gilt für eine Liste l'.
Zeige: Die Behauptung gilt auch für $l =$ `(make-pair y` l'`)` für beliebiges `y`.
Für den ersten Aufruf von `list-insert` gilt:

```
   (list-insert leq x (make-pair y l'))
=> (let ((head (first (make-pair y l'))))
     (if (leq x head)
         (make-pair x (make-pair y l'))
         (make-pair head (list-insert leq x (rest (make-pair y l')))))))
```

=>

```
(if (leq x y)
    (make-pair x (make-pair y l'))
    (make-pair y (list-insert leq x l')))
```

Es wird also mindestens eine Vergleichsoperation ausgeführt.
Fallunterscheidung:

- `(leq x y) => #t`
  Dann werden keine weiteren Vergleichsoperation ausgeführt.
  Also: $\mathcal{V}_i(l) = 1 \leq$ `(length` l`)`
- `(leq x y) => #f`
  Dann gilt: $\mathcal{V}_i(l) = 1 + \mathcal{V}_i(l') \overset{\text{IV}}{\leq} 1 +$ `(length` l'`)` $=$ `(length` l`)`

q.e.d.

Bemerkung: Die obere Schranke wird realisiert, d.h. es gibt `leq`, `y` und l mit $\mathcal{V}_i(l) =$ `(length` l`)`.
Beispiel: `(list-insert <= 4 (list 1 2 3))`

Definition: Sei $\mathcal{V}_s(l)$ die Anzahl der Vergleichsoperationen, die bei einem Aufruf `(list-sort leq` l`)` ausgeführten werden. Dabei ist `leq` beliebig aber fix.

Behauptung: Es gilt: $\mathcal{V}_s(l) \leq \frac{n}{2}(n-1)$, wobei $n =$ `(length` l`)`.

Beweis durch Terminduktion.

[Induktionsbasis] Nullstellige Symbole: $l =$ `empty`.

`(list-sort leq empty) => empty` und es wird kein Vergleich ausgeführt.

Also $\mathcal{V}_s(l) = 0 \leq 0 = \frac{\texttt{(length } l\texttt{)}}{2}(\texttt{(length } l\texttt{)} - 1)$.

[Induktionsschritt] Angenommen, die Induktionsbehauptung gilt für eine Liste l'.

Zeige: Die Behauptung gilt auch für $l =$ `(make-pair y` l'`)` für beliebiges `y`.

Für den ersten Aufruf von `list-sort` gilt:

```
   (list-sort leq l)
=>
   (list-insert leq (first (make-pair y l'))
                    (list-sort leq (rest (make-pair y l'))))
```

`=> (list-insert leq y (list-sort leq` l'`))`

Lemma: Für alle Listen l'' gilt: `(length` l''`)` $=$ `(length (sort leq` l''`))`
Beweis: Übung.

Insbesondere gilt also:

$$\mathcal{V}_i(\texttt{(list-sort leq } l'\texttt{)}) \leq \texttt{(length (list-sort leq } l'\texttt{))} = \texttt{(length } l'\texttt{)}$$

Damit ergibt sich mit $n =$ `(length` l`)` und $n' =$ `(length` l'`)`:

$$\begin{aligned}\mathcal{V}_s(l) &= \mathcal{V}_i(\texttt{(list-sort leq } l'\texttt{)}) + \mathcal{V}_s(l') \leq n' + \mathcal{V}_s(l') \\ &\overset{\text{IV}}{\leq} n' + \tfrac{n'}{2}(n'-1) = \tfrac{n'^2+n'}{2} = \tfrac{(n-1)^2+n-1}{2} = \tfrac{n}{2}(n-1)\end{aligned}$$

q.e.d

Bemerkung: Die obere Schranke wird realisiert, d.h. es gibt l mit `(length` l`)` $= n$ und $\mathcal{V}_s(l) = \frac{n}{2}(n-1)$. Beispiel: `(list-sort <= (list 32 16 8))`