

13. Terminierung und Laufzeit

13.1 Terminierungsbeweise

Rekursiv definierte Funktionen brauchen Terminierungsbeweise

Bisher: Argumentation: Argument über welches die Rekursion läuft, wird „kleiner“. Das ist sehr ungenau und muss präzisiert werden.

Probleme:

1. Rekursion über mehrere Argumente
2. manchmal wachsen rekursive Argumente
3. nichtlineare Rekursion mit verschiedener „Verkleinerung“
4. Kombination von 1. und 2. Ein rekursives Argument wächst, eines fällt

Präzisierung mittels wohlfundierter Relation

Definition (wohlfundierte Relation)

Menge X , $\succ \subseteq X \times X$

Kette ist eine Folge x_0, x_1, \dots von Elementen aus X mit $x_i \succ x_{i+1}$ (für $i \geq 0$)

\succ heißt wohlfundiert, wenn es nur endlich lange Ketten gibt

Beispiel: $>$ auf \mathbb{N} , nicht aber $>$ auf \mathbb{Z}

Intuitiv klar: kann man eine wohlfundierte Relation finden, so dass sie Argumente von aufeinander folgenden Aufrufen einer Funktion jeweils in der Relation stehen, so terminiert die Funktion.

Beispiel:

fun $p(n, x) = \text{if } n \times n > x \text{ then } n - 1 \text{ else } p(n + 1, x)$

Welche Kette von Paaren aus $\mathbb{Z} \times \mathbb{Z}$ durchläuft p ?

$(n_1, x_1), (n_2, x_2), \dots$ mit $x_i = x_{i+1}$, $n_{i+1} = n_i + 1$ und $n_i^2 \leq x$

Die folgende Relation ist wohlfundiert:

$(n_1, x_1) \succ (n_2, x_2)$ gdw $n_1 < n_2 \wedge x_1 = x_2 \wedge n_1^2 \leq x_1$

x -Komponente bleibt unverändert

n -Komponente wächst um 1 in jedem Aufruf

Die Terminierungsbedingung $n \times n > x$ wird sicher nach endlicher Zeit erreicht.

Beispiel: Fibonacci

fun $\text{fib } n = \text{if } n < 2 \text{ then } n \text{ else fib } (n - 1) + \text{fib } (n - 2)$

Die beiden Aufrufe produzieren die Folgen

$n, n - 1, \dots$

$n, n - 2, \dots$

Eine passende wohldefinierte Funktion:

$n_1 \succ n_2$ gdw $n_1 > n_2 \geq 0$ bricht nach endlich vielen Schritten ab

13.2 Laufzeit

Funktion p angewendet auf $x : t$

Wie kann man eine Funktion $\text{time}_p : t \rightarrow \mathbb{N}$ definieren, die die (exakte) Laufzeit der Anwendung von p auf x (in der Form Anzahl von Ausführungsschritten) angibt?

Die Definition soll unabhängig sein von

- dem ausführenden Rechner,
- der Implementierung der Programmiersprache durch einen Interpreter oder Übersetzer

Möglichkeit: operationale Semantik, hier Inferenzkalkül; gibt Anzahl von Auswertungsschritten an

Bei uns: Jede Anwendung einer Inferenzregel hat Kosten 1

Gesamtlaufzeit = Gesamtkosten = Größe des Inferenzbaums

Am Ende der Abstraktion von konstanten Beiträgen zur Laufzeit bleibt:

Anzahl der rekursiven Funktionsaufrufe

$\text{time}_p : t \rightarrow \mathbb{N}$ ist nicht wirklich interessant

Statt dessen $\text{time}_p : \mathbb{N} \rightarrow \mathbb{N}$

Argument ist die Größe der Funktionsargumente

Problem: time_p ist eventuell keine Funktion; denn eventuell braucht p für verschiedene Argumente der Größe n verschieden viele Schritte.

13.2.1. Laufzeit im schlechtesten Fall (worst case execution time)

$$\text{wc} - \text{time}_p(n) = \max_{x: |x|=n} \text{time}_p(x)$$

Die Laufzeit für das ungünstigste Argument der Länge n

Gibt es eine Funktion $f : \mathbb{N} \rightarrow \mathbb{N}$ mit

$$\forall n. \text{wc} - \text{time}_p(n) \leq f(n),$$

so ist f eine obere Schranke für die Laufzeit von p .

13.2.2. Laufzeit im besten Fall (best case execution time)

$$\text{bc} - \text{time}_p(n) = \min_{x: |x|=n} \text{time}_p(x)$$

Gibt es eine Funktion g mit $\text{bc} - \text{time}_p(n) \geq g(n)$ für alle n , so hat man eine **untere Schranke** für die Laufzeit von p .

13.2.3. Durchschnittliche (mittlere) Laufzeit (average case execution time)

Annahme: alle Argumente der Länge n treten gleich häufig auf.

$$\text{av-time}_p(n) = \frac{\sum_{x: |x|=n} \text{time}_p(x)}{\text{card}(\{x \mid |x|=n\})}$$

$\text{card } X$ = Mächtigkeit der Menge X

Damit haben wir drei Funktionen für die Laufzeit.

Abstrahieren von Termen kleinerer Ordnung und von Konstanten in der Definition dieser Funktionen

13.3 Asymptotische Notation

Die Größenordnung der Laufzeit wird angegeben durch eine Funktionsklasse, in der die Laufzeitfunktion liegt.

Definition (dominiert):

Seien $f, g \in \mathbb{N} \rightarrow \mathbb{R}_+$: wenn $\exists c > 0 \quad \exists n_0 \in \mathbb{N} \quad \forall n \geq n_0: g(n) \leq c \cdot f(n)$

f **dominiert** g , i.Z. $f \succeq g$,

Eigenschaften von „ \succeq “:

- Reflexivität: $\forall f \succeq f$
- Transitivität: $\forall f, g, h: f \succeq g$ und $g \succeq h$, dann $f \succeq h$

Definition (O , Ω (Omega), Θ (Theta)):

Sei $f \in \mathbb{N} \rightarrow \mathbb{R}_+$

- $O(f) = \{g \in \mathbb{N} \rightarrow \mathbb{R}_+ \mid f \succeq g\}$
ist die Klasse der Funktionen, die von f dominiert werden
- $\Omega(f) = \{g \in \mathbb{N} \rightarrow \mathbb{R}_+ \mid g \succeq f\}$
ist die Klasse der Funktionen, die f dominieren
- $\Theta(f) = O(f) \cap \Omega(f)$

$O(f)$ ist gut für die Definition von asymptotischen oberen Schranken

$\Omega(f)$ ist gut für die Definition von asymptotischen unteren Schranken

Ist $g \in \Theta(f)$, dann ist das Wachstum von g durch f ziemlich genau beschrieben. $g(n)$ verläuft in „Streifen“ $[c_1 \cdot f(n), c_2 \cdot f(n)]$ mit Konstanten $c_1 < c_2$.

Gesucht sind möglichst einfache Funktionen f mit $\text{wc-time}_p \in O(f)$ oder $\in \Theta(f)$

Beispiele:

- 1) $3n^4 + 5n^3 + 7n \leq 3n^4 + 5n^4 + 7n^4 = 15n^4$ für $n \geq n_0 = 1$
für $n \geq 1$ Mit $c = 15$ und $n_0 = 1$ gilt: $3n^4 + 5n^3 + 7n \in O(n^4)$
- 2) $1 + \sin n \in \Theta(1)$
- 3) $3n^2 \cdot \log n + 4n^2 + 3n \in \Theta(n^2 \cdot \log n)$
wobei $\log n = \begin{cases} 0 & \text{falls } n = 0 \\ \log_2 n & \text{sonst} \end{cases}$

Sätze: „ \subset “ echte Inklusion

- 1) $O(n \cdot \log n) \subset O(n^a) \subset O(n^b)$ für $1 < a < b$
- 2) $O(n^a) \subset O(b^n) \subset O(c^n)$ für $1 < a$ und $1 < b < c$
- 3) $O(1), \Theta(\log n), \Theta(n \cdot \log n), \Theta(n^2), \Theta(n^3), \Theta(2^n)$
sind disjunkt
- 4) $O(1) \subset O(\log n) \subset O(n) \subset O(n \cdot \log n) \subset O(n^2) \subset O(n^3) \subset O(2^n)$

13.4 Asymptotische Laufzeit

Uns interessieren asymptotische obere Schranken

$wc - time_p \in O(f)$ für welches f ?

$wc - time_p \in \Theta(f)$ für welches f ?

Solche f heißen asymptotische obere Schranken bzw. im Falle, dass es ein f mit

$wc - time_p \in \Theta(f)$ gibt, **scharfe** obere Schranke.

Sprachgebrauch:

- $O(1)$ konstante Laufzeit
- $\Theta(\log n)$ logarithmische Laufzeit
- $\Theta(n)$ lineare Laufzeit

Beispiele:

- | | | |
|------------------------|----------------|---------------------------|
| • Fakultätsfunktion | fact | $\Theta(n)$ |
| • Konkatenation | append (xs,ys) | $\Theta(xs)$ |
| • naives Spiegeln | nrev xs | $\Theta(xs ^2)$ |
| • cleveres Spiegeln | crev xs | $\Theta(xs)$ |
| • Suche in BST | lookup (s,b) | $O(\text{size } b)$ |
| • Suche in balanc. BST | | $O(\log(\text{size } b))$ |

Beispiele:

fun fib $n =$ if $n < 2$ then 1 else fib ($n - 1$) + fib ($n - 2$)

Jeder Aufruf verursacht zwei neue Aufrufe
 Also: Vermutung: exponentielles Wachstum
 fib n erzeugt fib n Aufrufe

1. Behauptung:

Es gibt Konstanten a, c und ein $n_0 \in \mathbb{N}$ mit $\text{fib } n \geq a \cdot c^n$ für alle $n \geq n_0$

Beweis: Induktion über n

Ind. Schritt: $\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)$

$$\geq a \cdot c^{n-1} + a \cdot c^{n-2}$$

Ind. Ann.

$$= a \cdot c^n \cdot \frac{c+1}{c^2}$$

$$\text{z.z.: } \frac{c+1}{c^2} \geq 1. \quad \text{Das ist äquivalent zu } c^2 \leq c+1 \text{ und } c^2 - c - 1 \leq 0$$

Lösen der quadratischen Gleichung $c^2 - c - 1 = 0$

$$\text{positive Lösung } c = \frac{\sqrt{5}+1}{2} \approx 1,618$$

Damit folgt für $c \leq \frac{\sqrt{5}+1}{2}$ die Behauptung

Ind. Anfang: Ziel: Bestimmung von a und n_0

$\text{fib } (2) = 2 \geq a \cdot c^2$ muss gelten

$$a \cdot c^2 = a \cdot \left(\frac{\sqrt{5}+1}{2} \right)^2 = a \cdot \frac{3+\sqrt{5}}{2}$$

$$\text{Wir wählen } a = \frac{1}{3+\sqrt{5}}$$

Gezeigt: $\text{fib } n \geq a \cdot c^n$

2. Behauptung:

$\text{fib } n \leq b \cdot d^n$

Wird wieder durch Induktion über n bewiesen.

Dabei ergeben sich die Konstanten $d = \frac{\sqrt{5}+1}{2}$ und b .

Insgesamt folgt: $\text{fib } n \in \Theta(f^n)$

Beispiel für allgemeine Rekursionsgleichung:

$$f(n) = F(f(1), f(2), \dots, f(n-1))$$

(bei fib 5 Parameter zu bestimmen)

Lösung ist eine Funktionsschar

Finden der Parameter bestimmt eine Funktion

→ enthält im Allgemeinen eine Anzahl von Parametern

13.5 Anwendung von SML

13.5.1 Balancierte BST

Erinnerung: Laufzeit von Operationen wie insert, lookup hängt von der Tiefe der Bäume ab.

Ziel: „balancierter“ Baum mit möglichst geringer Tiefe

Lösungen:

- AVL-Bäume
- Rot-Schwarz-Bäume
- 2-3, 2-5, a-b-Bäume

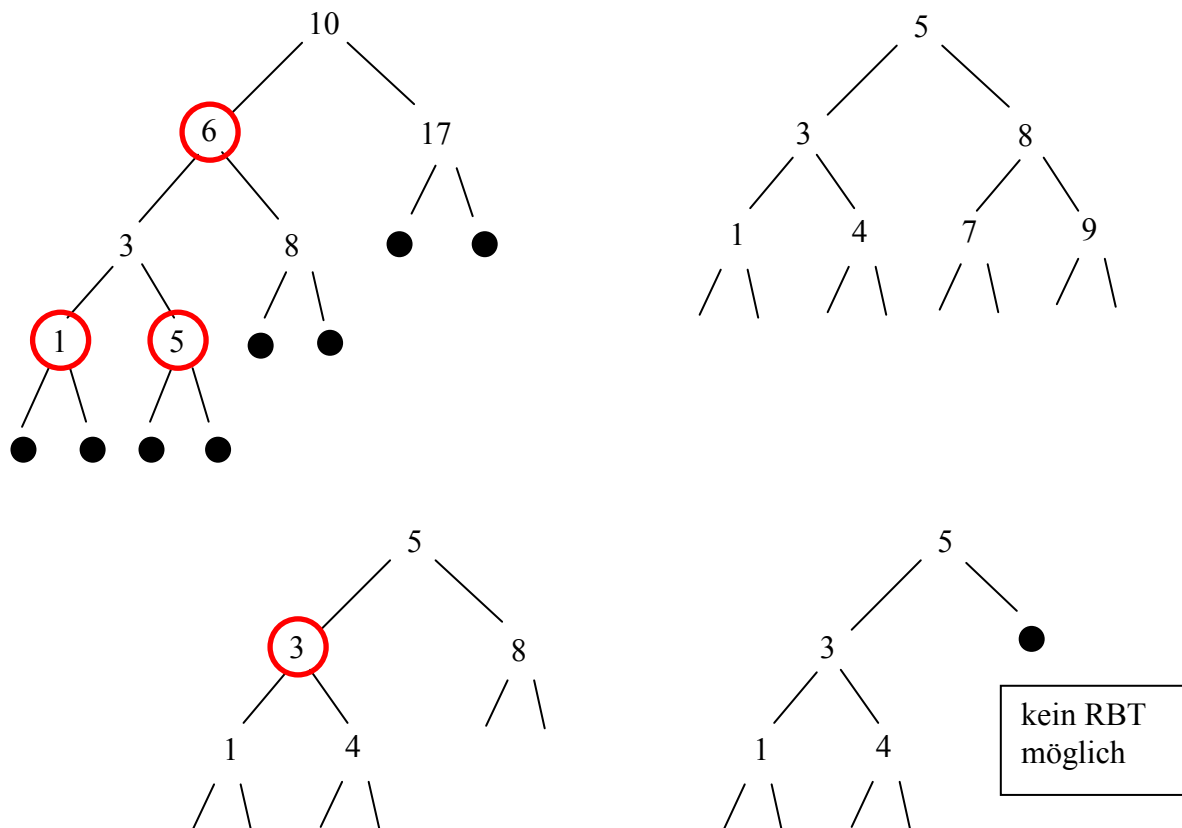
13.5.2 Rot-Schwarz-Bäume

Definition:

Ein Rot-Schwarz-Baum (=RBT) ist ein BST mit folgenden Eigenschaften:

- (I 1) Jeder Knoten hat eine zusätzliche Markierung Rot oder Schwarz (R,B)
- (I 2) Leere Knoten (Empty) sind immer schwarz
- (I 3) Kein roter Knoten hat ein rotes Kind
- (I 4) Jeder Pfad von der Wurzel zu einem Blatt enthält dieselbe Anzahl schwarzer Knoten

Beispiele:



Eigenschaften:

Intuitiv: Der längste Pfad in einem RBT (abwechselnd rote und schwarze Markierungen) ist höchstens doppelt so lange wie der kürzeste (nur schwarze Markierungen).

Satz:

Die Höhe eines RBT mit n inneren Knoten ist höchstens $2\log(n+1)$.

Beweis:

Sei $bh(x)$, die Schwarzhöhe des Knotens x , die Anzahl schwarzer Knoten auf einem Pfad von x zu einem Empty ohne x selbst. (bh wohldefiniert)

Lemma: Die Anzahl der inneren Knoten eines Teilbaums mit Wurzel x ist mindestens

$$2^{bh(x)} - 1.$$

Beweis durch vollständige Induktion über der Höhe von x

Ind.Anf.: Höhe=0

Teilbaum mit Wurzel x hat Größe $\geq 2^0 - 1 = 0$

Ind.Schritt: Sei x ein innerer Knoten mit Höhe > 0 und zwei Kindern

Jedes Kind hat Schwarzhöhe $bh(x)$ oder $bh(x)-1$

Klar: Höhe der Kinder ist kleiner als die von x selbst

\Rightarrow Größe des Teilbaums mit Wurzel x ist mindestens

$$\underbrace{2^{bh(x)-1} - 1}_{\text{I.V. linker Teilbaum}} + \underbrace{2^{bh(x)-1} - 1}_{\text{I.V. rechter Teilbaum}} + \underbrace{1}_x = \underline{2^{bh(x)} - 1} \quad \blacksquare \text{ (Lemma)}$$

Folgerung für den Satz: Sei h die Höhe eines RBT mit Wurzel x und n inneren Knoten

$$\Rightarrow n \geq 2^{bh(x)} - 1$$

$$\Rightarrow n \geq 2^{h/2} - 1$$

$$\Rightarrow h \leq 2\log(n+1) \quad \blacksquare$$

13.5.3 Implementierung von RBT in ML

datatype Colour = R | B

datatype 'a Rbtree = E

| N of Colour * 'a * 'a Rbtree * 'a Rbtree

Farbe hat keinen Einfluss auf die **Suche in RBT**

fun lookup (x, E) = false

| lookup (x, N(_, y, l, r)) =

if $x < y$ **then** lookup (x, l)

else if $y < x$ **then** lookup (x, r)

else true

Einfügen (insert) ist schwieriger, weil die Invarianten erhalten bleiben müssen

```

fun insert (x, t) =
  let fun ins (x, E) = N (R, x, E, E)
      | ins (x, t as N (c, y, l, r)) =
          if x < y then balance (c, y, ins (x, l), r)
          else if y < x then balance (c, y, l, ins (x, r))
          else t
  in val (_, y, l, r) = ins (x, t)
    in N (B, y, l, r)
    end
  end

```

Änderungen im Vergleich zu insert bei BST :

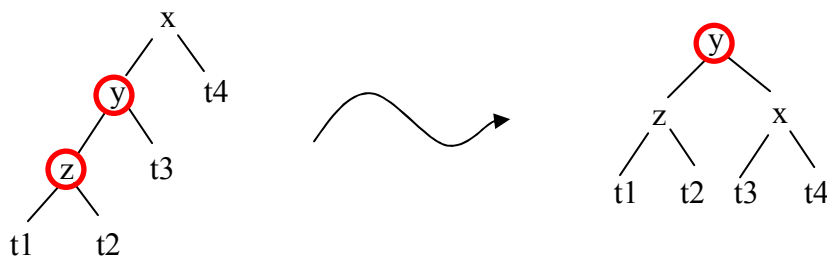
1. Neu eingefügte Knoten sind rot
2. Aufruf an Knotenkonstruktor wird ersetzt durch balance
3. Nach Einfügen Wurzel schwarz

13.5.4 Balancieren

Durch Einfügen eines roten Knotens kann nur (I 3) verletzt werden

⇒ Rotation

1.Fall :

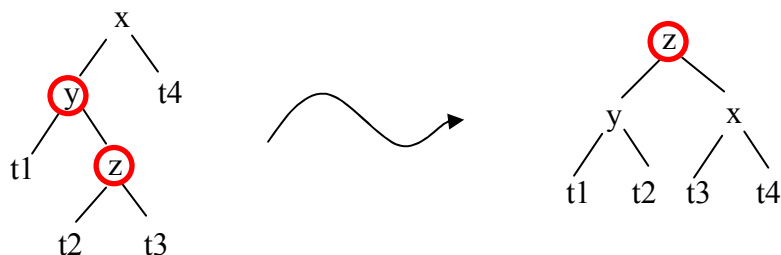


```

fun balance (B, x, N (R, y, N (R, z, t1, t2), t3), t4) =
  N (R, y, N (B, z, t1, t2), N (B, x, t3, t4))

```

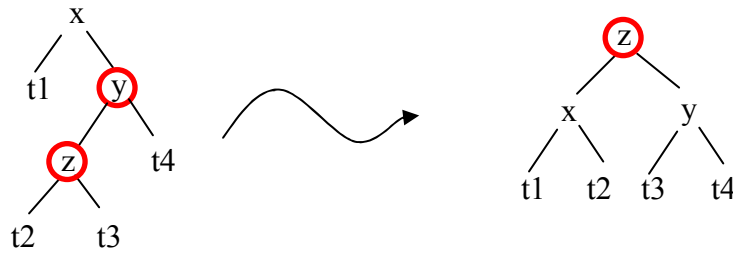
2.Fall :



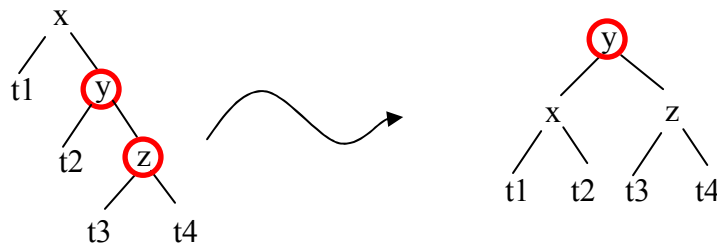
```

| balance (B, x, N (R, y, t1, N (R, z, t2, t3)), t4) =
  N (R, z, N (B, y, t1, t2), N (B, x, t3, t4))

```


3.Fall :

| balance (B, x, t1, N (R, y, N (R, z, t2, t3), t4)) =
 N (R, z, N (B, x, t1, t2), N (B, y, t3, t4))

4.Fall :

| balance (B, x, t1, N (R, y, t2, N (R, z, t3, t4))) =
 N (R, y, N (B, x, t1, t2), N (B, z, t3, t4))

sonst:

| balance s =
 N s

Nach der Balancierung eines Teilbaums kann die neue rote Wurzel wieder Kind eines roten Knotens sein.

⇒ Rekursive Rotation eventuell bis zur Wurzel, die dann schwarz gefärbt wird

Laufzeiten im schlechtesten Fall:

Wie bei BST: abhängig von der Tiefe

lookup: $O(\log n)$ (Satz!)

insert: $O(\log n)$ (Einfügen + Rotation jeweils $O(\log n)$)