Funktionale Programmierung

Zirkuläre Programme

Hans-Wolfgang Loidl, Andreas Abel

LFE Theoretische Informatik, Institut für Informatik,
Ludwig-Maximilians Universität, München

11. Mai 2009

Die Vorteile von Bedarfsauswertung

- **Modularisierung** des Codes
- Vermeidung unnützer Berechnungen
- Hoher Coolness-Faktor
- Verwendung **unendlicher und zirkulärer Datenstrukturen**

Definitionen

- Ein **zirkuläres** Programm erzeugt eine Datenstruktur, deren Berechnung von sich selbst abhängt.
- Solche Programme erfordern nicht-strikte Datenstrukturen.
- Diese werden in Sprachen mit nicht-strikter Semantik unterstützt, und können in Sprachen mit strikter Semantik simuliert werden.
- Anwendungsbeispiele: zirkuläre Listen, doppelt verkettete Listen, etc
- Zirkuläre Programme werden oft verwendet um mehrfaches Traversieren einer Datenstruktur oder den Aufbau von Zwischen-Datenstrukturen zu vermeiden.

Zirkuläre Datenstrukturen

Ein einfaches Beispiel einer zirkulären Datenstruktur ist die Liste *aller* natürlichen Zahlen:

$$nats = 1 : (map\ (+1)\ nats)$$

Beachte: Dieses Program benutzt die Datenstruktur, die es erzeugt, selbst als Eingabe.

Beispiel: Eliminierung von Duplikaten

Die Funktion *nub* aus Übung 1 eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

$$
\begin{aligned}
&\mathit{nub1} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{nub1}\ [\,] \qquad = [\,] \\
&\mathit{nub1}\ (x : xs) = x : (\mathit{nub1}\ (\mathit{filter}\ (\not\equiv x)\ xs))
\end{aligned}
$$

Beispiel: Eliminierung von Duplikaten

Die Funktion *nub* aus Übung 1 eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

$$
\begin{aligned}
&\mathit{nub1} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
&\mathit{nub1}\ [\,] \qquad = [\,] \\
&\mathit{nub1}\ (x : xs) = x : (\mathit{nub1}\ (\mathit{filter}\ (\not\equiv x)\ xs))
\end{aligned}
$$

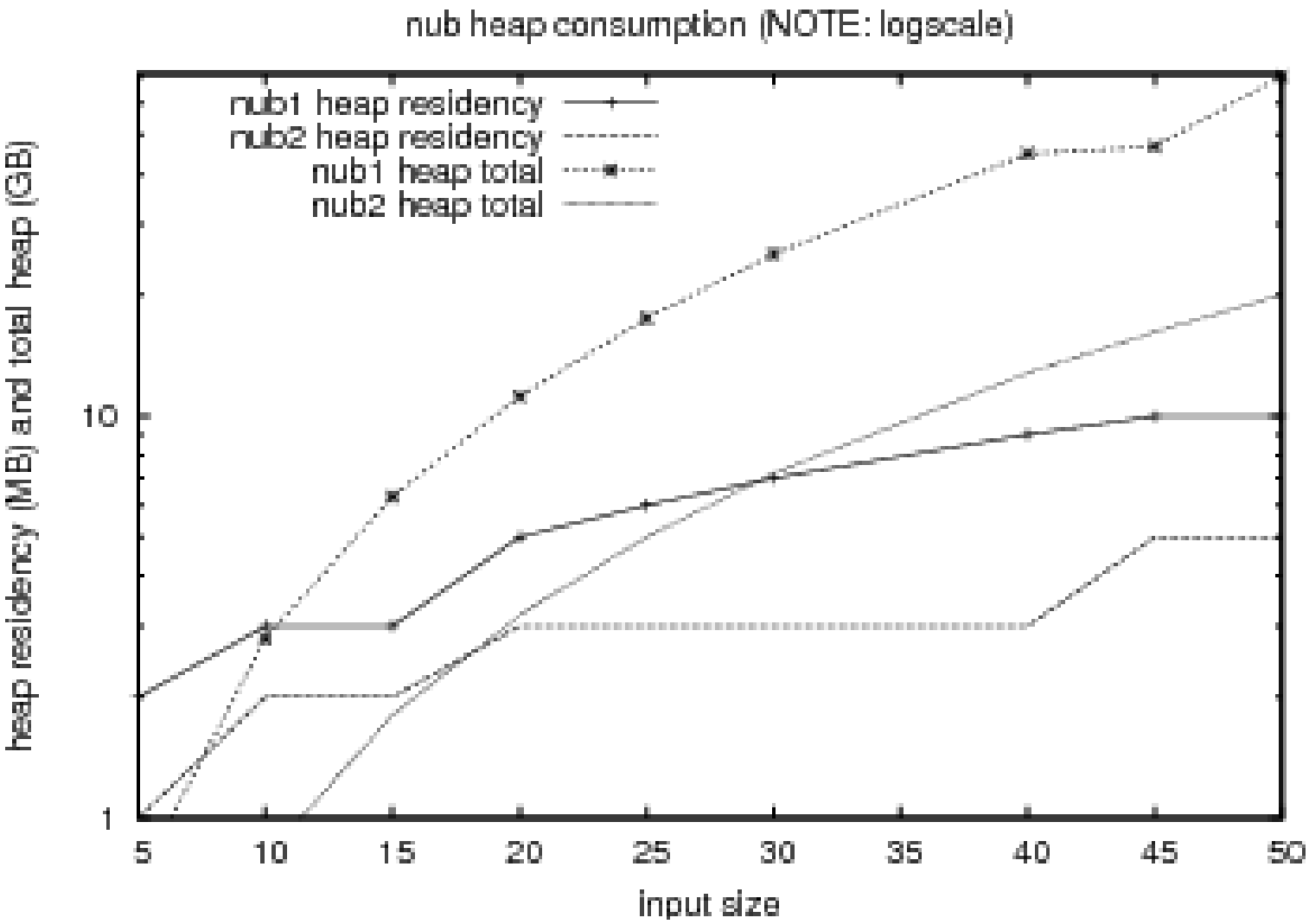
Diese Funktion erzeugt für jedes Listenelement der Eingabeliste eine Liste als Zwischen-Datenstruktur.

Beispiel: Eliminierung von Duplikaten

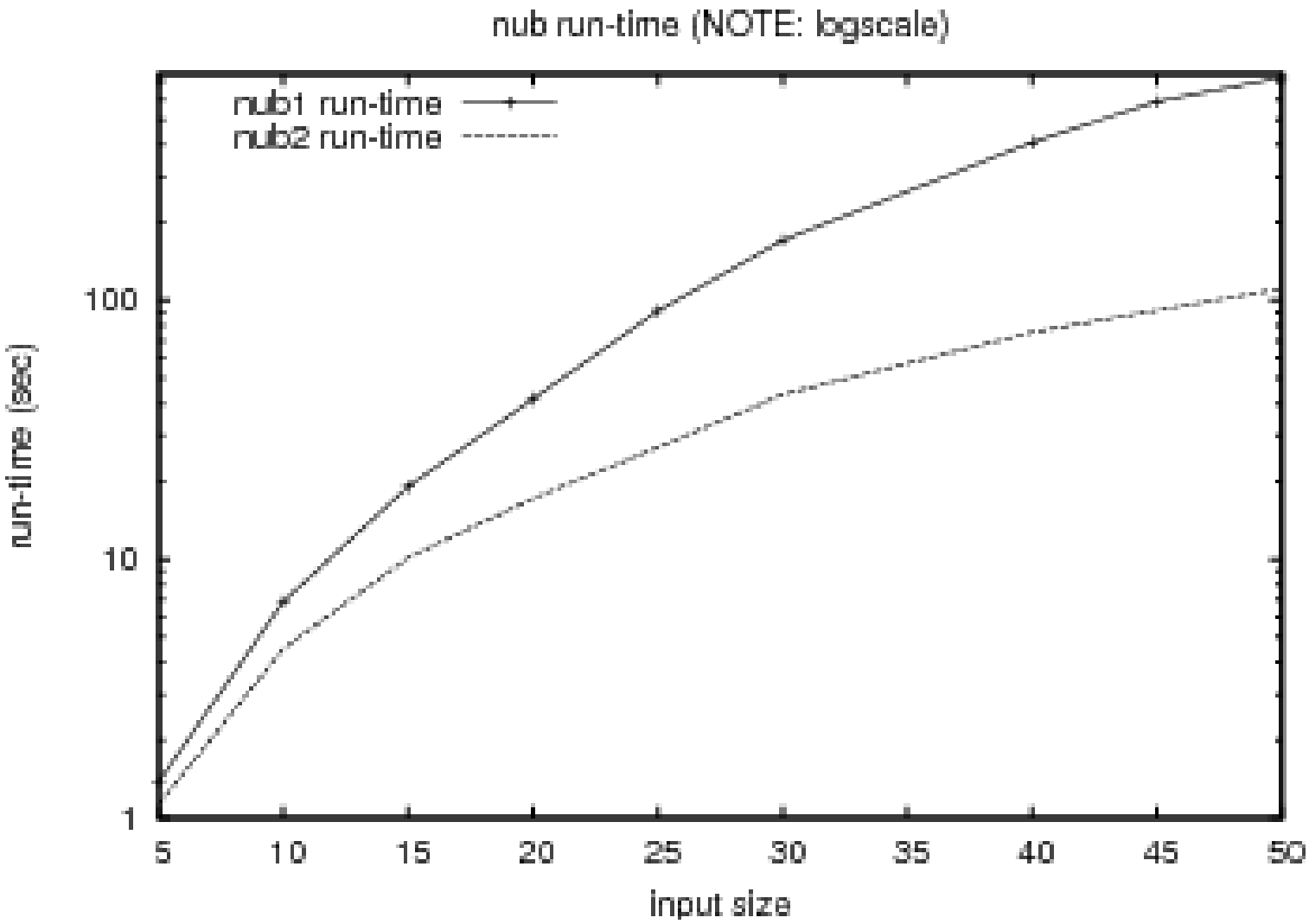
Eine bessere Implementierung vermeidet diese Listen:

$$
\begin{array}{l}
\mathit{nub2} :: [\mathit{Integer}] \to [\mathit{Integer}] \\
\mathit{nub2}\ \mathit{xs} = \mathit{res} \\
\quad \textbf{where}\ \mathit{res} = \mathit{build}\ \mathit{xs}\ 0 \\
\qquad \begin{array}{lll}
\mathit{build}\ [\,]\ n & & = [\,] \\
\mathit{build}\ (x : \mathit{xs})\ n & \mid \mathit{mem}\ x\ \mathit{res}\ n = \mathit{build}\ \mathit{xs}\ n \\
& \mid \mathit{otherwise} = x : (\mathit{build}\ \mathit{xs}\ (n+1)) \\
\mathit{mem}\ \_\ \_\ 0 & & = \mathit{False} \\
\mathit{mem}\ x\ (y : \mathit{ys})\ n & \mid x \equiv y = \mathit{True} \\
& \mid \mathit{otherwise} = \mathit{mem}\ x\ \mathit{ys}\ (n-1)
\end{array}
\end{array}
$$

Performanz


nub heap consumption (NOTE: logscale)
nub1 heap residency
nub2 heap residency
nub1 heap total
nub2 heap total
heap residency (MB) and total heap (GB)
10
1
5
10
15
20
25
30
35
40
45
50
input size

Performanz


nub run-time (NOTE: logscale)
nub1 run-time
nub2 run-time
run-time (sec)
100
10
1
5
10
15
20
25
30
35
40
45
50
input size

Beispiel: repmin

Gegeben: Binärer Baum von Integer Werten.
Gesucht: Binärer Baum mit der gleichen Struktur in dem die Werte der Blätter durch das kleinste Element im Baum ersetzt sind.

Naive Implementierung

Die naive Implementierung verwendet 2 separate Funktionen:

$$
\begin{aligned}
&\mathit{treemin} :: (\mathit{Ord}\ a) \Rightarrow \mathit{BinTree}\ a \to a \\
&\mathit{treemin}\ (\mathit{Leaf}\ x) = x \\
&\mathit{treemin}\ (\mathit{Node}\ l\ r) = \mathit{min}\ (\mathit{treemin}\ l)\ (\mathit{treemin}\ r)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{replace} :: \mathit{BinTree}\ a \to a \to \mathit{BinTree}\ a \\
&\mathit{replace}\ (\mathit{Leaf}\ \_)\ m = \mathit{Leaf}\ m \\
&\mathit{replace}\ (\mathit{Node}\ l\ r)\ m = \mathit{Node}\ (\mathit{replace}\ l\ m)\ (\mathit{replace}\ r\ m)
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{transform2p} :: (\mathit{Ord}\ a) \Rightarrow \mathit{BinTree}\ a \to \mathit{BinTree}\ a \\
&\mathit{transform2p}\ t = \mathit{replace}\ t\ (\mathit{treemin}\ t)
\end{aligned}
$$

Dieser Code traversiert den Baum 2-mal: einmal in *treemin* und einmal in *replace*.

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir Verschmelzen beide Funktionen in eine einzige:

$$
\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ x)\ m &&= (\mathit{Leaf}\ m, x)\\
&\mathit{repmin}\ (\mathit{Node}\ l\ r)\ m &&= (l', ml)\ `\mathit{comb}`\ (r', mr)\\
&&&\quad \textbf{where}\ (l', ml) = \mathit{repmin}\ l\ m\\
&&&\qquad (r', mr) = \mathit{repmin}\ r\ m\\
&&&\qquad \mathit{comb}\ (l', ml)\ (r', mr) =\\
&&&\qquad\quad (\mathit{Node}\ l'\ r', \mathit{min}\ ml\ mr)
\end{aligned}
$$

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir Verschmelzen beide Funktionen in eine einzige:

$$
\begin{aligned}
\mathit{repmin}\ (\mathit{Leaf}\ x)\ m &= (\mathit{Leaf}\ m, x) \\
\mathit{repmin}\ (\mathit{Node}\ l\ r)\ m &= (l', ml)\ `\mathit{comb}`\ (r', mr) \\
&\quad \textbf{where}\ (l', ml) = \mathit{repmin}\ l\ m \\
&\qquad (r', mr) = \mathit{repmin}\ r\ m \\
&\qquad \mathit{comb}\ (l', ml)\ (r', mr) = \\
&\qquad\quad (\mathit{Node}\ l'\ r', \mathit{min}\ ml\ mr)
\end{aligned}
$$

Mittels Gleichheitsschließen lässt sich einfach zeigen dass:

$$\mathit{repmin}\ t\ (\mathit{treemin}\ t) = (\mathit{replace}\ t\ (\mathit{treemin}\ t), \mathit{treemin}\ t)$$

Verbesserte Implementierung

Beobachtung: Die Strukturen beider Funktionen sind gleichartig.
Wir Verschmelzen beide Funktionen in eine einzige:

$$
\begin{aligned}
&\mathit{repmin}\ (\mathit{Leaf}\ x)\ m &&= (\mathit{Leaf}\ m, x)\\
&\mathit{repmin}\ (\mathit{Node}\ l\ r)\ m &&= (l', ml)\ `\mathit{comb}`\ (r', mr)\\
&&&\quad \textbf{where}\ (l', ml) = \mathit{repmin}\ l\ m\\
&&&\qquad (r', mr) = \mathit{repmin}\ r\ m\\
&&&\qquad \mathit{comb}\ (l', ml)\ (r', mr) =\\
&&&\qquad\quad (\mathit{Node}\ l'\ r', \mathit{min}\ ml\ mr)
\end{aligned}
$$

Nun können wir eine zirkuläre Datenstruktur verwenden um das Resultat der *treemin* Komponente mit der Eingabe der *replace* Komponente zu verknüpfen:

$$
\begin{aligned}
&\mathit{transform1p}\ t = \mathit{fst}\ p\\
&\quad \textbf{where}\ p = \mathit{repmin}\ t\ (\mathit{snd}\ p)
\end{aligned}
$$

Memoisation

- **Memoisation** ist eine Programmier- oder Implementierungstechnik, in der die Resultate früherer Aufrufe einer Funktion gespeichert ("memoised") werden, um wiederholtes Auswerten zu vermeiden.
- Im Allgemeinen verwendet man dazu eine Datenstruktur, wie eine Hash-Tabelle, und legt dort die bereits berechneten Werte ab.
- Dieser Ansatz ist von Vorteil, wenn die einzelnen Berechnungen sehr teuer sind, und den Administrationsaufwand der Tabellenverwaltung rechtfertigen.

Light-weight Memoisation

Zirkuläre Datenstrukturen bieten eine leichtgewichtige Alternative zu generellen Hash-Tabellen.

Anstatt eine Funktion direkt zu definieren, wird eine unendliche Liste aller Resultatwerte definiert. Der Funktionsaufruf wird durch eine Indexing-Operation auf der Liste ersetzt.

$$
\begin{aligned}
&\mathit{fibs} :: [\mathit{Int}] \\
&\mathit{fibs} = 1 : 1 : (\mathit{zipWith}\ (+)\ \mathit{fibs}\ (\mathit{tail}\ \mathit{fibs})) \\
&\mathit{fib} :: \mathit{Int} \to \mathit{Int} \\
&\mathit{fib}\ n = \mathit{fibs}\ !!\ n
\end{aligned}
$$

Termination von zirkulären Programmen

Die Termination von zirkulären Programmen zu beweisen ist oft nicht trivial.
Im Allgemeinen benutzt man das Konzept der Produktivität um die Termination von zirkulären Programmen zu zeigen.

Definition (maximal)

Gegeben eine partielle Ordnung $\sqsubseteq$ auf einer Menge D. Ein Element $x \in D$ ist *maximal* wenn

$$\forall y \in D.y \sqsubseteq x$$

$\sqsubseteq$ auf Funktionen und Listen wird komponentenweise definiert.

Termination von zirkulären Programmen

Definition (produktiv)

Produktivität von $x \in \sigma$ wird induktiv über die Struktur von σ definiert

- $\sigma = Int \vee \sigma = Bool$: x ist maximal
- $\sigma = \tau \to \tau'$: x bildet produktive Elemente in produktive Elemente ab
- $\sigma = [\tau]$: $\forall i.0 \sqsubseteq i \sqsubseteq \#x \to x!!i$ ist produktiv

Termination von zirkulären Programmen

Definition (Mengen-Produktivität)

Eine Liste $x \in [\sigma]$ ist A-produktiv ($A \in \mathbb{N}$), gdw $\forall a \in A.x!!a$ ist produktiv.

Intuition: A ist die Menge der Indizes von berechneten Listen-Elementen.

Termination von zirkulären Programmen

Definition (Segment-Produktivität)

Eine Liste $x \in [\sigma]$ ist n-produktiv ($n \in \mathbb{N}$), gdw x ist $\mathbb{N}^k$ produktiv.

Intuition: die ersten n Elemente der Liste sind berechnet.

Definition (Segment-Produktivität)

Eine Funktion $f \in [\sigma] \to [\tau]$ ist $v \in \mathbb{N} \to \mathbb{N}$ produktiv, gdw
$\forall k \in \mathbb{N}. x \in [\sigma]. x$ ist k produktiv $\to (f\ x)$ ist $(v\ k)$ produktiv

Intuition: die v Funktion gibt an, wie sich die Größe des berechneten Segments verändert.

Termination von zirkulären Programmen

Example

Die Funktion *map f* ist *id* produktiv, d.h. die Segmentgröße bleibt unverändert.

Example

Die List-cons Funktion : ist $(+1)$ produktiv, d.h. die Segmentgröße steigt um 1.

Termination von zirkulären Programmen

Theorem (Listen-Produktivität)

Sei eine Funktion $f \in [\sigma] \to [\tau]$ $v \in \mathbb{N} \to \mathbb{N}$-produktiv, and $\forall k \in \mathbb{N}.v\ k > k$. Dann ist der Fixpunkt von f eine produktive, unendliche Liste.

Intuition: Um zu zeigen, dass alle Element einer unendlichen Liste berechenbar sind, muss man zeigen, dass die Änderung der Segmentgröße immer ansteigt. D.h. in jeder Iteration steigt der berechnete Teil der Liste.

Termination von zirkulären Programmen

Example

Produktivität der memoisierenden Fibonacci Funktion.

- Die Funktion *zipWith* ist *min* produktiv, da sie eine binäre Funktion f komponentenweise auf die beiden Argumentlisten xs, ys anwendet. Wenn die ersten m Elemente von xs und die ersten n Elemente von ys berechnet sind, dann sind die ersten $min\ m\ n$ Elemente der Resultatliste berechnet.
- Wir wissen, dass die List-cons Funktion $: (+1)$ produktiv ist.
- Sei *fibs* eine k produktive Liste, und *vtl* die Produktivitätsfunktion von *tail*. Dann ist der Rumpf $1 : 1 : (zipWith\ fibs\ (tail\ fibs))\ 1 + 1 + min\ k\ (vtl\ k)$ produktiv.
- Weiters ist $vtl\ k = k - 1$ für alle $k > 0$.
- Daher ist die Produktivität des Rumpfs von *fibs*: $1 + 1 + min\ k\ (vtl\ k) = 2 + k - 1 = k + 1 > k$ □.

Struktur von Rot-Schwarz Bäumen

- **Rot-Schwarz Bäume** sind eine spezielle Form von binären Suchbäumen mit gefärbten Knoten.
- Sie ermöglichen Suche in logarithmischer Zeit.
- Einfügen erfordert eine **lokale** Balanzierung des Baums.
- Es gelten folgende Invarianten:
  - Kein roter Knoten hat einen roten Nachfolger.
  - Jeder Pfad von der Wurzel zu einem Blatt enthält die gleiche Anzahl von schwarzen Knoten.
- Die maximale Tiefe eines Rot-Schwarz-Baums mit n Elementen ist $2\lfloor \log(n+1) \rfloor$.

Datentyp

Die Datenstruktur eines Rot-Schwarz-Baums erweitert einen binären Baum um ein Feld, das die Farbe modelliert.

```
data Color = Red | Black
       deriving (Eq, Show)
data Ord a ⇒
   Tree a = Empty
      | Node Color (Tree a) a (Tree a)
       deriving (Eq, Show)
```

Suche

Die wichtigste Operation in einem binären Baum ist die Suche. Wir können denselben Algorithmus wie für einfache binäre Bäume verwenden.

$$
\begin{aligned}
&\mathit{elemOf} :: \mathit{Ord}\ a \Rightarrow a \to \mathit{Tree}\ a \to \mathit{Bool}\\
&\mathit{elemOf}\ x\ \mathit{Empty} = \mathit{False}\\
&\mathit{elemOf}\ x\ (\mathit{Node}\ \_\ l\ y\ r) \mid x \equiv y \quad = \mathit{True}\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid x < y \quad = x\ \text{`}\mathit{elemOf}\text{`}\ l\\
&\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = x\ \text{`}\mathit{elemOf}\text{`}\ r
\end{aligned}
$$

Einfügen

- Das Einfügen in einen Rot-Schwarz-Baum erfordert zusätzliche Operationen (Balanzierung), um zu garantieren, dass die beiden Invarianten erfüllt bleiben.
- Wir verwenden die Einfüge Operation für binäre Bäume, und erweitern sie folgendermaßen:
  - Ein neuer Knoten wird immer Rot gefärbt.
  - Die Wurzel des Baums muss immer Schwarz sein.
  - Nach jedem rekursiven Aufruf des Einfügens wird eine Balanzierungsfunktion aufgerufen.

Einfügen

```
insert :: Ord a ⇒ a → Tree a → Tree a
insert x t = Node Black a y b
  where ins Empty = Node Red Empty x Empty
        ins t@(Node col a y b)
          | x < y = balance col (ins a) y b
          | x > y = balance col a y (ins b)
          | otherwise = t
        (Node _ a y b) = ins t
```

Balanzieren des Baums

- Da der neue Knoten Rot ist, bleibt Invariante 2 unverändert.
- Aber Invariante 1 ist dadurch möglicherweise verletzt.
- Wir müssen daher den neuen Baum balanzieren.
- Dazu reicht es nur die lokale Umgebung eines roten Knotens zu betrachten.

Balanzierung

```
balance :: Ord a ⇒ Color → Tree a → a → Tree a → Tree a
balance Black (Node Red (Node Red a x b) y c) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balance Black (Node Red a x (Node Red b y c)) z d =
  Node Red (Node Black a x b) y (Node Black c z d)
balance Black a x (Node Red (Node Red b y c) z d) =
  Node Red (Node Black a x b) y (Node Black c z d)
balance Black a x (Node Red b y (Node Red c z d)) =
  Node Red (Node Black a x b) y (Node Black c z d)
balance c a x b = Node c a x b
```

Vergleich zu imperativen Implementierungen

- Der Code für das Einfügen ist viel kürzer als in klassischen imperativen Implementierungen.
- Das kommt einerseits von der kompakten Notation des Pattern Matching.
- Andererseits wird in imperativen Implementierungen eine größere lokale Umgebung, mit mehr Fällen betrachtet, um Operationen zu minimieren.
- Durch destruktive Operationen sind imperative Implementierungen effizienter.
- Eine ausgefeilte Garbage-Collection Politik kann den Overhead in der funktionalen Implementierung stark minimieren (generational GC).

Literatur

- R.S. Bird, "Using Circular Programs to Eliminate Multiple Traversals of Data, Acta Informatica, 21, 239–250 (1984)
- L. Allison, "Circular Programs and Self-Referential Structures", Software — Practice and Experience, 19(2), 99–109, Feb 1989.
- B.A. Sijtsma, "On the Productivity of Recursive List Definitions", TOPLAS, 11(4), 633–649, Oct 1989.
- C. Okasaki, "Purely Functional Data Structures", Cambridge University Press, 1998. ISBN 0-521-63124-6.