

Teil VII

Algorithmik I

14. Listen versus Arrays

- ▶ Zwei verschiedene Containertypen: Listen und Arrays
- ▶ Unterschiedliche Stärken und Schwächen:
 - ▶ Listen: flexibel — Erweiterung in konstanter Zeit (*Cons* aka '::');
 - ▶ Arrays: wahlfreier Zugriff in konstanter Zeit (*nth* aka $a.[i]$).
- ▶ Best of both worlds: flexible Arrays
 - ▶ Erweiterung in logarithmischer Zeit (*Cons* aka '::');
 - ▶ wahlfreier Zugriff in logarithmischer Zeit (*nth* aka $a.[i]$).
- ▶ Erste Implementierungsidee: Binärbäume

14. Wahlfreier Zugriff

- ▶ Welcher Knoten im Binärbaum korrespondiert zu welchem Index?



- ▶ *Idee*: die Zuordnung wird indirekt festgelegt, indem wir einen Baum auf eine Liste abbilden.
- ▶ *Spezifikation* des wahlfreier Zugriffs:

$$tree.[i] = (to-list\ tree).[i] \quad (12a)$$

$$(from-list\ list).[i] = list.[i] \quad (12b)$$

Wenn $to-list (from-list\ x) = x$, dann folgt (12b) aus (12a). (Beachte: wir überladen die Notation $xs.[i]!$)

- ▶ *Gesucht*: eine geschickte Definition von $to-list$!

14. Exploring the design space

Erste unstrittige (?) Festlegung: das Wurzelement ist das erste Element, das mit der Hausnummer 0. Aus diesem Grund setzen wir das Wurzelement vor den linken Teilbaum.

```
type Tree <'elem>
  | Leaf
  | Node of 'elem * Tree <'elem> * Tree <'elem>
```

Implementierung von *to-list*:

```
let rec to-list = function
  | Leaf      → []
  | Node (a, l, r) → a :: combine (to-list l, to-list r)
```

☞ Im Wesentlichen müssen wir festlegen, wie die beiden Listen für die Teilbäume kombiniert werden. Lassen wir uns damit etwas Zeit ...

14. Herleitung von *nth*

Referenzimplementierung: *nth* auf Listen.

```
let rec nth (list, i) =
  match list with
  | []      → None
  | x :: xs → if i = 0 then Some x else nth (xs, i - 1)
```

Herleitung von *nth* auf Binärbäumen. **Fall** *tree = Leaf*:

$$\begin{aligned}
 & \text{nth} (\text{Leaf}, i) \\
 = & \quad \{ \text{Spezifikation (12a)} \} \\
 & \text{nth} (\text{to-list Leaf}, i) \\
 = & \quad \{ \text{Definition von to-list} \} \\
 & \text{nth} ([], i) \\
 = & \quad \{ \text{Definition von nth} \} \\
 & \text{None}
 \end{aligned}$$

14. Herleitung von nth

Fall $tree = Node(a, l, r)$ **und** $i = 0$:

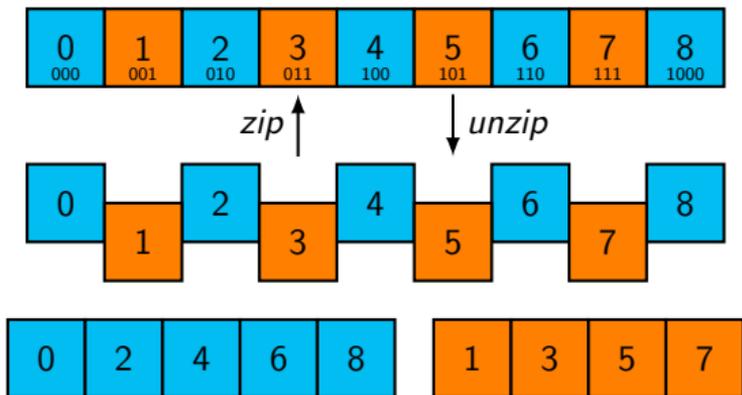
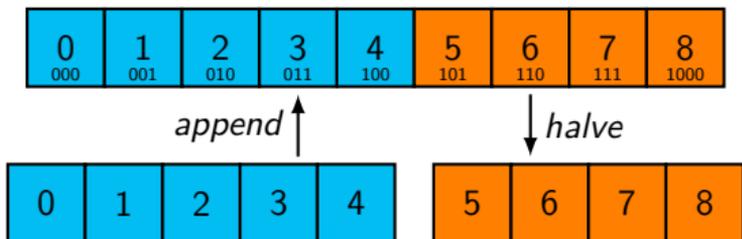
$$\begin{aligned} & nth(Node(a, l, r), 0) \\ = & \quad \{ \text{Spezifikation (13)} \} \\ & nth(to-list(Node(a, l, r)), 0) \\ = & \quad \{ \text{Definition von } to-list \} \\ & nth(a :: combine(to-list l, to-list r), 0) \\ = & \quad \{ \text{Definition von } nth \} \\ & Some a \end{aligned}$$

14. Herleitung von *nth*

Fall $tree = Node(a, l, r)$ **und** $i > 0$:

$$\begin{aligned}
 & nth(Node(a, l, r), i) \\
 = & \quad \{ \text{Spezifikation (13)} \} \\
 & nth(to-list(Node(a, l, r)), i) \\
 = & \quad \{ \text{Definition von } to\text{-list} \} \\
 & nth(a :: combine(to-list l, to-list r), i) \\
 = & \quad \{ \text{Definition von } nth \text{ und } i > 0 \} \\
 & nth(combine(to-list l, to-list r), i - 1) \\
 = & \quad \{ ? \}
 \end{aligned}$$

 Wir müssen festlegen, wie die beiden Listen für die Teilbäume kombiniert werden. Was ist eine geschickte Wahl?



14. Herleitung von *nth*

Eigenschaften von *nth*:

$$\begin{aligned} \text{nth}(\text{append}(xs, ys), i) &= \text{if } i < \text{length } xs \text{ then } \text{nth}(xs, i) \\ &\quad \text{else } \text{nth}(ys, i - \text{length } xs) \\ \text{nth}(\text{zip}(xs, ys), i) &= \text{if } i \% 2 = 0 \text{ then } \text{nth}(xs, i \div 2) \\ &\quad \text{else } \text{nth}(ys, i \div 2) \end{aligned}$$

☞ Verwenden wir *append*, benötigen wir Informationen über die Länge der Teillisten bzw. Größe der Teilbäume.

☞ Die Gleichung für *zip* setzt voraus, dass $0 \leq \text{length } xs - \text{length } ys \leq 1$. (Warum?)
Aber: Kenntnis der konkreten Längen ist nicht nötig.

14. Herleitung von *nth*

Fall $tree = Node(a, l, r)$ **und** $i > 0$:

$$\begin{aligned} & nth(zip(to-list\ l, to-list\ r), i - 1) \\ = & \{ \text{Eigenschaft, siehe oben} \} \\ & \mathbf{if}\ (i - 1) \% 2 = 0\ \mathbf{then}\ nth(to-list\ l, (i - 1) \div 2)\ \mathbf{else}\ \dots \\ = & \{ \text{Spezifikation (13)} \} \\ & \mathbf{if}\ (i - 1) \% 2 = 0\ \mathbf{then}\ nth(l, (i - 1) \div 2)\ \mathbf{else}\ \dots \end{aligned}$$

14. Implementierung von *nth*

Wir erhalten:

```

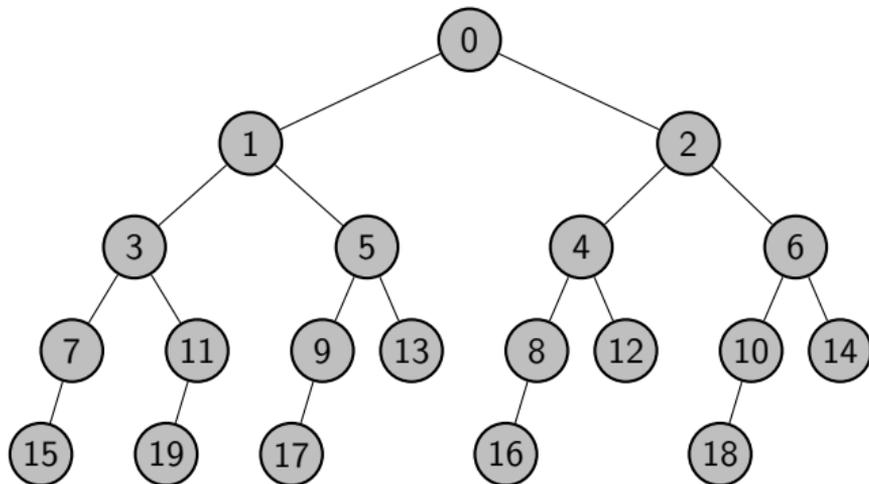
let rec nth (tree, i) =
  match tree with
  | Leaf          → None
  | Node (x, l, r) → if i = 0      then Some x
                    elif (i - 1) % 2 = 0 then nth (l, (i - 1) ÷ 2)
                    else nth (r, (i - 1) ÷ 2)
  
```

bzw.

```

let rec nth (tree, i) =
  match tree with
  | Leaf          → None
  | Node (x, l, r) → if i = 0      then Some x
                    elif i % 2 = 1 then nth (l, i ÷ 2)
                    else nth (r, i ÷ 2 - 1)
  
```

14. Nummerierung



14. Spezifikation von *cons*

Spezifikation der Erweiterung:

$$\textit{to-list} (\textit{cons} (a, \textit{tree})) = a :: \textit{to-list} \textit{tree} \quad (13)$$

Ziel: einen Ausdruck e mit $\textit{to-list} (\textit{cons} (a, \textit{tree})) = \textit{to-list} e$ herleiten, so dass e als Definition von $\textit{cons} (a, \textit{tree})$ dienen kann.

14. Herleitung von *cons*

Fall $tree = Leaf$:

$$\begin{aligned} & to-list (cons (a, Leaf)) \\ = & \{ \text{Spezifikation (13)} \} \\ & a :: to-list Leaf \\ = & \{ \text{Definition von } zip \} \\ & a :: zip ([], to-list Leaf) \\ = & \{ \text{Definition von } to-list \} \\ & a :: zip (to-list Leaf, to-list Leaf) \\ = & \{ \text{Definition von } to-list \} \\ & to-list (Node (a, Leaf, Leaf)) \end{aligned}$$

14. Herleitung von *cons*

Fall $tree = Node(b, l, r)$:

$$\begin{aligned}
 & to\text{-list}(cons(a, Node(b, l, r))) \\
 = & \quad \{ \text{Spezifikation (13)} \} \\
 & a :: to\text{-list}(Node(b, l, r)) \\
 = & \quad \{ \text{Definition von } to\text{-list} \} \\
 & a :: b :: zip(to\text{-list } l, to\text{-list } r) \\
 = & \quad \{ \text{Definition von } zip \} \\
 & a :: zip(b :: to\text{-list } r, to\text{-list } l) \\
 = & \quad \{ \text{Spezifikation (13)} \} \\
 & a :: zip(to\text{-list}(cons(b, r), to\text{-list } l)) \\
 = & \quad \{ \text{Definition von } to\text{-list} \} \\
 & to\text{-list}(Node(a, cons(b, r), l))
 \end{aligned}$$

14. Implementierung von *cons*

Wir erhalten:

```
let rec cons (a, tree) =  
  match tree with  
  | Leaf          → Node (a, Leaf, Leaf)  
  | Node (b, l, r) → Node (a, cons (b, r), l)
```

14. Invariante

Invariante: für jeden Knoten $Node(a, l, r)$ gilt, dass der linke Teilbaum l genau so viele Elemente wie der rechte Teilbaum r enthält oder ein Element mehr.

$$0 \leq size\ l - size\ r \leq 1$$

☞ Anderenfalls arbeitet *nth* nicht korrekt.

Die Erweiterung $cons(a, tree)$ erhält die Invariante.

$$| Node(b, l, r) \rightarrow Node(a, cons(b, r), l)$$

Enthalten l und r gleich viele Elemente, so ist der linke Teilbaum nach dem Einfügen um eins größer. Umgekehrt: war l vor dem Einfügen um eins größer, dann sind nach dem Einfügen beide Teilbäume gleich groß.

14. Braun Bäume

Die resultierenden Bäume heißen *Braun Bäume*.

Damit haben wir übrigens Knobelaufgabe #13 gelöst.

Die Struktur eines Braun Baums ist durch die Anzahl der Elemente eindeutig festgelegt. Ist zum Beispiel die Gesamtzahl gleich $2^k - 1$, dann ist der Braun Baum vollständig ausgeglichen.