Funktionale Programmierung

Zirkuläre Programme

Andreas Abel und Steffen Jost

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

8. Mai 2012

Vorteile von Lazy Evaluation

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

Definition

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.
- 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:

```
nums2 = 0 : (map (+1) num2)
```

Beachte:
Programm benutzt die erzeugte Datenstruktur selbst als Eingabe!

Rekursive Variante zur Erinnerung:

```
nums1  =  iterate (1+) 0

iterate f x  =   x : iterate f (f x)
```

Beispiel: Eliminierung von Duplikaten

Die Funktion *nub1* eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

```
nub1 :: [Integer] -> [Integer]
nub1 []          =  []
nub1 (x : xs)    =  x : (nub1 (filter (x /=) xs))
```

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

Beispiel: Eliminierung von Duplikaten

Die Funktion *nub1* eliminiert Duplikate aus einer Liste, ohne die Reihenfolge zu verändern:

```
nub1 :: [Integer] -> [Integer]
nub1 []          =  []
nub1 (x : xs)    =  x : (nub1 (filter (x /=) xs))
```

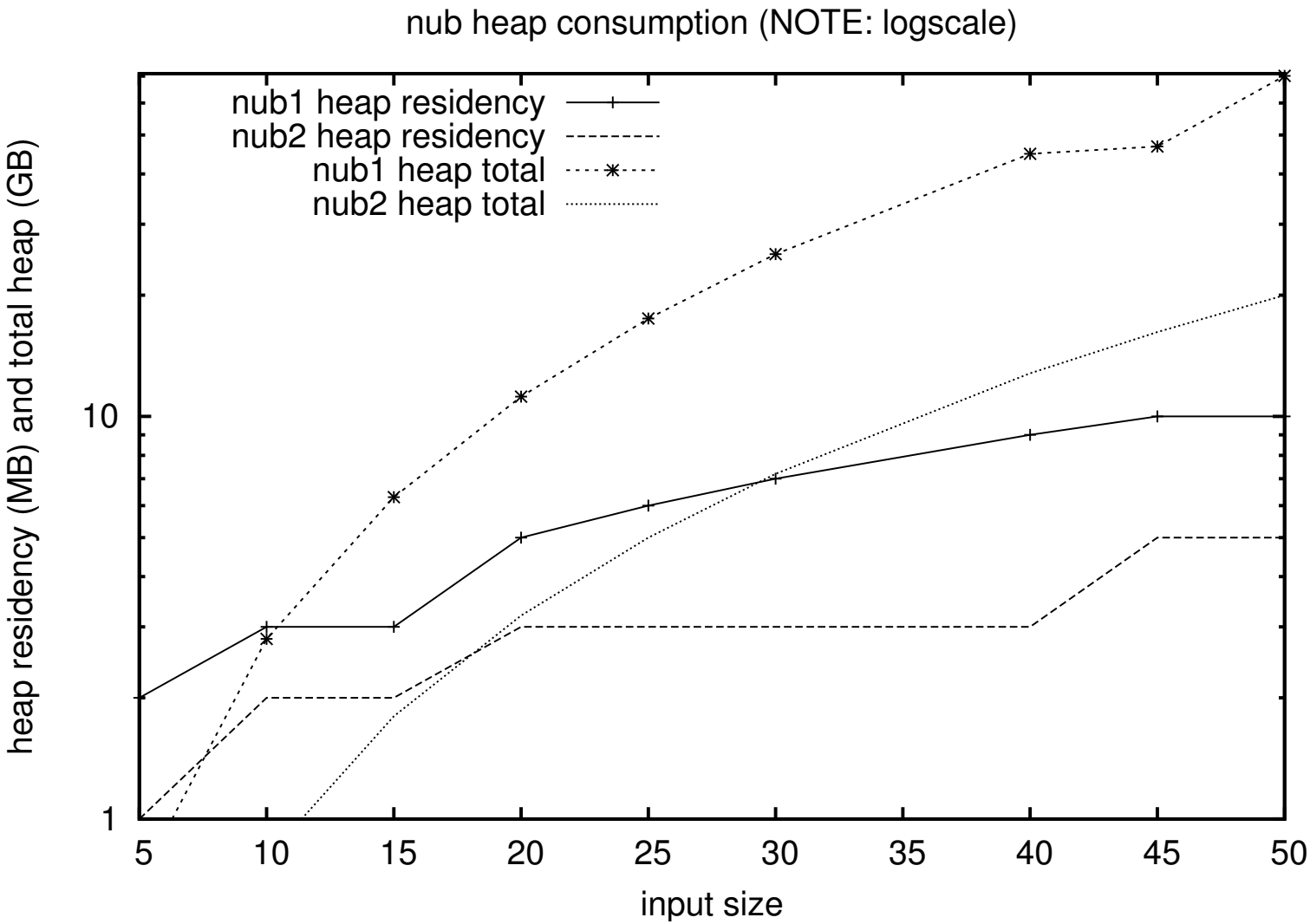
Diese Funktion erzeugt für jedes Listenelement der Eingabeliste eine Liste als Zwischen-Datenstruktur.

Beispiel: Eliminierung von Duplikaten

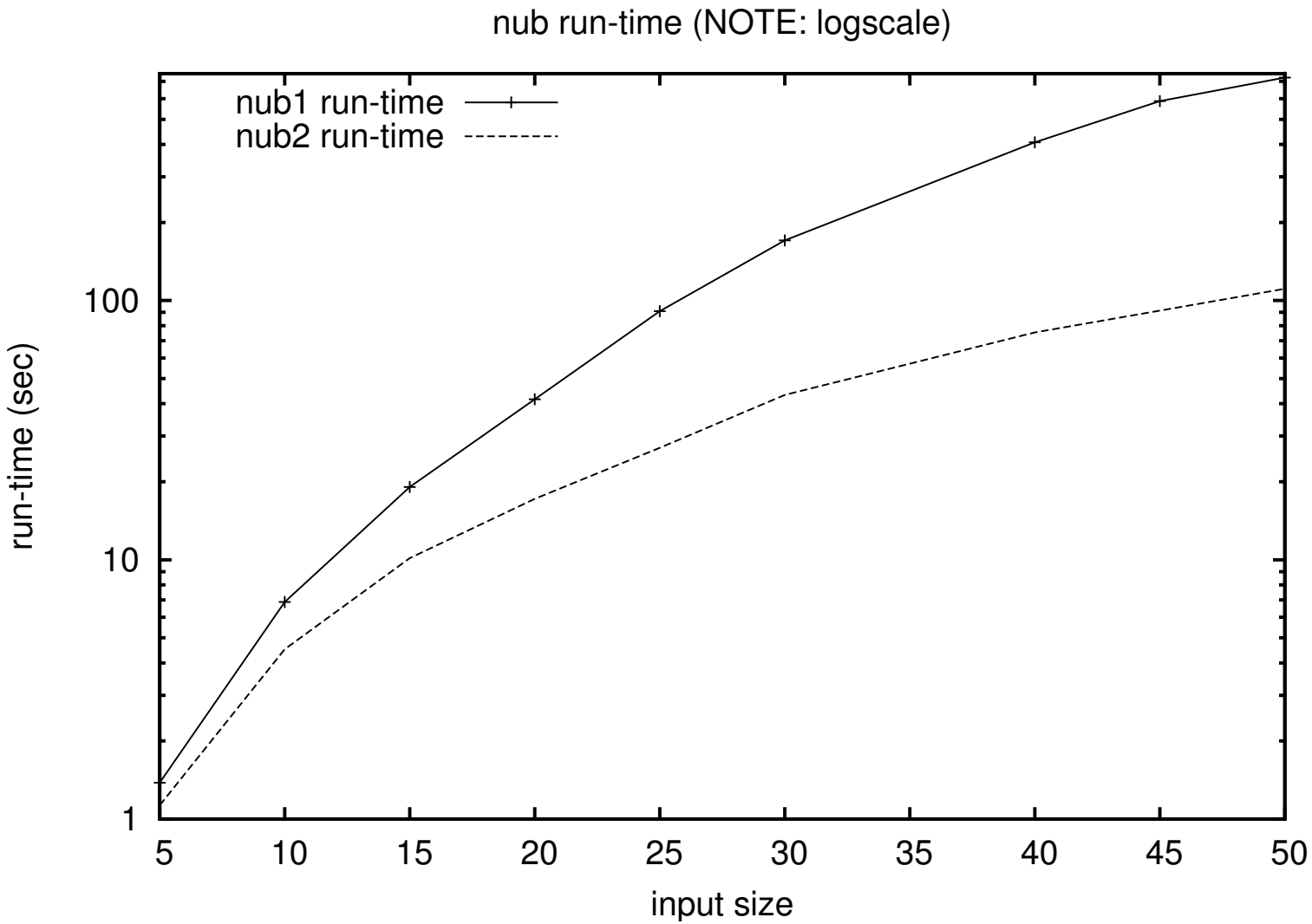
Eine bessere Implementierung vermeidet diese Listen:

```
nub2 :: [Integer] -> [Integer]
nub2 xs = res
  where
    res  =  build xs 0

    build [] n          =  []
    build (x : xs) n
      | mem x res n     =  build xs n
      | otherwise       =  x : (build xs (n + 1))

    mem _ _ 0           =  False
    mem x (y : ys) n
      | x == y          =  True
      | otherwise       =  mem x ys (n - 1)
```

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:

```
treemin :: (Ord a) => BinTree a -> a
treemin (Leaf x)    =  x
treemin (Node l r)  =  min (treemin l) (treemin r )

replace :: BinTree a -> a -> BinTree a
replace (Leaf _) m   = Leaf m
replace (Node l r) m = Node (replace l m) (replace r m)

transform2p :: (Ord a) => BinTree a -> BinTree a
transform2p t  =  replace t (treemin t)
```

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

Verbesserte Implementierung

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

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =  (l', ml) `comb` (r', mr)
  where
    (l', ml)  = repmin l m
    (r', mr)  = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```

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

```
transform1p t = fst p
  where p = repmin t (snd p)
```

Verbesserte Implementierung

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

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =  (l', ml) `comb` (r', mr)
  where
    (l', ml)  = repmin l m
    (r', mr)  = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```

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

```
transform1p t = fst p
  where p = repmin t (snd p)
```

Verbesserte Implementierung

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

```
repmin (Leaf x)   m  =  (Leaf m, x)
repmin (Node l r) m  =  (l', ml) `comb` (r', mr)
  where
    (l', ml)  = repmin l m
    (r', mr)  = repmin r m
    comb (l', ml) (r', mr) = (Node l' r', min ml mr)
```

Mittels Gleichheitsschließen lässt sich zeigen dass:

```
repmin t (treemin t) = (replace t (treemin t), treemin t)
```

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

```
transform1p t = fst p
  where p = repmin t (snd p)
```

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.

```
fibs :: [Integer]
fibs  =  1 : 1 : (zipWith (+) fibs (tail fibs))

fib :: Int -> Integer
fib n  =  fibs !! n
```

Wiederholte Aufrufe liefern deutlich schneller ein Ergebnis!

Termination von zirkulären Programmen

Der Beweis der Termination von zirkulären Programmen ist oft nicht trivial.

Im vorangegangen Beispiel kann man zeigen, dass immer nur auf vorherige Elemente der Datenstruktur zugegriffen wird, i.e. das n-te Element hängt von allen i-ten Elementen, mit $i < n$ ab.

Verallgemeinert nutzt man das Konzept der Produktivität um die Termination von zirkulären Programmen zu zeigen.

Produktivität

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.
Produktivität von $x \in \sigma$ induktiv über Struktur des Typs σ:

Definition (produktiv)

$x \in \sigma$ ist **produktiv**, wenn

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

Produktivität

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.

Produktivität

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 $\implies$ $(f\ x)$ ist $(v\ k)$ produktiv

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

Produktivität

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.

Produktivität

Theorem (Listen-Produktivität)

Sei eine Funktion $f \in [\sigma] \to [\sigma]$ welche $v \in \mathbb{N} \to \mathbb{N}$-produktiv ist, und $\forall k \in \mathbb{N} \,.\, v\ k > k$. 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.

Beispiel: Beweis der 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))
```

  $1 + 1 + \texttt{min k (vtl k)}$ produktiv.
- Weiters ist $\texttt{vtl k} = \texttt{k} - 1$ für alle $k > 0$.
- Daher ist die Produktivität des Rumpfs von `fibs`: $1 + 1 + \texttt{min k (vtl k)} = 2 + \texttt{k} - 1 = \texttt{k} + 1 > \texttt{k}$ □.

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.