

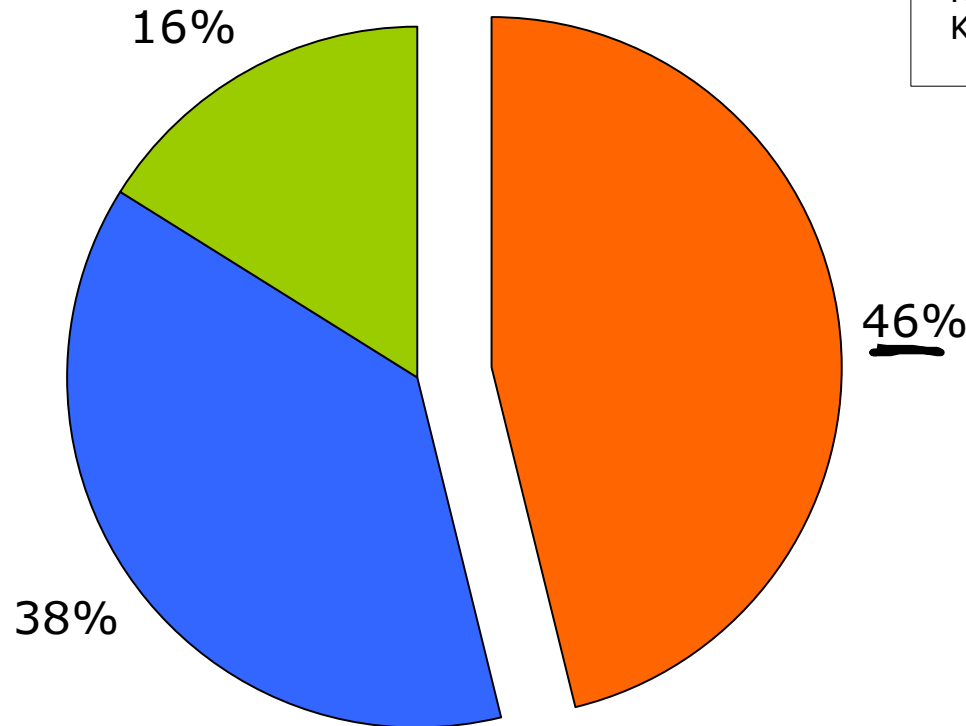
Übungen zu Informatik I Wintersemester 03/04

Übungsleiter: Dipl.-Inform. Tom Gelhausen



Fehler in Softwareprojekten

- Spezifikation ("errors in understanding the problem and in the choice of an algorithm to solve it")
- Implementierung ("initialization of fields, references, counting&calculation, range limits etc.")
- Rechtschreibfehler in Nachrichten oder Kommentaren, fehlende Kommentare etc.

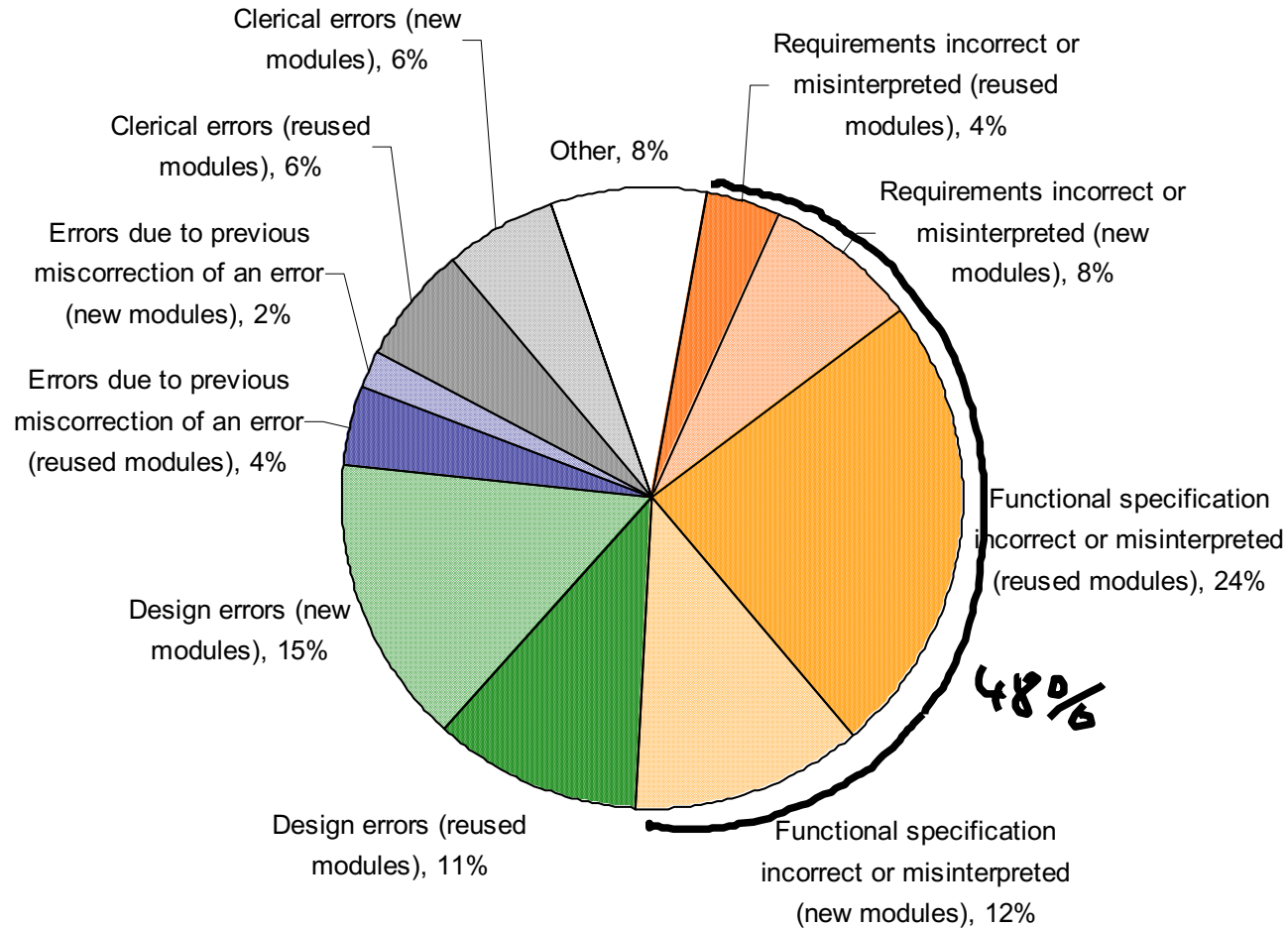


An Analysis Of Errors And Their Causes In System Programs

Studie von Albert Endres (damals IBM-Labor, Böblingen)

Analyse von Fehlern in Systemprogrammen, die Spezifikationen stammen also von Fachleuten





Software Errors And Complexity: An Empirical Investigation

Studie von Basili und Perricone



- Idee hinter Modulen
 - Module erlauben das Verbergen von Funktionen und anderen Entitäten vor dem Benutzer einer „Funktionsansammlung“
 - Die versteckten Entitäten sind nur für die exportierten Entitäten nützlich, sonst für niemanden
 - Durch das Verstecken wird als erst deutlich, welche Entitäten Bestandteil der Schnittstelle (Interface) sind, die der Benutzer der „Funktionsansammlung“ verwenden muss bzw. kann bzw. braucht

- Vergleich von Modulen und ADT'en:
 - Abstrakte Datentypen ermöglichen ebenfalls das Geheimnisprinzip (information hiding)
 - Verstecken des konkreten Datentyps (und der notwendigen Operationen) hinter einer abstrakten Schnittstelle
 - Realisierung der Abstrakten Algebren in der Praxis



▪ 3.3 Algebraische Strukturen und Algebren

• Gegeben:

- Σ : Signatur (also Menge von Operationen $\Sigma = \Sigma^{(0)} \cup \Sigma^{(1)} \cup \dots$)
- X : Elementaroperanden ($\Sigma^{(0)} \subseteq X$)
- \mathcal{T} : Menge der korrekten Terme zu Σ und X
- Q : Menge von Gesetzen (Axiome), die bedeutungstreue Umformungen $f \rightarrow f'$ definieren ($f, f' \in \Sigma$)

Beispiele (für Axiome): die Halbgruppen- und Monoidgesetze HG1, HG2 (nur bei kommutativen HG), die Gesetze V1-V10 der booleschen Algebra

- $3+3=9$ • Das Tripel $\mathcal{A} = (\mathcal{T}, \Sigma, Q)$ bildet eine **algebraische Struktur**, bzw. eine **abstrakte Algebra**. Dabei gilt für zwei Terme $t, t' \in \mathcal{T}$:
 $t \equiv t' \Leftrightarrow t$ kann nach den Gesetzen Q in t' umgeformt werden.
- **Beispiele:** Halbgruppen, Monoide, Verbände, Gruppen, Ringe, Körper, Vektorräume, boolesche Algebren



Typen auf dieser Folie weggelassen!

▪ 3.3 Konkrete Algebren

• Gegeben: abstrakte Algebra $\mathcal{A} = (\mathcal{F}, \Sigma, Q)$ mit $X \supseteq \Sigma^{(0)}$ Menge der Elementaroperanden

• **konkrete Algebra** (**Σ -Algebra**): Paar $A = (T, \Phi)$ wobei:

▫ **Trägermenge** T : Menge mit $X \subseteq T$

▫ $\Phi = \{f^{(n)}: T^n \rightarrow T \mid f^{(n)} \in \Sigma\}$, d.h. jedem n -stelligen Operationssymbol f wird eine Funktion $f^{(n)}: \underbrace{T \times \dots \times T}_{n \text{ mal}} \rightarrow T$ zugeordnet

▫ die Gesetze Q sind durch die Funktionen $f^{(n)}$ erfüllt

▫ A ist eine **Implementierung von \mathcal{A}**

• **Beispiel:** $(\mathbb{N}_0, +)$ implementiert ein kommutatives Monoid:

▫ die Signatur entspricht einem Monoid

▫ die Gesetze eines kommutativen Monoids sind erfüllt



- Betrachte Signatur 0: $\rightarrow \mathbb{N}$
succ: $\mathbb{N} \rightarrow \mathbb{N}$
 - Terme: $0, \text{succ}(0), \text{succ}(\text{succ}(0)), \dots$ \mathcal{T}
 - **Beobachtung**: Diese abstrakte Algebra hat keine Gesetze. Die Terme lassen sich nicht vereinfachen.
- Eine abstrakte Algebra $A = (\mathcal{T}, \Sigma, \emptyset)$ heißt **freie Termalgebra** zur Signatur Σ
- Wenn zusätzlich (zu $\Sigma^{(0)} \subseteq X$) noch $X = \Sigma^{(0)}$ gilt, spricht man von der **initialen** oder **Grundtermalgebra**, d.h. es gibt keine „anderen“ Elemente, aus denen Terme induktiv aufbauen sein könnten
- Konstruktoren in Haskell haben keinerlei Bedeutung
 - Sie spannen eine freie Termalgebra auf, es gibt keine Gesetze!
 - Es gibt keine „anderen“ Elemente, aus denen Terme aufgebaut sein könnten \Rightarrow Konstruktoren definieren initiale Grundtermalgebren ohne Gesetze



Theorie

Abstrakte Algebra

Konkrete Algebra

Grundtermalgebra

Axiomenmenge $Q \mapsto$

„in Haskell“

Abstrakter Datentyp

(konkreter) Datentyp

Konstruktoren

Funktionen (inkl. Sign.)

im Beispiel

kommutatives Monoid

$(\text{Nat}, +)$

Null, Succ(Null), Succ(Su ...

+

▪ $\Sigma = \Sigma^{(0)} \cup \Sigma^{(1)} \cup \Sigma^{(2)}$

• $\Sigma^{(0)} = \{ \text{“Null”} \}$

• $\Sigma^{(1)} = \{ \text{“Succ”} \}$

• $\Sigma^{(2)} = \{ \text{“op”} \}$

▪ $X = \{ \text{Null} \}$ ~~Succ?~~

▪ $\mathcal{T} = \{ \langle \text{Term} \rangle ::= \text{Null} \mid \text{Nachfolger } \langle \text{Term} \rangle \mid \text{'(op' } \langle \text{Term} \rangle \langle \text{Term} \rangle \text{')' } \}$

▪ $Q = \{ \text{algebraische Abgeschlossenheit} \\ \text{Assoziativgesetz} \\ \text{Kommutativgesetz} \\ \text{Einselement} \}$

```
data Nat = Null | Succ Nat deriving (Eq, Show)
plus :: Nat -> Nat -> Nat -- Abgeschlossenheit
plus Null a = a -- Einselement
plus a Null = a -- Einselement
plus (Succ a) b = plus a (Succ b) -- der Rest
plus (plus a b) c = plus a (plus b c) -- unzulässig
plus a (plus b c) = plus (plus a b) c -- unzulässig
plus a b = plus b a -- nicht erreichbar
```

plus $a b \in \mathbb{N}_0,$
 plus plus a b c = plus a plus b c,
 plus a b = plus b a
 plus Null a = a
 plus a Null = a }



Theorie

Abstrakte Algebra

Konkrete Algebra

Grundtermalgebra

Axiomenmenge $Q \mapsto$

„in Haskell“

Abstrakter Datentyp

(konkreter) Datentyp

Konstruktoren

Funktionen (inkl. Sign.)

im Beispiel

kommutatives Monoid

$(\text{Nat}, +)$

Null, Succ(Null), Succ(Su ...

+

- $\Sigma = \Sigma^{(0)} \cup \Sigma^{(1)} \cup \Sigma^{(2)}$

- $\Sigma^{(0)} = \{ \text{“Null”} \}$

- $\Sigma^{(1)} = \{ \text{“Succ”} \}$

- $\Sigma^{(2)} = \{ \text{“op”} \}$

- $X = \{ \text{Null} \}$

- $\mathcal{T} = \{ \langle \text{Term} \rangle ::= \text{Null} \mid \text{Nachfolger } \langle \text{Term} \rangle \mid \text{“op” } \langle \text{Term} \rangle \langle \text{Term} \rangle \}$

- $Q = \{ \text{algebraische Abgeschlossenheit} \\ \text{Assoziativgesetz} \\ \text{Kommutativgesetz} \\ \text{Einselement} \}$

```
type Nat = Integer
```

```
null :: Nat
```

```
succ :: Nat -> Nat
```

```
null = 0
```

```
succ n = n+1
```

```
plus :: Nat -> Nat -> Nat
```

```
plus a b = a + b
```

```
-- Abgeschlossenheit
```

```
-- alles
```

plus a b $\in \mathbb{N}_0$,

plus plus a b c = plus a plus b c,

plus a b = plus b a

plus Null a = a

plus a Null = a }



- Wir wollen einen Taschenrechner (o.ä.) programmieren und wollen in diesem Zusammenhang „Variablen“ verarbeiten
 - Wir müssen uns irgendwie merken, welchen Wert die jeweilige Variable aktuell beinhaltet. Diese Werte wollen wir in einer Datenstruktur „Speicher“ festhalten.
 - Wie sollen Variablen und Werte zueinander finden? z.B.:
 - Eine Liste von Variabel/Zahl-Paaren: $[(Var, Int)]$
 - Zwei Listen gleicher Länge/Reihenfolge: $([(Var)], [(Int)])$
 - Eine Funktion, die Variable auf Zahlen abbildet: $(Var \rightarrow Int)$
 - Unabhängig davon, welche Repräsentation nachher gewählt wird, wir fordern folgende Funktionen von unserem Speicher:
 - Initialisieren: `init :: Speicher`
 - Lesen: `lesen :: Speicher -> Var -> Int`
 - Schreiben: `schreiben :: Speicher -> Var -> Int -> Speicher`



▪ Problem:

- jede der vorgenannten Realisierungsmöglichkeiten birgt die Gefahr, dass ein besonders kluger Hacker irgendwelchen Unsinn mit unseren Datenstrukturen anstellt.
- Insbesondere die 2. Darstellung ist gefährdet wegen relativ kritischer Konsistenzbedingungen (Länge, Reihenfolge)

▪ Lösung:

- Einen Typ anbieten, der die Funktionen `init`, `lesen` und `schreiben` zur Verfügung stellt
- Die Implementierung des Typs wird versteckt
- Diese bezüglich der angebotenen Operationen limitierte Schnittstelle des Typs definiert einen Abstrakten Datentyp.



`init :: Speicher`

`lesen :: Speicher -> Var -> Int`

`schreiben :: Speicher -> Var -> Int -> Speicher`

- die **Signatur** definieren eine eindeutige Schnittstelle zwischen Benutzer und Implementierer unseres ADT
- Auf diese Schnittstelle müssen sich beide *einigen*, danach können beide unabhängig voneinander arbeiten, insbesondere
 - kann der Implementierer die Implementierung ändern in
 - eine schnellere Variante
 - eine weniger speicherintensive Variante
 - eine als korrekt bewiesene Variante
 - eine Variante, von der sich leichter der Aufwand bestimmen lässt

} später



▪ letzte Übung:

- `module Bee (beeKeeper, Ants(..), anteater) where ...`

„mit“: alle Konstruktoren
des Typs werden
mitexportiert
„ohne“: der Typ wird als
abstrakter Datentyp
exportiert

▪ Keine Konstruktoren exportiert

- ⇒ Als Benutzer können wir die Terme der Grundtermalgebra nicht bilden... ☹
- ⇒ Sollen wir ja auch gar nicht! Wir sollen ja die angebotene Funktionen (`init`) verwenden!



- Beispielimplementierung mit Liste von Variabel/Zahl-Paaren

```
module Speicher ( Speicher, init, lesen, schreiben ) where
```

```
      ↓Typ           ↓Konstruktor  
newtype Speicher = Speicher [ (Var,Int) ]
```

```
init :: Speicher  
init = Speicher []
```

```
lesen :: Speicher -> Var -> Int  
lesen (Speicher []) v = 0  
lesen (Speicher ((variable,wert):speicher)) v  
    | v==variable = wert  
    | otherwise = lesen (Speicher speicher) v
```

```
schreiben :: Speicher -> Var -> Int -> Speicher  
schreiben (Speicher speicher) v wert = Speicher ((v,wert):speicher)
```



▪ Beispielimplementierung mit Funktionen

```
module Speicher ( Speicher, init, lesen, schreiben ) where

  newtype Speicher = Speicher (Var -> Int)

  init :: Speicher
  init = Speicher (\v -> 0)

  lesen :: Speicher -> Var -> Int
  lesen (Speicher speicher) v = speicher v

  schreiben :: Speicher -> Var -> Int -> Speicher
  schreiben (Speicher speicher) v wert
    = Speicher (\w -> if var==w then wert else speicher w)
```



- **Typ Var definieren** (muß dem Modul Speicher schon bekannt sein!)

```
type Var = [Char]
```

- **Importieren des abstrakten Datentyps**

```
import Speicher
```

- **Leeren Speicher anlegen**

```
leererSpeicher = init
```

- **Komplexen Speicher anlegen**

```
s = schreiben (schreiben (schreiben init "u" 4) "v" 5) "w" 3
```

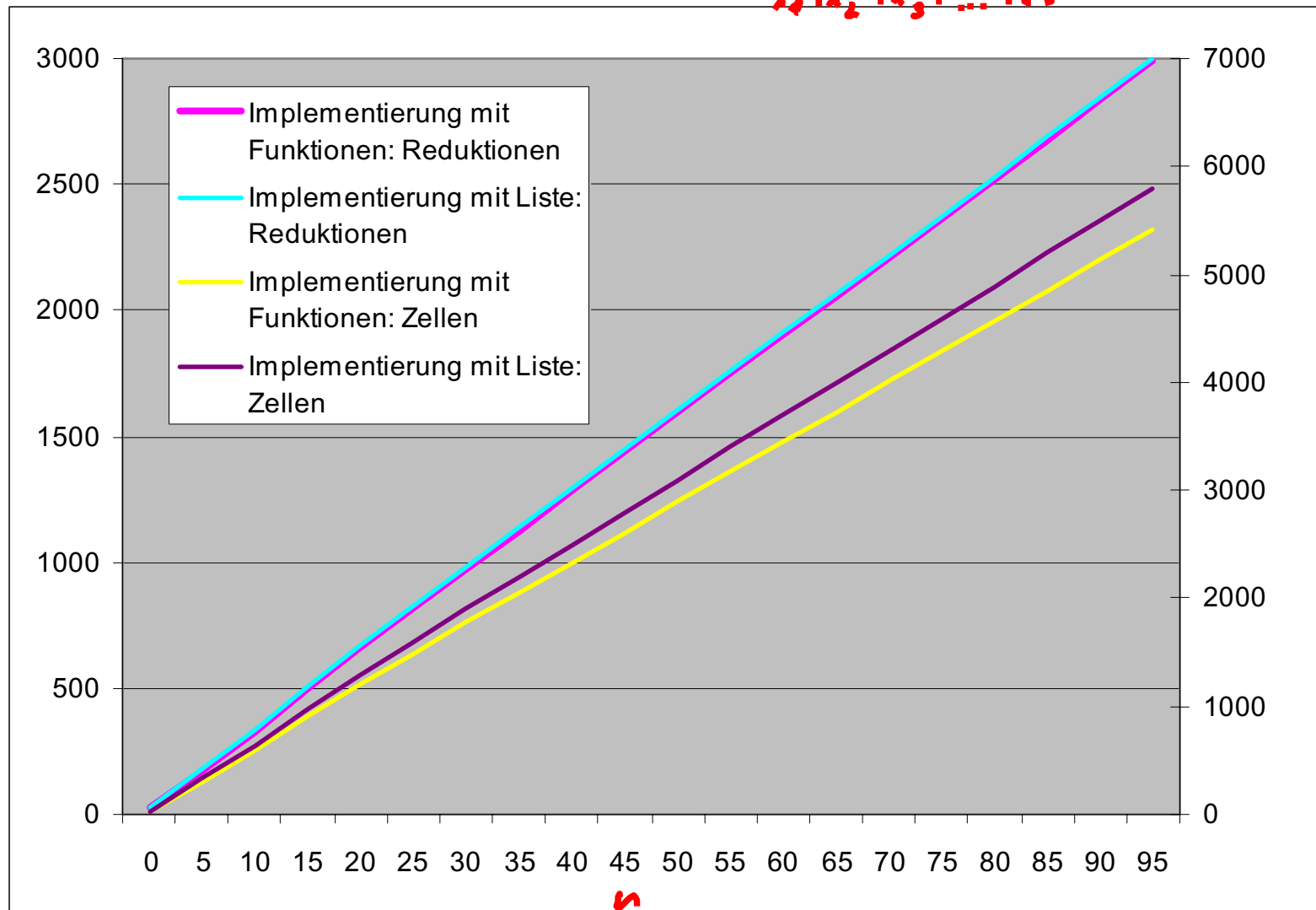
- **Speicher auslesen**

```
test = lesen s "v"
```

- **Aber:** Welche Implementierung wurde verwendet? Welche ist besser?



x_1, x_2, \dots, x_n

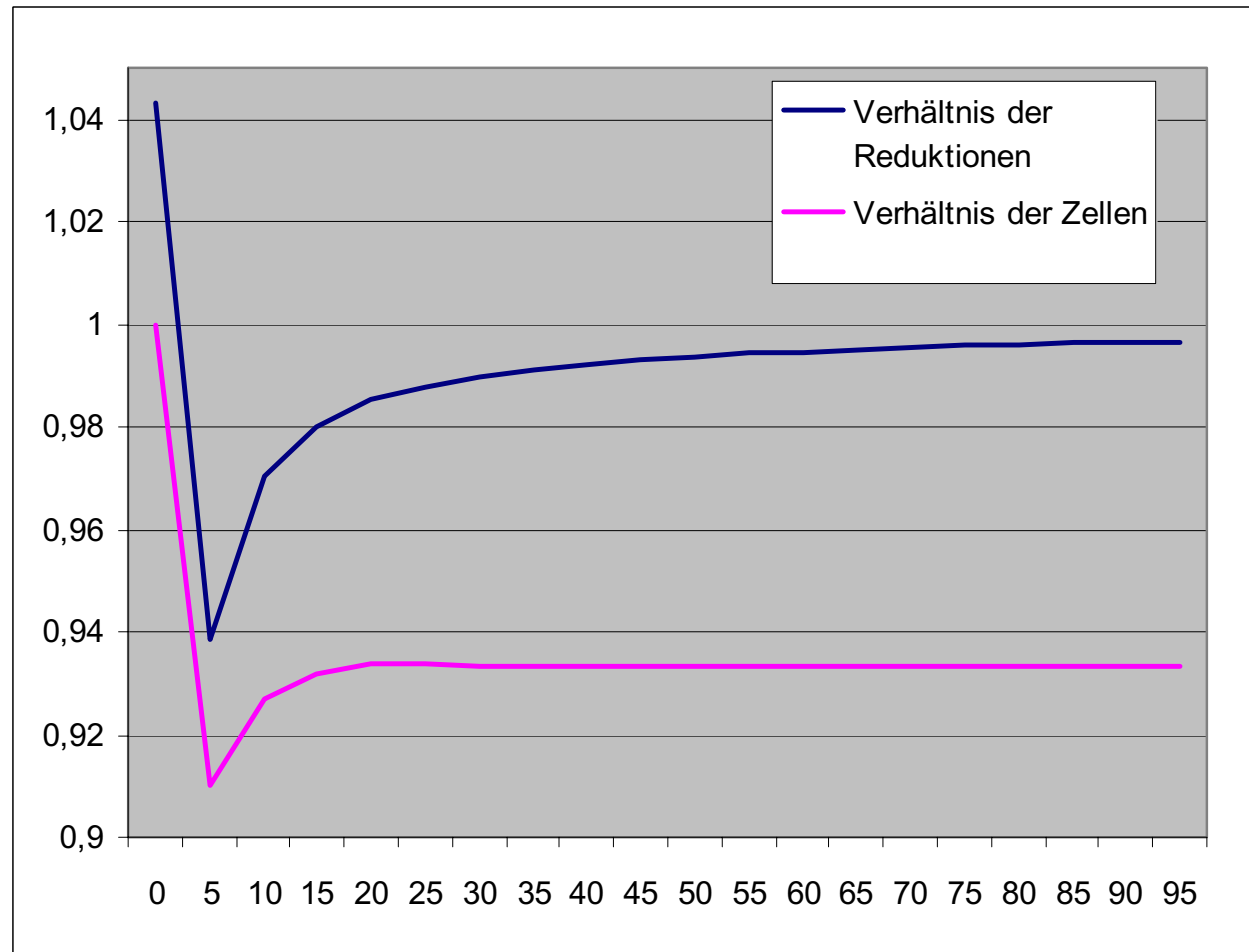


- Methode:

```
gen 0 = init.akizk  
gen n = schreiben (gen (n-1)) ("x" ++ show n) n  
test n = value (gen n) "x1"
```

- Angabe des Verhältnisses

Funktionsimplem.
Listenimplement.



- $O(1)$ für

- Elementaroperationen auf Int, Float, Double, Char, Boolean
 - Integer problematisch, da unbegrenzte Länge
- Listenoperator :
- Tupelbildung, Tupelkomponentenzugriff (a, b, c)
- Konstruktoren
- Mustererkennung
- Arrayzugriff (einen Wert lesen / schreiben)
- Wächter $/ \dots =$
- let, where
- Funktionsaufruf

- Stets Aufwände der inneren Ausdrücke berücksichtigen!



Beispiele für Aufwände

▪ `head (x:xs) = x` $O(1)$

▪ `tail (x:xs) = xs` $O(1)$

▪ `length [] = 0` $O(1)$
`length (x:xs) = 1 + length xs`
 $O(1)$ $O(1)$ $O(1)$

$n = \text{länge der Liste}$
 $f(n) = O(1) + f(n-1)$
 $= \sum_{i=0}^n c = n \cdot c \in O(n)$

▪ `take 0 _ = []` $O(1)$

`take _ [] = []` $O(1)$

`take n (x:xs) = x : (take (n-1) xs)`
 $O(1)$ $O(1)$ $O(1)$

$f(n) = O(1) + f(n-1)$ $O(n)$

▪ `drop 0 xs = xs`

`drop _ [] = []`

`drop n (x:xs) = drop (n-1) xs`



▪ `map f [] = []`

`map f (x:xs) = (f x) : (map f xs)`

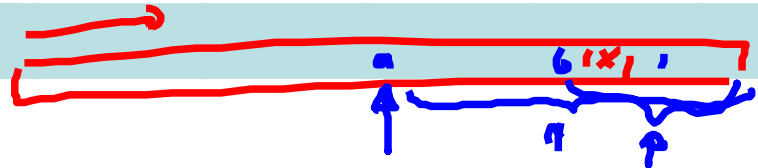
Handwritten annotations:
- A red circle around `f` in the first equation is labeled $O(1)$.
- A red arrow points from ϕ to `f x` in the second equation.
- Red underlines under `f x` and `map f xs` in the second equation are labeled $O(1)$ and $O(1)$ respectively.
- A red $n \cdot \phi$ is written to the right of the second equation.

▪ `filter f [] = []`

`filter f (x:xs) | f x = x : (filter f xs)`
`| otherwise = filter f xs`



Beispiele für Aufwände



▪ `bsearch [] _ = False`

`bsearch xs x | x < y = bsearch ls x`

`| x == y = True`

`| x > y = bsearch rs x`

where `ls = take m xs`

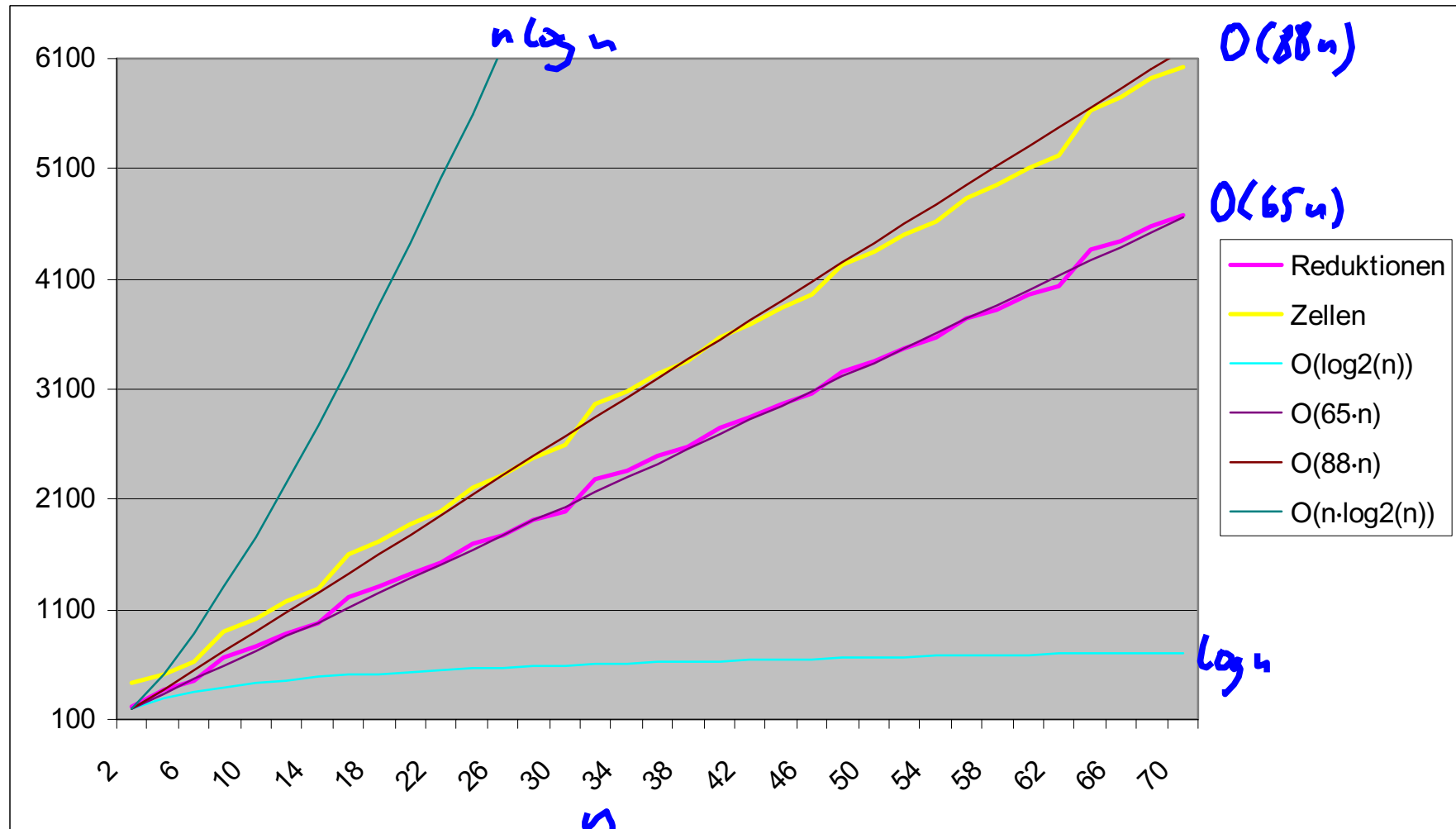
`(y:rs) = drop m xs`

`m = l `div` 2`

`l = length xs`

$O(\log n)$





Beispiele für Aufwände

▪ `bsearch [] _ = False`

$O(1) \checkmark$

`bsearch xs x | x < y = bsearch ls x`

$O(1)$

`| x == y = True` $O(1) \checkmark$

$O(1)$

`| x > y = bsearch rs x`

where `ls = take m xs`

`(y:rs) = drop m xs`

`m = l `div` 2` ~~$O(1)$~~

`l = length xs`
 $O(n)$



▪ Listenfunktionen

- $O(1)$
 - head, tail
- $O(n)$
 - length, take, drop, ...

▪ Funktionen höherer Ordnung

- ~~$O(n)$~~ + $n \cdot \varphi_f$ (φ_f Aufwand der inneren Funktion f)

$O(1)$ $O(n)$ $\in O(n)$
 $O(n)$ $O(n^2)$ $\in O(n^2)$

▪ Binärsuche

- $O(n)$ auf Listen
- $O(\log n)$ auf echten(!) Arrays



- Erster Ansatz:

fib 0 = 0 $O(1)$

fib 1 = 1 $O(1)$

fib (n + 2) = (fib (n+1)) + (fib n)

$f(f(f(f(\dots(1)))))$

$O(n) + O(n)$

„Main> fib 3

2

Main> fib 12

144

Main> fib 20

6765

Main> fib 100

Nach langer Zeit noch keine Lösung...“



- Verbesserter Ansatz:

$$\text{fib } n = \text{fst } (\text{fibH } n) \quad O(n)$$

$$\text{fibH } 0 = (0, 1) \quad O(1)$$

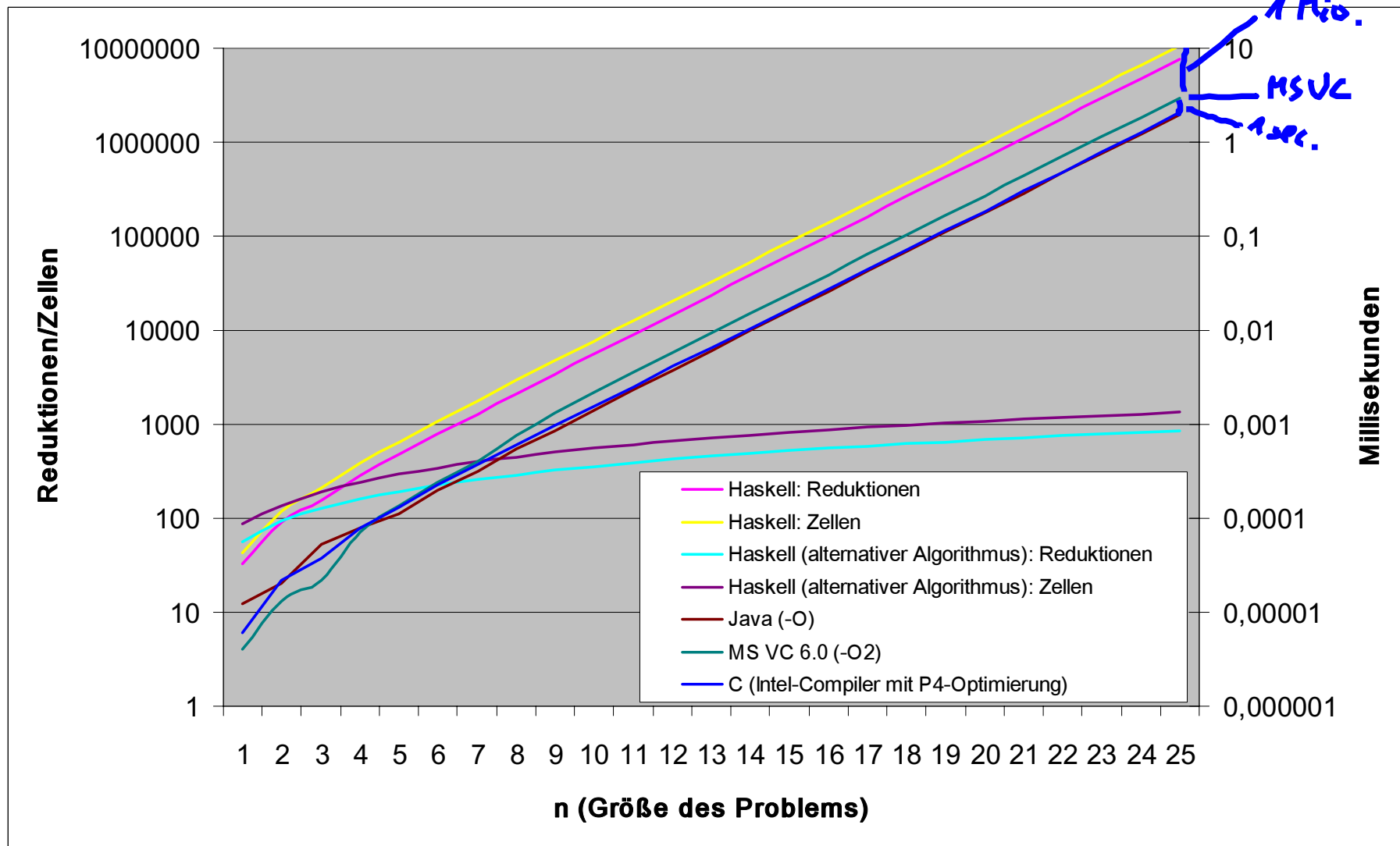
$$\text{fibH } (n + 1) = (b, a + b)$$

$O(n)$ $O(n)$

where $(a, b) = \text{fibH } n$

$$f(n) = O(1) + f(n-1) \in O(n)$$





- Erster Ansatz:

$$\text{fib } 0 = 0$$

$$\text{fib } 1 = 1$$

$$\text{fib } (n + 2) = (\text{fib } (n+1)) + (\text{fib } n)$$

$O(n)$ offenbar falsch!

$$t(1) = 1.$$

$$t(n) = t(t(n-1)) + t(t(n-2))$$

$$t(n-1) = t(t(n-2)) + t(t(n-3))$$

$$t(n-2) = t(t(n-3)) + t(t(n-4))$$

$$\rightarrow t(n) = t(t(n-2)) + t(t(n-3)) + t(t(n-3)) + t(t(n-4))$$



BCD?

$$fib(n) = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}}$$

nach H. Heuser



- Sowie die Musterlösungen finden Sie unter

<http://www.infoeins.de/uebungsblaetter.php>

