Inf 20.10.04

Inhalt:

- "Information", abstrakte Information, Syntax, Semantik am Beispiel Boolesche Terme und Boolescher Funktionen

- Funktionale Programmierung mit Haskell
  - Rekursion und Induktion
  - fundamentale und höhere Datentypen
  - Sortieralgorithmen
  - Laufzeitanalyse
  - Bäume in der Codierungstheorie
  - Klassenkonzept bei funktionalen Programmiersprachen
- Schaltnetze
  - Gatter
  - Addieren und Multiplikation
  - Multiplexer
- Schaltwerke
  - endliche Automaten
  - Flip-Flops

- von-Neumannsches Rechnermodell
  - Befehlsabarbeitungszyklus

Inf 22.10.04

Syntax und Semantik der Aussagelogik

Information und Informationssysteme

Informatik: Wissenschaft, Technik und Anwendung der maschinellen Verarbeitung, Speicherung, Übertragung und Darstellung von Informationen

Information: äußere Form (Darstellung, Syntax)
Bedeutung (abstrakte Information, Semantik)
Bezug zur realen Welt
Gültigkeit

Def.: Ein Informationssystem ist ein Tripel (R, A, I) bestehend aus einer Menge R von Repräsentationen, einer Menge A von abstrakten Informationen (semantisches Modell) und einer Funktion I, die jeder Darstellung eine abstrakte Information zuordnet (Interpretation)

Grundbegriffe:

- Ein Alphabet Σ ist eine endliche, nicht leere Menge von Symbolen
  z.B. $\Sigma_1 = \{0, 1\}$, $\Sigma_2 = \{0, 1, \dots, 9\}$, $\Sigma_3 = \{a, b, \dots, z\}$
- Ein Wort (String) über Σ ist eine endliche (möglicherweise leere) Folge von Symbolen aus Σ
  z.B. 0100110, 013942, bhgaz
  $|w| = n$ Länge des Worts
- Mit dem Symbol ε bezeichnet man das leere Wort (d.h. $\varepsilon \notin \Sigma$!)
- Σ^* bezeichnet die Menge aller Wörter über Σ : $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$
- Σ^+ " ohne ε
- Σ^k " mit der Länge k

- Konkatenation ist das "Hintereinandersetzen" von Wörtern (ohne Leerzeichen). Es wird das Symbol $\circ$ verwendet

  $\circ : \Sigma^* \times \Sigma^* \rightarrow \Sigma^*$

Rekursive Def. von Σ^*:

$\Sigma^* = \{\varepsilon\} \cup \Sigma \circ \Sigma^*$ repräsentiert eine Vorschrift zur Bildung von Σ^*

1) $\varepsilon \in \Sigma^*$
2) Wenn $x \in \Sigma$ und $w \in \Sigma^*$, dann ist ~~die~~ $x \circ w \in \Sigma^*$

Rekursive Definition der Länge

$|\varepsilon| = 0$

$|x \circ w| = |w| + 1$ für alle $x \in \Sigma$ und $w \in \Sigma^*$

- Eine formale Sprache ist eine Teilmenge $L \subseteq \Sigma^*$

  Bsp: $\Sigma = \{0,1\}$

  $\Sigma^* \supseteq Prim = \{w \in \Sigma^* \mid w$ ist Binärdarstellung einer Primzahl $\}$

- Das Wortproblem für eine formale Sprache:
  Entscheide, ob ein $w \in \Sigma^*$ auch zu L gehört oder nicht

Beispiele für Informationssysteme

1) R = Dezimalzahlen , A = Natürliche Zahlen

$\Sigma = \{0, \dots, 9\}$

$R = \{0\} \cup \{1, 2, \dots 9\} \circ \Sigma^*$

$I(15)$ = "Die Zahl fünfzehn"

Identifikation von Darstellung und abstrakter ~~A~~ Information

2) R = Dezimalzahlen , $A = \{1\}^*$ (Anzahlabstraktion)

$I(5) = 11111$

3) R = Binärdarstellung von nat. Zahlen

A = Dezimalzahlen

$\Sigma = \{0, 1\}$

$R = \{0\} \cup \{1\} \circ \Sigma^*$

$I(a_n a_{n-1} \dots a_1 a_0) = \sum_{i=0}^{n} a_i \cdot 2^i$

Wie bestimmt man die Binärdarstellung einer Zahl z?

Bsp.: $z = 25$

$z = 25$ ungerade $\rightarrow a_0 = 1$

$z = \lfloor \frac{25}{2} \rfloor = 12$ gerade $\rightarrow a_1 = 0$

$z = \lfloor \frac{12}{2} \rfloor = 6$ gerade $\rightarrow a_2 = 0$

$z = 3$ ungerade $\rightarrow a_3 = 1$

$z = 1$ ungerade $\rightarrow a_4 = 1$

$z = 0$ stop

Pseudocode zur Bestimmung der Binärdarstellung einer nat. Zahl $z > 0$

```
binrep = ε      leeres Wort
while z > 0
    if z mod 2 = 1 : binrep = 1∘binrep
    else :           binrep = 0∘binrep
    z = ⌊z/2⌋
return binrep
```

Inf 26.10.04

Binärbruch

$$1 \cdot 2^2 + 0 \cdot 2^1 + 1 \cdot 2^0 + 0 \cdot 2^{-1} + 1 \cdot 2^{-2} \longrightarrow 5,25$$

101,01

Pseudocode für Binärdarstellung einer Zahl $0 < r < 1$

```
binrep = 0.
~~r = 2 * r~~
~~#~~
while r ≠ 0
    r = 2 * r
    if r ≥ 1      binrep = binrep ∘ 1
                  r = r - 1
    else          binrep = binrep ∘ 0
```

Def.: Zwei Representationen $r_1, r_2 \in R$ heißen semantisch äquivalent, wenn $I(r_1) = I(r_2)$

Bsp.: Brüche

$R = \mathbb{Z} \times \mathbb{N}^+$ $\quad$ $A = \mathbb{Q}$ rationale Zahlen

$I(p,q) = \frac{p}{q}$

$I(22,33) = \frac{22}{33} = I(6,9) = I(2,3)$

Standardrepräsentant: gekürzter Bruch (Zähler und Nenner durch ggT teilen)

Boolesche Terme und Boolesche Funktionen

George Boole, engl. Mathematiker 1815 – 1864

Darstellungen: Boolesche Terme (B. Formeln)
Semantisches Modell: Boolesche Funktion (Funktionalität über Wahrheitswerte)
Interpretation: Auswertung von Termen

Aristoteles: Eine Aussage ist ein sprachliches Gebilde, von dem es sinnvoll ist zu sagen, es sei wahr oder falsch

Beispiel:	"7 ist eine Primzahl"	(wahre) Aussage
	"$\sqrt{2}$ ist eine rationale Zahl"	(falsche) Aussage
	"p ist eine Primzahl"	keine Aussage
	"Jede gerade Zahl ≥ 4 ist Summe von zwei Primzahlen"	Aussage (Antwort nicht bek.)
	"Dieser Satz ist falsch"	keine Aussage. Paradoxon, kann weder wahr noch falsch sein

Menge E von elementaren Aussagen mit festem Wahrheitswert, insbesondere true, false $\in E$

Menge V von Variablen als Platzhalter für Aussagen

Def.: Boolesche Terme über E und V werden rekursiv definiert:

(1) Jedes $a \in E$ (insbesondere true und false) und jede Variable $x \in V$ sind BT vom Rang 0

$$rg(a) = rg(x) = 0$$

→

(2) Ist t ein BT von Rang k, so ist $(\neg t)$ ein BT von Rang $k+1$

(3) Sind t_1 und t_2 BT von Rang k_1 und k_2, so sind auch $(t_1 \wedge t_2)$ und $(t_1 \vee t_2)$ BT vom Rang $1 + \max(k_1, k_2)$

(4) Minimalitätsprinzip: Jeder BT lässt sich durch eine Folge von Anweisungen des Typs (1), (2), (3) erzeugen

Zusatzbemerkung: Bei der Konstruktion von allgemeinen Formeln der Aussagelogik sind weitere Operationen erlaubt.

Implikation $t_1 \rightarrow t_2$ "t_1 impliziert t_2"

Äquivalenz $t_1 \leftrightarrow t_2$ "t_1 genau dann wenn t_2"

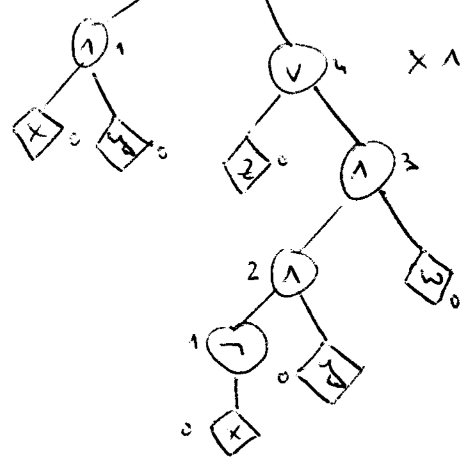
Antivalenz $t_1 \oplus t_2$ "Entweder t_1 oder t_2"

Vereinfachungen:

1) Verzicht auf äußere Klammern
2) die Reihenfolge für abnehmende Bindungsstärke der Operationen: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
3) Bei Operationen gleicher Stärke, die von links nach rechts geklammert sind, kann man die Klammern weglassen

$((x \rightarrow y) \rightarrow z)$ ist $x \rightarrow y \rightarrow z$

BT und ihren Aufbau kann man an einem Baum veranschaulichen



$((x \wedge y) \rightarrow (z \vee (((\neg x) \wedge y) \wedge w)))$

$x \wedge y \rightarrow z \vee \neg x \wedge y \wedge w$

Boolesche Terme

Inf 29.10.04

$\underline{E}, \forall, \underbrace{\neg, \wedge, \vee}_{BT}, \rightarrow, \leftrightarrow, \oplus$

Rechnen mit Wahrheitswerten (Boolesche Algebra)

$\mathbb{B} = \{0, 1\}$

Funktionen: not $\mathbb{B} \rightarrow \mathbb{B}$

not $(0) = 1$, not $(1) = 0$

b	not(b)
0	1
1	0

and, or, impl, equiv, exor : $\mathbb{B} \times \mathbb{B} \rightarrow \mathbb{B}$

b_1	b_2	and(b_1,b_2)	or(b_1,b_2)	impl(b_1,b_2)	equiv(b_1,b_2)	exor(b_1,b_2)
0	0	0	0	1	1	0
0	1	0	1	1	0	1
1	0	0	1	0	0	1
~~0~~1	1	1	1	1	1	0

Bsp.: Zeige, dass, für alle $b_1, b_2 \in \mathbb{B}$

impl(b_1, b_2) = or(not(b_1), b_2) : $b_1 \rightarrow b_2 = \neg b_1 \vee b_2$

b_1	b_2	impl(b_1,b_2)	not(b_1)	or(not(b_1),(b_2))
0	0	1	1	1
0	1	1	1	1
1	0	0	0	0
1	1	1	0	1

Idee der Interpretation von BT

- Betrachte Belegungen der Variablen aus dem BT

  $\beta : V \rightarrow \mathbb{B}$

- Interpretiere $\neg, \wedge, \vee$ durch neg, and, or

Induktiv $\beta: V \to \mathbb{B}$ gegeben

$I_\beta \; BT \to \mathbb{B}$

$I_\beta(a)$ schon definiert für alle $a \in E$ (insbs. $I_\beta(true)=1$, $I_\beta(false)=0$)

$I_\beta(v) = \beta(v)$

$I_\beta(\neg t) = not(I_\beta(t))$

$I_\beta(t_1 \wedge t_2) = and(I_\beta(t_1), I_\beta(t_2))$

$I_\beta(t_1 \vee t_2) = or(I_\beta(t_1), I_\beta(t_2))$

...

Def.: Zwei Terme t_1 und t_2 sind semantisch äquivalent (Schreibweise $t_1 \equiv t_2$), wenn $I_\beta(t_1) = I_\beta(t_2)$ für alle möglichen Belegungen β

$(x_1 \Rightarrow x_2) \equiv \neg x_1 \vee x_2$

$(x_1 \Leftrightarrow x_2) \equiv (x_1 \wedge x_2) \vee (\neg x_1 \wedge \neg x_2)$

$(x_1 \oplus x_2) \equiv (x_1 \wedge \neg x_2) \vee (\neg x_1 \wedge x_2)$

$$\left.\begin{aligned}(x_1 \vee x_2) &\equiv \neg(\neg x_1 \wedge \neg x_2) \\ (x_1 \wedge x_2) &\equiv \neg(\neg x_1 \vee \neg x_2)\end{aligned}\right\} \text{De Morgansche Regel}$$

Regeln

$\neg\neg x \equiv x$ Involutionsgesetz

$x \wedge y \equiv y \wedge x$, $x \vee y \equiv y \vee x$ Kommutativgesetz

$x \wedge (y \wedge z) \equiv (x \wedge y) \wedge z$, $(x \vee y) \vee z \equiv x \vee (y \vee z)$ Assoziativgesetz

$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$ Distributivgesetz

$x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$ "

$x \wedge x \equiv x$, $x \vee x \equiv x$ Idempotenz

$x \wedge (x \vee y) \equiv x$, $x \vee (x \wedge y) \equiv x$ Absorptionsgesetz

$\neg(x \wedge y) \equiv \neg x \vee \neg y$ De Morgansche Regel

$\neg(x \vee y) \equiv \neg x \wedge \neg y$ "

$x \wedge \neg x \equiv false$, $x \vee \neg x \equiv true$ Komplementäreigenschaften

$\to$

$x \vee false \equiv x$, $x \wedge true \equiv x$ Neutralitätsregeln

$x \wedge false \equiv false$, $x \vee true \equiv true$ Dominanzeigenschaft

Anwendung erfolgt oft über Substitutionen

Def.: $t[t_1 | x]$ bezeichnet den BT, der entsteht, wenn im Term t jedes Vorkommen der Variable x durch den Term t_1 ersetzt wird

Satz: Seien t_1 und t_2 semantisch äquivalente Terme, t ein beliebiger Term, x eine Variable. Dann gilt

$$t_1[t | x] \equiv t_2[t | x]$$

$$t[t_1 | x] \equiv t[t_2 | x]$$

Erfüllbarkeit von Booleschen Termen

Def.: Ein BT heißt erfüllbar, wenn es eine Belegung β gibt mit $I_\beta(t) = 1$

Ein BT heißt allgemeingültig (Tautologie), wenn für jede Belegung β ($I_\beta(t) = 1$) gilt

Ein BT heißt unerfüllbar (Kontradiktion), wenn für jede Belegung β $I_\beta(t) = 0$ gilt

Taut.	erfüllb. nicht allg.	ure aber gültig	Kontr.
t	s	$\neg s$	$\neg t$

Schweres alg. Problem (exp. Laufzeit mit brute force)

Inf. 3.11.04

Boolesche Funktionen

Def.: Eine n-stellige Boolesche Funktion ist eine Abb von $\mathbb{B}^n$ nach $\mathbb{B}$

$\mathbb{B}^n = \mathbb{B} \times \dots \times \mathbb{B} \ni (b_1, \dots b_n)$ Elemente sind n-Tupel von 0-1-Wert

• Boolsche Funktionen beschr. das Ein - Ausgabeverhalten bei n 0-1 Ausgängen und einen Ausgang

Jeder Booleschen Term t mit den Variablen $\{x_1, \dots x_n\}$ beschreibt eine Booleschen Funktion f_t, die wie folgt def. ist:

Jedes n-Tupel $(b_1, \dots b_n)$ wird als Belegung $\beta: \{x_1 \dots x_n\} \to \{0,1\}$ interpretiert mit $\beta(x_i) = b_i$ für $i = 0, \dots, n$

$f_t(b_1 \dots b_n) := I_\beta(t)$ f_t ist die abstr. Inf., die durch BT t repräsentiert wird.

Anzahl der n-stelligen BF

1) Anzahl der ~~Funktionen von~~ n-Tupel $(b_1, \dots b_n) \in \mathbb{B}^n$

$$|\mathbb{B}^n| = 2^n$$

2) Anzahl der Funktionen von A nach $\{0,1\}$ für eine Menge A mit m Elementen:

$$|\text{Funkt. } A \to \{0,1\}| = 2^m$$

3) Anzahl der n-st. BF: $2^{(2^n)}$

Konjunktive und disjunktive Normalformen

- Logische Operationen: $\neg, \wedge, \vee$
- Literal: Variable x_i oder deren Negation $\neg x_i$ (oft als $\overline{x_i}$)
- Minterm (Maxterm): Ein Minterm (Maxterm) ist eine Konjunktion (Disjunktion) von Literalen (z.B.: $(x_1 \wedge \neg x_2 \wedge x_4)$, $(\neg x_2 \vee x_5)$)
- vollst. Minterm (Maxterm) bei vorgegebenem n:

  Für jedes $i = 1, \dots n$ tritt entweder x_i oder $\neg x_i$ in dem Term auf
- Konjunktive Normalform: Konjunktion von Maxtermen "KNF"
- Disjunktive Normalform: ...

Satz: Für jede Boolesche Funktion $f: \mathbb{B}^n \to \mathbb{B}$ gibt es eine DNF und eine KNF, die f reproduzieren, d.h.

$$f_{dnf(f)} = f = f_{knf(f)}$$

Beobachtung: Für jede Belegung $\beta\{x_1, \dots x_n\} \to \{0,1\}$ repräsentiert durch $(b_1, \dots b_n) = (\beta(x_1), \dots, \beta(x_n))$ gibt es einen vollst. Minterm $mi = mi(b_1 \dots b_n)$ und einen vollst. Maxterm $ma = ma(b_1 \dots b_n)$ mit der Eigenschaft, dass mi für β den Wert 1 annimmt und für jede Belegung $\beta' \neq \beta$ den Wert 0 annimmt und ~~ma für β~~ $ma(\beta) = 1$,

$ma((\beta') \neq (\beta)) = 1$

Sei $f: \mathbb{B}^n \to \mathbb{B}$ gegeben, dann def. wir:

$$\mathrm{dnf}(f) = \bigvee_{(b_1,\dots,b_n) \in f^{-1}(1)} \mathrm{mi}(b_1,\dots,b_n)$$

$$\mathrm{knf}(f) = \bigwedge_{(b_1,\dots,b_n) \in f^{-1}(0)} \mathrm{ma}(b_1,\dots,b_n)$$

Behauptung: $f = f_{\mathrm{dnf}(f)} = f_{\mathrm{knf}(f)}$

Beweis für dnf(f):

Betrachte Tupel $(b_1,\dots,b_n)$; Fall 1 $f(\overbrace{b_1 \dots b_n}^{b}) = 1$

Fall 2 $f(b) = 0$

Fall 1: $b \in f^{-1}(1)$, d.h. $\mathrm{mi}(b)$ tritt in dnf(f) auf

Auswertung von dnf(f) unter der Belegung b ergibt eine 1 für mi(b) und damit insgesamt 1 ✓

Fall 2: $b \notin f^{-1}(1)$

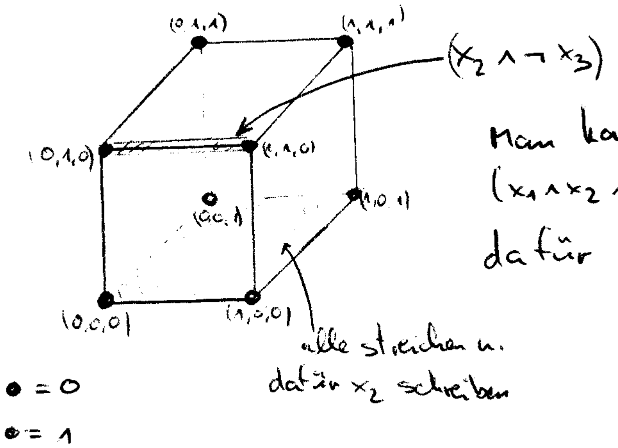
Alle Minterme $\mathrm{mi}(c_1 \dots c_n)$ in dnf(f) ergeben 0 bei Belegung b und damit insgesamt 0 ✓

Inf. 5.11.04

Nachteil: Mind. eine der kanonischen NF hat exponentielle Größe (oft beide)

Vereinfachungen möglich?
→ Nicht immer, aber man sollte es versuchen

Man kann eine BF durch einen n-dimensionalen Würfel darstellen



Man kann $(x_1 \wedge x_2 \wedge \neg x_3) \vee (\neg x_1 \wedge x_2 \wedge \neg x_3)$ streichen und dafür $(x_2 \wedge \neg x_3)$ dazuschreiben

Allgemeine Methode: Suche zwei Minterme (Maxterme), die sich nur in einem Literal unterscheiden und sonst identisch sind, streiche beide Terme und ersetze durch den gemeinsamen Teil.

Lemma: Für jeden BT t und jede Variable x gilt

1) $(t \wedge x) \vee (t \wedge \neg x) \equiv t$
2) $(t \vee x) \wedge (t \vee \neg x) \equiv t$

Beweis:

$(t \wedge x) \vee (t \wedge \neg x) \equiv (t \vee t) \wedge (t \vee \neg x) \wedge (x \vee t) \wedge (x \vee \neg x)$

$\equiv t \wedge (t \vee \neg x) \wedge (\ldots$

$\equiv \ldots$

Resolutionskalkül

Ziel: Algo um zu entscheiden, ob eine KNF erfüllbar ist.

1) Literale $\ell, \bar{\ell}$

Maxterme werden als Klauseln geschrieben, das sind Mengen von Literalen

$$x_1 \vee \neg x_2 \vee x_5 \rightarrow \{x_1, \bar{x}_2, x_5\}$$

KNF wird als Klauselmenge dargestellt.

$$\alpha = (x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_4) \wedge x_2 \wedge (x_2 \vee x_4)$$

$$K_\alpha = \{\{x_1, x_3\}, \{\bar{x}_1, \bar{x}_4\}, \{x_2\}, \{x_2, x_4\}\}$$

2) Resolventenbildung

Wenn K und K' Klauseln sind und u ein Literal, sodass $\ell \in K$ und $\bar{\ell} \in K'$, dann nennt man die Klausel $K \setminus \{\ell\} \cup K' \setminus \{\bar{\ell}\}$ einen Resolventen aus K und K'

3) Resolutionslemma: Ist K_α die Klauselmenge eines BT in KNF_α und R ein Resolvent aus zwei Klauseln $K, K' \in K_\alpha$, dann ist α genau dann erfüllbar, wenn der von $K_\alpha \cup \{R\}$ beschriebene Term erfüllt ist

Beweis:

- $K_\alpha \cup \{R\}$ erfüllbar $\Rightarrow \alpha$ erfüllbar

  Eine erfüllende Belegung für $K_\alpha \cup \{R\}$ erfüllt jede Klausel, damit auch K_α

- K_α erfüllbar $\Rightarrow K_\alpha \cup \{R\}$ erfüllbar

  Sei β eine erfüllende Belegung für K_α (dann insb. auch K, K')

  $R = K \setminus \{\ell\} \cup K' \setminus \{\bar{\ell}\}$ zeigen, dass β auch erfüllende Belegung für R ist:

  a) $\beta(\ell) = 0$, dann muss in K ein Literal ℓ' existieren und $\beta(\ell') = 1$

  $\Rightarrow \ell' \in K \setminus \{\ell\} \Rightarrow \ell' \in R \Rightarrow R$ wird erfüllt

b) $\beta(\ell) = 1 \Rightarrow \beta(\bar{\ell}) = 0 \Rightarrow \exists \ell' \in K' \ \beta(\ell') = 1 \Rightarrow \ell' \in R$

$\Rightarrow R$ erfüllt

10. 11. 04

4) Resolutionssatz: Eine KNF α ist nicht erfüllbar (Kontradiktion) genau dann, wenn man in endlich vielen Schritten aus K_α und den Resolventen die leere Klausel ableiten kann

Beweis: $\Rightarrow$ aufwendig

$\Leftarrow$ Angenommen, man kann die leere Klausel $\square$ ableiten, dann gibt es davor Klauseln K_1, K_2 mit $K_1 = \{\ell\}$ und $K_2 = \{\bar{\ell}\}$, dann ist $\ell \wedge \bar{\ell}$ nicht erfüllbar und damit ist α nicht erfüllbar

Ableitung aller Resolventen durch systematisches Suchen

$K^{(0)} = K_\alpha$ Res(K) = Menge aller Resolventen, die man aus zwei Klauseln aus K ableiten kann

$K^{(1)} = Res(K^{(0)}) \cup K^{(0)}$

$K^{(i+1)} = Res(K^{(i)}) \cup K^{(i)}$

$\vdots$

$K^* = \bigcup_{i=1}^{\infty} K^{(i)}$

Ist $K^{(i)} = K^{(i+1)}$, dann ist $K^* = K^{(i)} \rightarrow$ Abbruchbedingung

Pseudocode:

```
K = K_α
while (□ ∉ K and Res(K) \ K ≠ ∅)
      K = Res(K) ∪ K
if □ ∈ K return "unerfüllbar"
else return "erfüllbar"
```

Symbol | repräsentiert die Operation NAND

$x | y \equiv \neg(x \wedge y)$

Satz: $\{ | \}$ ist vollständige Basis

Bew.: $x \wedge y \equiv \neg(x | y)$

$\equiv (x|y) | (x|y)$

Begründung, warum eine Signatur nicht vollständig ist, kann aufwändig sein

$\{\oplus\}$ nicht vollständig:

man kann keine Tautologie darstellen

Hinweise:

1) Es darf immer nur ein Literal gestrichen werden
2) Für DNF kann geprüft werden, ob sie eine Tautologie ist

α DNF	$\neg\alpha$
$\alpha = (x_1 \wedge x_3) \vee (x_2 \wedge \neg x_3)$	$\neg\alpha = \neg(x_1 \wedge x_3) \wedge \neg(x_2 \wedge \neg x_3)$ $\equiv (\neg x_1 \vee \neg x_3) \wedge (\neg x_2 \vee x_3)$ KNF

$\neg\alpha$ unerfüllbar $\Leftrightarrow$ α Tautologie

Logische Signaturen u. vollst. Basen

Motivation:

Boolesche Funktion $\xrightarrow{\text{DNF/KNF}}$ Boolescher Term $\xrightarrow{\text{Gatter f. } \neg, \wedge, \vee}$ Schaltnetz

De Morgan: Man kann entweder auf $\wedge$-Gatter oder auf $\vee$-Gatter verzichten
Ohne $\neg$-Gatter geht es nicht!

Def.: Eine log. Signatur ist eine Menge von Symbolen, die als Boolesche Operationen bzw. als Boolesche Konstanten interpretiert werden können

Boolesche Signatur: $\{true, false, \neg, \wedge, \vee\}$
true und false für Spezialfall, dass dnf(f) mit ~~dnf~~ $f^{-1}(1) = \emptyset$
bzw. knf(f) mit $f^{-1}(0) = \emptyset$
eigtl.: $true \equiv \neg x_1 \vee x_1$ $false \equiv \neg x_1 \wedge x_1$

Def.: Eine log. Signatur, mit der man jede BF repräsentieren kann, ist vollständig (vollständige Basis)

$\{\oplus\}$ ist nicht vollständig

$x_1 \oplus x_1 \qquad (x_1 \oplus x_2) \oplus x_1$

$\oplus$ ist assoz. u. kommutativ

$\Rightarrow$ Jeder Term kann äquivalent umgeformt werden zu

$$\underbrace{x_1 \oplus \dots \oplus x_1}_{a_1} \oplus \underbrace{x_2 \oplus \dots \oplus x_2}_{a_2} \oplus \dots \oplus \underbrace{x_n \oplus \dots \oplus x_n}_{a_n}$$

falls $a_1, a_2, \dots a_n$ gerade $\Rightarrow$ Term ist Kontradiktion

falls a_i ungerade, für jede Belegung β und $\beta'(x_i) \neq \beta(x_i)$ und $\beta'(x_j) = \beta(x_j)$ für alle $j \neq i$ ergibt sich

$I_\beta(t) \neq I_{\beta'}(t)$

$\Rightarrow$ Tautologie ist nicht darstellbar

Terminduktion

Terminduktion ist eine verallgemeinerte Induktion angewendet auf den Rang von Termen

Induktionsanfang: Beweis für Terme vom Rang 0, d.h. Variablen u. Konstanten aus der Signatur

Induktionsschritt: $t = t_1 \text{ op } t_2$, op aus der Signatur

$rg(t) > rg(t_1), rg(t_2)$

Zeige, dass wenn Aussage wahr für t_1 u. t_2, dann auch wahr für t.

Beispiel:

<u>Beh.:</u> $\{\neg, \oplus\}$ ist nicht vollständig

<u>Bew.:</u> z.z., dass jeder $\{\neg, \oplus\}$-Term entweder Tautologie oder genau auf der Hälfte der Belegungen gleich 1 ist.

$P(t)$: Es gibt eine endliche Teilmenge $A \subseteq \{x_1, .. x_n\}$, sodass entweder

0) Für jede Belegung β gilt: $I_\beta(t) = 0 \iff$ hat in A ungerade Anzahl von 1

1) Für jede Belegung β gilt: $I_\beta(t) = 1 \iff$ " —

Beweis mit Terminduktion:

Anfang: $t = x_i$ $\quad A = \{x_i\} \rightsquigarrow$ Fall 1

Induktionsschritt:

a) $P(t)$ ist wahr mit Teilmenge A

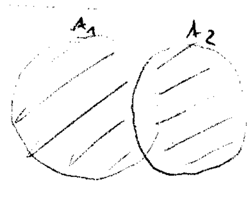
$t' = \neg t$ $\quad P(t_1)$ mit Teilmenge A

Fall 0 für $t \Rightarrow$ Fall 1 für t'

Fall 1 für $t \Rightarrow$ Fall 0 für t'

b) Angenommen $P(t_1)$, $P(t_2)$ wahr mit Teilmengen A_1, A_2

$t' = t_1 \oplus t_2$ $\quad A' = (A_1 \setminus A_2) \cup (A_2 \setminus A_1)$



Begründung

β hat in A' ungerade Anzahl von 1, dann hat β entweder eine in A_1 ungerade und eine in A_2 gerade Anzahl von 1

t_1	t_2	t
Fall 0	Fall 0	Fall 1
Fall 0	Fall 1	Fall 0
Fall 1	Fall 0	Fall 0
Fall 1	Fall 1	Fall 1

Prädikatenlogik

Prädikate und Quantoren

Def.: Ein Prädikat ist ein sprachliches Gebilde (Aussageform) mit Variablen aus bestimmten Bereichen, sodass bei Ersetzung aller Variablen durch Werte aus den entsprechenden Bereichen Aussagen ~~enthalten~~ entstehen (wahr oder falsch)

Bsp.: $P(x) =$ "x ist Primzahl" Bereich $\mathbb{N}$

$P(7) \to$ wahr $\quad P(9) \to$ falsch

$Q(x,y) =$ "$x \leq y$" Bereich $\mathbb{Z}$

$Q(3,6) \to$ wahr $\quad Q(-3,-6) \to$ falsch

Quantoren:

Allquantor $\forall x$ ~~für~~ für alle x...

Existenzquantor $\exists x$ es existiert ein x...

Quantoren binden freie Variablen

Aussageformen in denen alle Variablen gebunden sind, sind Aussagen

Regeln: $\neg \exists x\ P(x) \equiv \forall x\ \neg P(x)$

$\neg \forall x\ P(x) \equiv \exists x\ \neg P(x)$

$\forall x\ (P(x) \wedge Q(x)) \equiv \forall x\ P(x) \wedge \forall x\ Q(x)$

$\exists x\ (P(x) \vee Q(x)) \equiv \exists x\ P(x) \vee \exists x\ Q(x)$

Funktionale Programmierung am Beispiel von Haskell

Algorithmen

Ein algo. Problem besteht aus einer Menge IN von Eingaben, einer Menge OUT von Ausgaben (Lösungen, Ergebnisse) und einer spezifizierten Regel, die jeder Eingabe eine Ausgabe zuordnet, kurz eine Abb.

$f: IN \to OUT$

Algorithmus: Verfahren, das für jede Eingabe in einer endlichen Reihe von elementaren Rechenschritten die entsprechende Ausgabe bestimmt. Jeder Schritt ist ~~die~~ durch die Eingabe und die Ergebnisse der vorherigen Schritte eindeutig bestimmt (deterministisches Verfahren)

Haskell

Fundamentale Datentypen

Int, Float, Bool, Char

+, −, * für Int

- Operationen können auch als Funktionen verwendet werden:

  5 + 7

  (+) 5 7

- Signatur beschreibt Definitionsbereich und Wertebereich einer Funktion

  Wertebereich als letzter Datentyp in der Signatur

  $$\text{maxvondrei} :: \underbrace{\text{Int} \rightarrow \text{Int} \rightarrow \text{Int}}_{\text{Eing.}} \rightarrow \underbrace{\text{Int}}_{\text{Ausg.}}$$

- Nach der Signatur folgt Definition der Funktion

  Gültungsbereich einer Definition wird durch Einrücken markiert (zwingend!)

- Funktionsnamen beginnen immer mit einem kleinen Buchstaben

Datentyp Int

$-2^{31}, \dots, -1, 0, 1 \dots 2^{31}-1$

Darstellung von neg. Zahlen durch 32 bit Darstellung

Möglichkeiten der Darstellung neg. Zahlen.

1) $b_{31} = 1$ $\quad z = -\sum_{i=0}^{30} b_i 2^i$ nicht gebr.

2) $b_{31} = 1$ $\quad z = -\sum_{i=0}^{30} \bar{b}_i 2^i$ 1-Komplement (nicht bei Haskell)

3) $z = -(b_{31} \cdot 2^{31}) + \sum_{i=0}^{30} b_i 2^i$ 2-Komplement (bei Haskell)

Vorteil: Addition von positiven und negativen Zahlen mit normaler Additionsmethode

Bsp.:

−5	1	1	…	1	1	0	1	1
+4	0	0	…	0	1	0	0	0
	1	1	…	1	1	1	1	1

Ganzzahlige Division

Satz: Für jedes $n \in \mathbb{Z}$ und jedes $d \in \mathbb{N}^+$ gibt es eindeutig bestimmte Zahlen $q, r \in \mathbb{Z}$ mit

$n = q \cdot d + r$ und $0 \leq r < d$

Bsp.: $n = 29 \quad d = 6 \quad \Rightarrow 29 = 4 \cdot 6 + 5$

$n = -29 \quad d = 6 \quad \Rightarrow -29 = (-4) \cdot 6 + (-5) \quad +6 \quad -6$

$= (-5) \cdot 6 + 1$

mod zur Beschleunigung des Euklidischen Algos (ggT)

ggT(29,6) = ggT(23,6) = ... = ggT(5,6)

↑ mod 29 6

= ggT(..)

Erweiterung auf neg. durch Nutzung von abs

```
ggT :: Int -> Int -> Int
ggT a b
 | a == 0              = abs b
 | b == 0              = abs a
 | a >= b && b >= 0    = ggT (mod a b) b
 | b >= a && a >= 0    = ggT (mod b a) a
 | otherwise           = ggT (abs a) (abs b)
```

Satz: Für bel. $a, b \in \mathbb{Z}$ gibt es Zahlen $s, t \in \mathbb{Z}$ so dass $ggT(a,b) = s \cdot a + t \cdot b$

Bsp: $ggT(29,6) = 1 \quad \Rightarrow 1 = s \cdot 29 + t \cdot 6$

$29 = 4 \cdot 6 + 5$ | $5 = 29 - 4 \cdot 6$

$6 = 1 \cdot 5 + \underline{\underline{1}}$ | $1 = 6 - 1 \cdot 5$

$\underbrace{5 = 5 \cdot 1 + \underline{\underline{0}}}_{ggT(29,6)=1}$

$1 = 6 - 1 \cdot 5 = 6 - 1(29 - 4 \cdot 6)$

$= 5 \cdot 6 - 29$

Rekursion

Eine Funktion ist primitiv rekursiv, wenn zur Berechnung von f(n) nur der Wert von f(n-1) und n verwendet werden.

Allgemeine Formen der Rekursion:

f(n) ruft f(n-1), f(n-2), ... auf

Beispiel: Fibonacci-Zahlen

```
fib :: Int -> Int
fib n
  | n == 0      = 0
  | n == 1      = 1
  | otherwise   = fib (n-1) + fib (n-2)
```

Beobachtungen:

1) Schnelles Wachstum der Fib-Zahlen
2) Laufzeit wächst auch sehr stark

zu 1) $f(n) = f(n-1) + f(n-2) \geq f(n-1)$ $\Rightarrow$ monotone Folge

$f(n) = f(n-1) + f(n-2) \leq 2 \cdot f(n-1)$

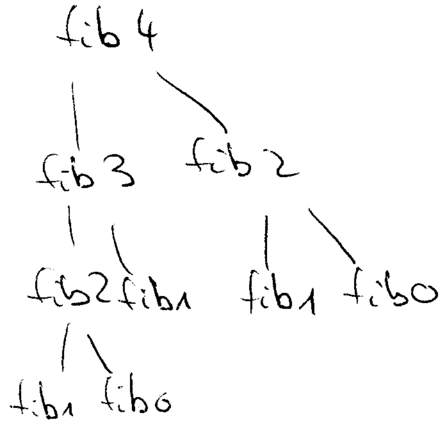
$f(n) =$ " $\geq 2 \cdot f(n-2)$

$f(2n+1) \geq 2 \cdot f(2n-1) \geq 2 \cdot 2 \cdot f(2n-3) \ldots$

$\geq 2^n \cdot 1$

$f(m) \geq 2^{\lfloor \frac{m}{2} \rfloor} \geq 2^{\frac{m-1}{2}} = \sqrt{2}^{\,m-1}$

Laufzeit für fib



$t(0)$	1 Aufruf
$t(1)$	1
$t(2)$	3
$t(3)$	5
$t(4)$	9

Satz: $f(n+1) \le t(n) \le f(n+3)-1$

Beweis mit vollst. Induktion

n	0	1	2	3	4
$f(n)$	0	1	1	2	3
$t(n)$	1	1	3	5	9

Ind.-Anfang für $n=0$ und $n=1$ aus der Tabelle

Ind-Schritt von n auf $n+1$

...

geschlossene Formel für Fib-Zahlen

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

Laufzeit von Rekursionen

$n! = n \cdot (n-1)!$

$0! = 1$

Zur Berechnung von $n!$ werden n Multiplikationen benötigt
$\rightarrow$ Lineare Zeit

ggT mit $ggT(a,b) = ggT(a-b, b)$
größere der beiden Zahlen wird in jedem Schritt um mind. 1 kleiner
$\Rightarrow$ lineare Laufzeit $(T(a,b) \in O(\max(a,b)))$

ggT mit $ggT(a,b) = ggT((\mathrm{mod}\ a\ b)\ b)$
nach zwei Schritten mind. Halbierung der größeren Zahl.
Nach $2 \cdot \log(\max(a,b))$ Schritten fertig

Float

Typ für Gleitkommazahlen

3.1415	wissenschaftliche Notation
0.25	1.25e6
-42.5	42.3e-5
5	

Vordefinierte Funktionen z.B. für ln, sin, cos, Arctan, ...

wichtig:

floor, ceiling :: Float → Int   zum auf-/abrunden

Operationen $\underbrace{+,-,*}_{\text{auch für Int}}$, /

Der Compiler entscheidet nach Typanalyse, welche Operation (Int/Float) verwendet wird.

Explizite Typumwandlung möglich durch die Funktion

fromInt :: Int → Float

Abgeleitete Datentypen

a) Tupel

b) Listen

Tupel: Typ, um zwei, drei, oder eine andere feste Anzahl von Daten geordnet zusammenzufassen

Syntax: (Typ1, Typ2, ..., Typ k)

Tupel können zur Definition von eigenen Datentypen verwendet werden

type PointInSpace = (Float, Float, Float)

↑

Konvention: Großbuchstabe

Tupel als Eingabe – kein Vorteil zur Verwendung der Einzelkomponenten

Tupel als Ausgabe – Verwendung als Zwischenspeicher

Beispiel: Fibonacci-Zahlen mit primitiver Rekursion auf Tupeln

```
fibStep :: (Int, Int) -> (Int, Int)   -- ( f(n-2), f(n-1) -> f(n-1), f(n
fibStep (a,b) = (b, a+b)

fibPair :: Int -> (Int, Int)   -- gibt n-tes Fib-tupel wieder
fibPair 0 = (0,1)
fibPair n = fibStep (fibPair(n-1))

fastFib :: Int -> Int   -- n-te Fibzahl
fastFib n = fst.fibPair n
```

Funktion zum Zugriff auf erste Komponente

Komposition von Funktionen

Zusammenfassung Tupel

- neue Datentypen
- Effizienzsteigerung
- Zusammenfassung verschiedener Datentypen
- feste Anzahl

<u>Liste</u>:

- neue Datentypen
- Effizienzsteigerung
- Zusammenfassung nur gleichartiger Datentypen
- variable Länge

Liste ist geordnete Kollektion von Daten eines einzigen Typs in beliebiger Anzahl

Syntax [Typ]

Beispiel [1,5,9,3] :: [Int]

Abgeleitete Datentypen

Tupel: Typ, um zwei, drei oder eine andere feste Anzahl von Daten geordnet zusammenzufassen.

Syntax: (Typ1, Typ2, ..., Typ k)

Beispiel: (12, 15) :: (Int, Int)

(3, -4, 5) :: (Int, Int, Int)

(3, '3') :: (Int, Char)

Tupel können zur Definition von eigenen Datentypen verwendet werden:

```
type PointInSpace = (Float, Float, Float)
     ↑
     Großbuchstabe (Konvention)
```

Tupel als Eingabe - kein Unterschied zur Verwendung der Einzelkomponenten

Tupel als Ausgabe - Verwendung als Zwischenspeicher

Beispiel: Fibonacci-Zahlen mit primitiver Rekursion auf Tupeln

```
fibStep :: (Int, Int) -> (Int, Int) --
fibStep (a, b) = (b, a+b)

fibPair :: Int -> (Int, Int) -- n -> (f(n), f(n+1))
fibPair 0 = (0,1)
fibPair n = fibStep (fibPair (n-1))
```

firstFib :: Int -> Int
firstFib n = fst . fibPair (n)

(1)

Funktion zum Zugriff auf erste Komponente in einem 2-Tupel

(1) Komposition von Funktionen

Tupel	Liste
+ neue Datentypen	——— +
+ Effizienzsteigerung	——— +
+ Zusammenfassung verschiedener Datentypen	Zusammenfassung gleichartiger Datentypen −
− feste Anzahl	variable Länge +

Liste: Geordnete Kollektion von Daten eines bestimmten Typs in beliebiger Anzahl

Syntax: [Typ]

Beispiele: [1,5,9,3] :: [Int]

[(1,2),(3,5),(4,1)] :: [(Int, Int)]

Spezialfall: [Char] als String

verkürzte Schreibweise für Strings ['a','b','a','x'] = "abax"

Für aufzählbare Typen wie Int, Char, Float gilt es zusätzlich die folgenden Listenbeschreibungen.

[2..7] = [2,3,4,5,6,7]

[b..d] = [b,c,d] = "bcd" / [2,5..12] = ~~[2,4~~ [2,5,8,11]

Informatik A 3.12.04

vordefinierte, polymorphe Listenfunktionen aus Prelude.hs

polymorph - vielgestaltig: für Listen über beliebige Datentypen verwendbar

a Variable für Datentyp

Operationen :, !!, ++

(:) :: a -> [a] -> [a]   Element an Listenanfang einfügen

'd' : "rei" ⇝ "drei"

2 : [1..3] ⇝ [2,1,2,3]

(!!) :: [a] -> Int -> a   k-tes Element aus Liste ausgeben

"drei" !! 3 ⇝ 'i'

[1..5] !! 3 ⇝ 4

(++) :: [a] -> [a] -> [a]   Listen verketten (Konkatenieren)

"Ket" ++ "te" ⇝ "Kette"

length :: [a] -> Int   Länge

length [] ⇝ 0

length "drei" ⇝ 4

reverse :: [a] -> [a]   Umkehrung

revese "drei" ⇝ "ierd"

head, last :: [a] -> a   erstes/letztes Element ausgeben

init, tail :: [a] -> [a]   letztes/erstes Element abschneiden

concat :: [[a]] -> [a]   Konkatenieren der Listen

concat ["gemein", "sam", "keit"] ⇝ "gemeinsamkeit" and, or :: [Bool] -> Bool   Konjunktion, Disjunktion von allen Werten der Liste

sum, product :: [Int] -> Int   Summe / Produkt aller Elemente
auch Float

Wichtig für die Arbeit mit Listen:

1) Jede nichtleere Liste kann mit dem Konstruktor : erzeugt werden

$$\overbrace{\underbrace{('d':\underbrace{('r':\underbrace{('e':\underbrace{('i':[\,])}_{\text{"i"}})}_{\text{"ei"}})}_{\text{"rei"}})}_{\text{"drei"}}}^{\text{"drei"}} \equiv 'd':'r':'e':'i':[\,]$$

Folgerung: jede nichtleere Liste kann in der Form
x : xs geschrieben werden
(x: erstes El., xs: Restliste)

2) Besondere Rolle von Strings für die Ein- und Ausgabe

Funktionen show und read wandeln Elemente eines Types in einen String um (show), bzw. umgekehrt (read)

show (2 + 33) ⇝ "35"
show (True || False) ⇝ "True"
(read "True") :: Bool ⇝ True
(read "35") :: Int ⇝ 35

5) Mit $elem :: a \to [a] \to Bool$
kann man testen ob ein Element in der Liste auftritt

$$elem\ 3\ [1..5] \rightsquigarrow True$$

Programmierung auf Listen

Beispiel: Berechne für Int-Liste die Summe der Quadrate
Idee: 0 für leere Liste, für nichtleere Listen berechne Quadrat des i-ten El. + Quadrats. d. Restl.

Umsetzung:
```
sqsum :: [Int] -> Int
sqsum [] = 0
sqsum (x:xs) = x*x + sqsum xs
```

Ablauf für sqsum [2,5]

[2,5] passt nicht zum Muster [], aber zum Muster x:xs 2:[5]

$\rightsquigarrow 2*2 + sqsum\ [5]$
$\rightsquigarrow 2*2 (+ 5*5 + sqsum\ [])$
$\rightsquigarrow 2*2 + (5*5 + 0) = 29$

Mechanismus des Pattern-Matching (Musterzuordnung)
Mehrere Definitionen für eine Funktion gegeben (mit eingeschränktem Bereich)
Wie bei Fallunterscheidung wird für eine Eingabe getestet, ob sie auf das "Muster" der ersten Definition passt, sonst auf das Muster der zweiten, dritten, ...

Welche Muster kann man verwenden?

- feste Werte für primitive Datentypen
  (z.B. 0, 1, True)
- eine Variable für einen Wert einer Datent.
  (z.B. n, x, var, ...)
- Eine "Wildcard" _ für ~~jedes~~ irgendein Argument
- Tupelmuster (x, y) oder (x_1, x_2, x_3, x_4)
- Listenmuster [], [x], [x, y], [x, y, ...], ...
- Muster mit Konstruktoren, hier für Listen:

  (x : xs)

Beispiele:

1) Prüfe, ob eine Int-Liste aufsteigend sortiert ist

```
checksort :: [Int] -> Bool
checksort []  = True
checksort [x] = True
checksort (x:y:xs) = x <= y && checksort (y:xs)
```

2) Listenrekursion und Induktion
   Was berechnet die folgende Funktion:

```
magic :: [Int] -> Int
magic [] = 0
magic (x:xs) = x - magic xs
```

alternierende Summe:

magic [1,2,3,4] = 1 - 2 + 3 - 4 = -2

Sortieren mit Listenrekursion

Aufgabe: Sortiere Elemente einer gegebenen Liste in aufsteigende Reihenfolge

1) Insertionsort

Idee: Beginne mit leerer Liste und füge nacheinander alle Elemente der Eingabeliste an die richtige Stelle in der sortierten Liste ein

Code:

```
ins :: Int -> [Int] -> [Int]
ins x [] = [x]
ins x (y:ys)
  | x <= y      = x : (y:ys)
  | otherwise   = y : (ins x ys)

iSort :: [Int] -> [Int]
iSort [] = []
iSort (x:xs) = ins x (iSort xs)
```

Laufzeit ist proportional zur Anzahl der Vergleiche (worst case, d.h. Anzahl der Vergleiche im schlechtesten Fall)

ins x [— n Vergleiche, wenn x größer als alle Elemente in [

x: Element; [: Liste d. Länge n

Vergleiche für iSort bei Liste der Länge n

$$(n-1) + (n-2) + \dots + 1 = \frac{n(n-1)}{2}$$

$\Rightarrow$ Anzahl d. Vergleiche $\leq \frac{n(n-1)}{2}$

$\geq$ ~~...~~ $\frac{n(n-1)}{2}$ bei absteigend sortierter Liste

$\Rightarrow\ = \frac{n(n-1)}{2}$ im schl. Fall

$$t(n) \leq c \cdot \frac{n(n-1)}{2} \leq c' n^2$$

Schreibweise: $t(n) \in O(n^2)$

n^2 ist asymptotische obere Schranke der Laufzeit

$$t(n) \geq c \frac{(n-1)n}{2} \geq c'' n^2$$

$\Rightarrow\ t(n) \in \Omega(n^2)$ untere Schranke

$c' n^2 \leq t(n) \leq c'' n^2 \Rightarrow t(n) \in \Theta(n^2)$
scharfe Schranke

Quick Sort:

Idee: Wähle erstes Element als Pivot-Element
Bilde Liste 1 der Elemente, die $\leq$ Pivot
Bilde Liste 2 der Elemente, die $\geq$ Pivot
Sortiere Liste 1 und Liste 2 rekursiv und
bilde sort. Liste + [Pivot] + sort. Liste 2

Implementierung von Quick Sort mit List-Comprehension

[y | y <- ys , y <= x]

wähle aus der Liste ys alle Elemente die $\leq x$ sind

Laufzeit ist Listenlänge + Zeit für Auswertung

Code:

```
qSort :: [Int] -> [Int]
qSort [] = []
qSort (x:xs) = qSort [y | y <- xs, y ≤ x] ++ [x] ++
               qSort [y | y <- xs, y > x]
```

Laufzeit in Anzahl d. Vergleiche:

$$t(n) \leq (n-1) + (n-1) + \underset{\text{Liste 1}}{t(k)} + \underset{\text{Länge Liste 2}}{t(n-1-k)}$$

Behauptung $t(n) \leq n^2$

Beweis mit vollständiger Induktion

IA: leere Liste - 0 Vergleiche

IV: Wahr für alle $k < n$

IS: $t(n) \leq n^2$

$$t(n) \leq (n-1) + (n-1) + t(k) + t(\underbrace{n-1-k}_{<n}) \quad \text{nach IV}$$

$$\leq (n-1) + (n-1) + k^2 + (n-1-k)^2 \leq 2(n-1) + (n-1)^2$$

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

$$= n^2 - 1 \leq n^2 \qquad \square$$

Informatik A

17.12.04

$\text{map} :: \underbrace{(a \to b)}_{\text{Typ von } f} \to \underbrace{[a]}_{\text{Typ v. xs}} \to \underbrace{[b]}_{\text{Ausgabetyp}}$

$\text{filter} :: \underbrace{(a \to \text{Bool})}_{\text{Typ v. } p} \to \underbrace{[a]}_{\text{Typ v. xs}} \to \underbrace{[a]}_{\text{Ausgabetyp}}$

$\text{filter1} :: (a \to a \to a) \to [a] \to a$

$\text{filter1 } f \ [x] = x$
$\text{filter1 } f \ (x:xs) = f \ x \ (\text{filter } f \ xs)$

Beispiel: Operation || auf Bool

$f \ x \ y = x \,||\, y$ oder (||)

filter1 (||) xs
$\qquad\qquad$ [Bool] $\rightsquigarrow$ Disjunktion aller Listenwerte

Problem: Fehlermeldung bei leerer Liste

Viele Operationen besitzen neutrales Element:

z.B.: 0 für Addition $\quad x + 0 = x$
1 für Multiplikation $\quad x \cdot 1 = x$
True für Konjunktion $\quad x \&\& \text{True} = x$
False für Disjunktion $\quad x \,||\, \text{False} = x$
0 für max über $\mathbb{N}$ $\quad \max(0, x) = x$

$\rightsquigarrow$ Gib für Faltung der leeren Liste neutrales Element aus

Erweiterung der Syntax notwendig:

$$\text{foldr} :: \underbrace{(a \to a \to a)}_{\text{Operation}} \to \underbrace{a}_{\text{neutr. El.}} \to \underbrace{[a]}_{\text{Eingabeliste}} \to \underbrace{a}_{\text{Ausgabe}}$$

etwas allgemeiner: Operation kann auch nach Typ variieren

$$\text{foldr} :: (a \to b \to b) \to b \to [a] \to b$$

Rechts- und Linksfaltung

$$a_1 \otimes (a_2 \otimes (\ldots (a_{n-1} \otimes a_n)))) \quad \text{Rechtsfaltung}$$

$$((((a_1 \otimes a_2) \otimes a_3) \otimes \ldots) \otimes a_n \quad \text{Linksfaltung}$$

$$\text{foldl} :: \underbrace{(b \to a \to b)}_{\text{Operation}} \to \underbrace{b}_{\text{n. El.}} \to \underbrace{[a]}_{\text{Eing.}} \to \underbrace{b}_{\text{Ausg.}}$$

Beispiele

1) Minimum (vordefiniert)

minimum xs = foldr1 min xs

2) Quadratischer Abstand eines Punkts $(a_1, a_2, \ldots, a_n) \in \mathbb{R}^n$ zum Ursprung

$$a_1^2 + (a_2^2 + \ldots (+ a_n^2))$$

mit Rechtsfaltung: Operation

f x y = x * x + y

sqdist xs = foldr f 0 xs

3) Insertion-Sort

Operation: $ins :: a \to \underbrace{[a]}_{b} \to \underbrace{[a]}_{b}$ leere Liste ab „neutrales El.“

iSort xs = foldr ins [] xs

iSort [3,1,2] → ins 3 $\underbrace{\text{foldr ins [] [1,2]}}$

ins 1 $\underbrace{\text{foldr ins [] [2]}}$

ins 2 $\underbrace{\text{foldr ins [] []}}_{[]}$

⇝ [2] ⇝ [1,2] ⇝ [1,2,3]

4) Polynomauswertung mit dem Horner-Schema

$p(d) = a_n d^n + a_{n-1} d^{n-1} + \ldots + a_1 d + a_0$

$2n-1$ Multiplikation + n Additionen

Horner-Schema

$p(d) = (((\ldots(a_n \cdot d + a_{n-1})d \ldots)d + a_2)d + a_1)d + a_0$

n-Additionen und n Multiplikationen

für d=10

f x y = x * $\cancel{d}^{10}$ + y

foldl f 0 xs berechnet p(10)

für beliebige Werte von d mit lokalen Definitionen

g :: Float → Float → Float → Float

$\quad$ d $\quad$ x $\quad$ y $\quad$ x*d+y

g d x y = x*d +y

horner d xs = foldl f 0 xs

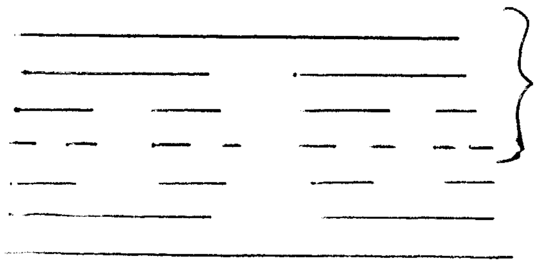
$\quad$ where f x y = g d x y

Sortieren

- Insertion Sort
- Quicksort

```
quicksort :: (Ord a) => [a] -> [a]
quicksort [] = []
quicksort (x:xs) = quicksort [y | y <- xs , y <= x]
                   ++ [x]
                   ++ quicksort [y | y <- xs, y > x]
```

- MergeSort



$\log_2 n$ Schichten

```
merge :: (Ord a) => [a] -> [a] -> [a] -> [a]

merge [] ys = ys
merge xs [] = xs
merge (x:xs) (y:ys)
   | x <= y = x : (merge xs (y:ys))
   | otherwise = y : (merge (x:xs) ys)
```

Funktionen höherer Ordnung

.. haben Funktionen als Argumente

- ```
  map :: (a -> b) -> [a] -> [b]
  map f [] = []
  map f (x:xs) = (f x) : (map f xs)
  ```

- ```
  filter :: (a -> Bool) -> [a] -> [a]
  filter p [] = []
  filter p (x:xs) | p x       = x : (filter p xs)
                  | otherwise = filter p xs
  ```

Inf. 5.1.05

Codierungstheorie

Aspekte:

1) Codierte Informationen sollen so kurz (kompakt) wie möglich sein

2) Übertragungsfehler sollten erkannt werden / korrigierbar sein

3) Geheimhaltung unterstützen

Teilgebiete

1) Datenkompression

2) Fehlererkennende u. fehlerkorrigierende Codes

3) Kryptographie

Modell: Informationsquelle

Reininformation i_0

$$\begin{tikzcd}
i_0 & i_0 \arrow[d, "\text{Quellkodierung}"] \\
i_1 \arrow[u, "\text{Quellendek.}"] & i_1 \arrow[d, "\text{Kanalkodierung}"] \\
\tilde{i}_2 \arrow[u, "\text{Kanaldekod. mit Korrektur}"] & i_2 \arrow[l, "\text{Übertr.-Kanal}"', "\text{(gestört)}"]
\end{tikzcd}$$

Kanalkodierung durch Blockkodes (alle Codewörter haben gleiche Länge)

Def.: Eine Blockcodierung (der Länge n) ist eine injektive Funktion $c: I \to Q^n$, wobei I eine Informationsmenge ist und Q ein endliches Alphabet ("Kanalalphabet")
Das Bild $Im(c) = \{c(i) \in Q^n \mid i \in I\}$ nennt man den Code
$|Q| = q$, hier $q = 2$ $Q = \{0, 1\}$

Def.: Seien $v = (v_1, \dots, v_n)$ und $w = (w_1, \dots, w_n) \in Q^n$
Der Hamming-Abstand zwischen v und w ist die Anzahl der Stellen, an denen sich die Tupel unterscheiden
$d(v, w) = |\{i \mid 1 \le i \le n \text{ und } v_i \ne w_i\}|$
Der Hamming-Abstand hat alle Eigenschaften eines Abstandsmaßes ($d \ge 0$; $d = 0 \Leftrightarrow v = w$; Symmetrie; Dr.-Ungl.)

Def.: $C \subseteq Q^n$, dann ist der Minimalabstand von C ist def. als
$d(C) = \min\{d(c, d) \mid c, d \in C \text{ und } c \ne d\}$

Angenommen, ein Codewort c wird gesendet und $v \in Q^n$ wird empfangen
1) Ist $v \notin C$, dann ist ein Fehler aufgetreten
2) Wenn höchstens ein Fehler aufgetreten ist und c ist das einzige Codewort mit Abstand 1 zu v, dann wurde c gesendet

Def.: Für jedes $v \in Q^n$ definieren wir die Kugel mit Radius t um v
$B_t(v) = \{w \in Q^n \mid d(v, w) \le t\}$
z.B.: $B_1((1,0,1)) = \{(1,0,1), (0,0,1), (1,1,1), (1,0,0)\}$

Def.: Ein Code C ist k-fehlererkennend, wenn bei jedem empfangenen Wort, das höchstens k Fehler enthält, erkannt wird, dass Übertragungsfehler aufgetreten sind.

Ein Code C ist k-fehlerkorrigierend, wenn bei jedem empfangenen Wort, dass $\leq k$ Fehler enthält, die Fehler korrigiert werden können, d.h. das gesendete Codewort eindeutig feststeht.

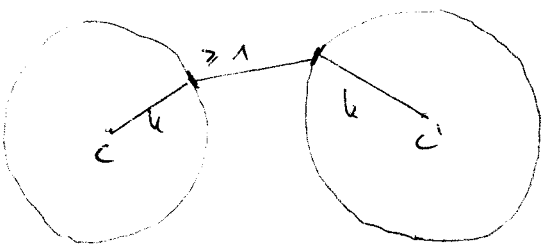
Satz: C ist k-fehlerkorrigierend $\iff \forall\, c \neq c' \in C \quad B_k(c) \cap B_k(c') = \emptyset$

$\iff d(C) \geq 2k+1$

c wird gesendet, v wird empfangen mit $\leq k$ Fehlern

$\Rightarrow v \in B_k(c)$, damit nicht in $B_k(c')$

$\Rightarrow c$ eindeutig bestimmt

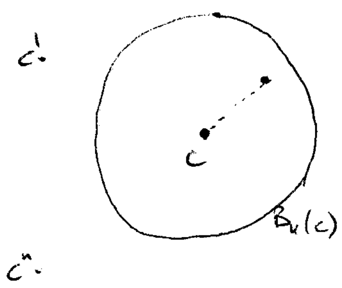


Mindestabstand: $2k+1$

Satz: Der Code C ist k-fehlererkennend

$\iff \forall c \in C \quad B_k(c) \cap (C \setminus \{c\}) = \emptyset$

$\iff d(C) \geq k+1$



Standardcodierungen

$I = Q^m$, d.h. Informationen liegen bereits als Tupel vor.

1) Codierung mit Paritätsbit

$c_{par} : Q^m \to Q^{m+1}$

$c_{par}(v_1, \dots, v_m) = (v_1, \dots, v_m, p)$ $\qquad p = (v_1 + \dots + v_m) \bmod 2$

Code c_{par} $\qquad d(c_{par}) = 2$

$d(c_{par}(v), c_{par}(v')) \geq d(v, v')$, d.h. ≥ 2 wenn $d(v,v') \geq 2$,
$= 2$ wenn $d(v,v') = 1$

c_{par} ist 1-fehlererkennend

2) Doppelcodierung

$c_2 : Q^m \to Q^{2m}$ (einfache Verdopplung)

$d(c_2) = 2 \Rightarrow c_2$ ist 1-fehlererkennend

3) Dreifachcodierung

$c_3 : Q^m \to Q^{3m}$

$d(c_3) = 3 \Rightarrow c_3$ ist 1-fehlerkorrigierend

4) Doppelcodierung mit Paritätsbit

$c_{2,par} : Q^m \to Q^{2m+1}$ (Paritätsbit von ursprünglichem Wert)

$d(c_{2,par}) = 3 \Rightarrow$ 1-fehlerkorrigierend

5) Kreuzsicherungscode

$c_{kr} : Q^m \to Q^{m+2\ell}$; $m = \ell^2$

$$\begin{array}{|cccc|c} v_1 & v_2 & \dots & v_\ell & \overline{p}_1 \text{ - Paritätsbit} \\ v_{\ell+1} & \dots & \dots & v_{2\ell} & \vdots \\ \vdots & & & \vdots & \vdots \\ v_{(\ell-1)\ell+1} & \dots & & v_{\ell^2} & \overline{p}_\ell \\ \hline p_1 & \dots & & p_\ell & \end{array}$$

$c_{kr}(v_1 \dots v_m) = (v_1, \dots, v_m, p_1, \dots p_\ell, \overline{p}_1, \dots \overline{p}_\ell)$

Zeigen, dass $d(C_{kr}) = 3$ d.h.

$d(C_{kr}(v), C_{kr}(w)) \geq 3$ für alle $v \neq w \in Q^m$

1) $d(v,w) \geq 3 \Rightarrow d(C_{kr}(v), C_{kr}(w)) \geq 3$

2) $d(v,w) = 2 \Rightarrow d(C_{kr}(v), C_{kr}(w)) \geq 4$

3) $d(v,w) = 1 \Rightarrow d(C_{kr}(v), C_{kr}(w)) = 3$

C_{kr} ist 1-fehlerkorrigierend

Informationsrate

Def: Die Informationsrate eines Codes $C \subseteq \{0,1\}^n$ ist der Quotient $\frac{\log_2 |C|}{n}$ (allgemein $\frac{\log_{|Q|}|C|}{n}$)

Für $C_{2,p} : Q^m \to Q^{2m+1}$ $\quad \frac{\log_2 2^m}{2m+1} \approx \frac{1}{2}$

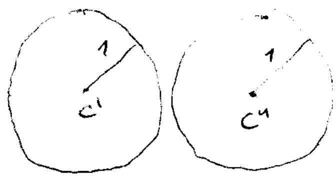
Für $C_{kr} : Q^{\ell^2} \longrightarrow Q^{\ell^2+2\ell}$

$|C_{kr}| = 2^{\ell^2} \quad n = \ell^2 + 2\ell$

$$\frac{\log_2 2^{\ell^2}}{\ell^2+2\ell} = \frac{\ell^2+2\ell-2\ell}{\ell^2+2\ell} = 1 - \frac{2\ell}{\ell^2+2\ell} \approx 1 - \frac{2}{\ell}$$

Was ist best mögliche Informationsrate für 1-fehlerkorrigierende Codes?

$C \subseteq Q^n$



$c' = \underbrace{(1,0,0,1,\dots)}_{n}$

$|B_1(c')| = n+1$

$|C| \cdot (n+1) \leq 2^n \Rightarrow |C| \leq \frac{2^n}{n+1}$

$\log_2 |C| \leq \log_2\left(\frac{2n}{n+1}\right) = \log_2(2n) - \log_2(n+1) = n - \log_2(n+1)$

bestmögliche Informationsrate

$$\frac{n - \log_2(n+1)}{n} = 1 - \frac{\log_2(n+1)}{n}$$ „perfekter Code"

<u>Quellencodierung</u>:

Ziel: komplexe Information so codieren, dass die codierte Information möglichst kurz ist

Idee: Codes variabler Länge, häufige Symbole haben kurze Codierung

Problem: Entweder man braucht zusätzliche Trennsymbole (nicht kompakt) oder man muss Trennungsstellen algorithmisch erkennen

Lösung: Präfixcodes

Def.: Eine Codierung variabler Länge ist eine injektive Funktion $c: A \to \{0,1\}^*$ (Menge endl. Wörter über diesem Alphabet); $C = \{c(a) \mid a \in A\}$ Code

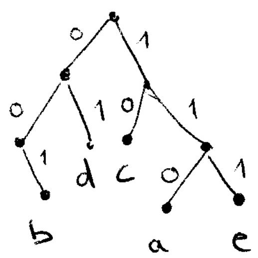
Def.: $C \subseteq \{0,1\}^*$ ist ein Präfixcode, wenn kein Codewort $c(a)$ Präfix eines anderen Codeworts $c(b)$ ist.

Beispiele:

1) Jeder Blockcode ist Präfixcode
2) $\{01, 001, 100, 110, 111\}$ ist Präfixcode
3) $\{0, 10, 11, 101\}$ ist kein Präfixcode

Präfixcodes können mit binären Bäume dargestellt werden

für Bsp 2:



Codewörter <u>nur</u> in Blättern

Codierung eines Worts $a_1 \dots a_n \in A^*$ durch

$c(a_1)c(a_2)\dots c(a_n)$ einfache Konkatenation

abad $\rightarrow$ 11000111001

Decodierung durch Baumdurchlauf
Wenn Blatt erreicht wird, Symbol auslesen und zurück zur Wurzel

Satz (Ungleichung von Kraft):

(1) Sei $|A| \geq 2$ und $c: A \rightarrow \{0,1\}^*$ ein Präfixcode, sei $|c(a)|$ die Länge des Codeworts $c(a)$, dann gilt die folgende Ungleichung

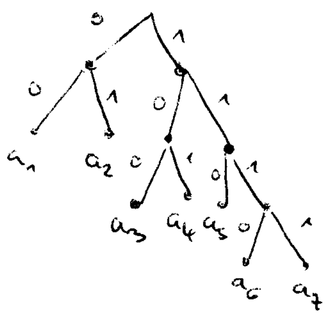
$$\sum_{a \in A} \frac{1}{2^{|c(a)|}} \leq 1$$

(2) Sei $n \geq 2$ und $m_1 \dots m_n \in \mathbb{N}^+$, dann gibt es einen Präfixcode $c: \{a_1, \dots, a_n\} \rightarrow \{0,1\}^*$ mit $|c(a_i)| = m_i$ für $i = 1, \dots, n$ falls $\sum_{i=1}^{n} \frac{1}{2^{m_i}} \leq 1$

Beweisidee (1): Induktion nach $|A|$

(2): Konstruktiver Beweis

Beispiel $\underset{m_1}{2}, \underset{m_2}{2}, 3, 3, 3, 4, \underset{m_7}{4}$ | $\frac{1}{2^2} + \frac{1}{2^2} + 3 \cdot \frac{1}{2^3} + 2 \cdot \frac{1}{2^4} = 1 \leq 1$



Inf. 12.01.05

Zu codierendes Alphabet $A = \{a_1, \dots, a_n\}$ mit Wahrscheinlichkeits-verteilung $p: A \to [0,1]$ mit $\sum_{i=1}^{n} p(a_i) = 1$

Sei $c: A \to \{0,1\}^*$ eine Codierung (Prefixcode) von A, dann ist die erwartete Länge eines Codeworts $\mu(c) = \sum_{i=1}^{n} p(a_i) \cdot c(a_i)$

Def.: Ein Präfixcode $c: A \to \{0,1\}^*$ ist optimal für (A,p), wenn für jede andere Codierung $c', A \to \{0,1\}^*$ gilt, sodass $\mu(c) \leq \mu(c')$

$$\mu(c) = \sum_{i=1}^{n} p(a_i) \cdot |c(a_i)| = \sum_{i=1}^{n} p(a_i) \cdot d_T(a_i)$$; d_T = Tiefe von Blatt a_i im Codebaum

Einführung von gewichteten binären Bäumen

1) Blättern werden Gewichte zugeordnet
2) innere Knoten bekommen als Gewicht die Summe der Gewichte aller Blätter unter diesem Knoten

Hier: Gewichte der Blätter sind Wahrscheinlichkeiten $p(a_i)$

$\Rightarrow$ Wurzel hat Gewicht 1

Huffman Algorithmus

Gegeben: A mit Verteilung p

Initialisierung: Liste von n Bäumen erstellen: Jeder Baum nur ein Knoten mit Markierung a_i und Gewicht $p(a_i)$

Wiederhole $(n-1)$ mal: Streiche aus der Liste die beiden Bäume mit geringsten Gewicht und füge neuen Baum ein, dessen Wurzel die gestrichenen Bäume als Kinder hat

Gib den Baum aus der Liste aus.

Lemma 1: Sind a, b die Symbole aus A mit den geringsten Wahrscheinlichkeit, dann gibt es eine optimale Codierung, sodass a, b Zwillingsblätter sind

Beweisidee: jeder innere Knoten hat zwei Kinder, d.h. jedes tiefste Blatt hat einen Zwilling

Lemma 2: Wenn T Baum einer optimalen Codierung ist für (A, p) und a, b als Zwillingsblätter auftauchen, dann ist der folgende Baum auch optimal: Streiche a, b heraus und markiere den Vater mit einem neuen Symbol z

optimal für $(A \setminus \{a,b\}) \cup \{z\}$ mit p'

$$p'(x) = \begin{cases} p(x) & \text{für } x \neq a, b \\ p(a) + p(b) & \text{für } x = z \end{cases}$$

Algebraische Typen in Haskell

Ziel: Zusammenfassung verschiedenartiger Objekte zu einem neuen Typen

Beispiele:

Monat = { Januar, ..., Dezember }

Kraftfahrzeug = { Motorrad, Bus, ... }

Figur = { Kreis, Rechteck, ... }

Syntax:

```
data Jahreszeit = Fruehling | Sommer | Herbst | Winter
data Bool = True | False
```

Konstruktoren können auch mit Parametern ausgestattet werden

Beispiel:

1) data Mensch = Person String Int

type Name = String

type Jahr = Int

data Mensch = Person Name Jahr

Person "Anna" 74 zum Anlegen eines Objekts

2) data Figur = Kreis Float | Rechteck Float Float

Konstruktoren sind Funktionen

Rechteck :: Float Float -> Figur

Allgemeine Form der Definition eines algebraischen Typs

data Typname = $Constr_1$ t_{11} t_{12} .. t_{1k} |
$Constr_2$ t_{21} t_{22} .. t_{2k} |
...
$constr_n$ t_{n1} t_{n2} .. t_{nk}

Rekursion möglich, d.h. definierte Typ kann auch als Parameter verwendet werden

1) Palindrome "anna" abxba

data Palindrom = Empty |
Single Char |
Composed Char Palindrom

Composed 'x' single a steht für xax

1) Palindrome

data palindrome = Empty |
Single Char |
composed char palindrom

ab x b a

↗ single ↳ composed b x

toString :: Palindrom → String

toString (Empty) = []

toString (Single a) = [a]

toString (composed a pd) = [a] ++ toString pd ++ [a]

2) Binäre Bäume

Menge von Knoten mit Vater-Sohn-Bez. Jeder Knoten hat höchstens zwei Kinder; ein linkes und/oder ein rechtes Blatt (Blatt = kinderlose Knoten)

Knoten mit mind. einem Kind = innere Knoten

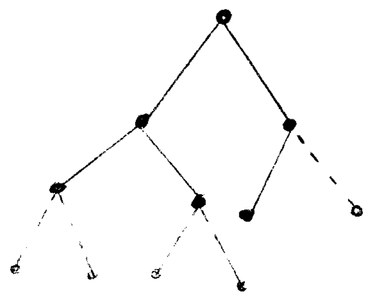
~~Knoten mit mind. einem Kind~~

Ein bin. Baum ist echt, wenn jeder innere Knoten genau zwei Kinder hat

Algebraischer Typ für echte bin. Bäume:

data BinTree = Leaf |
Node BinTree BinTree

allgemeine bin Bäume



unechte bin. Bäume werden zu bin. Bäumen erweitert

data NTree = Nil |
 Node NTree NTree

---

für Huffman-Codierung

data HTree = Leaf Float Char |
 Node HTree HTree Float

Größe und Tiefe von bin Bäumen (NTree)

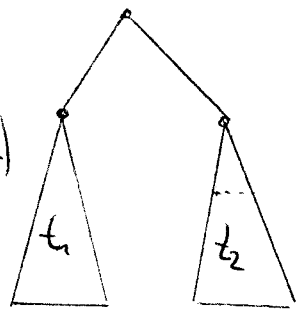
Größe = Anzahl der nicht-Nil-Knoten

Tiefe (Höhe = Länge des längsten Weges von Wurzel zu einem Blatt (Nil-Knoten)

size :: NTree -> Int

size (Nil) = 0

size (Node t_1 t_2) = 1 + (size t1) + (size t2)



height :: NTree -> Int

height (Nil) = 0

height (Node t1 t2) = 1 + max (height t1) (height t2)

Zusammenhang zwischen Höhe und Größe

Satz: Für jeden NTree gilt $\text{height}(T) \leq \text{size}(T) \leq 2^{\text{height}(T)}$

h 

h+1 Knoten

⇒ ≥h innere Knoten

vollst. Induktion

...

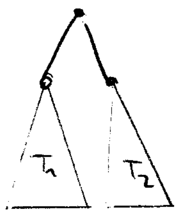
Fortführung vollst. ~~Information~~ Induktion

Anfang: Für $h(T) = 0$

$s(T) = 0 < 2^0 = 1$

IV.: $s(T) < 2^{h(T)}$ für alle Bäume mit $h(T) \leq n$

IS.: Sei T ein NTree der Höhe $n+1$



$h(T_1) \leq n \qquad h(T_2) \leq n$

nach I.V.:

$s(T_1) < 2^n \qquad s(T_2) < 2^n$

$\Rightarrow s(T_1) \leq 2^n - 1 \qquad s(T_2) \leq 2^n - 1$

$$s(T) = 1 + s(T_1) + s(T_2) \leq 1 + 2^n - 1 + 2^n - 1$$
$$\leq 2^n + 2^n - 1 < 2^{n+1} = 2^{h(T)}$$

Anmerkung: Algebraische Typen (insbs. auch rekursive) können polymorph definiert werden z.B.:

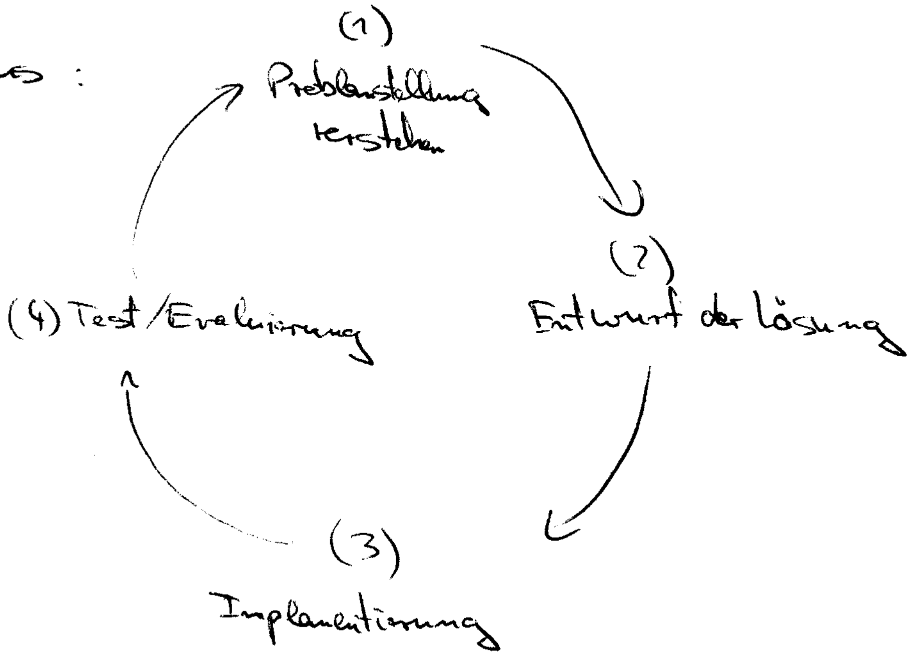
Tree a = Nil | Node a (Tree a) (Tree a)

Durchlaufen von Bäumen (Tree-Traversal)

Inorder, Postorder, Preorder

<u>Modularisierung</u>.

Entwicklungszyklus:



19.1.05

– alg. Typen

1) Aufzählender Typ

Bsp. data Bool = True | False

data Ordering = LT | EQ | GT

Funktionen mit Pattern matching

2) Produkttyp (-> Tupelbildung)

data People = Person Name Age

Typname Konstruktor

type Name = String

type Age = Int

alternativ:

type People = (Name, Age)

3) Alternativen (mengentheor. Vereinigung)

data Shape = Circle Float | Rectangle Float Float

Konstrukt

4) Rekursive / polymorphe alg. Typen

Listen [a] über Typ a

Binäre Bäume

data Tree a = Nil | Node a (Tree a) | Node b (Tree a)

deriving (Eq, Ord, Show, Read)

bestimmt Funktionalität von a wird „hochgezogen" auf Bäume

→ rekursiv def. Fkt. ⇔ Induktionsbeweis

Modularisierung in Haskell

Parnas (1972): "Information Hiding Principle"

heißt: Zerlegung in Teilaufgaben (Module)

lokale Entwurfsentscheidungen innerhalb eines Moduls sollten

- für die Benutzung des Gesamtsystems irrelevant sein
- von außen nicht sichtbar sein

Kenntnis einer Spezifikation des Moduls ist notwendig und hinreichend für die korrekte Benutzung

→ hierarchische Zerlegung ("Wer nutzt was von wem")

→ Import / Export

Syntax:

Modulnamen: großer Anfangsbuchstaben

```
module Modulname where
    function1 :: ...
    :
```

} "sichtbaren" Definitionen des Moduls

Import eines Moduls

```
module Modul1 where
    import Modul2   -- sichtbaren Def. von Modul2 können genutzt werden
    function :: ...
    :
```

Import / Export - Kontrollen

Explizite Angabe dessen, was imp / exp. wird

module Modul1 (Def) where

geht auch:

module Modul1 (..., module Modul2, ...)

import Modul2 hiding (...)

import qualified Modul1 - kann auf Modul1.function zugreifen

-> standardmäßig wird Prelude als Modul importiert

Huffman-Codierung

Präfixcode $c: A \to \{0,1\}^+$

Ziel: Datenkompression

$a \to p(a)$ Häufigkeit mit $\sum_{a \in A} p(a) = 1$

=> Baumdarstellung

-> Greedy - Verfahren

Präfixcode <-> Codewortbaum

Codewort für a <-> Markierung auf Weg Wurzel zu Blatt a

Huffman <-> Knoten zusätzlich markiert mit Häufigkeit aller Blätter darunter

Hauptschritte:

- Häufigkeit der Zeichen im Text feststellen
- Verschmelzen und sortieren von Bäumen
- Codebaum in Codetabelle übersetzen

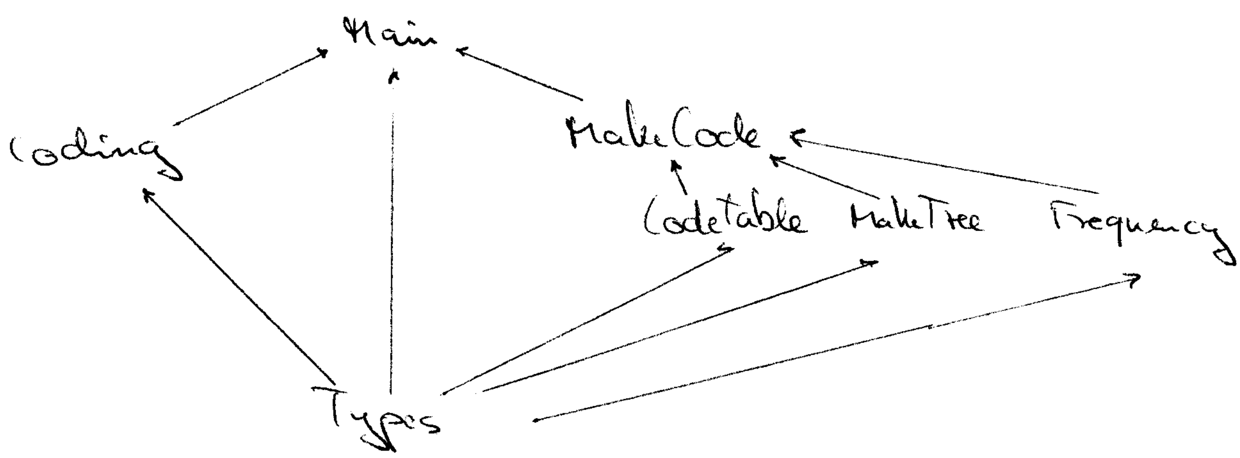
module MakeCode

- Codieren / Decodieren

Welche Typen braucht man?

```
data Bit = L | R
type HCode = [Bit]
data Tree = Leaf Char Int | Node Int Tree Tree
type Table = [(Char, HCode)]
```

Bilden Modul Types



Klassen in Haskell

- Klassen in Haskell sind Typklassen zur Zusammenfassung von bestimmten Datentypen mit gemeinsamen Eigenschaften, nämlich dass bestimmte Funktionen definiert sind
- Entsprechung in Java-Interface
- Haskell-Klasse ist Liste von Funktionsname mit Signatur
- Datentyp ist Exemplar (Instanz) einer Klasse, wenn alle Funktionen der Klassenliste definiert ist
- Es gibt eine Reihe vordefinierter Klassen Eq, Ord

Klassendefinition:

```
class ClassName a where
    Liste von Funktionen und Signatur
```

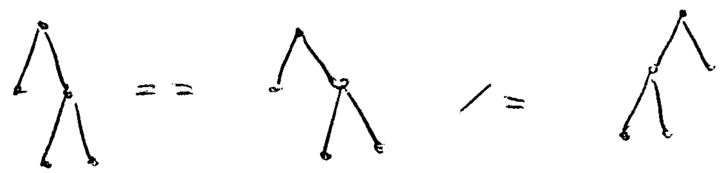
Beispiel:

```
class Eq a where
    (==), (/=) :: a -> a -> Bool
    x /= y = not (x == y)
    x == y = not (x /= y)
```

Alle primitiven Datentypen sind Instanzen von Eq
Tupel und Listen sind Instanzen von Eq

Algebraische Typen werden zu Exemplaren von Eq, wenn man das fordert

```
data BinTree = Nil | Node BinTree BinTree
    deriving (Eq)
```



Node Nil (Node Nil Nil) Node Nil (Node Nil Nil) ~> True

type Bruch = (Int, Int)   (4,3) /= (8,6) -> nicht gewünscht

data Bruch = Bruch Int Int

mit deriving (Eq) gleicher Effekt wie oben

2. Variante: ohne deriving (Eq)

```
instance Eq Bruch where
    (Bruch a b) == (Bruch c d) = (a*d == c*b)
```

Ord setzt Eq voraus und enthält Vergleichsoperatoren

```
class (Eq a) => Ord a where
  (<), (<=), (>), (>=) :: a -> a -> Bool
  max, min :: a -> a -> a
```

Enum setzt Ord voraus

```
class (Ord a) => Enum a where
  toEnum :: Int -> a
  fromEnum :: a -> Int
```

Für Char sind chr und ord Synonyme für toEnum und fromEnum §

Show: neben Eq zweite Basisklasse

- fordert, dass ein Objekt in einen String umgewandelt werden kann
- Wird zur Bildschirmanzeige verwendet
- Alle Basistypen sind Exemplare von Show
- Tupel und Listen — " —

(sind a, b Exemplare von Show, dann auch (a, b) und [a]

Standardmechanismus für algebraische Typen ähnlich wie bei deriving(Show)

2. Variante ohne deriving(Show).

instance Show Bruch where

show (Bruch) a b =

Bei polymorphen Typen (z.B. binäre Bäume und Markierungen) kann deriving(Show) verwendet werden, wenn der verwendete Typ a selbst Exemplar von Show ist

data (Eq a, Show a) => BinTree a = Nil | Node (BinTree a) (BinTree a)

optional

— deriving (Eq, Show)

Lazy Evaluation

Grundprinzip von Haskell: Argumente einer Funktion werden erst dann evaluiert, wenn sie tatsächlich gebraucht werden

Konsequenz: Man kann mit unendlichen Listen arbeiten

Einfachster Fall einer Auswertung ist eine Termersetzung

Bsp.: f x y = x + y   Auswertung von f (6-4) (f 5 3)

(6-4) + (f 5 3)

2 + (f 5 3)

2 + (5 + 3)

2 + 8

10

g x y = x + 8   Auswertung von g 3 (g 8 5)

3 + 8

11

Duplizierte Argumente werden nur einmal ausgewertet

h: Int -> Float -> Float -> Float

h n x y

| n 70 = x * x

| otherwise = y * y

Auswertung von h 5 (3+1) (h 0 8 6)

(3+1) * (3+1)

4 * 4

Auswertung bei Fallunterscheidungen etc.

- Guards werden in gegebener Reihenfolge ausgewertet, bis zum ersten True
- Danach Auswertung der Funktion für diesen Fall

Analog Auswertung für verschiedene Muster

Auswertung von Listen

Grundsatz: Es wird nur der Teil einer Liste erzeugt, auf den andere Funktionen zugreifen

Bsp.: Minimales Element durch Insertion Sort und herausnehmen des ersten Elements

head . insSort [8,2,6]

head (ins 8 insSort [2,6])

ins 2 insSort [6]

ins ~~Sort~~ 6 insSort []

ins 6 []

6 : []

2 : 6 : []

(2 : (ins 8 (6 : [])))

2 (ins 8 (6 : []))

wird nicht ausgewertet

Auswertung von List-Comprehension

allgemeine Form

$[e \mid g_1, \dots g_k]$

g_i sind entweder Generatoren $p \leftarrow \mid Exp$

oder Bedingungen

e ist ein Ausdruck, der alle generierten Elemente verwenden darf

Regel: Generatoren von links nach rechts auswerten, Bedingungen prüfen

Abbruch des speziellen Falls bei False

Beispiel: pairs : [Int] -> [Int] -> [(Int, Int)]

pairs xs ys = [(x,y)] | x ← xs , y ← ys]

pairs [1,2,3] [4,5] = [(1,4), (1,5), (2,4) ...]

bei pairs xs ys = [(x,y) | y ← ys , x ← xs] andere Reihenfolge

Beispiel 2: Pythagoräische Zahlentripel $(x,y,z) \in \mathbb{N}^3$ so dass $x^2 + y^2 = z^2$

o.B.d.A $1 \leq x < y$

[(x,y,z) | x ← [2..n] , y ← [x+1..n] , z ← [y+1..n],

x*x + y*y == z*z]

Was passiert, wenn ob. Grenze n nicht explizit verwendet wird, d.h. bei unendlichen Listen

Syntax einer unendlichen Liste durch [2..] → [2,3,4,...]

oder durch [2,4..] → [2,4,6,...]

$[(x,y,z) \mid x \leftarrow [2..], y \leftarrow [x+1..], z \leftarrow [y+1..], x*x+y*y==z*z$

erzeugt leere Liste

besser: $z \leftarrow [4..]$ an den Anfang setzen

$z \leftarrow [4..]$, $y \leftarrow [3..z-1]$, $x \leftarrow [2..y-1]$

Beispiel 3: Liste der Primzahlen

1) $[x \mid x \leftarrow [2..], \text{prim } x]$

prim x = foldr (&&) True [(mod x y) > 0 | y ← [2..x-1]]

2) Sieb des Eratosthenes

sieve (x:xs) = x : sieve [y | y ← xs, (mod x y) > 0]
primes = sieve [2..]

2 : sieve [y | y ← [3..], (mod y 2) > 0]

2 : 3 : sieve [y | y ← [4..], (mod y 2) > 0, (mod y 3) > 0]

...

Programmierung mit Eingabe / Ausgabe

Standardeingabe : Tastatur
Standardausgabe : Bildschirm + Dateien

I/O - Aktion hat Rückgabetyp IO String bei Eingaben
IO Char
IO Int
IO () ohne Rückgabe (→ Ausgabe)

Spezielle Aktionen

```
getline :: Io String        (Stringeingabe von Tastatur)
getChar :: Io Char
putStr :: String -> IO()
putStrLn :: String -> IO()        (mit Zeilenumbruch)
print :: Show a => a -> IO()
      print = putStrLn . show
return :: a -> IO a        (keine IO-Aktion, sondern nur Rückgabe)
```

Folge von Aktionen mit do-Anweisung

```
put2times :: String -> IO()
put2times s = do putStrLn s
                 putStrLn s


reverse2lines :: IO()
reverse2lines = do line <- getLine
                   line2 <- getLine
                   putStrLn (reverse line2)
                   putStrLn (reverse line1)
```

Typumwandlung von Strings

```
read :: Read a => String -> a
getInt :: Io Int
getInt = do line <- getLine
            return (read line :: Int)
```

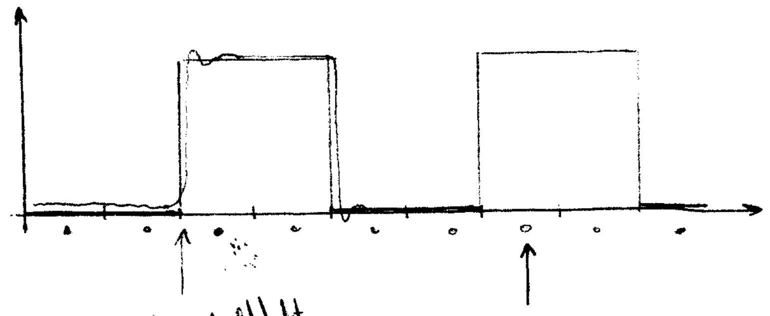
Inf 2.7.05

Rechnerstrukturen

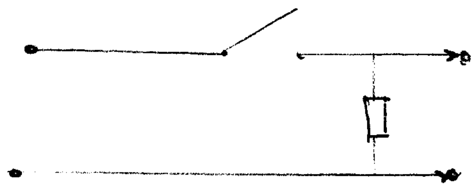
Transistoren, Gatter und Schaltnetze

Spannungswerte: 0 (auch Null, Masse)

1 (3,3V bei MOS)



Realisierung durch Schalter



Schalter offen $\Rightarrow u_{out} = 0$

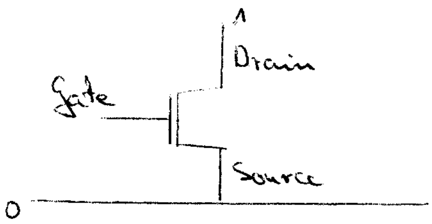
Schalter geschlossen $\Rightarrow u_{out} = u_{in}$

Reihenschaltung von Schalter $\Rightarrow$ logisches Und

Parallelschaltung von " $\Rightarrow$ logisches Oder

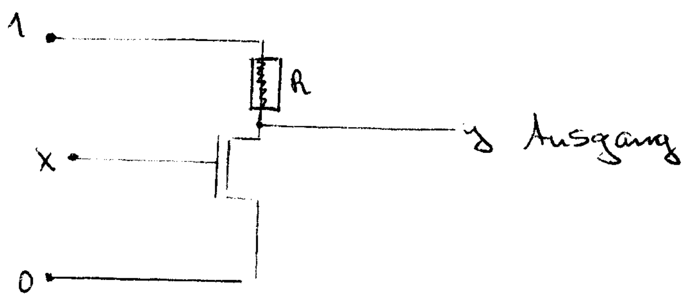
MOS-Transistor (Metall-Oxid-Semiconductor)

schematisch



Gatespannung beeinflusst Widerstand zwischen Source und Drain

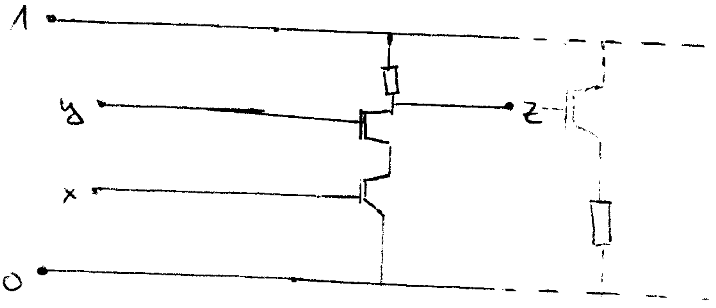
Gatespannung
0 $\longrightarrow$ sehr großer W.
1 $\longrightarrow$ sehr kleiner W.



$x = 0 \Rightarrow y = 1$

$x = 1 \Rightarrow y = 0$

Negationsgatter



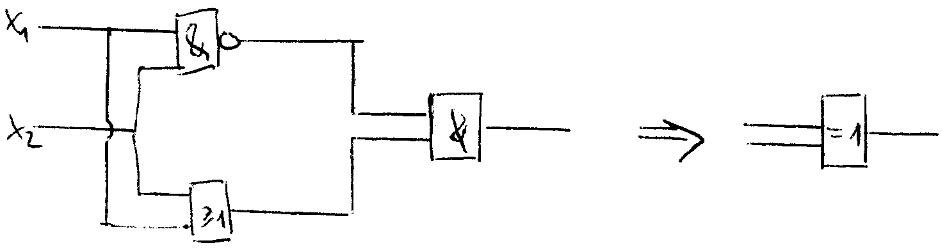
$y = x = 0 \Rightarrow z = 0$

Sonst $\Rightarrow z = 1$

Nand-Gatter

And - Gatter

Schaltung für Antivalenz



Strukturelles Modell für Schaltnetze

Gerichteter azyklischer Graph

Knotenmenge: Ein- und Ausgabeknoten
Verzweigungsknoten
Gatter

gerichtete Kanten: von Eingabeknoten oder Verzweigungsausgang
oder Gatterausgang
zu Ausgabeknoten oder Verzweigungseingang
oder Gattereingang

Gütekriterien: Aufwand (Anzahl der Transistoren / Gatter)

Zeitverhalten (Signallaufzeit, max Anzahl von Gattern auf einem gerichteten Weg)

Entwurfsaufwand

allgemeines Vorgehen zum Entwurf von Schaltungen:

1) Beschreibung als Boolesche Funktion in Tabellenform
2) DNF oder KNF bilden
3) Vereinfachung durch Termzusammenfassungen
4) Umsetzung in Gatter

Mooresches Gesetz (Exponentialgesetz der Mikroelektronik):

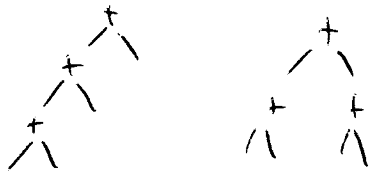
- Anzahl der Transistoren auf einem Prozessorchip verdoppelt sich ca. alle zwei Jahre
- Anzahl der Tr. auf Speicherchips verdoppelt sich ca. alle 18 Monate
- Doppelte Leistung zum gleichen Preis auch ca. 2 Jahre

Multiplikation von 2 n-bit Zahlen

(n-1) Addition von 2n-bit Zahlen

(n-1) · Additionszeit (naiv)

rekursiv über binären Baum: $\log_2 n$ · Additionszeit



	Carry-ripple t.	Carry-lookahead
t_n	$O(n \log_2 n)$	$O(\log_2^2 n)$
s_n	$O(n^2)$	$O(n^{1+\log_2(3)})$

Multiplikation von negativen Zahlen

Problem: 2-Kompliment nicht geeignet

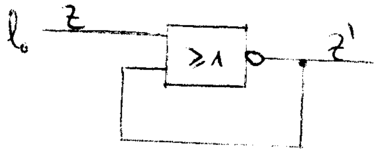
Lösung:
1) Vorzeichen der Faktoren unterscheiden
2) negative Zahlen in Betrag umwandeln
3) Multiplikation ausführen
4) Vorzeichen des Produkts analysieren, evt. durch Komplementbildung in neg. Zahl umwandeln

Schaltwerke

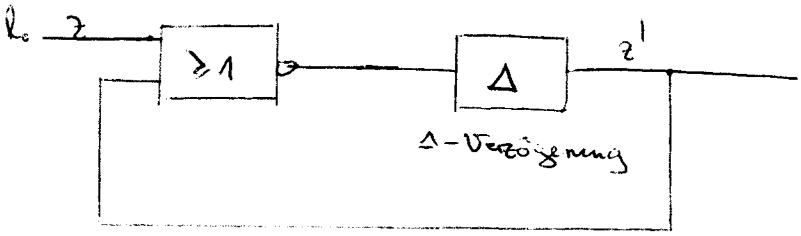
Schaltnetze: gerichteter azyklischer Graph, d.h. Rückkopplung verboten

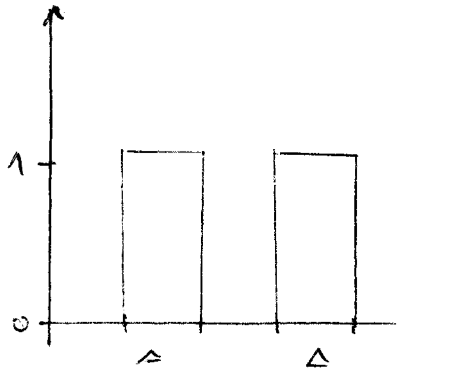
Schaltwerk: gerichteter (nicht azyklischer) Graph, d.h. Rückkopplungen erlaubt und sogar beabsichtigt zur Konstruktion von selbsterhaltenden Zuständen (Speicher)

Einführungsbeispiel: Rückgekoppeltes NOR



$$z = 0 \Rightarrow \begin{cases} z' = 0 & \text{für } z' = 1 \\ z' = 1 & \text{für } z' = 0 \end{cases}$$ Widerspruch?





$z(t) = 0 \Rightarrow z'(t+\Delta) = 1$

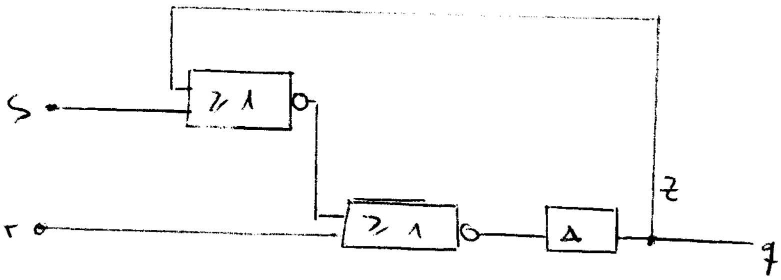
$\Rightarrow z(t+\Delta) = 1 \Rightarrow z'(t+2\Delta) = 0$

$\Rightarrow z(t+2\Delta) = 0$ $\longrightarrow$ instabil

Ziel: Zustände, die sich selbst stabilisieren

Beispiel: Speicher für ein Bit

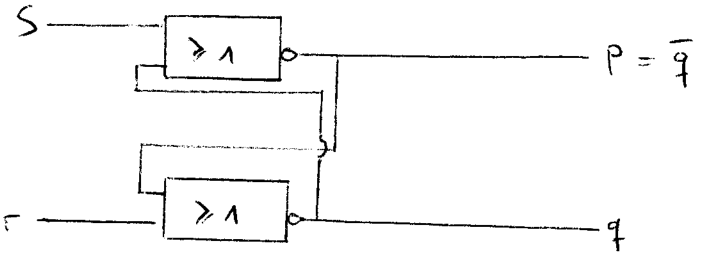
- zwei Eingänge s(set) r(reset)
- Impuls auf s soll Zustand q auf 1 setzen
- Impuls auf r — " — 0 "



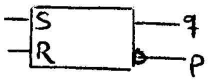
$\left.\begin{array}{l} s(t) = 1 \\ z(t+\Delta) = 1 \end{array}\right\}$ bleibt stabil, bis $r(t') = 1$

$s(t)$	$r(t)$	$z(t)$	$q(t+\Delta) = z(t+\Delta)$	
0	0	0	0	stabil
0	0	1	1	
1	0	0	1	setzen
1	0	1	1	~~setzen~~ stabil
0	1	0	0	
0	1	1	0	setzen

Andere Darstellung der Schaltung:

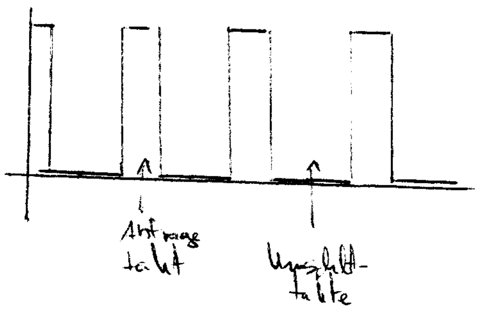


„Asynchrones RS-FlipFlop“

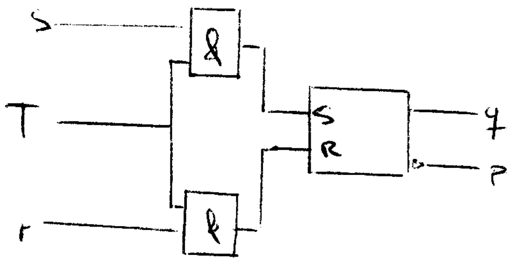
Schaltsymbol: 

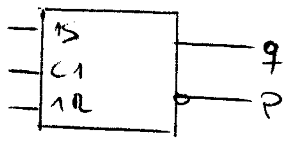
Zentraler Taktgeber

⇒ Synchrone Schaltwerke

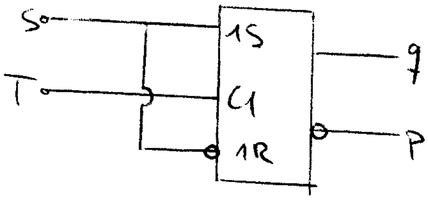


Pegelgesteuertes RS-Latch



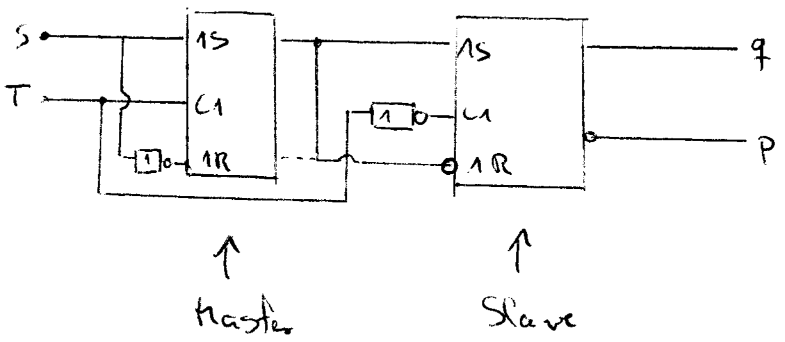
Schaltsymbol: 

Was passiert bei bei r = s = 1



D-Latch / D-Flipflop

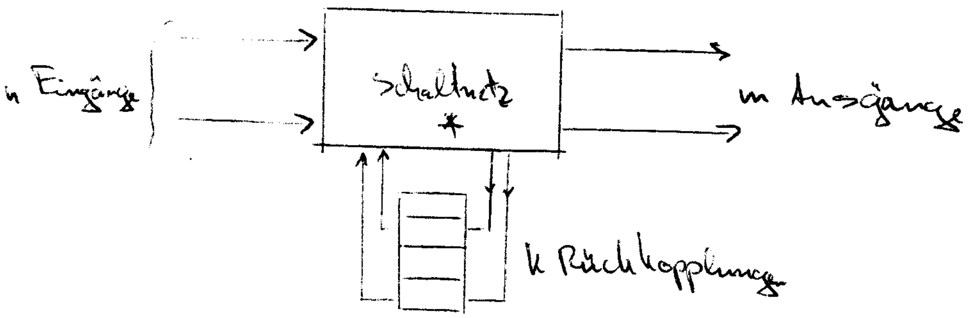
Master-Slave-Flipflop



taktflankengesteuertes FlipFlop

Modellierung von synchronen Schaltwerken durch endliche Automaten

- alle Rückkopplungen über getaktete Verzögerungsglieder
- Schaltwerke ohne Rückkopplungen sind Schaltnetze

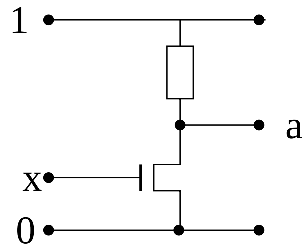


Ein Schaltnetz realisiert Funktion $f: \mathbb{B}^n \to \mathbb{B}^m$

Das Schaltnetz * realisiert Funktion $f: \mathbb{B}^{n+k} \to \mathbb{B}^{m+k}$

Nach außen sichtbar: $f: (\mathbb{B}^n)^* \longrightarrow (\mathbb{B}^m)^*$

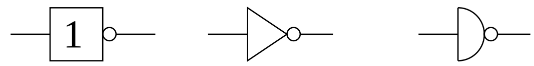
Schaltung	Funktion	symbolische Darstellung: DIN	amerik.	alt

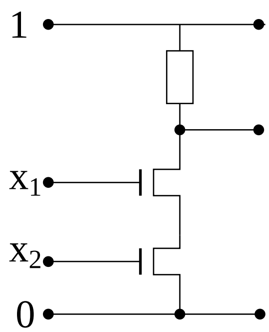


Negation

NOT

$a = \neg x$

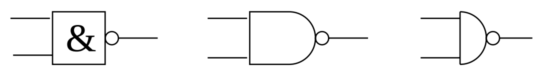


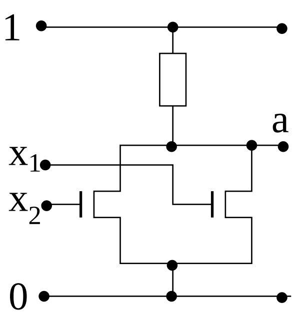


Nicht−Und

NAND

$a = \neg(x_1 \wedge x_2)$

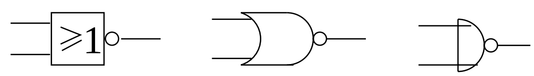


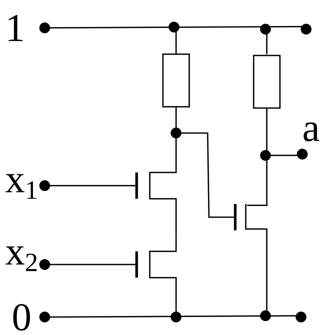


Nicht−Oder

NOR

$a = \neg(x_1 \vee x_2)$

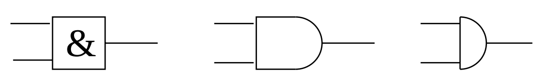




Und

AND

$a = x_1 \wedge x_2$

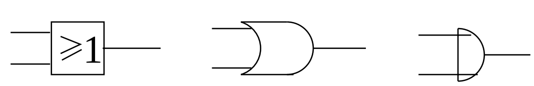


Kombination von

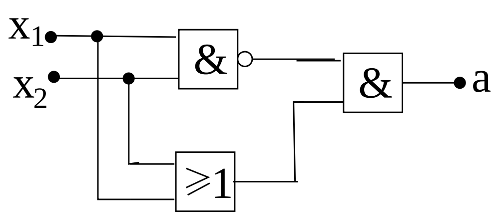
NOR und NOT

Oder OR

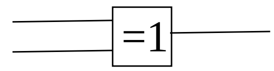
$a = x_1 \vee x_2$



Ein Schaltnetz für die Antivalenz EXOR



symbolisch



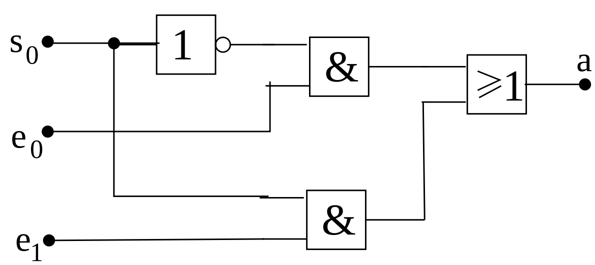
Schaltnetze für Multiplexer MUX

Ein Multiplexer hat $2^n + n$ Eingänge n Steuerleitungen und 2^n Datenleitungen

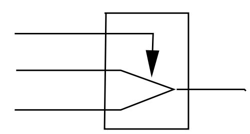
Die Eingabewerte der Steuerleitungen codieren die Nummer der Datenleitung, deren Wert ausgegeben werden soll

n = 1: Steuersignal s_0 Datensignale e_0 e_1 Ausgang a

s_0	a
0	e_0
1	e_1

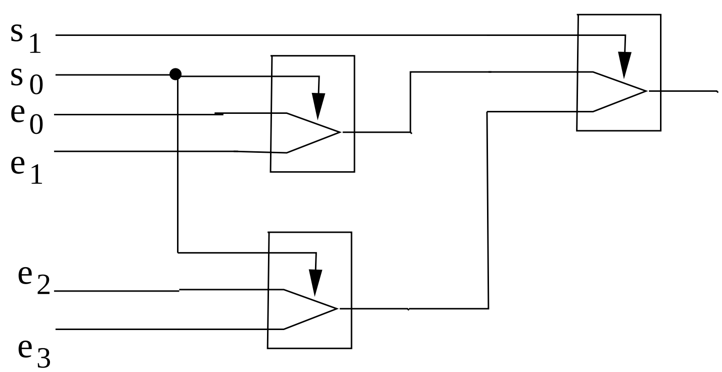


symbolisch



n = 2: Steuersignale s_0 s_1 Datensignale e_0.... e_3 Ausgang a

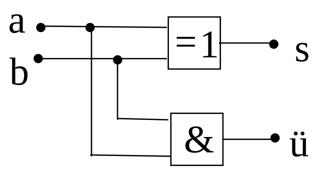
s_0	s_1	a
0	0	e_0
0	1	e_1
1	0	e_2
1	1	e_3

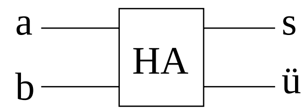


Halbaddierer und Volladdierer

1Bit−Halbaddierer berechnen Summe s und Übertrag ü

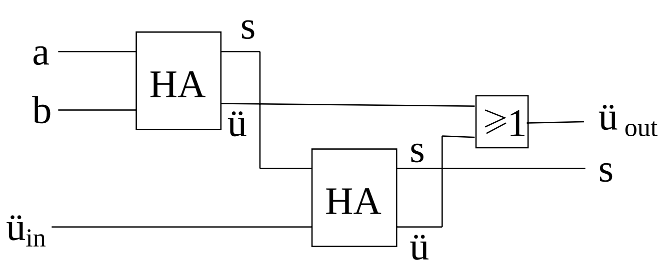
$s = a \oplus b$ (Antivalenz) $ü = a \ \& \ b$

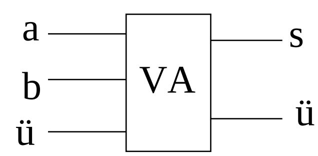




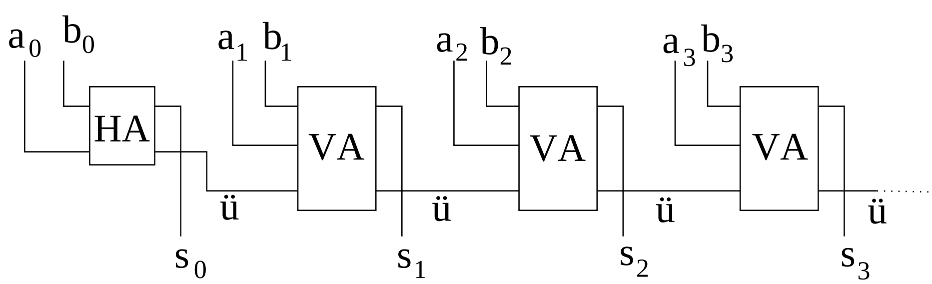
1Bit−Volladdierer beziehen in die Berechnung einen übergebenen Übertrag $ü_{in}$ ein: $s = a \oplus b \oplus ü_{in}$

$ü_{out}$ = " mindestens zwei aus a, b, $ü_{in}$ "





Einfacher nBit−Addierer (Carry−ripple−Addierer)



Rekursiver Aufbau eines schnellen Carry−lookahead−Addierers