

# Grundlagen der Mathematik

## Inhaltsverzeichnis

<b>1 Formale Sprachen</b>	<b>1</b>
1.1 Objektsprache, Metasprache . . . . .	1
1.2 Formale Sprachen . . . . .	1
1.3 Abstrakte Syntaxbäume . . . . .	1
<b>2 Lambda-Kalkül</b>	<b>2</b>
2.1 Variablensubstitution . . . . .	2
2.2 Lambda-Ausdrücke . . . . .	2
2.3 Auswertung . . . . .	3

## 1 Formale Sprachen

### 1.1 Objektsprache, Metasprache

Hinfort ist es sinnvoll zwischen Objektsprache und Metasprache zu unterscheiden. Die Ausdrücke einer Objektsprache werden als Objekte behandelt über die man spricht. Die Metasprache ist die Sprache in der man über diese Objekte spricht. Zur Unterscheidung von werden Ausdrücke der Objektsprache zunächst in Anführungszeichen gesetzt. Demnach lässt z. B. der folgende metasprachliche Satz formulieren:

Die Aussage »Heute hat es geregnet.« war gestern wahr gewesen.

Natürlich kann es mehrere Ebenen von Metasprachen geben:

Die Aussage »Die Aussage »Heute hat es geregnet.« war gestern wahr gewesen.« wahr gestern wahr gewesen.

### 1.2 Formale Sprachen

Wir können auf ein Blatt Papier oder in den Computer eine Zeichenkette aus Symbolen schreiben. Um genau zu sein, müssen wir uns zunächst überlegen welche Symbole erlaubt sein sollen. Die erlaubten Symbole fassen wir zu einer Menge  $\Sigma$  zusammen und bezeichnen diese als *Alphabet*. Aus den Symbolen lassen sich nun Ketten von Symbolen bilden, die wir als *Wörter* bezeichnen. Ein Wort ist demnach ein Tupel

$$a := (a_1, a_2, \dots, a_n), \quad (a_k \in \Sigma) \quad (1.1)$$

das in vielen Fällen auch kurz als  $a_1 a_2 \dots a_n$  geschrieben wird. Z. B. schreiben wir »Boot« anstelle

von (B, o, o, t). Dazu passend wird die Konkatenation zweier Wörter  $a$  und  $b$  als  $ab$  geschrieben. Also

$$ab := (a_1, \dots, a_m, b_1, \dots, b_n) \quad (1.2)$$

für  $a = (a_1, \dots, a_m)$  und  $b = (b_1, \dots, b_n)$ .

Die Konkatenation von mehr als zwei Wörtern ist ganz klar und lässt sich präzise rekursiv definieren:

$$\text{cat}_{k=1}^n [a^{(k)}] := \text{cat}_{k=1}^{n-1} [a^{(k)}] a^{(n)}. \quad (1.3)$$

Man definiert auch noch das sogenannte leere Wort  $\varepsilon$  als leeres Tupel, kurz  $\varepsilon := ()$ . Das leere Wort ist zugleich der Wert der leeren Konkatenation:

$$\text{cat}_{k=1}^0 [a^{(k)}] := \varepsilon. \quad (1.4)$$

Verschiedene Wörter lassen sich nun zu einer Menge  $L$  zusammenfassen, die man als *Sprache* bezeichnet.

Die Menge aller möglichen Konkatenationen (einschließlich der leeren Konkatenation) von Symbolen aus  $\Sigma$  bezeichnet man als *kleinsche Hülle*  $\Sigma^*$ . Schließt man die leere Konkatenation aus, so spricht man von der *positiven kleinschen Hülle*  $\Sigma^+$ .

Jede Sprache  $L$  über dem Alphabet  $\Sigma$  ist demnach eine Teilmenge  $L \subseteq \Sigma^*$ .

### 1.3 Abstrakte Syntaxbäume

Wenn ein mathematischer Ausdruck syntaktisch analysiert wurde, kann das Ergebnis dieser Analyse als ein sogenannter *abstrakter Syntaxbaum*, kurz *AST* für engl. *abstract syntax tree* dargestellt werden.

Dem Ausdruck »2x« ordnen wir z. B. den AST

$$(*, 2, x)$$

zu und »2x + 4« ist dementsprechend

$$(+, (*, 2, x), 4).$$

Hier ein paar Beispiele:

1.  $f(x)$  wird zu  $(f, x)$ ,
2.  $f(x, y)$  wird zu  $(f, x, y)$ ,
3.  $x^2 + a$  wird zu  $(+, (^, x, 2), a)$ ,
4.  $2(a + b)$  wird zu  $(*, 2, (+, a, b))$ ,
5.  $a + b + c$  wird zu  $(+, (+, a, b), c)$ ,
6.  $a + b + c$  wird zu  $(+, a, b, c)$ ,
7.  $x^2 = x$  wird zu  $(=, (^, x, 2), x)$ ,
8.  $A \wedge B$  wird zu  $(\text{and}, A, B)$ .

Bei der Darstellung von abstrakten Syntaxbäumen im Computer gibt es zunächst zwei Varianten. In Lisp wird der AST zum Ausdruck » $f(x, y)$ « als verkettete Liste

$$(f \ x \ y)$$

dargestellt. Hier kann nämlich mit `first` (auch `car` genannt) der Funktionsbezeichner `f` erhalten werden und mit `rest` (auch `cdr` genannt) die Liste der Argumente.

Bei der Verwendung von dynamischen Feldern anstelle von verketteten Listen sind die Operationen `first` und `rest` eventuell etwas ineffizienter. Die Darstellung ist z. B.:

```
["f", "x", "y"]
```

In einer effizienten statisch typisierten Programmiersprache ließe sich natürlich ein Datentyp für AST-Knoten formulieren, der auf extra die Bedürfnisse, welche sich ergeben, zugeschnitten ist.

## 2 Lambda-Kalkül

### 2.1 Variablensubstitution

Angenommen, bei  $t$  und  $u$  handelt es sich um Ausdrücke bzw. abstrakte Syntaxbäume. Dann bedeutet die Schreibweise

$$t \ [x := u]$$

die Ersetzung jedes Vorkommens der Variable  $x$  im Baum  $t$  durch den Baum  $u$ . Verwendet man keine abstrakten Syntaxbäume, so müssen dabei jedoch im Zusammenhang mit Vorrangregeln die Regeln der Klammersetzung beachtet werden. Z. B. ist

$$(a * b) \ [b := x + y] = a * (x + y) \quad (2.1)$$

und nicht  $a * x + y$ .

Eine solche Variablensubstitution soll außerdem nur für Variablen durchgeführt werden, welche *frei*, d. h. ungebunden vorkommen. Man spricht daher von einer *freien Variablensubstitution*. Variablen können ausschließlich durch lambda-Ausdrücke gebunden werden. Was ein lambda-Ausdruck ist, wird später erklärt.

### 2.2 Lambda-Ausdrücke

Einer Liste wie  $[a, b, c, d]$  ordnen wir den AST

```
([], a, b, c, d)
```

zu. Somit gehört zur einelementigen Liste  $[x]$  der Baum  $([], x)$ . Zur leeren Liste  $[]$  gehört der Baum  $([],)$ , wobei die runden Klammern obligatorisch sind. Der Begriff *Liste* ist gleichbedeutend mit *Tupel*.

Einen AST der Form

```
(lambda, ([], x), t)
```

bzw. einen Ausdruck

$$\lambda([x], t) \quad (2.2)$$

wobei  $t$  ein beliebiger AST ist, wollen wir als lambda-Ausdruck bezeichnen. Anstelle von  $\lambda([x], t)$  schreiben wir kürzer  $|x| \ t$ . Alternative Schreibweisen sind  $\lambda x. t$  oder  $x \mapsto t$ .

Dabei verlangt man nun, dass  $|x|$  schwächer bindet als alle anderen Operationen und rechtsassoziativ ist. Rechtsassoziativ bedeutet, dass die Klammerung immer auf der rechten Seite gesetzt wird. Z. B. ist

$$|x| \ |y| \ x + y = |x| \ (|y| \ (x + y)). \quad (2.3)$$

Ein lambda-Ausdruck lässt sich nun auf einen AST *applizieren*. Man definiert

$$(|x| \ t)(u) := (t \ [x := u]). \quad (2.4)$$

Z. B. ist

$$(|x| \ x^2)(4) = 4^2 = 16 \quad (2.5)$$

und

$$\begin{aligned} &(|x, y| \ x \cdot y)(a + b, c + d) \\ &= (x \cdot y) \ [x := a + b, y := c + d] \\ &= (a + b)(c + d). \end{aligned} \quad (2.6)$$

Applikation ist linksassoziativ. D. h. es gilt

$$f(x)(y) = (f(x))(y). \quad (2.7)$$

Man kann nun die freie Variablensubstitution präzise definieren. Für Ausdrücke  $s, t, u$  und Variablen  $x, y$  gelten folgende Regeln:

- (s1)  $x \ [x := u] = u$ .
- (s2)  $y \ [x := u] = y$  wenn  $x \neq y$ .
- (s3)  $s(t) \ [x := u] = (s[x := u])(t[x := u])$ .
- (s4)  $(|x| \ t) \ [x := u] = |x| \ t$ .
- (s5)  $(|y| \ t) \ [x := u] = |y| \ (t[x := u])$  wenn  $x \neq y$  und  $FV(u)$  disjunkt zu  $BV(|y| \ t)$  ist.

Mit  $FV(t)$  bezeichnet man die Menge der *freien Variablen* von  $t$ , engl. *free variables*. Dieser Operator ist wie folgt definiert:

- 1.  $FV(x) = \{x\}$  für jede Variable  $x$ .
- 2.  $FV(s(t)) = FV(s) \cup FV(t)$ .
- 3.  $FV(|x| \ t) = FV(t) \setminus \{x\}$ .

Mit  $BV(t)$  bezeichnet man die Menge der *gebundenen Variablen* von  $t$ , engl. *bounded variables*. Dieser Operator ist wie folgt definiert:

- 1.  $BV(x) = \{\}$  für jede Variable  $x$ .
- 2.  $BV(s(t)) = BV(s) \cup BV(t)$ .
- 3.  $BV(|x| \ t) = BV(t) \cup \{x\}$ .

Entsprechend können wir noch  $AV(t)$ , die Menge aller Variablen von  $t$  definieren:

1.  $AV(x) = \{x\}$  für jede Variable  $x$ .
2.  $AV(s(t)) = AV(s) \cup AV(t)$ .
3.  $AV(|x| t) = AV(t) \cup \{x\}$ .

Außerdem definieren wir noch  $PV(t)$ , die Menge der in Ausdrücken präsenten Variablen, wie folgt:

1.  $PV(x) = \{x\}$  für jede Variable  $x$ .
2.  $PV(s(t)) = PV(s) \cup PV(t)$ .
3.  $PV(|x| t) = PV(t)$ .

Man beachte dass eine Variable in einem Ausdruck sowohl frei als auch gebunden vorkommen kann. Mit anderen Worten: Eine disjunkte Zerlegung von  $AV(t)$  in  $FV(t)$  und  $BV(t)$  ist nicht immer möglich.

Nun gibt es auch lambda-Ausdrücke mit mehr als einem Argument. Wir können nun

$$(|x, y| t) := (|x| |y| t) \quad (2.8)$$

definieren. Man bezeichnet eine solche Umformung als *Currying* oder *Schönfinkeln*.

Bei mehr als einem Argument muss man in solchen Fällen aufpassen, wo die selben Variablen sowohl frei als auch gebunden vorkommen. Der Ausdruck

$$(|x, y| x + y)(y, x) \quad (2.9)$$

unterscheidet sich z. B. von

$$((x + y)[x := y])[y := x]. \quad (2.10)$$

Hier müssen beide Substitutionen »gleichzeitig« vorgenommen werden. Präziser gesagt müssen die gebundenen Variablen vor der Substitution umbenannt werden. Betrachten wir nämlich den geschönfinkelten Ausdruck

$$(|x| |y| x + y)(y)(x). \quad (2.11)$$

Hier erhält man nach Definition nun

$$((|y|x + y)[x := y])(x), \quad (2.12)$$

aber die Substitution kann wegen Regel (s5) nicht ausgeführt werden, denn  $FV(y) = \{y\}$  ist nicht disjunkt zu  $BV(|y| x + y) = \{y\}$ . Benennt man die gebundene Variable  $y$  nun in  $a$  um, so ergibt sich

$$\begin{aligned} & ((|a| x + a)[x := y])(x) \\ &= (|a| y + a)(x) = y + x. \end{aligned} \quad (2.13)$$

Wir hatten gesagt, Variablen können ausschließlich durch lambda-Ausdrücke gebunden werden. Variablenbindungen ohne lambda-Bindung müssen immer

in solche mit lambda-Bindung umformuliert werden. Z. B. ist

$$\sum_{k=m}^n a_k = \text{sum}(m, n, |k| a_k) \quad (2.14)$$

und

$$\int_a^b f(x) dx = \text{integral}(a, b, |x| f(x)). \quad (2.15)$$

Für die Quantoren der Prädikatenlogik ergibt sich:

$$\forall x \in M [P(x)] = \text{all}(M, |x| P(x)), \quad (2.16)$$

$$\exists x \in M [P(x)] = \text{any}(M, |x| P(x)). \quad (2.17)$$

Das bedeutet beim Allquantor z. B., dass der Baum

$$(\text{all}, (\text{in}, x, M), (P, x))$$

zum Baum

$$(\text{all}, M, (\text{lambda}, ([], x), (P, x)))$$

umformuliert wird.

## 2.3 Auswertung

Bei der Auswertung von lambda-Termen kann es sein, dass wir in einen sogenannten *infiniten Regress* gelangen. Man betrachte z. B.

$$\text{Regress} = (|x| x(x))(|x| x(x)).$$

Die Auswertung führt wieder auf den ursprünglichen Term, wird also niemals enden. Oder man betrachte:

$$\begin{aligned} & (|x| 1+x(x))(|x| 1+x(x)) \\ &= 1+(|x| 1+x(x))(|x| 1+x(x)) \\ &= 1+(1+(|x| 1+x(x))(|x| 1+x(x))) \\ &= \text{usw.} \\ &= 1+(1+(1+(\dots))). \end{aligned}$$

Das Ergebnis dieser Auswertung wäre ein unendlich großer AST. Es ist nun aber so, dass bei einer bestimmten Auswertung ein AST nicht benötigt wird, weil er bei der Auswertung einfach entfällt. Solches kann z. B. bei einer Projektion geschehen. Die Funktion

$$p := |x| |y| x$$

projiziert auf das erste Argument. Daher ist

$$p(0)(\text{Regress}) = 0,$$

obwohl der problematische Term *Regress* vorkommt. Wertet man zuerst *Regress* aus, so gelangt man in den *infiniten Regress*. Wertet man hier aber von links nach rechts aus, so gelangt man zu

$$\begin{aligned}
& (|x| \mid |y| \mid x)(0)(\text{Regress}) \\
& = (|y| \mid 0)(\text{Regress}) = 0.
\end{aligned}$$

Wir wollen hierbei von einer sogenannten *Lazy Evaluation* (dt. *Bedarfsauswertung*) sprechen, die sich von der normalen *Eager Evaluation* unterscheidet.

Normalerweise werden in  $p(x, y)$  die Argumente  $x, y$  auszuwertet und danach wird  $p$  auf die Werte appliziert. Der Ansatz für die alternative Vorgehensweise ist wie folgt: Zuerst wird  $p$  auf die Argumente appliziert, wobei jedes Vorkommen einer Variablen in  $p$  gegen eine Referenz (einen »Zeiger«) auf den entsprechenden AST ausgetauscht wird. Nun wird das Ergebnis weiter ausgewertet, wobei problematische Terme eventuell durch Projektionen verloren gegangen sind.

Ein Problem hierbei ist jedoch, dass sich der Rechenaufwand bei mehrfachem Vorkommen einer Variablen erhöht. Man tut daher folgendes: Bei der ersten Auswertung des ASTs wird dieser für alle Referenzen gegen das Ergebnis der Auswertung ausgetauscht.