Aussagen als Typen

Curry-Howard für Eilige

In der Aussagenlogik sind zusammengesetzte Aussagen aus einem Satz von Grundverknüpfungen konstruierbar, meist UND, ODER, NICHT.

In der Aussagenlogik sind zusammengesetzte Aussagen aus einem Satz von Grundverknüpfungen konstruierbar, meist UND, ODER, NICHT.

Betrachten wir $A \wedge B$.

In der Aussagenlogik sind zusammengesetzte Aussagen aus einem Satz von Grundverknüpfungen konstruierbar, meist UND, ODER, NICHT.

Betrachten wir $A \wedge B$. Wertetabelle:

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

In der Aussagenlogik sind zusammengesetzte Aussagen aus einem Satz von Grundverknüpfungen konstruierbar, meist UND, ODER, NICHT.

Betrachten wir $A \wedge B$. Wertetabelle:

A	B	$A \wedge B$
0	0	0
0	1	0
1	0	0
1	1	1

Betrachten wir A, B als Zahlen und multiplizieren diese, kommt doch dasselbe Ergebnis bei heraus.

Wie verhält es sich dann mit $A \vee B$?

Wie verhält es sich dann mit $A \vee B$? Addieren?

Wie verhält es sich dann mit $A \vee B$? Addieren?

Wertetabelle:

A	B	$A \vee B$	$A + B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	2

Wie verhält es sich dann mit $A \vee B$? Addieren?

Wertetabelle:

A	B	$A \vee B$	$A + B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	2

Wir könnten $A + B - AB$ rechnen, dann stimmt es.

Wie verhält es sich dann mit $A \vee B$? Addieren?

Wertetabelle:

A	B	$A \vee B$	$A + B$
0	0	0	0
0	1	1	1
1	0	1	1
1	1	1	2

Wir könnten $A + B - AB$ rechnen, dann stimmt es.

Aber angenommen, uns würde nur interessieren ob da null herauskommt oder nicht, dann könnten wir uns die Subtraktion von AB sparen.

Wir teilen die natürlichen Zahlen in zwei Äquivalenzklassen auf, einmal 0 für falsch, und die restlichen Zahlen $x \neq 0$ für wahr.

Wir teilen die natürlichen Zahlen in zwei Äquivalenzklassen auf, einmal 0 für falsch, und die restlichen Zahlen $x \neq 0$ für wahr.

Dann klappt es:

A	B	AB	$A+B$
0	0	0	0
0	x	0	x
x	0	0	x
x	x	x	x

(In der Tabelle steht x für die Äquivalenzklasse.)

Eine entsprechende Operation gibt es auch für die Implikation $A \to B$.

Eine entsprechende Operation gibt es auch für die Implikation $A \to B$.

Na?

Eine entsprechende Operation gibt es auch für die Implikation $A \to B$.

Na? Das ist B^A.

Eine entsprechende Operation gibt es auch für die Implikation $A \to B$.

Na? Das ist B^A. Wertetabelle:

A	B	B^A	$A \to B$
0	0	1	1
0	x	1	1
x	0	0	0
x	x	x	1

Eine entsprechende Operation gibt es auch für die Implikation $A \to B$.

Na? Das ist B^A. Wertetabelle:

A	B	B^A	$A \to B$
0	0	1	1
0	x	1	1
x	0	0	0
x	x	x	1

Tolle Spielerei. Und was bringt uns das jetzt?

Aus der Mengenlehre, speziell der Kardinalzahltheorie ist bekannt, dass zu jeder dieser Operationen eine einfache Mengenkonstruktion gehört.

Aus der Mengenlehre, speziell der Kardinalzahltheorie ist bekannt, dass zu jeder dieser Operationen eine einfache Mengenkonstruktion gehört.

Die Entsprechnungen sind:

Aussagen	Zahlen	Mengen
$A \wedge B$	AB	$A \times B$
$A \vee B$	$A + B$	$A \sqcup B$
$A \to B$	B^A	$\mathrm{Abb}(A, B)$

Aus der Mengenlehre, speziell der Kardinalzahltheorie ist bekannt, dass zu jeder dieser Operationen eine einfache Mengenkonstruktion gehört.

Die Entsprechnungen sind:

Aussagen	Zahlen	Mengen
$A \wedge B$	AB	$A \times B$
$A \vee B$	$A + B$	$A \sqcup B$
$A \to B$	B^A	$\mathrm{Abb}(A, B)$

Mit $A \sqcup B$ ist die disjunkte Vereinigung

$$(\{0\} \times A) \cup (\{1\} \times B) \cong (\{\text{Grün}\} \times A) \cup (\{\text{Blau}\} \times B)$$

gemeint. Das heißt, die Elemente werden vorher eingefärbt, damit bekannt bleibt, aus welcher der Mengen ein Element der Vereinigung entstammt.

Aus der Mengenlehre, speziell der Kardinalzahltheorie ist bekannt, dass zu jeder dieser Operationen eine einfache Mengenkonstruktion gehört.

Die Entsprechnungen sind:

Aussagen	Zahlen	Mengen
$A \wedge B$	AB	$A \times B$
$A \vee B$	$A + B$	$A \sqcup B$
$A \to B$	B^A	$\mathrm{Abb}(A, B)$

Mit $A \sqcup B$ ist die disjunkte Vereinigung

$$(\{0\} \times A) \cup (\{1\} \times B) \cong (\{\text{Grün}\} \times A) \cup (\{\text{Blau}\} \times B)$$

gemeint. Das heißt, die Elemente werden vorher eingefärbt, damit bekannt bleibt, aus welcher der Mengen ein Element der Vereinigung entstammt.

Mit $\mathrm{Abb}(A, B)$ ist die Menge der Abbildungen von A nach B gemeint.

Eine Menge kann die leere Menge $\emptyset$ sein, oder eine nichtleere Menge. Wir haben $|\emptyset| = 0$ und $|X| = x \neq 0$ für $X \neq \emptyset$.

Eine Menge kann die leere Menge $\emptyset$ sein, oder eine nichtleere Menge. Wir haben $|\emptyset| = 0$ und $|X| = x \neq 0$ für $X \neq \emptyset$.

Das ist genau das, was wir haben wollen, denn

$$|A \times B| = |A| \cdot |B|,$$
$$|A \sqcup B| = |A| + |B|,$$
$$|\operatorname{Abb}(A, B)| = |B|^{|A|}.$$

Für die Allquantifizierung $\forall_{x \in M} P(x)$ brauchen wir unbedingt auch eine Entsprechung, wie später klar werden wird.

Für die Allquantifizierung $\forall_{x \in M} P(x)$ brauchen wir unbedingt auch eine Entsprechung, wie später klar werden wird.

Sei M zunächst endlich, sagen wir

$$M := \{x_0, x_1, x_2, x_3\}.$$

Dann gilt

$$\forall_{x \in M} P(x) = P(x_0) \wedge P(x_1) \wedge P(x_2) \wedge P(x_3).$$

Für die Allquantifizierung $\forall_{x \in M} P(x)$ brauchen wir unbedingt auch eine Entsprechung, wie später klar werden wird.

Sei M zunächst endlich, sagen wir

$$M := \{x_0, x_1, x_2, x_3\}.$$

Dann gilt

$$\forall_{x \in M} P(x) = P(x_0) \wedge P(x_1) \wedge P(x_2) \wedge P(x_3).$$

Die Entsprechung ist

$$P(x_0) \times P(x_1) \times P(x_2) \times P(x_3),$$

wobei $P \colon M \to \{0, 1\}$ gegen $P' \colon M \to U$ zu ersetzen ist, so dass

$$P(x) \iff |P'(x)| \neq \emptyset.$$

Die Elemente dieses Produkts sind Tupel

$$(y_0, y_1, y_2, y_3).$$

Die Elemente dieses Produkts sind Tupel

$$(y_0, y_1, y_2, y_3).$$

Ein solches Tupel können wir auch als Abbildung

$$\{0 \mapsto y_0, 1 \mapsto y_1, 2 \mapsto y_2, 3 \mapsto y_3\}$$

betrachten. Das ist schlicht die Indizierung des Tupels.

Die Elemente dieses Produkts sind Tupel

$$(y_0, y_1, y_2, y_3).$$

Ein solches Tupel können wir auch als Abbildung

$$\{0 \mapsto y_0, 1 \mapsto y_1, 2 \mapsto y_2, 3 \mapsto y_3\}$$

betrachten. Das ist schlicht die Indizierung des Tupels.

Die Menge M enthält nun jedes Element genau einmal. Dann können wir doch auch M selbst als Indexmenge verwenden, richtig? Das Tupel ist demnach kodierbar als

$$\{x_0 \mapsto y_0, x_1 \mapsto y_1, x_2 \mapsto y_2, x_3 \mapsto y_3\}.$$

Das sind Abbildungen $x \mapsto y(x)$, wobei $x \in M$ und $y(x) \in P'(x)$.

Das sind Abbildungen $x \mapsto y(x)$, wobei $x \in M$ und $y(x) \in P'(x)$.

So eine Konstruktion gibt es in der Mathematik tatsächlich. Es handelt sich um das allgemeine Mengenprodukt

$$\prod_{x \in M} P'(x) := \{f : M \to \bigcup_{x \in M} P'(x) \mid \forall x \in M : f(x) \in P'(x)\}.$$

Aber was bringt uns das denn jetzt?

Aber was bringt uns das denn jetzt?

Betrachten wir die aussagenlogische Formel

$$A \wedge B \rightarrow A.$$

Wir wollen beweisen dass das eine Tautologie ist.

Aber was bringt uns das denn jetzt?

Betrachten wir die aussagenlogische Formel

$$A \wedge B \rightarrow A.$$

Wir wollen beweisen dass das eine Tautologie ist.

Es handelt sich genau dann um eine Tautologie, wenn

$$\forall_A \forall_B (A \wedge B \rightarrow A)$$

eine wahre Aussage ist.

Aber was bringt uns das denn jetzt?

Betrachten wir die aussagenlogische Formel

$$A \wedge B \to A.$$

Wir wollen beweisen dass das eine Tautologie ist.

Es handelt sich genau dann um eine Tautologie, wenn

$$\forall_A \forall_B (A \wedge B \to A)$$

eine wahre Aussage ist. Das ist wiederum genau dann der Fall, wenn die entsprechende Menge

$$\Pi_A \Pi_B (A \times B \to A)$$

nichtleer ist, wobei wir $A \times B \to A$ für $\mathrm{Abb}(A \times B, A)$ schreiben.

Ein Element dieser Menge ist konstruierbar – die Projektion auf das linke Element, das ist

$$\text{proj}_0 := A \mapsto B \mapsto (a, b) \mapsto a \text{ mit } (a, b) \in A \times B,$$

bzw. anders geschrieben

$$\text{proj}_0(A)(B)((a, b)) := a.$$

Ein Element dieser Menge ist konstruierbar – die Projektion auf das linke Element, das ist

$$\text{proj}_0 := A \mapsto B \mapsto (a, b) \mapsto a \text{ mit } (a, b) \in A \times B,$$

bzw. anders geschrieben

$$\text{proj}_0(A)(B)((a, b)) := a.$$

Es gibt daher mindestens dieses Element proj_0. Die Menge ist somit nichtleer, die ursprüngliche Formel daher bewiesen.

Irgendwie erinnert das an ein Programm, wie man es in einer Programmiersprache konstruieren kann.

Irgendwie erinnert das an ein Programm, wie man es in einer Programmiersprache konstruieren kann.

Standard ML

```
val proj0: 'a * 'b -> 'a = fn (a, b) => a;
```

Alternativ:

```
fun proj0((a, b): 'a*'b): 'a = a;
```

Irgendwie erinnert das an ein Programm, wie man es in einer Programmiersprache konstruieren kann.

Standard ML

```
val proj0: 'a * 'b -> 'a = fn (a, b) => a;
```

Alternativ:

```
fun proj0((a, b): 'a*'b): 'a = a;
```

Rust

```
fn proj0<A, B>((a, _b): (A, B)) -> A {a}
```

Das heißt, den Mengen entsprechen Typen? Ein Typsystem hilft uns dabei, korrekte Terme zu konstruieren. Dann hilft es uns auch, korrekte Beweise zu konstruieren?

Das heißt, den Mengen entsprechen Typen? Ein Typsystem hilft uns dabei, korrekte Terme zu konstruieren. Dann hilft es uns auch, korrekte Beweise zu konstruieren?

Ja, das geht. Die genannten Programmiersprachen schaffen das aber nur teilweise. Wir wollen ja Funktionen im mathematischen Sinn haben.

Das heißt, den Mengen entsprechen Typen? Ein Typsystem hilft uns dabei, korrekte Terme zu konstruieren. Dann hilft es uns auch, korrekte Beweise zu konstruieren?

Ja, das geht. Die genannten Programmiersprachen schaffen das aber nur teilweise. Wir wollen ja Funktionen im mathematischen Sinn haben.

Das heißt:

- Die Funktion muss einem Argument immer denselben Wert zuordnen, denn eine mathematische Funktion ordnet einem Element *genau einen* Wert zu. Eine solche Funktion nennt man *rein*, – *rein funktional*.
- Die Funktion muss immer terminieren und damit einen Wert liefern, denn eine mathematische Funktion ordnet *jedem* Element einen Wert zu. Eine solche Funktion nennt man *total*.

Das heißt, den Mengen entsprechen Typen? Ein Typsystem hilft uns dabei, korrekte Terme zu konstruieren. Dann hilft es uns auch, korrekte Beweise zu konstruieren?

Ja, das geht. Die genannten Programmiersprachen schaffen das aber nur teilweise. Wir wollen ja Funktionen im mathematischen Sinn haben.

Das heißt:

- Die Funktion muss einem Argument immer denselben Wert zuordnen, denn eine mathematische Funktion ordnet einem Element *genau einen* Wert zu. Eine solche Funktion nennt man *rein*, – *rein funktional*.
- Die Funktion muss immer terminieren und damit einen Wert liefern, denn eine mathematische Funktion ordnet *jedem* Element einen Wert zu. Eine solche Funktion nennt man *total*.

Würden wir partielle Funktionen zulassen, könnte man zu jedem Typ als Term ein Programm einfügen, welches sich in einer Endlosschleife verfängt, und hätte damit sofort einen »Beweis« für jede beliebige Aussage.

Zudem muss das Typsystem reichhaltig genug sein, um die gewünschten Mengenkonstruktionen überhaupt ausdrücken zu können.

Zudem muss das Typsystem reichhaltig genug sein, um die gewünschten Mengenkonstruktionen überhaupt ausdrücken zu können.

Das Typsystem muss erlauben:

- algebraische Typen,
- abhängige Typen.

Zudem muss das Typsystem reichhaltig genug sein, um die gewünschten Mengenkonstruktionen überhaupt ausdrücken zu können.

Das Typsystem muss erlauben:

- algebraische Typen,
- abhängige Typen.

Abhängige Typen sind eine fortgeschrittene Form von parametrischer Polymorphie, bei denen Typen nicht nur über Typen parametrisiert werden können, sondern auch über Werte.

Das heißt, wenn wir so eine pedantische Typprüfung bauen würden, könnten wir. . . ?

Das heißt, wenn wir so eine pedantische Typprüfung bauen würden, könnten wir. . . ?

Solche Systeme wurden bereits vor langer Zeit geschaffen. Der Theorembeweisassistent Coq enthält eine Gallina genannte Sprache, die zentraler Gegenstand des Programmkerns zur Verfikiation der konstruierten Beweise ist.

Das heißt, wenn wir so eine pedantische Typprüfung bauen würden, könnten wir. . . ?

Solche Systeme wurden bereits vor langer Zeit geschaffen. Der Theorembeweisassistent Coq enthält eine Gallina genannte Sprache, die zentraler Gegenstand des Programmkerns zur Verfikiation der konstruierten Beweise ist.

Gallina

```
Definition proj0: forall(A B: Type), A*B -> A :=
  fun (A B: Type) (t: A*B) =>
    match t with (a, b) => a end.
```

Zwischen Aussagen und Typen wird eine enge Entsprechung erkennbar, die im *Curry-Howard-Isomorphismus* ihre Präzisierung findet.

Zwischen Aussagen und Typen wird eine enge Entsprechung erkennbar, die im *Curry-Howard-Isomorphismus* ihre Präzisierung findet.

Aussagen	**Typen**
Konjunktion — $A \wedge B$	$A \times B$ — Produkt
Disjunktion — $A \vee B$	$A + B$ — Summe
Implikation — $A \to B$	$A \to B$ — Funktionentyp
Allquant. — $\forall_{x \in M} P(x)$	$\Pi_{x:M} P(x)$ — abhängige Summe
Existenzquant. — $\exists_{x \in M} P(x)$	$\Sigma_{x:M} P(x)$ — abhängiges Produkt

Zwischen Aussagen und Typen wird eine enge Entsprechung erkennbar, die im *Curry-Howard-Isomorphismus* ihre Präzisierung findet.

Aussagen	**Typen**
Konjunktion — $A \wedge B$	$A \times B$ — Produkt
Disjunktion — $A \vee B$	$A + B$ — Summe
Implikation — $A \to B$	$A \to B$ — Funktionentyp
Allquant. — $\forall_{x \in M} P(x)$	$\Pi_{x:M} P(x)$ — abhängige Summe
Existenzquant. — $\exists_{x \in M} P(x)$	$\Sigma_{x:M} P(x)$ — abhängiges Produkt

Zum Beweis einer Aussage konstruiert man zum entsprechenden Typ einen Term, dessen Wert von diesem Typ ist. Der Wert, den wir *Zeuge* nennen, belegt dass der Typ durch mindestens einen Wert *bewohnt* ist. Dies erbringt den Beweis der ursprünglichen Aussage.

Ausblick

Ausgelassen wurden die folgenden Themen:

- Summentypen und Pattern-Matching (Fallunterscheidung).
- Die Negation $\neg A$ muss als $A \to 0$ kodiert werden.
- Wie verhält es sich mit der Gleichheit?
- Man erhält auf diese Weise intuitionistische Logik. Ist auch klassische Aussagenlogik möglich?

Literatur

- Stefan Müller-Stach: »Äquivalenz und Wahrheit«, Kapitel 9: »Typentheorie und ihre Semantik«.
- Benjamin C. Pierce et al.: »Software Foundations. Volume 1: Logical Foundations«.

Ende.