Funktionen und Relationen

Hans Jürgen Ohlbach

Keywords: Funktionen: Typen, Definitions- und Wertebereich, Komposition, Eigenschaften, Isomorphien, Berechenbarkeit, Visualisierungen; Relationen, Prädikate: Konstruktionen, Eigenschaften, Ordnungsrelationen, Äquivalenzrelationen, Visualisierungen, Datenstrukturen.

Inhaltsverzeichnis

1 Einführung

Funktionen sind Abbildungen[1], die ein oder mehrere Objekte auf ein weiteres Objekt abbilden. Aus der Schule kennt man in erster Linie Funktionen, die Zahlen auf Zahlen abbilden. Die Additionsfunktion, z.B. bildet zwei reelle Zahlen auf die Summe der beiden Zahlen ab. Der Funktionsbegriff ist aber nicht nur auf reelle Zahlen beschränkt. Eine Funktion *AnzahlPrimfaktoren*(n), z.B. berechnet für eine *natürliche Zahl* n die Anzahl ihrer Primfaktoren, ebenfalls eine natürliche Zahl. Es ist z.B. *AnzahlPrimfaktoren*$(6) = 2$ (weil $6 = 2 \cdot 3$). Es gibt natürlich auch Funktionen, die auf ganz anderen Objekten als Zahlen operieren. Als Beispiel bildet die Funktion $VaterVon(x)$ einen Menschen auf dessen Vater ab, z.B. $VaterVon(AlbertEinstein) = HermannEinstein$. Gerade in der Informatik spielen Funktionen, die nicht auf Zahlen operieren eine viel größere Rolle, als Funktionen, die auf Zahlen operieren.

[1] Die Begriffe *Funktion* und *Abbildung* bedeuten i.A. das gleiche.

Relationen dagegen sind Beziehungen zwischen Objekten. Auch hier kennt man aus der Schule in erster Linie Relationen zwischen Zahlen. Die Kleiner-Relation ($<$) auf Zahlen, z.B. testet, ob eine Zahl kleiner ist als eine andere. $3 < 5$ ist wahr, $5 < 3$ ist falsch. Man kann allerdings auch eine Relation als Funktion auffassen, die Objekte auf $wahr$ oder $falsch$ abbildet. $3 < 5 = wahr$ und $5 < 3 = falsch$.

Aber natürlich gibt es auch Relationen zwischen ganz anderen Objekten als Zahlen. Anstelle der Funktion $VaterVon$ könnte man z.B. die Relation $istVaterVon$ definieren. Dann wäre
$istVaterVon(HermannEinstein, AlbertEinstein)$ wahr und
$istVaterVon(AlbertEinstein, HermannEinstein)$ falsch.

Welche engen Beziehungen zwischen Funktionen und Relationen bestehen, wird in den folgenden Kapiteln noch genauer besprochen.

In den meisten Programmiersprachen wird in der Tat nicht zwischen Funktionen und Relationen unterschieden. Relationen werden als Funktionen definiert, mit dem Ergebnistyp $boolean$, welcher für $wahr$ oder $falsch$ steht.

Wenn in mathematischen Texten, und im Folgenden auch in diesem Text, von Funktionen und Relationen die Rede ist, dann sind damit wirklich Funktionen und Relationen gemeint. Das ist im Gegensatz zu informatorischen Texten, wo oft zwischen *Syntax* und *Semantik* unterschieden werden muss. Ein Computer verarbeitet zunächst nur Zeichenketten. Für ihn besteht ein großer Unterschied zwischen, z.B. $plus(3, 4)$ und $3 + 4$, während Menschen i.A. in beiden Fällen darunter die Additionsfunktion verstehen. $plus$ und $+$ sind dabei verschiedene syntaktische Varianten derselben Additionsfunktion. Man sagt dann, die Additionsfunktion ist die *Semantik* der *Funktionssymbole* $plus$ und $+$. Ein mathematischer Text besteht natürlich auch nur aus Zeichenketten. Daher muss man eine syntaktische Darstellung benutzen. Wenn es nicht explizit gesagt wird, soll aber in diesem Text damit automatisch deren Semantik gemeint sein. Das äußert sich u.a. darin, dass wir von *Funktionen* und *Relationen* sprechen, und nicht von *Funktionssymbolen* und *Relationssymbolen*.

2 Funktionen

Funktionen bilden eine Anzahl von Objekten, die *Argumente* auf ein Objekt, den Funktionswert, ab. Die Anzahl von Argumenten ist die *Stelligkeit* (engl. *arity*) der Funktion. Die Additionsfunktion, z.B. hat die Stelligkeit 2, die $VaterVon$-Funktion hat die Stelligkeit 1 ($VaterVon(AlbertEinstein) = HermannEinstein$). Manchmal erlaubt man auch 0-Stellige Funktionen. Diese hängen von keinem Argument ab, d.h. sie sind Konstanten. Ein Beispiel ist die Zahl π.

2.1 Argumenttypen, Werttypen

Die allermeisten Funktionen sind nur für bestimmte *Argumenttypen* (engl. *domain type*, oder kurz *domain*) definiert, und berechnen auch nur *Werte* von bestimmten *Typen* (engl. *range type*, oder kurz *range*). Die Additionsfunktion als Beispiel ist nur für Zahlen definiert, und berechnet auch nur Zahlen.

$AlbertEinstein + HermannEinstein$ macht keinen Sinn, und ist daher undefiniert. Argument- und Werttypen sind wichtige Informationen über eine Funktion, die man meist wie in folgendem Beispiel notiert:

$$+ : \mathbb{R}, \mathbb{R} \to \mathbb{R}$$

oder alternativ:

$$+ : \mathbb{R} \times \mathbb{R} \to \mathbb{R}$$

($\mathbb{R}$ steht für die reellen Zahlen.
$\mathbb{R} \times \mathbb{R}$ ist das *Kreuzprodukt* der reellen Zahlen, d.h. die Menge $\{(x, y) \mid x, y \in \mathbb{R}\}$).

In getypten Programmiersprachen, wie z.B. Java, müssen Argument- und Werttypen von neu definierten Funktionen immer angegeben werden. Dann kann der Compiler schon feststellen, ob ihre Verwendung typgerecht ist oder nicht. (In manchen Sprachen hat der Compiler einen sog. Typinferenzmechanismus. Das erspart dem Programmierer, Domain- und Rangetypen, die aus anderen Informationen ableitbar sind, einzugeben.)

Ist eine Funktion f für eine Domain T definiert, und K ist eine Teilmenge von T (d.h. $K \subseteq T$), dann kann man auch die Einschränkung der Funktion auf K definieren:

$$f_{|K}(x) = \begin{cases} f(x) & \text{falls } x \in K \\ \text{undefiniert} & \text{sonst} \end{cases}$$

Das ist natürlich auch für mehrstellige Funktionen möglich. Z.B. ist $+_{|\mathbb{N}}$ die Additionsfunktion, eingeschränkt auf die natürlichen Zahlen $\mathbb{N}$.

2.2 Totale und Partielle Funktionen, Definitions- und Wertebereich

Funktionen, die für *jeden* Wert ihrer Domain einen Funktionswert liefern, nennt man *totale Funktionen*. Ein Beispiel, bei dem das nicht der Fall ist, ist die Divisionsfunktion auf reellen Zahlen. Die liefert für alle Paare von reellen Zahlen einen Wert, ausgenommen wenn der Divisor $= 0$ ist. Dort ist die Division nicht definiert. Eine Funktion, die nicht für alle Werte ihrer Domain einen Funktionswert berechnet, nennt man daher eine *partielle Funktion*.

Die Teilmenge von Argumenten, für die die Funktionen einen Wert berechnet, nennt man den *Definitionsbereich*. Z.B. ist der Definitionsbereich der Divisionsfunktion: $\mathbb{R} \times \mathbb{R} \setminus \{0\}$, das ist die Menge der Paare von reellen Zahlen, außer den Paaren, wo das zweite Element 0 ist.

Bei totalen Funktionen ist der Definitionsbereich immer der gesamte Argumenttyp.

Formal kann man den Definitionsbereich einer n-stelligen Funktion f definieren:

$$\textit{Definitionsbereich}(f) = \{(x_1, \ldots, x_n) | \exists y : \ f(x_1, \ldots, x_n) = y\}$$

Für partielle Funktionen kann man immer ihre Einschränkung auf die Werte des Definitionsbereichs definieren, auf die sie einen Funktionswert berechnet. Die so eingeschränkte Funktion ist dann immer eine totale Funktion.

In Programmiersprachen äußert sich *Partialität* entweder in Form von Fehlermeldungen (Bsp.: „Division durch 0 ist nicht erlaubt“), oder das Programm gerät für die undefinierten Werte in eine Endlosschleife.

Funktionen müssen nicht für alle Elemente des Wertetyps tatsächlich einen Funktionswert liefern. Die trigonometrische Sinusfunktion, z.B. bildet reelle Zahlen auf reelle Zahlen ab, d.h. ihr Typ ist $\sin : \mathbb{R} \to \mathbb{R}$. Konkret werden aber nur Werte aus dem Intervall $[-1, +1]$ berechnet. Der *Wertebereich* dieser Funktion ist daher das Intervall $[-1, +1]$.

Formal kann man den Wertebereich einer n-stelligen Funktion f definieren:

$$\textit{Wertebereich}(f) = \{y \mid \exists x_1, \ldots, x_n : \ f(x_1, \ldots, x_n) = y\}$$

Es ist die Menge aller Elemente des Wertetyps, die Funktionswerte irgendwelcher Argumente sind.

2.3 Komposition von Funktionen

Die Werte von Funktionsaufrufen können natürlich wieder Argumente von weiteren Funktionen sein. D.h. eine Funktion $f_1(x_1, \ldots, x_n)$ kann einen Wert y_1 liefern, und y_1 kann wiederum Argument einer zweiten Funktion $f_2(y_1, z_1 \ldots, z_k)$ sein.

In logischer Schreibweise schreibt man das als *Term*: $f_2(f_1(x_1, \ldots, x_n), z_1 \ldots, z_k)$.

Beispiel:
$KinderVon(\, VaterVon(AlbertEinstein), MutterVon(AlbertEinstein))$ bezeichnet $AlbertEinstein$ zusammen mit seinen Geschwistern.

Sind alle beteiligten Funktionen einstellig, dann lassen sich Ketten von Funktionsaufrufen bilden: $f_n(f_{n-1}(\ldots f_1(x)\ldots)$. Hierfür hat sich eine Kurzschreibweise eingebürgert, um für solche Ketten von Funktionsaufrufen einen neuen Namen definieren zu können: $f = f_1 \circ \ldots \circ f_n$. Mit dieser Definition steht $f(x)$ für $f_n(f_{n-1}(\ldots f_1(x)\ldots)$. f ist dann die *Funktionskomposition* der Funktionen $f_1, \ldots, f_n$.

Eine Funktionskomposition ist natürlich nur dann möglich wenn der Rangetyp von f_i mit dem Domaintyp von f_{i+1} übereinstimmt, oder wenigstens eine Teilmenge davon ist.

Die Komposition $VaterVon \circ$ *AnzahlPrimfaktoren* würde z.B. keinen Sinn machen.

2.4 Kommutativität, Assoziativität, Distributivität

Bei vielen zweistelligen Funktionen, auch Verknüpfungen genannt, ist die Reihenfolge der Argumente in der Tat egal. Z.B. ist $2{+}3 = 3{+}2$ oder allgemein $\forall x, y : \ x{+}y = y{+}x$. Bei der Subtraktionsfunktion ist das nicht so: $2 - 3 \neq 3 - 2$. Funktionen, bei denen die Reihenfolge der Argumente egal ist, bezeichnet man als *kommutativ*. Formal:

$$\textit{kommutativ}(f) \quad gdw. \quad \forall x, y : \ f(x, y) = f(y, x).$$

Eine weitere Eigenschaft von einigen Verknüpfungen ist, dass die Klammerung egal ist.
Z.B. $2 + (3 + 4) = (2 + 3) + 4$ oder allgemein: $\forall x, y, z : \ x + (y + z) = (x + y) + z$.
Solche Funktionen nennt man *assoziativ*. Formal:

$$\textit{assoziativ}(f) \quad gdw. \quad \forall x, y, z : \ f(x, f(y, z)) = f(f(x, y), z).$$

Assoziative Funktionen lassen sich realisieren, indem man als Argumente eine beliebig lange Liste zulässt. Z.B. anstelle von $a_1 + (a_2 + (a_3 + \ldots + a_n \ldots)))$ berechnet man $+(a_1, \ldots, a_n) = \Sigma_{i=1}^{n} a_i$. Ist die Funktion zusätzlich noch kommutativ (wie z.B. die Additionsfunktion), dann ist sogar die Reihenfolge der Elemente in der Liste egal.

Distributivität ist eine Eigenschaft, die zwei Verknüpfungen betrifft. In der Arithmetik haben Addition und Multiplikation diese Eigenschaft: $\forall x, y, z : \ x \cdot (y + z) = x \cdot y + x \cdot z$.

Formal gilt:

$$\textit{distributiv}(f, g) \quad gdw. \quad \forall x, y, z : \ f(x, g(y, z)) = g(f(x, y), f(x, z)).$$

Die Richtung $x \cdot (y + z) \mapsto x \cdot y + x \cdot z$ bezeichnet man üblicherweise als *Ausmultiplizieren*, während man die Rückrichtung $x \cdot y + x \cdot z \mapsto x \cdot (y + z)$ als Ausklammern bezeichnet.

Distributivität hilft oft, wenn man Terme auf eine Normalform bringen will. Zum Beispiel kann man mittels distributiver Umformungen arithmetische Terme als Summe von Produkten umschreiben.

2.5 Injektivität

Eine Funktion, die *nicht* zwei oder mehr Argumentwerte auf denselben Funktionswert abbildet, nennt man *injektiv*. Formal:

$$\textit{injektiv}(f) \quad \text{gdw.} \quad \forall x, y : \ f(x) = f(y) \Rightarrow x = y \quad \text{gdw.} \quad \forall x, y : \ x \neq y \Rightarrow f(x) \neq f(y)$$

Die einfachste injektive Funktion ist die Nachfolgefunktion für natürliche Zahlen. Sie bildet eine Zahl n auf $n + 1$ ab. Ein einfaches Gegenbeispiel wäre die $VaterVon$-Funktion. Sie ist nicht injektiv, denn zwei verschiedene Menschen können natürlich denselben Vater haben.

Bei mehrstelligen Funktionen kann man auch Injektivität in bestimmten Argumenten definieren. Gilt z.B. $\forall x, y, z : \ f(x, z) = f(y, z) \Rightarrow x = y$ für die zweistellig Funktion f, dann ist f injektiv im ersten Argument. Die Additionsfunktion ist injektiv im ersten und zweiten Argument. Es gilt ja: $x + z = y + z \Rightarrow x = y$, sowie $z + x = z + y \Rightarrow x = y$.

In der Informatik sind injektive Funktionen absolut notwendig wenn man Objekten eindeutige Identifikatoren geben muss. Für eine Studentendatenbank, z.B. ist es notwendig, die Studenten eindeutig zu identifizieren. Der Name ist i.A. nicht eindeutig. Oft gibt es z.B. eine ganze Reihe von Michael Schmidts. *Name(Student)* wäre dann *keine* injektive Funktion. Stattdessen benutzt man die Funktion *Matrikelnummer(Student)*, die man injektiv macht, indem man jedem Studenten eine eindeutige Matrikelnummer gibt. Ein anderes Beispiel wäre *Kennzeichen(Auto)*, wo auch sicher gestellt wird, dass nicht zwei Autos das gleiche Kennzeichen haben.

2.6 Surjektivität

Die bekannte Sinusfunktion aus der Trigonometrie bildet reelle Zahlen auf reelle Zahlen ab. Allerdings werden nur Zahlen im Intervall [-1,1] berechnet. Es gibt z.B. keine Zahl x, so dass $\sin(x) = 2$ ist. Anders ist es bei der Tangensfunktion. Für jede reelle Zahl y gibt es eine Zahl x, so dass $y = \tan(x)$ ist (tatsächlich gibt es unendlich viele solche Zahlen x). Die Tangensfunktion schöpft den gesamten Wertebereich der reellen Zahlen aus, während die Sinusfunktion das nicht tut. Daher bezeichnet man die Tangensfunktion als *surjektiv*, nicht aber die Sinusfunktion.
Formal: Für eine Funktion $f : D \to R$:

$$\textit{surjektiv}(f) \quad gdw. \quad \forall y \in R\, \exists x \in D :\ f(x) = y.$$

Beispiele: Die $VaterVon$-Funktion ist *nicht* surjektiv, denn nicht jeder Mensch ist ein Vater.
Die $KindVon$-Funktion, dagegen, ist surjektiv, denn jeder Mensch ist ein Kind von einem anderen Menschen (das wird sich ändern, wenn es die ersten geklonten Menschen gibt).

Manchmal werden auch die Begriffe *linkstotal* und *rechtstotal* verwendet. Linkstotal ist das, was wir als *total* bezeichnet haben, d.h. jedes Element der Domain hat einen Funktionswert. Rechtstotal ist synonym für surjektiv, d.h. jedes Element der Range ist Funktionswert eines Domainelements.

2.7 Bijektivität

Eine einstellige Funktion, die gleichzeitig injektiv und surjektiv ist, nennt man *bijektiv*, oder *Bijektion*. Bijektive Funktionen definieren eine eines-zu-eins Beziehung zwischen Domain und Range. Die Funktion $TanzpartnerVon$ in einer Tanzschule wäre dann bijektiv, wenn es gleich viele Männer wie Frauen gäbe, und bei Damenwahl niemand auf der Bank sitzen bleibt. Dann ist jedem Mann eine eindeutige Tanzpartnerin zugeordnet, und umgekehrt jeder Frau ein eindeutiger Tanzpartner.

Sind Domain und Range endlich, wie in der Tanzschule, dann kann es nur dann bijektive Funktionen geben, wenn Domain und Range gleich viele Elemente haben. Das ist anders wenn Domain und Range unendlich sind. Die Funktion *verdoppeln*$(x) = 2 \cdot x$ bildet jede natürliche Zahl in $\mathbb{N}$ auf eine gerade Zahl in $2\mathbb{N}$ ab. Als Funktion *verdoppeln* : $\mathbb{N} \to 2\mathbb{N}$ ist sie bijektiv, denn jede natürliche Zahl wird auf ein eindeutiges Doppel abgebildet (Injektivität), und jede gerade Zahl ist das Doppelte einer natürlichen Zahl (Surjektivität).

Obwohl die geraden Zahlen eine echte Teilmenge der natürlichen Zahlen sind, betrachtet man, wegen der eindeutigen Zuordenbarkeit, beide Mengen als gleichmächtig. Das ist eine der vielen Merkwürdigkeiten der unendlichen Mengen.

2.8 Isomorphien

Jede bijektive Funktion zwischen zwei Mengen definiert eine *Isomorphie* (Strukturgleichheit) zwischen den beiden Mengen. Im obigen Tanzschulenbeispiel ist es die Funktion $TanzpartnerVon$, die eine Isomorphie zwischen den beteiligten Männern und Frauen definiert. Diese Art von Isomorphie

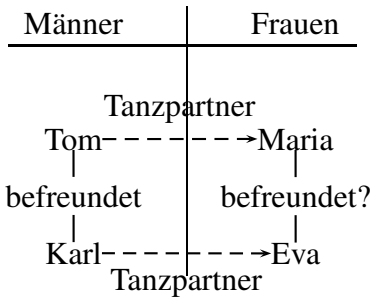
ist allerdings sehr grob. Sie berücksichtigt keinerlei Eigenschaften der Menschen und auch keine Beziehungen zwischen ihnen.

Man kann die Isomorphie verfeinern, indem man Eigenschaften der Elemente der beteiligten Mengen berücksichtigt. Im Tanzschulenbeispiel könnten man z.B. nur Paare gleicher Größe haben wollen. Dazu bräuchte man die Funktion $\textit{GrößeVon} : \textit{Mensch} \to \mathbb{R}$ und die Bedingung:

$$\forall x \in \textit{Tänzer} : \ \textit{GrößeVon}(x) = \textit{GrößeVon}(\textit{TanzpartnerVon}(x)).$$

Wenn diese Bedingung erfüllbar ist, dann hätte man eine verfeinerte Isomorphie zwischen den Männern und Frauen in der Tanzschule. Natürlich kann man das auf weitere Eigenschaften der Tänzer ausweiten, z.B. Haarfarbe, Bildung usw. Partnervermittlungen versuchen verzweifelt, solche Isomorphien herzustellen. Es funktioniert aber i.A. nur sehr eingeschränkt.

Eine andere Art der Verfeinerung einer Isomorphie berücksichtigt Beziehungen zwischen den Elementen der beiden beteiligten Mengen. Man könnte im Tanzschulenbeispiel vielleicht haben wollen, dass jeweils befreundete Männer und Frauen auch Tanzpartnern sind. Wenn z.B. Tom und Maria Tanzpartner sind, und Karl mit Tom befreundet ist, dann möchte man vielleicht, dass auch die Tanzpartnerin von Karl, nämlich Eva, mit Maria befreundet ist.



Man bräuchte dazu eine Relation $\textit{befreundetMit} : \textit{Mensch} \times \textit{Mensch}$, wo wir, um das Beispiel einfacher zu machen, annehmen, dass nur Männer mit Männern befreundet sind, und Frauen mit Frauen. Die Zusatzbedingung für die Isomorphie wäre dann:

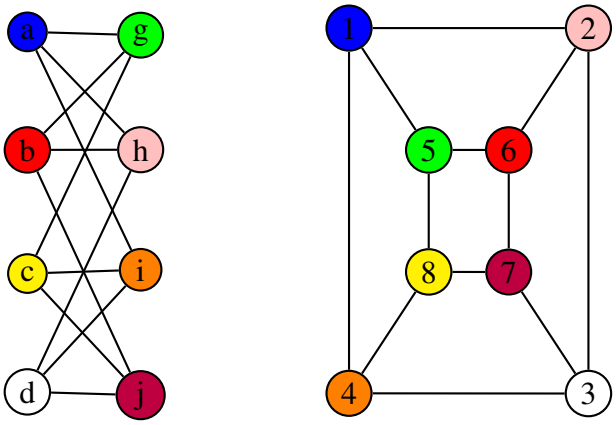
$$\begin{aligned} \forall x, y, z : \ & y = \textit{TanzpartnerVon}(x) \wedge \textit{befreundetMit}(x, z) \\ \Rightarrow \exists v : \ & v = \textit{TanzpartnerVon}(z) \wedge \textit{befreundetMit}(y, v) \end{aligned}$$

Solche Bedingungen beschreiben genau *Isomorphie zwischen Graphen*. Ein Graph besteht aus Knoten und einer Nachbarschaftsbeziehung zwischen den Knoten. Im obigen Tanzschulenbeispiel gäbe es zwei Graphen, die Männer mit der $\textit{befreundetMit}$-Relation, sowie die Frauen, ebenfalls mit der $\textit{befreundetMit}$-Relation als Nachbarschaftsbeziehung.

Zwei Graphen G_1 und G_2 sind dann isomorph wenn es eine bijektive Abbildung zwischen den Knoten der beiden Graphen gibt, die zusätzlich garantiert, dass benachbarte Knoten in G_1 auch auf benachbarte Knoten in G_2 abgebildet werden, und umgekehrt.

Die beiden Graphen G_1 und G_2 sind isomorph, obwohl ihre Darstellung völlig unterschiedlich ist.

G_1 G_2 Abbildung



p(a) = 1
p(b) = 6
p(c) = 8
p(d) = 3
p(g) = 5
p(h) = 2
p(i) = 4
p(j) = 7

Wenn ihre Kanten Gummibänder wären, dann könnte man, ohne sie zu zerreißen, die Knoten von G_1 so verschieben, dass G_2 daraus entstünde. Man kann überprüfen, dass z.B. die Nachbarn von a in G_1, nämlich g,h,i auf die Nachbarn von p(a) = 1, nämlich 5,2 und 4 abgebildet werden. Das gilt für alle Knoten in G_1 und umgekehrt, auch für alle Knoten in G_2.

2.9 Berechenbarkeit von Funktionen

Für Informatiker ist natürlich von besonderen Interesse, welche Funktionen überhaupt von unseren heutigen Computern berechenbar sind. Gibt es eventuell Funktionen, die nicht berechenbar sind? Gibt es vielleicht Funktionen, die zwar von unseren heutigen Computern nicht berechenbar sind, aber vielleicht von Berechnungsmodellen, die stärker sind als unsere heutigen Computer? Dies alles sind Fragen aus dem Gebiet der Berechenbarkeitstheorie. Daher sollen hier nur einige kurze Andeutungen gemacht werden.

Zunächst einmal kann man folgendes feststellen: Unsere heutigen Computer arbeiten auf Bitfolgen. Sie transformieren Bitfolgen. Jede Bitfolge kann man aber als Binärdarstellung einer natürlichen Zahl auffassen. Daher reduziert sich die Fragestellung auf: Welche Funktionen von natürlichen Zahlen in natürliche Zahlen sind berechenbar?

Selbst Partnervermittlungen im Internet machen nichts anderes als Funktionen auf natürlichen Zahlen zu berechnen. Die Daten, die ein Kunde über sich eingibt werden binär gespeichert, und es wird als Ergebnis die binäre Darstellung eines anderen Kunden berechnet, der besonders gut zu dem ersten passt.

Dass die Berechnungsprobleme selbst für natürliche Zahlen nicht so einfach sind, lässt sich anhand einiger kleiner Beispiele illustrieren.

Die folgende Funktion:

$$Collatz(n) = \begin{cases} 1 & \text{falls } n = 1 \\ Collatz(n/2) & \text{falls } n \text{ gerade ist} \\ Collatz(3 \cdot n + 1) & \text{sonst} \end{cases}$$

liefert, soweit man bisher weiß, für jedes n als Ergebnis die 1. Es hat aber noch niemand geschafft, das für alle n zu beweisen.

Ein weiteres Beispiel ist die folgende Funktion:

$$h_\pi(n) = \begin{cases} 1 & \text{falls die Zahl 7 } n \text{ mal hintereinander in der Dezimaldarstellung von } \pi \text{ vorkommt} \\ 0 & \text{sonst} \end{cases}$$

Da die Dezimaldarstellung von π unendlich lang ist, kann es für die Zahl 7 entweder beliebig lange Ketten von 7ern darin geben, oder es kommen maximal k viele 7er hintereinander vor, z.B. $k = 100$. Im ersten Fall gilt $h_\pi(n) = 1$, unabhängig von n. Im zweiten Fall gilt

$$h_\pi(n) = \begin{cases} 1 & \text{falls } n \leq k \\ 0 & \text{sonst} \end{cases}$$

In beiden Fällen ist die Berechnungsvorschrift total einfach. Es hat aber noch kein Wissenschaftler herausgefunden, welche davon jetzt gilt.

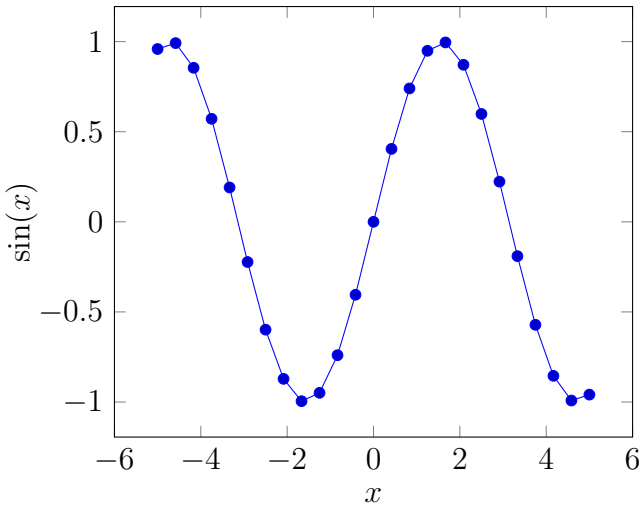
Bislang gilt noch die Church-Turing These:

„Alles was man intuitiv berechnen kann, kann man auch mit einer Turingmaschine berechnen."

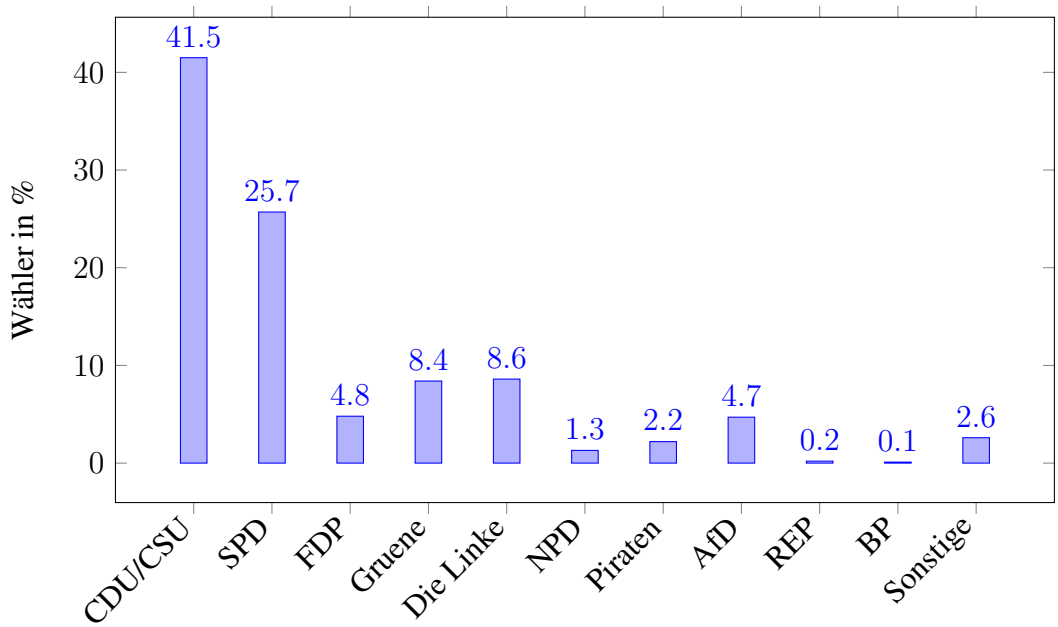
Eine Turingmaschine ist ein von Alan Turing erfundenes extrem einfaches Berechnungsmodell (siehe das Miniskript über Turingmaschinen). Unsere Computer sind zwar um ein Vielfaches schneller als eine Turingmaschine, können aber auch nur die gleichen Funktionen berechnen, wie die Turingmaschinen.

2.10 Visualisierung von Funktionen

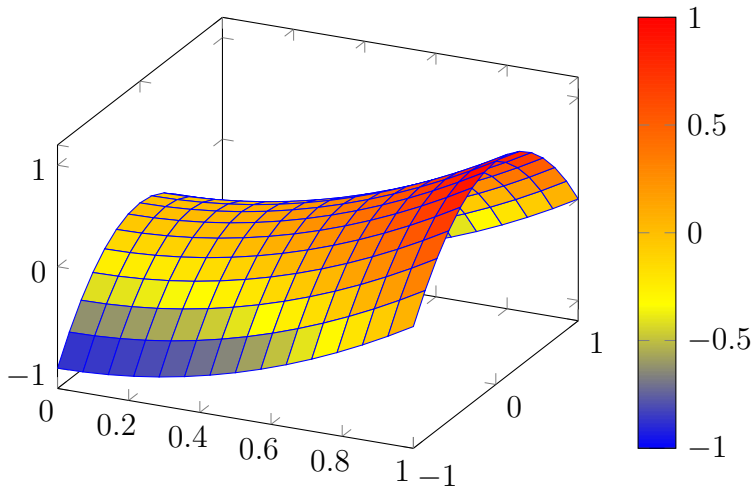
Die Visualisierung von Funktionen $\mathbb{R} \to \mathbb{R}$ kennt man schon aus der Schule. Ein Beispiel ist die Visualisierung der Sinusfunktion:



Wenn die Domain nicht die reellen Zahlen sind, sondern eine endliche diskrete Menge von Objekten, dann hat man auch schon viele Beispiele von Funktionsvisualisierungen gesehen. Die Ergebnisse der Bundestagswahl 2013 als Funktion $Parteien \to [0, 100]$ lässt sich z.B. folgendermaßen visualisieren:



Funktionen $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$ sind schon deutlich schwieriger zu zeichnen. Hier ist z.B. die Funktion $f(x, y) = x^2 - y^2$:



Hat die Domain 3 oder mehr Dimensionen, wird die Visualisierung nahezu unmöglich.

3 Relationen

Die bekanntesten Arten von Relationen sind sicherlich die zweistelligen Relationen. In der Arithmetik sind es u.A. die Vergleichsrelationen $<, \leq, =, >, \geq$. In Alltagssituationen sind es Relationen wie $befreundetMit$ oder $verheiratetMit$ oder $KindVon$ usw. In der Informatik hat man insbesondere die Vererbungsrelation zwischen Klassen im objektorientierten Programmieren.

Dreistellige Relationen sind schon deutlich seltener. Die Aussage „Frankfurt liegt *zwischen* München und Hamburg“ braucht z.B. eine dreistellige Relation $zwischen$.

Da man alle Relationen auch als Abbildungen auf die Wahrheitswerte $wahr$ und $falsch$ auffassen kann, macht es auch Sinn, einstellige Relationen zu betrachten.
$istGerade(2) = wahr$ und $istGerade(3) = falsch$ wäre ein typisches Beispiel. Sogar nullstellige Relationen gibt es, nämlich die Konstanten $wahr$ und $falsch$.

3.1 Schreibweisen

Die Standardschreibweise von n-stelligen Relationen ist die *Präfixschreibweise*: $r(x_1, \ldots x_n)$. Bei zweistelligen Relation benutzt man dagegen häufiger die *Infixschreibweise*, $x_1\ r\ x_2$. Insbesondere in der Arithmetik ist das üblich, z.B. $x < y$. Eine Infixschreibweise von dreistellige Relationen ist im Prinzip auch möglich, z.B. „Frankfurt $liegtZwischen$ München und Hamburg“, wird aber in formalen Texten kaum benutzt.

3.2 Definitionsbereich von Relationen

Für Relationen ist es im Prinzip nicht notwendig, eine Domain oder Definitionsbereich anzugeben. Eine Relation wie *istVerwandtMit* würde man normalerweise nur auf Menschen anwenden wollen. Wenn man es aber unbedingt auf andere Dinge anwenden wollte, z.B. *istVerwandtMit(2,3)* dann würde man das einfach als $falsch$ erklären.

In getypten Programmiersprachen wird man trotzdem gezwungen, für Relationen Argumenttypen festzulegen. Damit kann ein Compiler schon feststellen, dass z.B. in einem Programmstück *istVerwandtMit(2,3)* etwas falsch sein muss.

Auch die mathematische Definition von n-stelligen Relationen, nämlich als Teilmenge eines kartesischen Produkts

$$r \subseteq D_1 \times, \ldots, \times D_n$$

greift auf Definitionsbereiche $D_1, \ldots, D_n$ zurück.

3.3 Konstruktion von Relationen

Relationen kann man auf die unterschiedlichsten Weisen definieren. Eine Möglichkeit ist, aus gegebenen Relationen neue Relationen zu konstruieren. Insbesondere aus zweistellige Relationen werden häufig neue Relationen gemacht.

3.3.1 Komposition von zweistelligen Relationen

Eine Definition wie „Enkel sind Kindeskinder“ lässt sich mathematisch mithilfe einer $KindVon$-Relation ausdrücken:

$$\forall x, z: \ EnkelVon(x, z) \Leftrightarrow \exists y: \ KindVon(x, y) \wedge KindVon(y, z)$$

d.h. z ist ein Enkel von x genau dann wenn z Kind eines Kindes (y) ist.

Ganz allgemein lässt sich aus zwei zweistelligen Relationen r und s ($r = s$ ist auch möglich) eine neue Relation rs bilden:

$$\forall x, z: \ rs(x, z) \Leftrightarrow \exists y: \ r(x, y) \wedge s(y, z).$$

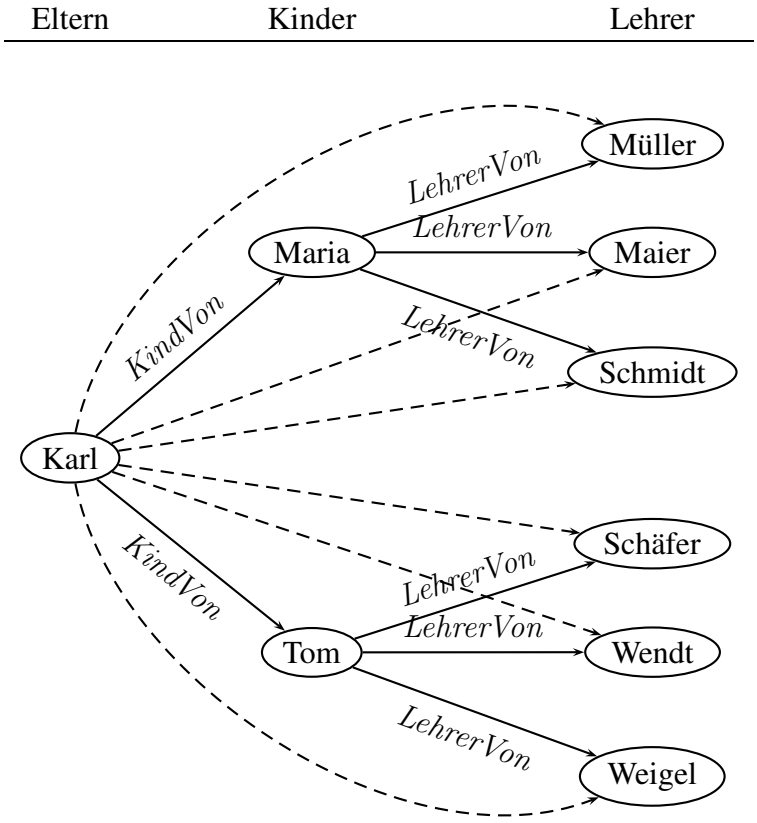
Abkürzend schreibt man dafür: $rs = r \circ s$.
$r \circ s$ ist die *Komposition* von r und s.

Diese Konstruktion kann man beliebig fortsetzen, z.B.

$$UrenkelVon = KindVon \circ KindVon \circ KindVon$$

Als weiteres Beispiel, die Relation $KindVon \circ LehrerVon$ würde die Beziehung zwischen Eltern mit allen Lehrern aller ihrer Kinder definieren.

Die folgende Graphik zeigt ein Beispiel.
Die gestrichelten Pfeile symbolisieren die $KindVon \circ LehrerVon$-Relation.



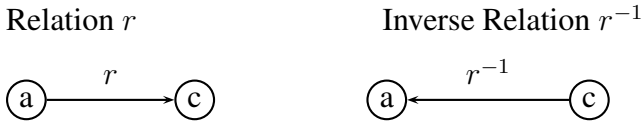
3.3.2 Inverse zweistellige Relationen

Jede zweistellige Relation r kann man auf einfache Weise in ihr Inverses r^{-1} umkehren:

$$\forall x, y: \; r^{-1}(x, y) \Leftrightarrow r(y, x).$$

Solche Umkehrungen kommen häufig vor: $<^{-1} = >$, $KindVon^{-1} = ElterVon$ usw.

In der graphischen Darstellung von Relationen mit Pfeilen zwischen den Objekten erhält man die Inverse Relation, indem man einfach die Pfeilrichtung umkehrt.



3.3.3 Das Komplement einer Relation

Das Komplement einer Relationen besteht aus allen Tupeln, die *nicht* in der Relation selbst stehen. Hierbei macht es Sinn, den Definitionsbereich einer Relation mit in die Definition des Komplements einzubeziehen.

Formal gilt für eine n-stellige Relation $r \subseteq (D_1 \times \ldots \times D_n)$:

$$Komplement(r) = \{(x_1, \ldots, x_n) \subseteq (D_1 \times \ldots \times D_n) \mid r(x_1, \ldots, x_n) \text{ gilt nicht}\}$$

Typische Beispiele auf Zahlen sind: $Komplement(<) = \geq$ und $Komplement(>) = \leq$.

Falls man hier den Definitionsbereich der Relation nicht mit einbeziehen würde, dann würde z.B. auch $Einstein\ Komplement(<)\ BMW$ gelten, da ja $Einstein < BMW$ nicht gilt.

3.3.4 Funktionen $\leftrightarrow$ Relationen

Jede n-stellige Funktion f kann man in eine $n+1$-stellige Relation r_f umwandeln:

$$\forall x_1, \ldots, x_n, y:\ r_f(x_1, \ldots, x_n, y)\ gdw.\ y = f(x_1, \ldots, x_n).$$

Wenn man die Funktion f graphisch darstellen will, dann entspricht r_f gerade dem Graphen der Funktion.

Umgekehrt, falls von der Definition von Relationen ausgegangen wird, werden Funktionen als *linkstotale* und *rechtseindeutige* Relationen definiert. D.h. eine $n+1$-stellige Relation $r(x_1, \ldots, x_n, x_{n+1})$ kann man als Funktion auffassen, falls es *für jedes* n-Tupel $x_1, \ldots, x_n$ (linkstotal) *genau ein* (rechtseindeutig) x_{n+1} mit $r(x_1, \ldots, x_n, x_{n+1})$ gibt. Dann kann man stattdessen auch schreiben $x_{n+1} = f_r(x_1, \ldots, x_n)$.

Solche Relationen bezeichnet man auch als *funktional*.

Wenn die Relation nicht funktional ist, kann man daraus auch eine Funktion machen, die muss aber statt einem Element *eine ganze Liste* von Elementen liefern.

$$\forall x_1, \ldots, x_n:\ f_r(x_1, \ldots, x_n) = \{y \mid r(x_1, \ldots, x_n, y)\}$$

In Programmiersprachen nutzt man das, indem man sog. Collection-Typen einführt (Listen, Sets, Arrays usw.), in denen man beliebig viele Elemente speichern kann. Die Funktionen müssen dann solche Collection-Typen zurückliefern.

Eine einstellige Funktion $VaterVon : Mensch \rightarrow Mensch$ kann man in eine zweistellige Relation $VaterVon : Mensch \times Mensch$ umwandeln, und auch umgekehrt, da ja jeder nur einen Vater hat. Eine Relation $KindVon : Mensch \times Mensch$ könnte man dagegen nicht direkt in eine Funktion $KindVon : Mensch \rightarrow Mensch$ umwandeln, da Menschen ja mehr als ein Kind haben können. Man kann sie in eine Boolesche Funktion $KindVon : Mensch \times Mensch \rightarrow boolean$ umwandeln, oder in eine Funktion, die eine ganze Liste von Kindern liefert: $KindVon : Mensch \rightarrow List(Mensch)$.

3.3.5 n-stellige Relationen $\mapsto$ zweistelligen Relationen

Zweistellige Relationen haben in der Tat eine Sonderstellung, denn mit einer geschickten Umformung lassen sich beliebige mehrstellige Relationen in (mehrere) zweistellige Relationen umbauen. Als Beispiel betrachten wir die dreistellig Relation $zwischen$, z.B. $zwischen$(***München, Frankfurt, Hamburg***).
Einem solchen Triple könnten wir einen eigenen Namen geben, z.B.
$MFH =$ (***München, Frankfurt, Hamburg***).
Jetzt definieren wir drei zweistellige Relationen: $Ausgangsstadt$, $Zwischenstadt$, $Endstadt$ und formen die obige $zwischen$-Beziehung um:
$Ausgangsstadt(MFH,$ ***München***$)$
$Zwischenstadt(MFH,$ ***Frankfurt***$)$
$Endstadt(MFH,$ ***Hamburg***$)$

Die allgemeine Umwandlungsvorschrift für n-stellige Relationen r ($n > 2$) ist daher folgendermaßen:

1. Definiere n neue zweistellig Relationen $r_1, \ldots, r_n$.
2. Für jedes konkrete n-Tupel $r(a_1, \ldots, a_n)$
   (a) erzeuge einen eindeutigen Namen a,
   (b) ersetze $r(a_1, \ldots, a_n)$ durch $r_1(a, a_1)$,..., $r_n(a, a_n)$.

In objektorientierten Programmiersprachen wird diese Konstruktion implizit angewandt. Betrachten wir eine Klassendefinition einer Klasse `Student`, mit den Attributen `Name`, `Semester`, `Matrikelnummer`. Ein konkretes `Student` Objekt würde man typischerweise mit einer Konstruktormethode erzeugen:
`Student mayer = new Student(Mayer,3,123456).`

Mathematisch können wir diese Klasse als 3-stellige Relation auffassen:
`Student: String × Integer × Integer`,
und für unseren konkreten Studenten `mayer` hätten wir die Relation
`Student(Mayer,3,123456).`

Üblicherweise definiert man dann in der Klasse `Student` getter-Funktionen: `getName`, `getSemester`, `getMatrikelnummer`, die man aufrufen könnte `getName(mayer)`, `getSemester(mayer)`, `getMatrikelnummer(mayer)`.
(In Java würde man aber schreiben:
`mayer.getName()`, `mayer.getSemester()`, `mayer.getMatrikelnummer()`).

Mathematisch könnte man die getter-Funktionen dann auch als zweistellige Relationen auffassen:

$name$:	$Student \times String$
$semester$:	$Student \times Integer$
$matrikelnummer$:	$Student \times Integer$.

Verglichen mit der oben definierten Umwandlungsprozedur haben wir jetzt die Beziehung:
Die getter-Funktionen entsprechen den generierten zweistelligen Relationen.
Das mit der Konstruktormethode erzeugte Objekt (`Student mayer`) entspricht dem Namen a für das ganze Tripel (im Beispiel: (`Mayer`,3,123456)).

3.4 Eigenschaften von zweistelligen Relationen

Die wichtigsten Eigenschaften von zweistelligen Relationen r sind:
$reflexiv(r)$ $\quad gdw. \forall x: \ r(x,x)$
$symmetrisch(r)$ $gdw. \forall x,y: \ r(x,y) \Rightarrow r(y,x)$
$transitiv(r)$ $\quad gdw. \forall x,y,z: \ r(x,y) \wedge r(y,z) \Rightarrow r(x,z)$

Beispiele:
Die $\leq$-Relation auf Zahlen ist reflexiv, da ja $\forall x: \ x \leq x$ gilt.
Sie ist nicht symmetrisch, wohl aber transitiv.
Es gilt ja $\forall x,y,z: \ x \leq y \wedge y \leq z \Rightarrow x \leq z$.

Eine $befreundetMit$-Relation auf Menschen wäre reflexiv, wenn jeder mit sich selbst befreundet wäre (manche Menschen hassen sich selbst; die sind wohl nicht mit sich selbst befreundet). Sie könnte auch symmetrisch sein, wenn jeder, mit dem man befreundet ist, auch mit einem selbst befreundet ist (was wohl auch nicht immer der Fall ist). Sie ist definitiv nicht transitiv, denn die Freunde meiner Freunde müssen nicht immer auch meine Freunde sein.

Weitere mögliche Eigenschaften von zweistelligen Relationen sind:
$irrreflexiv(r)$ $\quad gdw. \forall x: \ \neg r(x,x)$
$asymmetrisch(r)$ $\quad gdw. \forall x,y: \ r(x,y) \Rightarrow \neg r(y,x)$
$antisymmetrisch(r)$ $gdw. \forall x,y: \ r(x,y) \wedge r(y,x) \Rightarrow x = y$

Bei einer irreflexiven Relation gibt es *kein einziges* Element, das mit sich selbst in der Relation steht. Das ist nicht gleichbedeutend mit *nicht reflexiv*. Bei einer nicht reflexiven Relation kann es auch Elemente geben, die mit sich selbst in der Relation stehen, es dürfen aber nicht alle sein. Prominente irreflexive Relationen sind die $<$- und $>$-Relationen auf Zahlen.

Der Unterschied zwischen asymmetrisch und antisymmetrisch ist: Bei asymmetrischen Relationen darf für *kein einziges Paar* mit $r(x,y)$ auch $r(y,x)$ gelten. Daher darf es auch keine reflexiven Elemente geben, d.h. $r(x,x)$ darf für kein Element gelten. Bei antisymmetrischen Relationen darf es dagegen reflexive Elemente geben, d.h. es darf $r(x,x)$ für einige x gelten. Die $<$- und $>$-Relationen auf Zahlen sind asymmetrisch, während die $\leq$- und $\geq$-Relationen auf Zahlen antisymmetrisch sind.

Eine für die Informatik weniger wichtige Eigenschaft ist *Dichtheit*:

$$dicht(r) \Leftrightarrow \forall x,z: \ r(x,z) \Rightarrow \exists y: \ r(x,y) \wedge r(y,z)$$

Eine dichte Relation hat zwischen je zwei Elementen, die in der Relation stehen noch ein drittes Element (das, um die Definition präziser zu machen, ungleich der anderen beiden Element sein muss). Die $<$-Relation ist dicht auf den reellen und rationalen Zahlen, aber nicht auf den ganzen Zahlen.

3.5 Ordnungsrelationen

Ordnungsrelationen in der Mathematik sind Verallgemeinerungen der kleiner-gleich Beziehungen. Sie erlauben es, Elemente einer Menge miteinander zu vergleichen und in eine Hierarchie zu bringen.

3.5.1 Halbordnung

Eine Halbordnung (oder partielle Ordnung) ist eine reflexive, antisymmetrische und transitive zweistellige Relation. In der Mathematik ist die Teilmengenbeziehung $\subseteq$ eine Halbordnung.
Sie ist reflexiv: $\forall X : X \subseteq X$,
antisymmetrisch: $\forall X, Y : (X \subseteq Y \wedge Y \subseteq X) \Rightarrow X = Y$,
und transitiv: $\forall x, y, z : (X \subseteq Y \wedge Y \subseteq Z) \Rightarrow X \subseteq Z$.

Ersetzt man Reflexivität und Antisymmetrie durch Irreflexivität, dann erhält man eine *strenge Halbordnung*. Typische strenge Halbordnungen sind hierarchische Befehlsstrukturen in Organisationen. Jeder in der Hierarchie kann allen seinen Untergebenen befehlen (Transitivität), er wird aber üblicherweise nicht sich selbst befehlen (Irreflexivität). Es kann aber mehrere Chefs geben, die sich nicht gegenseitig befehlen dürfen.

Gilt in einer Halbordnung noch zusätzlich, dass es keine unendlichen echt absteigenden Ketten gibt, dann spricht man von einer *wohlfundierten Ordnung* oder *Noetherschen Ordnung* (nach der Mathematikerin Emmi Noether). Wohlfundierte Ordnungen haben immer ein kleinstes Elemente am Ende der Ketten. Jede Halbordnung auf einer endlichen Menge ist naturgemäß eine wohlfundierte Ordnung. Daher macht sich der Unterschied erst auf unendlichen Mengen bemerkbar.

Z.B. ist die unendliche Menge $\{1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \ldots\}$ nicht wohlfundiert. Sie hat kein kleinstes Element.

Wohlfundierte Ordnungen spielen eine Rolle bei Terminierungsbeweisen von Algorithmen, d.h. bei Beweisen, dass ein Algorithmus für jede Eingabe irgendwann anhält. In so einem Beweis bildet man die Daten, die der Algorithmus bearbeitet, auf eine wohlfundierte Ordnung ab, und zeigt, dass der Algorithmus in dieser Ordnung immer nach endlichen Schritten absteigt, bis er irgendwann ein kleinstes Element erreicht, wo er dann stoppt.

3.5.2 Totalordnungen

Gilt in einer Halbordnung $\leq$ zusätzlich noch die *Totalität*, d.h. $\forall x, y : x \leq y \vee y \leq x$, dann spricht man von einer *Totalordnung*. In diesem Fall ist jeder mit jedem vergleichbar. Das geht nur, wenn alle Elemente in einer *linearen Kette* angeordnet sind. Typische Vertreter sind die natürlichen Zahlen mit der $\leq$-Relation.

Totalordnungen werden gebraucht, sobald man irgendwelche Objekte sortieren möchte. Dann muss man beliebige Objekte vergleichen können, um eine Anordnung festzulegen.

3.5.3 Abgeleitete Ordnungen

Oft möchte man auf Basis einer grundlegenden Ordnung komplexere Objekte ordnen, z.B. aus der alphabetischen Ordnung von Buchstaben eine Ordnung für die Namen in einem Telefonbuch ableiten. Dafür gibt es verschiedene Möglichkeiten.

Lexikographische Ordnung: Hierbei geht man von einer vorgegebenen Ordnung einer Basismenge aus, z.B. der alphabetischen Ordnung von Buchstaben: A $>$ a $>$ B $>$ b $\ldots >$ Z $>$ z. Damit lassen sich Zeichenketten, wie z.B. Namen in einem Telefonbuch, oder Wörter in einem Lexikon ordnen:

Um zwei Zeichenketten, z.B. „Schmidt“ und „Schneider“ zu vergleichen, läuft man beide Zeichenketten synchron von links nach rechts durch, bis man zur ersten Position kommt, wo die Zeichen verschieden sind. Diese beiden Zeichen vergleicht man mit der Basisordnung. Im Beispiel ergibt sich „Schmidt“ $<$ „Schneider“, weil m $<$ n ist. Sind beide Zeichenketten unterschiedlich lang, und eine Zeichenkette ist ein Teil der anderen, wie bei „Schmid“ und „Schmidt“, dann kommt die kürzere nach vorne: „Schmid“ $<$ „Schmidt“. Sind die beiden Zeichenketten gleich, entscheidet meist der Zufall über die Reihenfolge.

Mit derselben Methode kann man beliebige n-Tupel $(x_1, \ldots, x_n)$ vergleichen, sobald man eine Ordnung $<$ für die x_i hat. $(x_1, \ldots, x_n)$ vergleicht man mit $(y_1, \ldots, y_m)$, indem man von links nach rechts x_i mit y_i vergleicht. Beim ersten Paar (x_k, y_k) mit unterschiedlichen Werten, entscheidet $x_k < y_k$ über die Anordnung der beiden Tupel. Ist ein Tupel kürzer als das andere, und das kürzere stimmt mit dem Anfangsstück des anderen überein, dann kommt das kürzere nach vorne.

Multisetordnung: Eine Multisetordnung ist im Prinzip auch eine lexikographische Ordnung, allerdings mit einer Vorverarbeitung, die aus einer Menge von Objekten n-Tupel erzeugt, die dann lexikographisch geordnet werden. Damit lassen sich Mengen von Objekten in eine Reihenfolge bringen.

Die Berechnung des Medaillenspiegels nach internationalen Wettbewerben, wie den Olympischen Spielen geschieht nach dieser Methode. Ausgangspunkt ist eine Ordnung auf den Medaillen: Gold $>$ Silber $>$ Bronze. Diese Reihenfolge bestimmt die Reihenfolge in den Medaillentripel für die einzelnen Länder: (Anzahl Goldmedaillen, Anzahl Silbermedaillen, Anzahl Bronzemedaillen). Diese Tripel lassen sich dann lexikographisch ordnen. Dadurch kommen diese oft unintuitiven Medaillenspiegel zustande mit z.B.
(2 Goldmedaillen, 0 Silbermedaillen, 0 Bronzemedaillen) $>$
(1 Goldmedaille, 10 Silbermedaillen, 20 Bronzemedaillen).

Die Bundesligatabelle wird nach einer Variante einer Multisetordnung berechnet.
Dabei ist die Ausgangsordnung:
Erzielte Punkte $>$ Tordifferenz $>$ Anzahl Tore $>$ Gesamtergebnis aus Hin- und Rückspiel im direkten Vergleich $>$ Anzahl Auswärtstore im direkten Vergleich $>$ Anzahl aller Auswärtstore.

Das ergibt für jeden Verein ein 6-Tupel, die allerdings nicht unabhängig voneinerander vorberechnet werden können. Wenn z.B. die Reihenfolge für BVB und FC Bayern berechnet werden soll, dann ergeben sich die Einträge für „Gesamtergebnis aus Hin- und Rückspiel im direkten Vergleich“ und „Anzahl Auswärtstore im direkten Vergleich“ aus den Spielen der beiden Vereine gegeneinander. [2] Das reicht aber aus, um eine sortierte Reihenfolge herzustellen, da bei der Sortierung nur jeweils zwei Objekte miteinander verglichen werden müssen.

[2] Wenn dabei alle Einträge gleich sind, gibt es ein Entscheidungsspiel.

3.6 Äquivalenzrelationen

Relationen, die reflexiv, symmetrisch und transitiv sind, bezeichnet man als *Äquivalenzrelationen*. Die einfachste Äquivalenzrelation ist die Gleichheit auf beliebigen Objekten. Es gilt ja:

$$\begin{array}{ll} \forall x: & x = x \\ \forall x, y: & x = y \Rightarrow y = x \\ \forall x, y, z: & (x = y \wedge y = z) \Rightarrow x = z \end{array}$$

Viele andere Äquivalenzrelationen ergeben sich indirekt aus der Gleichheit. In unserem Tanzschulenbeispiel könnten wir z.B. definieren: Zwei Tänzer sind äquivalent, wenn sie gleich alt sind, oder gleich groß, oder gleiche Haarfarbe haben.

Eine Äquivalenzrelation zerlegt die Menge, auf der sie definiert ist, in zueinander disjunkte *Äquivalenzklassen*. Eine Äquivalenzklasse enthält alle Elemente, die zueinander äquivalent sind. Die Anzahl von Äquivalenzklassen bezeichnet man als *Index* der Äquivalenzrelation. Die Äquivalenzrelation, die sich aus der Haarfarbe der Tänzer in der Tanzschule ergibt, zerlegt die Tänzer in Äquivalenzklassen von Leuten mit gleicher Haarfarbe. Der Index der Relation wäre die Anzahl von verschiedenen Haarfarben.

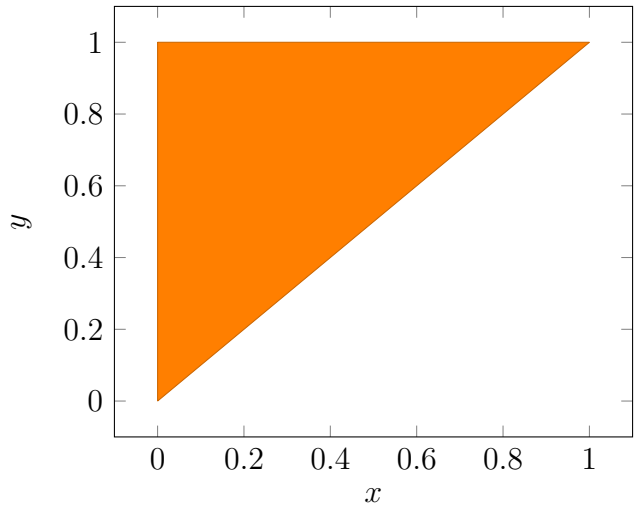
Äquivalenzklassen können auch verfeinert werden. Nehmen wir als Äquivalenzrelation für die Tänzer: *im gleichen Jahrzehnt geboren*. Dann bekommen wir als Äquivalenzklassen die in den 50ern Geborenen, die in den 60ern Geborenen usw. Eine Verfeinerung wäre: *im gleichen Jahr geboren*. Dann würden die in den 50ern Geborenen nochmal weiter zerlegt in die in 1950 Geborenen, in 1951 Geborenen usw.

3.7 Visualisierung von Relationen

n-stellige Relationen definieren die Teilmenge des n-stelligen Kreuzproduktes aller Objekte, für die die Relation wahr ist.

Eine einstellige Relation wie z.B. $istGerade$, definiert eine Teilmenge aller Objekte, nämlich die geraden natürlichen Zahlen. Die kann man natürlich ganz einfach durch eine Gerade, auf der die geraden Zahlen markiert sind, visualisieren.

Eine zweistellige Relation auf dichten Domains wie z.B. den reellen Zahlen lässt sich durch ein zweidimensionales Koordinatensystem zeichnen, in der der Bereich der Paare (x, y), für die die Relation wahr ist, gekennzeichnet wird. Als Beispiel zeichnen wir die $\leq$-Relation auf den Intervallen [0,1]:



Für alle Paare x, y im farbigen Bereich gilt $x \leq y$. Die $<$-Relation würde fast genauso aussehen, es fehlten nur die Elemente der Diagonalen.

3-stellige Relationen wären gerade noch als Wolken in einem 3-dimensionalen Koordinatensystem darzustellen. Relationen in mehr als 3 Dimensionen sind aber kaum noch darstellbar.

Die Flächendarstellung von Relationen illustriert auch sehr schön den Unterschied zwischen zweistelligen Relationen und einstelligen Funktionen. Funktionen erscheinen in dieser Darstellungen als ausdehnungslose Striche (die Funktionsgraphen), während Relationen als größere Bereiche erscheinen.

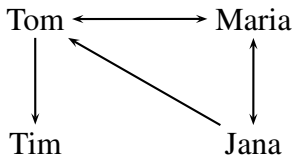
3.7.1 Relationen als gerichtete Graphen

Für zweidimensionale Relationen auf diskreten Mengen hat sich eine Darstellung als gerichtete Graphen als sehr nützlich erwiesen. Die Elemente der Domain bilden die Knoten des Graphen, und die Relationsbeziehungen bilden die Kanten.

Als Beispiel betrachten wir wieder unsere Tanzschule und eine *befreundetMit*-Relation zwischen den Tänzern. Da die Menge der Tänzer endlich ist, lässt sich die Relation zunächst als Menge von Paaren aufschreiben, z.B.:

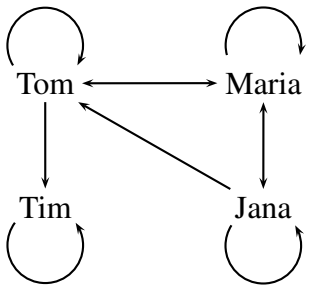
befreundetMit(Tom,Maria), *befreundetMit*(Maria,Tom), *befreundetMit*(Tom,Tim),
befreundetMit(Maria, Jana), *befreundetMit*(Jana,Maria), *befreundetMit*(Jana,Tom)

Der entsprechende Graph sieht dann so aus:



Verschiedene Eigenschaften der Relationen lassen sich in dieser Darstellung auch sehr schön visualisieren. Symmetrie äußert sich darin, dass die Pfeile jeweils in beide Richtungen gehen. Ist die Relation generell symmetrisch, dann ist der Graph ungerichtet. D.h. die Kanten haben keine Richtung, oder anders gesagt, die Pfeile, die die Kanten angeben, gehen immer in beide Richtungen.

Reflexivität äußert sich darin, dass reflexive Elemente einen Pfeil zu sich selbst haben. Sind z.B. alle Person von oben auch mit sich selbst befreundet, müsste das so aussehen.



Die Darstellung transitiver Relationen wird meist sehr unübersichtlich. Eine transitive Relation ist z.B. die $istVorfahre$-Relation, die jede Person mit all seinen Vorfahren verbindet. Zeichnet man den Stammbaum einer Person als gerichteten Graphen, dann würde die Visualisierung der $istVorfahre$-Relation jede Person mit all seinen Vorfahren verbinden. Das lässt man i.A. bleiben.

3.8 Datenstrukturen für zweistellige Relationen / Graphen

Für zweistellige Relationen *auf diskreten Mengen* wurden unterschiedliche Datenstrukturen entwickelt, die auch unterschiedlichen Zwecken dienen. Die einfachste Datenstruktur ist die Auflistung, wie z.B.

$befreundetMit$(Tom,Maria), $befreundetMit$(Maria,Tom), $befreundetMit$(Tom,Tim),
$befreundetMit$(Maria, Jana), $befreundetMit$(Jana,Maria), $befreundetMit$(Jana,Tom)

Diese findet man in logischen Programmiersprachen wie Prolog.

Alternativ kann man für jeden Knoten des entsprechenden Graphen die Menge der Nachbarknoten auflisten (Adjazenzlisten genannt):

Tom	$\mapsto$	{Maria,Tim}
Maria	$\mapsto$	{Jana,Tom}
Jana	$\mapsto$	{Maria,Tom}

In dieser Datenstruktur kann man sehr leicht durch den Graphen navigieren, indem man von einem Knoten über dessen Nachbarschaftsliste zu einem Nachbarknoten geht.

Eine dritte Darstellung sind die sog. *Adjazensmatrizen*. Das sind zweidimensionale Matrizen, wobei die Dimensionen für die Objekte stehen, und die Matrixeinträge 1 sind, wenn die Zeilen- und Spalteneinträge in der Relation stehen, ansonsten sind sie 0.

Für unsere Tanzschule von oben wäre die Matrixdarstellung der $befreundetMit$-Relation:

	Tom	Maria	Tim	Jana
Tom	0	1	1	0
Maria	1	0	0	1
Tim	0	0	0	0
Jana	1	1	0	0

In dieser Datenstruktur äußert sich Reflexivität darin, dass die Diagonale mit 1en befüllt ist. Symmetrie äußert sich darin, dass die Matrix symmetrisch an der Diagonalen ist.

4 Prädikate

In der Grammatik, der Philosophie und in der Logik benutzt man auch den Begriff *Prädikat*. In der Grammatik braucht man ihn, um Satzstrukturen zu definieren: Ein Satz besteht aus Subjekt, Prädikat und Objekt (Bsp. Tom *studiert* Informatik). Das Prädikat stellt also eine Beziehung zwischen Subjekt und Objekt her.

In der Philosophie ist ein Prädikat zunächst eine Eigenschaft eines Objekts, die wahr oder falsch sein kann. Der Begriff wurde aber auch auf Beziehungen zwischen mehreren Objekten ausgedehnt.

In logischen Systemen, z.B. der Prädikatenlogik, unterscheidet man ganz strikt zwischen Syntax und Semantik. Syntax ist die textuelle Darstellung von Sachverhalten. Die textuelle Darstellung muss so sein, dass sie ein Computer verarbeiten kann. Ein Teil einer solchen Logik sind *Prädikatensymbole*, wie z.B. *studiert*. Diese kann man mit Argumenten verbinden: *studiert(Tom,Informatik)*. Die *Interpretation* dieses Textes wäre dann: *Tom* und *Informatik* werden auf konkrete Objekte abgebildet, und *studiert* wird auf eine zweistellige Relation zwischen diesen Objekten abgebildet.

Man kann also unterscheiden:
studiert ist ein Prädikatensymbol.
studiert(Tom,Informatik) ist ein Prädikat.
Die Interpretation von *studiert* ist eine Relation.
Die Interpretation von *studiert(Tom,Informatik)* ist entweder wahr oder falsch.

Nicht alle Texte halten sich an diese Unterscheidung. Insbesondere wenn nicht so genau zwischen Syntax und Semantik unterschieden wird, vermischt sich die Verwendungen dieser Begriffe. Oft wird daher auch „Prädikat“ und „Relation“ synonym verwendet.

Stichwortverzeichnis