Analytische Geometrie im $\mathbb{R}^3$
- Modellierung mit *DrScheme (DMdA)* -

Gernot Lorenz

Version 2.0, November 2009

Die funktionale Modellierung geeigneter Themen zeigt einerseits weitgehende Gemeinsamkeiten in Mathematik und Informatik auf, andererseits können auch die unterschiedlichen Vorgehensweise aufgezeigt werden. Angesichts der Tatsache, dass ein integrierter Mathematik- und Informatikunterricht in den allgemeinbildenden Schulen, etwa in der Sekundarstufe II, ist in keinem der Bundesländer vorgesehen ist, bietet sich im Folgenden ein Beispiel dafür, wie im Informatikunterricht auch Mathematik gelernt und/oder vertieft werden kann.
Im Mittelpunkt steht die Datenmodellierung eines funktionalen Programmieransatzes zum Thema *Analytische Geometrie im* $\mathbb{R}^3$ in der Sekundarstufe II (in Auswahl). Das für die Umsetzung eingesetzte Entwicklungssystem *DrScheme*[1] eignet sich in der Lernsprachenversion *DMdA*[1] für die funktionale Modellierung in besonderer Weise aufgrund seiner didaktischen Konzepte wie *systematische Konstruktionsanleitung* und *Funktionsverträge als Bestandteil des Quellcodes*.
Das Thema bietet ein reichhaltiges Übungsgelände für den Umgang mit zusammengesetzten Datentypen wie Verbund und Liste sowie mit Funktionen höherer Ordnung (*higher-order-functions*).

1 Einführung

1.1 Allgemeines

Analytische Geometrie im $\mathbb{R}^3$ (auch im $\mathbb{R}^2$) bietet eine direkte Anschaulichkeit und eine enorme praktische Anwendbarkeit, etwa im Sinne von Problemstellungen wie „Schneiden sich die beiden Geraden?", „Welchen Winkel bildet die Gerade mit der z-Achse?", „Wo schneidet die Gerade die Ebene?" usw. Das heißt, in der Schulmathematik geht es wie in der Ingenieurswissenschaften bei die diesem Thema weit über die innermathematische Beschäftigung hinaus zur praktischen Problemlösung.
Letztere steht im Zentrum der Informatik, und da einerseites beim praktischen Problemlösen innermathematische Aspekte wie Äquivalenzen, Existenzbeweise etc. außen vor bleiben und andererseits Punktmengen wie Punkt, Gerade oder Ebene selbst mathematische Modelle der (idealisierten) Wirklichkeit sind, könnte eine Eins-zu-Eins-Übernahme der mathematischen Modelle naheliegen. Es wird sich zeigen, dass ein solche Vorgehen nicht immer hilfreich ist, vielmehr müssen z.T. andere Modellierungen gefunden werden im Rahmen der Vorgaben durch z.B. die benutzte Programmiersprache, Aufwandsuntersuchungen von Algorithmen usw.
Inhaltlich beschränkt sich das Thema auf die Untersuchung von Strukturen wie Punkte, Vektoren und Geraden, kann aber erweitert werden auf Kugeln, Kreise usw.

[1] Programmierumgebung für Anfänger *http://www.plt.org*

1.2 Didaktische Aspekte

Im diesem Beitrag geht es um Anregungen zum Programmieren-Lernen für Unterricht oder Übungen in der Sekundarstufe II und/oder im Anfangssemester an Hochschulen, und zwar speziell für einen *funktionalen Ansatz*. Dieser eignet sich für die o.a. gestellten Fragen in besonderer Weise, da hier numerische oder BOOLEsche Werte als Ergebnisse erwartet werden.
Von daher stehen aus Sicht des Programmier-Unterrichtes der Einsatz von zusammengesetzten Datentypen wie Liste und Verbund (*record*) für die Datenmodellierung sowie Funktionen höherer Ordnung (*higher order functions*) und das Erzeugen neuer Funktionen, auch anonymer Funktionen („Schönfinkeln") durch Funktionen im Mittelpunkt.
Die didaktischen „Bonbons" des *DMdA*[1]-Konzeptes (*„Die Macht der Abstraktion"*) sind dabei

- die systematische Entwicklung einer Funktion mittels einer klar definierten *Konstruktionsanleitung*
- die vorgeschriebenen *Verträge* für Funktionen und Datentypen, die als Bestandteil des Quellcodes den Programmierer - im vorliegenden Fall den Schüler - zu gründlicher Planung und Disziplin zwingen

1.3 Codierung

Allgemein

Die Codierung der Modelle bzw. Entwürfe erfolgt mit der *DrScheme*-Version 4.2.2, bei der sowohl bei den *DMdA*-Sprachen (es wird das Sprachlevel *Die Macht der Abstraktion* benutzt) als auch bei den *HtDP*-Sprachen[2] im Gegensatz zu früheren Versionen Testfälle als Bestandteil des Quellcodes mit `(check-expect <Funktionsaufruf> <Rückgabewert>))` aufgeführt werden.
Bei den *DMdA*-Sprachen kommt hinzu, dass Verträge für Funktionen wie z.B. und `(: skalar-produkt (vektor vektor -> real))` oder bei der Typ-Definition wie z.B. `(define gerade (contract (mixed gerade-2pf gerade-prf)))` ebenfalls Bestandteil des Quellcodes sind.

DMdA vs. HtDP

Für Leser mit Vorkenntnissen in den *HtDP*-Sprachen[2] sind bei *DMdA* folgende Unterschiede zu beachten:

1. Funktionsverträge sind in *DMdA* Bestandteile des Quellcodes und haben die Form
   `(: <Bezeichner> (<Argument> ... -> <Rückgabewert>))`

2. Eine Funktionsdefinition hat im Gegensatz zu *HtDP* die Form

   ```
   (define <Bezeichner>
      (lambda <Parameterliste>
         <Rumpf>))
   ```

3. Im Gegensatz zu den *HtDP*-Sprachen lauten die Wahrheitswerte nicht `true` und `false`, sondern in *DMdA* `#t` und `#f`, also wie in Standard-Scheme
4. Die Prüfung auf Gleichheit mit `equal?` ist erst *DMdA*-Stufe *Die Macht der Abstraktion mit Zuweisungen* möglich. Für *Die Macht der Abstraktion - Anfänger* und *Die Macht der Abstraktion* wird statt dessen das gewöhnliche Gleichheitszeichen = benutzt.
5. Auf der hier benuzten Sprachebene *Die Macht der Abstraktion* steht die Funktion `apply` nicht zur Verfügung. Statt dessen wird, wenn möglich, `fold` verwendet.

2 Datenmodellierung

Die in Frage kommenden Objekte wie Punkt sowie die Punktmengen Gerade, Ebene und Kugel sind bereits mathematische Modelle, für die eine geeignete Daten-Repäsentation in der vorliegenden Programmiersprache gesucht werden muss.

2.1 Punkte und Vektoren

Punkte und Vektoren im $\mathbb{R}^3$ sind datentechnisch äquivalent, es handelt sich jeweils um Tripel von reellen Zahlen:

$$P\,(X \mid Y \mid Z) \qquad \equiv \qquad \vec{p} \;= \begin{pmatrix} x \\ y \\ z \end{pmatrix}$$

Als zusammengesetzter Datentyp bietet sich ein *Verbund* an, der in den DMdA-Sprachstufen folgendermaßen konstruiert wird:

```
(define-record-procedures vektor
  make-vektor vektor?
  (vektor-x vektor-y vektor-z))
```

Listing 1: Zahlentripel als Verbund

Somit liegt eine Analogie zum 2-dimensionalen Typ `posn` der HtDP-Sprachen vor.

Dennoch soll im Folgenden ein anderer Weg verfolgt werden: Wie bereits in der Einführung verdeutlicht worden ist, soll hier der Umgang mit *Funktionen höherer Ordnung* geübt werden, was die Bearbeitung von Listen voraussetzt. Deshalb wird ein Punkt oder Vektor durch eine 3-elementige *Liste* repräsentiert:

```
(define vektor
  (contract
   (combined (list real)
             (predicate (lambda (l) (= (length l) 3))))))
```

Listing 2: Zahlentripel als Liste mit 3 rellen Zahlen

Hier wird eine *DMdA*-Spezialität deutlich: Die Typ-Definition erfolgt durch einen Vertrag, in dem der Typ „reelle" Zahlenliste mit dem Prädikat/Bedingung, dass diese genau drei Elemente enthalten muss, kombiniert.

2.2 Grundoperationen als Funktionen

Die Funktion `v-addition` für die ***Vektoraddition*** verbraucht zwei Listen und liefert eine Liste zurück, also

```
(: vektor+ (vektor vektor -> vektor))

(check-expect (vektor+ (list 1 0 -4) (list -2 1 3)) (list -1 1 -1))
```

Da die Elemente zweier gleichlanger Listen paarweise addiert werden sollen, bietet sich der Einsatz von `map` an:

```
(define vektor+
  (lambda (v1 v2)
    (map + v1 v2)))
```

Der Einfachheit kann man auch eine *Vektorsubtraktion* definieren, wenn man eine s-Multiplikation mit -1 und Vektorraddition vermeiden will:

```
(: vektor- (vektor vektor -> vektor))

(check-expect (vektor- (list 1 0 -4) (list -2 1 3)) (list 3 -1 -7))
```

```
(define vektor-
  (lambda (v1 v2)
    (map - v1 v2)))
```

Die ***s-Multiplikation*** oder *Skalar-Multiplikation* verbraucht als Skalar eine reelle Zahl und als Vektor eine Liste und liefert eine Liste als Vektor zurück, also

```
(: s-multiplikation (real vektor -> vektor))

(check-expect (s-multiplikation 3 (list 1 0 -4)) (list 3 0 -12))
```

Da jedes Element der Vektor-Liste mit dem Skalar multipliziert werden soll, können wir die anonyme Funktion `(lambda (komponente) (* skalar komponente))` mittels `map` auf `vektor` anwenden:

```
(define s-multiplikation
  (lambda (s v)
    (map (lambda (komponente)(* s komponente)) v)))
```

Für das ***Skalarprodukt*** ergibt sich aus der Definition

$$\vec{v_1} \cdot \vec{v_2} = \begin{pmatrix} v_{1x} \\ v_{1y} \\ v_{1z} \end{pmatrix} \cdot \begin{pmatrix} v_{2x} \\ v_{2y} \\ v_{2z} \end{pmatrix} := v_{1x}v_{2x} + v_{1y}v_{2y} + v_{1z}v_{2z} \quad \in \mathbb{R}$$

mit Hilfe von `map` und `fold` (da in diesem Sprachlevel `apply` nicht zur Verfügung steht):

```
(: skalar-produkt (vektor vektor -> real))

(check-expect (skalar-produkt (list 1 1 1) (list 1 2 3)) 6)
(check-expect (skalar-produkt (list 1 1 0) (list 0 0 1)) 0)

(define skalar-produkt
  (lambda (v1 v2)
    (fold 0 + (map * v1 v2))))
```

Der ***Betrag/Länge eines Vektors*** ergibt sich sofort aus $\sqrt{\vec{v} \cdot \vec{v}}$:

```
(: betrag (vektor -> real))

(check-expect (betrag (list 1 2 2)) 3)

(define betrag
  (lambda (vektor)
    (sqrt (skalar-produkt vektor vektor))))
```

Das ***Vektorprodukt***, auch „Kreuzprodukt" genannt, existiert aufgrund seiner Definition bekanntlich nur im 3-dimensionalen Raum

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_1 \\ a_2 \\ a_3 \end{pmatrix} \times \begin{pmatrix} b_1 \\ b_2 \\ b_3 \end{pmatrix} := \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix}$$

mit den Eigenschaften:

- Für $\vec{n} = \vec{a} \times \vec{b}$ gilt: $\vec{n} \perp \vec{a}$ und $\vec{n} \perp \vec{b}$
- $|\vec{a} \times \vec{b}| = |\vec{a}| \cdot \vec{b}| \cdot \sin\phi \quad$ mit $\phi = \sphericalangle(\vec{n}, \vec{a})$
- $\vec{a} \cdot (\vec{b} \times \vec{c}) = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = det(\vec{a}, \vec{b}, \vec{c}) \qquad$ (Spatprodukt $\equiv$ Determinante, s.u.)

Das *Vektorprodukt* selbst wird als Funktion zweier Listen modelliert:

```
(: vektor-produkt (vektor vektor -> vektor))
```

mit z.B.

```
(check-expect (vektor-produkt (list 1 1 1) (list 1 2 3)) (list 1 -2 1))
(check-expect (vektor-produkt (list 0 1 0) (list 1 0 0) ) (list  0 0 -1))
(check-expect (vektor-produkt (list 0 0 1) (list 0 1 0)) (list -1 0 0))
```

Wir wählen einen Weg, der den Einsatz von `map` ermöglicht. Dazu betrachten wir folgende Umformung:

$$\vec{a} \times \vec{b} = \begin{pmatrix} a_2b_3 - a_3b_2 \\ a_3b_1 - a_1b_3 \\ a_1b_2 - a_2b_1 \end{pmatrix} = \begin{pmatrix} a_2b_3 \\ a_3b_1 \\ a_1b_2 \end{pmatrix} - \begin{pmatrix} a_3b_2 \\ a_1b_3 \\ a_2b_1 \end{pmatrix} = \begin{pmatrix} a_2 \\ a_3 \\ a_1 \end{pmatrix} \otimes \begin{pmatrix} b_3 \\ b_1 \\ b_2 \end{pmatrix} - \begin{pmatrix} a_3 \\ a_1 \\ a_2 \end{pmatrix} \otimes \begin{pmatrix} b_2 \\ b_3 \\ b_1 \end{pmatrix}$$

wobei die Verknüpfung $\otimes$ eine komponentenweise Multiplikation zweier Vektoren ist, deren Ergebnisse wieder einen Vektor bilden. Schreiben wir die vier Spaltenvektoren im letzten Term als Listen, erhalten wir

```
(vektor- (⊗  (list a2 a3 a1) (list b3 b1 b2)) (⊗  (list a3 a1 a2) (list b2 b3 b1)))
```

Modifizieren wir jetzt den Term mit den aus der Technischen Informatik bekannten Ring-Schiebe-Register-Operatoren `rol` (= rotate left, also Links-Schieben) und `ror` (= rotate right, also Rechts-Schieben), erhalten wir

```
(vektor-
    (⊗ (rol (list a1 a2 a3 )) (ror (list b1 b2 b3)))
    (⊗ (ror (list a1 a2 a3 )) (rol (list b1 b2 b3))))
```

Ersetzt man `(⊗ liste1 liste2)` durch `(map * liste1 liste2)`, bleiben nur noch die Hilfsfunktion `rol` und `ror` zu definieren:

```
;---„Rotiert" Liste=Vektor nach links
(: rol (vektor -> vektor))

(check-expect (rol (list 2 3 4)) (list 3 4 2))

(define rol
  (lambda (vektor)
    (append (rest vektor) (list (first vektor)))))
```

und

```
;---„Rotiert" Liste=Vektor nach rechts
(: ror (vektor -> vektor))

(check-expect (ror (list 2 3 4)) (list 4 2 3))

(define ror
  (lambda (vektor)
    (reverse (rol (reverse vektor)))))
```

Insgesamt ergibt sich

```
(define vektor-produkt
  (lambda (v1 v2)
    (map -
         (map * (rol v1) (ror v2))
         (map * (ror v1) (rol v2)))))
```

Für bestimmte Anwendungen ist auch das ***Spatprodukt***

$$(\vec{a} \times \vec{b}) \cdot \vec{c}$$

nützlich. Es besonderer Vorteil liegt darin, dass man aufgrund der Beziehung

$$(\vec{a} \times \vec{b}) \cdot \vec{c} = \begin{vmatrix} a_1 & b_1 & c_1 \\ a_2 & b_2 & c_2 \\ a_3 & b_3 & c_3 \end{vmatrix} = det(\vec{a}, \vec{b}, \vec{c})$$

auch 3x3-Determinanten berechnen kann, weshalb wir die zugehörige Funktion `spat-produkt`auch `det` nennen können:

```
(: spat-produkt (vektor vektor vektor -> real))

(check-expect (spat-produkt (list 1 1 1) (list 1 2 3) (list 1 1 1)) 0)
(check-expect (spat-produkt (list 1 0 0) (list 0 1 0) (list 0 0 1)) 1)

(define spat-produkt
  (lambda (v1 v2 v3)
    (skalar-produkt (vektor-produkt v1 v2) v3)))
```

Schließlich erlaubt die Determinantenberechnung im $\mathbb{R}^3$ auch die Untersuchung von Linearen Gleichungssystemen (LGS) der Form

$$x \cdot \vec{a} + y \cdot \vec{b} + z \cdot \vec{c} = \vec{d}$$

mit Hilfe der *CRAMERschen Regel*, wobei für die Lösungsmenge als reelles Zahlentripel gilt:

$$\mathfrak{L} = \left[x = \frac{det(\vec{d}, \vec{b}, \vec{c})}{det(\vec{a}, \vec{b}, \vec{c})},\ y = \frac{det(\vec{a}, \vec{d}, \vec{c})}{det(\vec{a}, \vec{b}, \vec{c})},\ z = \frac{det(\vec{a}, \vec{b}, \vec{d})}{det(\vec{a}, \vec{b}, \vec{c})} \right]$$

Existiert keine eindeutige Lösung, kann soll die gesuchte Funktion mit der Zeichenkette `"nicht (eindeutig) lösbar)"` anworten.
Das ergibt folgenden Vertrag:

```
(: cramer (vektor vektor vektor vektor -> (mixed string (list real))))
```

mit z.B.

```
(check-expect
 (cramer (list 2 3 1) (list -5 1 -3) (list 7 -2 5) (list 1 4 2))
 (list 21/13 45/13 28/13))
```

und der Schablone

```
(define cramer
  (lambda (v1 v2 v3 v4)
    (cond
      (.....nicht lösbar..... <Zeichenkette>)
      (else
       <Lösungsmenge>))))
```

Nicht lösbar ist das LGS, wenn $det(\vec{a}, \vec{b}, \vec{c}) = 0$, also muss geprüft werden, ob das Spatprodukt der drei Vektoren Null ergibt.
Damit ergibt sich insgesamt

```
(define cramer
  (lambda (v1 v2 v3 v4)
    (cond
      ((zero? (spat-produkt v1 v2 v3)) "nicht (eindeutig) lösbar)"
      (else
       (list
        (/ (spat-produkt v4 v2 v3) (spat-produkt v1 v2 v3))
        (/ (spat-produkt v1 v4 v3) (spat-produkt v1 v2 v3))
        (/ (spat-produkt v1 v2 v4) (spat-produkt v1 v2 v3)))))))
```

2.3 Punktmengen: Beispiel *Gerade*

Bei Punktmengen wie Gerade, Ebene und Kreis gibt es Beschreibungen mit und ohne Parameter, letztere sind je nach dem mit Skalar- oder/und Vektorprodukt möglich:

Für eine *Gerade* im $\mathbb{R}^3$ sind folgende Beschreibungen bzw. Darstellungen üblich:

1. 2-Punkteform, mit Parameter
   $g: \quad \vec{p} = \vec{a} + t \cdot (\vec{a} - \vec{b}) = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + t \cdot \left[\begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} - \begin{pmatrix} b_x \\ b_y \\ b_z \end{pmatrix} \right] \quad \text{mit } t \in \mathbb{R}$
   Kennzeichnung: 2 Ortsvektoren $\vec{a}, \vec{b}$
2. Punktrichtungsform, mit Parameter
   $g: \quad \vec{p} = \vec{a} + t \cdot \vec{v} = \begin{pmatrix} a_x \\ a_y \\ a_z \end{pmatrix} + t \cdot \begin{pmatrix} v_x \\ v_y \\ v_z \end{pmatrix} \quad \text{mit } t \in \mathbb{R}$
   Kennzeichnung: Ortsvektor $\vec{a}$, Richtungsvektor $\vec{v}$
3. parameterfreie Form, mit *Vektorprodukt*
   $g: \quad \vec{v_0} \times (\vec{p} - \vec{a}) = \vec{0}$
   Kennzeichnung: Ortsvektor $\vec{a}$, „Einheits"-Richtungsvektor $\vec{v_0}$, d.h. $|\vec{v_0}| = 1$

Eine Analyse dieser Übersicht bringt folgende Erkenntnisse:

- In allen drei Darstellungen ist eine Gerade durch 2 Vektoren gekennzeichnet
- Die Parameterdarstellungen 1. und 2. dienen der Berechnung eines Punktes $P \in g$ aus einem vorgegebenen Parameter t
- Die parameterfreie Darstellung 3. ermöglicht
  - die Prüfung der Frage „$P \in g$?"
  - Abstandsberechungen $d(P, g) = |\vec{v_0} \times (\vec{p} - \vec{a})|$

Also kann eine `gerade` als ein *gemischter Datentyp* modelliert werden:
Eine Gerade ist gekennzeichnet durch

- 2 Ortsvektoren oder
- 1 Ortsvektor und 1 Richtungsvektor

Daraus ergibt sich folgender Vertrag:

```
(define gerade
  (contract
   (mixed
    gerade-2pf       ; 2 Ortsvektoren
    gerade-prf)))    ; 1 Ortsvektor, 1 Richtungsvektor
```

Listing 3: Vertrag für den Typ *Gerade*

Die Ortsvektor zeigen auf Punkte der Geraden, also können wir den zugehörigen Zahlentripeln auch von Punkten sprechen, und entsprechend ergeben sich die beiden Einzeltypen `gerade-2pf` und `gerade-prf`, die sinnvollerweise als *Verbund* mit jeweils zwei Komponenten für die beiden Vektoren definiert werden:

```
(: make-gerade-2pf (vektor vektor -> gerade-2pf))
; Eine Gerade ist gekennzeichnet durch 2-Punkte/Vektoren
(define-record-procedures gerade-2pf
  make-gerade-2pf gerade-2pf?
  (gerade-2pf-punkt1       ; 1. Ortsvektor („Stützvektor")
   gerade-2pf-punkt2))     ; 2. Ortsvektor

(: make-gerade-prf (vektor vektor -> gerade-prf))
; Eine Gerade ist gekennzeichnet durch einen Punkt
; und einen Richtungsvektor
(define-record-procedures gerade-prf
  make-gerade-prf gerade-prf?
  (gerade-prf-punkt        ; Ortsvektor („Stützvektor")
   gerade-prf-richtung))   ; Richtungsvektor
```

Beispiele (für Testfälle im Folgenden)

```
(define g0a (make-gerade-2pf (list 1 0 -4) (list -1 1 -1)))
(define g0b (make-gerade-prf (list 1 0 -4) (list -2 1 3)))
(define g1 (make-gerade-prf (list 0 0 0) (list 1 0 1)))
(define g2 (make-gerade-prf (list 1 0 0) (list -1 0 1)))
(define g3 (make-gerade-prf (list 0 1 0) (list 1 0 1)))
(define g4 (make-gerade-prf (list 1 0 0) (list -1 0 1)))
```

Es bleibt die Frage, wie man eine „Geradengleichung" erhält, und zwar zunächst in Parmeterform. Es ist also eine Funktion zu konstruieren, die zu einer gegebenen Geraden g eine Geradengleichung produziert, z.B. in der „Punkt-Richtungsform" 2.:

```
(: g-gleichung-mit-parameter (gerade -> (real -> vektor)))
```

Es handelt sich also um eine Funktion, die Funktionen erzeugt:

```
(define g-gleichung-mit-parameter
  (lambda (g)
    <Parameterform von g>))
```

Da sich die beiden Parameterformen 1. und 2. strukturell nicht unterscheiden, sondern nur durch die Art der kennzeichnenden Vektoren, können wir abstrahieren und sie als eine anonyme Funktion formulieren

```
; Parameterform von 2.:
(lambda (t)
  (vektor+ (gerade-punkt g) (s-multiplikation t (gerade-richtung g))))
```

wobei

```
(: gerade-punkt (gerade -> vektor)) ; Ortsvektor oder „Stützvektor"

(check-expect (gerade-punkt g0a) (list 1 0 -4))
(check-expect (gerade-punkt g0b) (list 1 0 -4))
```

und

```
(: gerade-richtung (gerade -> vektor))

(check-expect (gerade-richtung g0a) (list -2 1 3))
(check-expect (gerade-richtung g0b) (list -2 1 3))
```

zwei Funktionen sind, die aus `g` als gemischtem Datentyp jeweils die zutreffenden Vektoren extrahieren.
Dabei muss jeweils eine Fallunterscheidung durchgeführt werden:

```
(define gerade-punkt
  (lambda (g)
    (cond
      (< 2-Punkteform?> .....)
      (< Punkt-Richtungsform?> .....))))
```

und

```
(define gerade-richtung
  (lambda (g)
    (cond
      (< 2-Punkteform?> .....)
      (< Punkt-Richtungsform?> .....))))
```

Durch Abfragen mit den Prädikatoren `gerade-2pf?` und `gerade-prf?` und Auswählen der zutreffenden Verbund-Komponenten von `g` erhält man für `gerade-punkt`:

```
(define gerade-punkt
  (lambda (g)
    (cond
      ((gerade-2pf? g)
       (gerade-2pf-punkt1 g))  ; ...-punkt2 wäre auch möglich
      ((gerade-prf? g)
       (gerade-prf-punkt g)))))
```

Bei der 2-Punkte-Form ergibt die Differenz `(vektor- (gerade-2pf-punkt2 g) (gerade-2pf-punkt1 g))` der beiden Ortsvektoren einen Richtungsvektor, womit diese auf die Punkt-Richtungsform zurückgeführt wird:

```
(define gerade-richtung
  (lambda (g)
    (cond
      ((gerade-2pf? g)
       (vektor- (gerade-2pf-punkt2 g) (gerade-2pf-punkt1 g)))
      ((gerade-prf? g)
       (gerade-prf-richtung g)))))
```

Damit ist `g-gleichung-mit-parameter` komplett:

```
(define g-gleichung-mit-parameter
  (lambda (g)
    (lambda (t)
      (vektor+ (gerade-punkt g) (s-multiplikation t (gerade-richtung g))))))
```

Listing 4: Geradengleichung mit Parameter

was man mit Testfällen bestätigen kann:

```
(check-expect ((g-gleichung-mit-parameter g0a) 1) (list -1 1 -1))
(check-expect ((g-gleichung-mit-parameter g0b) 1) (list -1 1 -1))
(check-expect ((g-gleichung-mit-parameter g2) 3) (list -2 0 3))
```

Wie bereits oben erwähnt, kann auch die parameterfreie Geradendarstellung (3. in der o.a. Übersicht), nämlich $g: \quad \vec{v_0} \times (\vec{p} - \vec{a}) = \vec{0}$ für manche Anwendungen nützlich sein, z.B. Bestimmung des Abstandes Gerade-Punkt mit $d(g, P) = |\vec{v_0} \times (\vec{p} - \vec{a})|$ oder Klärung von $P \in g$?

Für die entsprechende Funktion

```
(: g-gleichung-ohne-parameter (gerade -> (vektor -> vektor)))
```

Die Gleichung stellt einen BOOLEschen Ausdruck dar, der je nach dem, ob $P \in g$. Deshalb wird nur der Term der linken Seite (ohne Betrag) in Scheme codiert.

```
(define g-gleichung-ohne-parameter  ;
  (lambda (g)
    (....<Bestimmung des Einheits-Richtungsvektors>....
      (lambda (p)
        (vektor-produkt v0 (vektor- p (gerade-punkt g)))))))
```

Der Einhheits-Richtungsvektor $\vec{v_0} = \dfrac{v}{|\vec{v}|}$ wird als lokale Variable aus dem Richtungsvektor der Geraden

```
(let ((v0 (s-multiplikation
           (/ 1 (betrag (gerade-richtung g)))
           (gerade-richtung g))))
```

ermittelt, insgesamt

```
(define g-gleichung-ohne-parameter  ;
  (lambda (g)
    (let
        ((v0 (s-multiplikation            ; v0 = Einheits-Richtungs-Vektor
              (/ 1 (betrag (gerade-richtung g)))
              (gerade-richtung g))))
      (lambda (p)
        (vektor-produkt v0 (vektor- p (gerade-punkt g)))))))
```

Listing 5: Geradengleichung ohne Parameter

3 Anwendungen: Konstruktion der Analyse-Funktionen (Auswahl)

Hinweis: Die Anwendungsfunktionen werden i.A. nicht gemäß der ausführlichen Konstruktionsanleitung entwickelt.

Lineare Abhängigkeit, allgemein
Bekanntlich ist im $\mathbb{R}^3$ eine Menge von $n \geq 4$ Vektoren stets l.a. und der Fall $n = 1$ ist trivial.
Zum Prüfen auf Lineare Abhängigkeit bieten folgende Möglichkeiten:

1. Je eine eigene Funktion für $n = 2$ und $n = 3$
2. Eine Funktion mit Fallunterscheidung für $n = 2$ und $n = 3$
3. Eine Funktion für $n \geq 1$

Im Folgenden werden die Fälle 1. und 3. vorgestellt

Lineare Abhängigkeit von 2 Vektoren
Es bieten sich zwei Varianten an:

- mit *Skalarprodukt*
  Eine Überprüfung der Linearen Abhängigkeit einer Menge von zwei Vektoren mittels des Satzes

  $$\{\vec{a},\ \vec{b}\} \quad l.a. \quad \Leftrightarrow \quad |\vec{a} \cdot \vec{b}| = |\vec{a}||\vec{b}|$$

  liefert i.A. falsche Ergebnisse, weil $|\vec{a} \cdot \vec{b}|$ exakte Scheme-Zahlen, dagegen $|\vec{a}|$ oder $|\vec{b}|$ wegen der Quadratwurzel in `betrag` inexakte Scheme-Zahlen. Deshalb müssen beide Seiten quadriert werden, d.h. $|\vec{a} \cdot \vec{b}|^2 = (\vec{a} \cdot \vec{a})(\vec{b} \cdot \vec{b})$ wird als Kriterium genommen:

  ```
  (: la-2? (vektor vektor -> boolean))
  ; prüft, ob 2 Vektoren l.a. sind, mittels |v1 v2|= |v1||v2|
  ; da |ab| exakt, aber |a||b| i.a. inexakt ist, erfolgt Quadratur

  (check-expect (la-2? (list 1 2 3) (list 1 0 1)) #f)
  (check-expect (la-2? (list 1 2 3) (list 0 0 0)) #t)

  (define la-2?
    (lambda (v1 v2)
      (= (* (skalar-produkt v1 v2) (skalar-produkt v1 v2))
         (* (skalar-produkt v1 v1) (skalar-produkt v2 v2)))))
  ```

  Hier ist abzuwägen, ob es Vorteile im Hinblick auf das Folgende bringt, die Funktion nicht mit den beiden Einzelvektoren als Parameter zu modellieren, sondern mit einer Liste aus den beiden Vektoren, also mit einem Vertrag der Form `(: la-2? ((list vektor) -> boolean))`

- mit *Vektorprodukt*
  Das Kriterium

  $$\{\vec{a},\ \vec{b}\} \quad l.a. \quad \Leftrightarrow \quad \vec{a} \times \vec{b} = \vec{0}$$

  kann hier nicht direkt umgesetzt werden, da auf der Sprachebene „Die Macht der Abstraktion angepasst" `equal?` nicht zur Verfügung steht. Da somit nur Zahlen mit „=" verglichen werden können, erfolgt ein Übergang auf den Betrag:

  $$\{\vec{a},\ \vec{b}\} \quad l.a. \quad \Leftrightarrow \quad |\vec{a} \times \vec{b}| = 0$$

  ```
  (: la-2? (vektor vektor -> boolean))
  ; prüft, ob 2 Vektoren l.a. sind, mittels |v1 x v2|= 0

  (check-expect (la-2? (list 1 2 3) (list 1 0 1)) #f)
  ```

```
(check-expect (la-2? (list 1 2 3) (list 0 0 0)) #t)

(define la-2?
  (lambda (v1 v2)
    (zero? (betrag (vektor-produkt v1 v2)))))
```

Auch hier ist abzuwägen, ob es Vorteile im Hinblick auf das Folgende bringt, die Funktion nicht mit den beiden Einzelvektoren als Parameter zu modellieren, sondern mit einer Liste aus den beiden Vektoren, also mit einem Vertrag der Form `(: la-2? ((list vektor) -> boolean))`

Lineare Abhängigkeit von 3 Vektoren

Eine einfache Überprüfung kann mit dem Spatprodukt oder der Determinante der Vektoren erfolgen:

$$\{\vec{a},\ \vec{b}\} \quad l.a. \quad \Leftrightarrow \quad (\vec{a} \times \vec{b}) \cdot \vec{c} = det(\vec{a}, \vec{b}, \vec{c}) = 0$$

also

```
(: la-3? (vektor vektor vektor -> boolean))

(check-expect (la-3? (list 1 0 0) (list 0 1 0) (list 0 0 1)) #f)
(check-expect (la-3? (list 1 0 0) (list 0 1 0) (list 1 1 0)) #t)

(define la-3?
  (lambda (v1 v2 v3)
    (zero? (spat-produkt v1 v2 v3))))
```

Lineare Abhängigkeit von n Vektoren

Da die Anzahl der Vektoren variiert, werden keine Einzelvektoren als Parameter verwendet, sondern eine Liste von Vektoren:

```
(: la-n? ((list vektor) -> boolean))
```

Es werden zwei Varianten vorgestellt mit folgenden Testfällen:

```
(check-expect (la-n? (list (list 1 2 3) (list 2 3 4) (list 2 4 6))) #t)
(check-expect (la-n? (list (list 1 2 3) (list 2 3 4) (list 1 4 6))) #f)
(check-expect (la-n? (list (list 1 2 3) (list 2 3 4) (list 0 0 0))) #t)
(check-expect (la-n? (list (list 1 2 3) (list 1 4 6))) #f)
(check-expect (la-n? (list (list 1 2 3) (list 2 4 6))) #t)
(check-expect (la-n? (list (list 1 2 3) (list 0 0 0))) #t)
(check-expect (la-n? (list (list 1 2 3))) #f)   ; n=1, falls nicht Nullvektor
(check-expect (la-n? (list (list 0 0 0))) #t)   ; n=1, Nullvektor
```

- ohne *Vektorprodukt*

  Es müssen 4 Fälle unterschieden werden, die Fälle $n \geq 4$ und $n = 1$ sind trivial:

  ```
  (define la-n?
    (lambda (liste)
      (cond
        ((>= (length liste) 4) #t) ; n>=4 im R3 immer l.a.
        ((= (length liste) 1) (zero? (betrag (first liste))))
          ; n=1: Nullvektor ist l.a., sonst l.u.
        ((= (length liste) 2) .........) ; n=2
        (else    ; =3
         .........))))
  ```

  Im Falle $n = 2$ wird die o.a. Funktion `la-2?` auf die beiden Elemente der Liste angewendet. Für den `else`-Fall, also $n = 3$, wird folgende Tatsache benutzt, dass eine Menge von 3 Vektoren genau dann l.u. ist, wenn alle Vektoren paarweise l.u. sind, oder dass eine Menge von 3 Vektoren genau dann l.a., wenn mindestens ein Paar l.a. ist, was man wieder mit `la-2?` überprüfen kann.

```
(define la-n?
  (lambda (liste)
    (cond
      ((>= (length liste) 4) #t) ; n>=4 im R3 immer l.a.
      ((= (length liste) 1) (zero? (betrag (first liste))))
        ; n=1: Nullvektor ist l.a., sonst l.u.
      ((= (length liste) 2) (la-2? (first liste) (first (rest liste)))) ; n=2
      (else    ; =3
       (or
        (la-2? (first liste) (first (rest liste)))
        (la-2? (first liste) (first (rest (rest liste))))
        (la-2? (first (rest liste)) (first (rest (rest liste))))))))))
```

- mit *Vektorprodukt*

  Diese Variante unterscheidet sich von Variante 1 lediglich dadurch, dass der `else`-Fall, also $n = 3$, durch die schon definierte Funktion `la-3?` ersetzt, in der das Spatprodukt steckt, das wiederum das Vektorprodukt enthält:

```
(define la-n?
  (lambda (liste)
    (cond
      ((>= (length liste) 4) #t) ; n>=4 im R3 immer l.a.
      ((= (length liste) 1) (zero? (betrag (first liste))))
        ; n=1: Nullvektor ist l.a., sonst l.u.
      ((= (length liste) 2) (la-2? (first liste) (first (rest liste)))) ; n=2
      (else    ; =3
       (la-3?
        (first liste) (first (rest liste)) (first (rest (rest liste)))))))))
```

Winkel zwischen zwei Vektoren in Bogenmaß

```
(: winkel-rad (vektor vektor -> real))

(check-expect (winkel-rad (list 1 0 0) (list 2 0 0)) 0)
(check-within (winkel-rad (list 1 0 0) (list 0 1 0)) 1.57 0.01) ; Pi/2 inexakt

(define winkel-rad
  (lambda (v1 v2)
    (acos
     (/
      (skalar-produkt v1 v2)
      (* (betrag v1) (betrag v2))))))
```

Winkel zwischen zwei Vektoren in Grad

Da die Kreiszahl π in DMdA leider nicht implementiert ist, muss sie selbst definiert werden:

```
(define kreiszahl 3.141592653589793)
(: winkel-grad (vektor vektor -> real))

(check-expect (winkel-grad (list 1 0 0) (list 2 0 0)) 0)
(check-within (winkel-grad (list 1 0 0) (list 0 1 0)) 90 0.01)

(define winkel-grad
  (lambda (v1 v2)
    (* 180 (/ (winkel-rad v1 v2) kreiszahl))))
```

Winkel zwischen zwei Geraden in Bogenmaß

```
(: g-winkel-rad (gerade gerade -> real))

(check-within (g-winkel-rad g0a g1) 1.38 0.01)
(check-within (g-winkel-rad g1 g2) 1.57 0.01) ; pi/2 = inexakte Zahl !
```

```
(check-within (g-winkel-rad g1 g3) 0 0.01) ; inexakt

(define g-winkel-rad
  (lambda (g h)
    (winkel-rad (gerade-richtung g) (gerade-richtung h))))
```

Senkrechte Geraden

```
(: senkrecht? (gerade gerade -> boolean))

(check-expect (senkrecht? g0a g0b) #f)
(check-expect (senkrecht? g3 g4) #t)

(define senkrecht?
  (lambda (g h)
    (zero? (skalar-produkt (gerade-richtung g) (gerade-richtung h)))))
```

Parallele Geraden

```
(: parallel? (gerade gerade -> boolean))

(check-expect (parallel? g0a g0b) #t)
(check-expect (parallel? g1 g2) #f)
(check-expect (parallel? g1 g3) #t)
(check-expect (parallel? g1 g1) #t)

(define parallel?
  (lambda (g h)
    (la-2? (gerade-richtung g) (gerade-richtung h))))
```

Lage zweier Geraden zueinander

```
(: geraden-lage (gerade gerade -> string))

(check-expect (geraden-lage g1 g2) "g, h windschief")
(check-expect (geraden-lage g1 g3) "g||h & g<>h")
(check-expect (geraden-lage g2 g2) "g=h")

(define geraden-lage
  (lambda (g h)
    (cond
      ((la-2? (gerade-richtung g) (gerade-richtung h))
       (cond
         ((la-2? (gerade-richtung g)
                 (vektor- (gerade-punkt g) (gerade-punkt h))) "g=h")
         (else
          "g||h & g<>h")))
      (else
       (cond
         ((la-n? (list (gerade-richtung g) (gerade-richtung h)
                       (vektor- (gerade-punkt g) (gerade-punkt h))))
          "g^h<>{}")
         (else
          "g, h windschief"))))))
```

Abstand Gerade-Punkt

```
(: abstand-g-P (gerade vektor -> real))

(check-within (abstand-g-P g0A (list 0 0 0)) 1.73 0.01)
(check-within (abstand-g-P g2 (list 0 0 0)) 0.707 0.01 )  ; genau: (/ (sqrt 2) 2)
(check-expect (abstand-g-P g1 (list 0 0 0)) 0)
(check-within (abstand-g-P g3 (list 0 0 0)) 1 0.01)

(define abstand-g-P
```

```
  (lambda (g P)
    (betrag ((g-gleichung-ohne-parameter g) P))))
```

Abstand zweier Geraden

Bekanntlich haben zwei Geraden

$$g : \vec{p} = \vec{a} + t \cdot \vec{v} \qquad \text{und} \qquad h : \vec{p} = \vec{b} + s \cdot \vec{u}$$

den Abstand

$$d(g,h) = \begin{cases} |\vec{v_0} \times (\vec{a} - \vec{b})| & \text{falls} \quad g \mid\mid h \\ |(\vec{v} \times \vec{u})_0 \cdot (\vec{a} - \vec{b})| & \text{sonst} \end{cases}$$

Das ergibt

```
(: abstand-g-g (gerade gerade -> real))

(check-within (abstand-g-g g0A g1) 0.96 0.01)
(check-expect (abstand-g-g g1 g4) 0)
(check-within (abstand-g-g g1 g3) 1 0.01 )  ; inexakt

(define abstand-g-g
  (lambda (g h)
    (let ((differenz (vektor- (gerade-punkt g) (gerade-punkt h)))
          (v0 (s-multiplikation
               (/ 1 (betrag (gerade-richtung g)))
               (gerade-richtung g))))
      (cond
        ((parallel? g h) (betrag (vektor-produkt v0 differenz)))
        (else
         (let ((z (vektor-produkt (gerade-richtung g) (gerade-richtung h))))
           (abs (skalar-produkt
                 (s-multiplikation (/ 1 (betrag z)) z)
                 differenz)))))))))
```

4 Ausblick

Für weitere Punktmengen wie Ebene und Kugel oder gar Kegelschnitte kann in ähnlicher Weise vorgegangen werden.

Bei der *Ebene* werden die Modellierungsmöglichkeiten komplexer:
Es gibt nicht nur drei Parameterformen (3 Ortsvektoren, 2 Ortsvektoren und 1 Richtungsvektor, 1 Ortsvektor und 2 Richtungsvektoren), sondern auch zwei parameterfreie Formen: die Normalformen mit Skalarprodukt, wie z.B. die HESSEsche Normalenformen, und Formen mit Vektorprodukt $E : (\vec{v} \times \vec{u}) \cdot (\vec{p} - \vec{s}) = 0$.
Entsprechend wächst der Umfang der möglichen Anwendungsfunktionen: Zu den Untersuchungen zweier Ebenen wie z.B. *Lage zweier Ebenen zueinander* oder *Winkel zwischen Ebenen* kommen die Mischfälle Gerade-Ebene wie z.B. *Lage Geraden und Ebene zueinander* oder *Winkel zwischen Ebene und Gerade* usw.

Es gibt also noch viel zu tun.

Literatur

[1] *Klaeren, Herbert* und *Sperber, Michael*, **Die Macht der Abstraktion** – Einführung in die Programmierung –, 1. Auflage, Teubner, 2007, ISBN-Nr.978-3-8351-0155-5

[2] *Felleisen, M.*, u.a., **How to Design Programs** – An Introduction to Programming and Computing –, MIIT Press, 2001, ISBN-Nr. 0-262-06218-6, online-Version (engl.): *http://www.htdp.org*