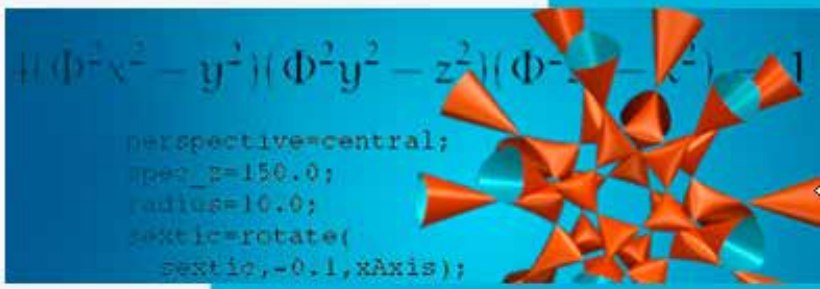
Philipps Universität Marburg

Deklarative Programmierung

Sommersemester 2018

Prof. Christoph Bockisch
(Programmiersprachen und –werkzeuge)
Steffen Dick, Alexander Bille, Johannes Frankenau,
Patrick Frömel, Niclas Schmidt, Jonas Stettin,
Robert Tran, Julian Velten



[Skript 6, 7]

Werte mit mehreren Eigenschaften

- Problemdomäne besteht aus Entitäten
- Repräsentation von Entität durch Wert
- Wert hat einen Datentyp

- Bisher:
  - Werte sind atomar
  - Repräsentieren genau eine Eigenschaft

Werte mit mehreren Eigenschaften

- Allgemein haben Entitäten mehrere Eigenschaften
- Trotzdem Repräsentation durch einen Wert!

- Beispiel:
  - Entität als Ergebnis einer Funktion
  - WorldState mit mehreren Eigenschaften

- Gödelisierung:
  - Codierung mehrerer Eigenschaften als ein atomarer Wert
  - Beispielsweise als Zeichenkette
  - In der Praxis: Konvertierung zu Aufwändig

Datentyp-Strukturen

- “Strukturen“ oder ”Records”
  - Kombination mehrerer Teilwerte zu einem Wert (Datum)
  - Jeder Teilwert einzeln zugreifbar

- Beispiel: Struktur Position
  - Teilwerte
    - x-Koordinate
    - y-Koordinate
  - Typ dieser Struktur: Number x Number

Entlehnt aus der Mathematik:
Kreuzprodukt von Mengen.
Daher Produkttyp

- Übrigens:
  Summentypen entsprechen der Vereinigung von Mengen:
  Tax: Number + String

Beispiel

- BSL stellt Struktur zur Repräsentation von Positionen bereit

- Eine posn ist ein Wert
- posn hat zwei Komponenten: x-Koordinate, y-Koordinate

- Die Struktur wird durch Funktionen definiert

- Erzeugung von Instanzen: make-posn
- Signatur Number Number → Posn

(make-posn 3 4)

Entwurf von Strukturen

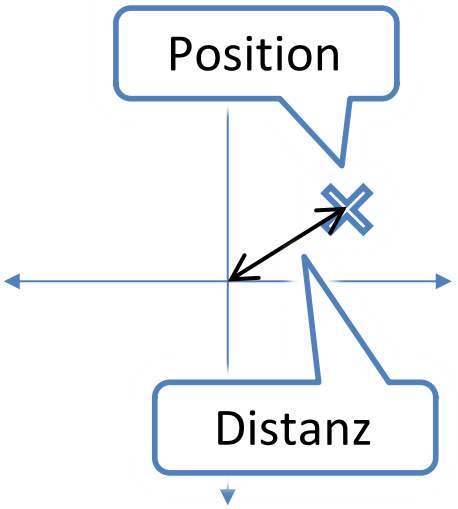
- Datentypen der Teilwerte
- Interpretation für die Teilwerte

```
(define-struct posn (x y))
; A Posn is a structure: (make-posn Number Number)
; interp. the number of pixels from left and from top
```

Instanzen von Strukturen sind Werte

- Verwendung von Instanzen von Strukturen
  - Als Argument
  - Als Rückgabewert
  - Genauso wie andere Werte (Zahlen, Strings, etc.)

```
; Posn -> Number
; to compute the distance of a-posn to the origin
(check-expect (distance-to-0 (make-posn 0 5)) 5)
(check-expect (distance-to-0 (make-posn 7 0)) 7)
(check-expect (distance-to-0 (make-posn 3 4)) 5)
(check-expect (distance-to-0 (make-posn 8 6)) 10)
(define (distance-to-0 a-posn) 0)
```



Nur ein Stub!

Verwendung von Struktur-Instanzen

- Funktionen, die Struktur definieren
  - Erzeugung von Instanzen
  - Zugriff auf Komponenten

- Zugriff auf x-Koordinate: posn-x
- Signatur Posn → Number

```
> (posn-x (make-posn 3 4))
3
```

- Zugriff auf y-Koordinate: posn-y
- Signatur Posn → Number

```
 > (posn-y (make-posn 3 4))
4
```

Definition von Strukturtypen

- Nicht fest eingebaut (nicht primitiv)
- Benutzerdefiniert
- Sprache bietet Mechanismus zur Definition

- BSL:

(define-struct *StructureName* (*FieldName*$_1$ ... *FieldName*$_2$))

- Beispiel:

(define-struct posn (x y))

Definition von Strukturtypen

- Bei Definition einer Struktur stellt Racket automatisch Hilfsfunktionen bereit

- Konstruktor
  - Erzeugt Instanz der Struktur
  - Name: make-*StructureName*
- Selektoren für jeden Teilwert
  - Extrahiert aus Struktur-Instanz die angegebene Komponente
  - Name: *StructureName-FieldName*
- Strukturprädikat
  - Überprüft ob ein Wert eine Instanz der Struktur ist
  - Name: *StructureName*?

Verschachtelte Strukturen

- Strukturen sind Datentypen
- Instanzen von Strukturen sind Werte

- Können überall verwendet werden wo Datentypen/Werte verlangt werden
- Auch in der Definition von Strukturen

Verschachtelte Strukturen

- Beispiel

```
(define-struct vel (deltax deltay))

; posn vel -> posn
; computes position of loc after applying v
(check-expect (move (make-posn 5 6) (make-vel 1 2))
(make-posn 6 8))
(define (move loc v)
   (make-posn
      (+ (posn-x loc) (vel-deltax v))
      (+ (posn-y loc) (vel-deltay v))))
```

Erwartet Position und Bewegung. Eigentlich zwei Eigenschaften einer Entität, bspw. Ball.

Verschachtelte Strukturen

- Beispiel: Ball hat die Eigenschaften
  - x-Koordinate, y-Koordinate, Delta-x und Delta-y
  - Definition als Struktur mit vier Eigenschaften:

```
(define-struct ball (x y deltax deltay))
```

- Problem
  - Zusammenhang der Felder geht verloren
  - Funktion move erwartet Posn und Vel
  - Instanzen hiervon müssen erst erzeugt werden

```
(move (make-posn (ball-x some-ball) (ball-y some-ball))
(make-vel (ball-deltax some-ball) (ball-deltay some-ball)))
```

Verschachtelte Strukturen

- Besser:
  - Verwendung der Strukturen Posn und Vel in der Definition von Ball
  - Erhalt von logischen Strukturen
  - Wiederverwendung von Hilfsfunktionen auf Teil-Strukturen

```
(define-struct ball (loc vel))
(define some-ball
   (make-ball (make-posn 5 6) (make-vel 1 2)))
(move (ball-loc some-ball) (ball-vel some-ball))
```

Wiederverwendung von Strukturen

```
(define-struct ball (loc vel))
; a Ball is a structure: (make-ball Posn Vel)
; interp. the position and velocity of a ball
```

Teildaten: Position und Velocity im Kommentar angegeben. Nicht verbindlich.

- Annahme bei Definition von Ball: 2-dimensionaler Raum
- Können wir anstatt Posn und Vel auch andere Typen verwenden?
  - Number: Position in 1-dimensionalen Raum
  - Number: Geschwindigkeit im 1-dimensionalen Raum

Das geht, aber ...

```
; a Ball1d is a structure: (make-ball Number Number)
; interp. the position and velocity of a 1D ball
```

Wiederverwendung von Strukturen

- Wiederverwendung möglich bei demselben Aufbau
- Gemeinsamer Aufbau von
  - Position, Velocity, Ball
  - Paare von Werten
  - Z.B. möglich, alle Werte als Position zu codieren

```
; a Vel is a structure: (make-posn Number Number)
; interp. the velocity vector of a moving object

; a Ball is a structure: (make-posn Posn Vel)
; interp. the position and velocity of a ball
```

Wiederverwendung von Strukturen

- Bei Wiederverwendung
  - Datentypen können nicht durch Strukturprädikat unterschieden werden
  - Nicht alle Funktionen über Struktur können angewendet werden: Möglicherweise werden Erwartungen verletzt

```
; Ball -> Ball
; computes a new ball at a position of ball‘s posn after applying ball‘s vel
(define (move ball)
   (make-ball (make-posn (+ (posn-x (ball-loc ball))
                            (vel-deltax (ball-vel ball)))
                         (+ (posn-y (ball-loc ball))
                            (vel-deltay (ball-vel ball))))
              (ball-vel ball)))
(define some-ball (make-ball (make-posn 5 6) (make-vel 1 2)))
>(move some-ball)
(make-ball (make-posn 6 8) (make-vel 1 2))
```

Wiederverwendung von Strukturen

- Bei Wiederverwendung
  - Datentypen können nicht durch Strukturprädikat unterschieden werden
  - Nicht alle Funktionen über Struktur können angewendet werden: Möglicherweise werden Erwartungen verletzt

```
; Ball -> Ball
; computes a new ball at a position of ball‘s posn after applying ball‘s vel
(define (move ball)
   (make-ball (make-posn (+ (posn-x (ball-loc ball))
                            (vel-deltax (ball-vel ball)))
                         (+ (posn-y (ball-loc ball))
                            (vel-deltay (ball-vel ball))))
              (ball-vel ball)))
(define some-1d-ball (make-ball 1 2))
>(move some-1d-ball)
posn-x: expects a posn, given 1
```

Was passiert bei Wiederverwendung?

Wiederverwendung von Strukturen

- Wiederverwendung nur bei gemeinsamem semantischen Konzept

- Oder bei Fehlen von semantischem Konzept
- Beispiel Lisp: Darstellung aller Produkttypen durch geschachtelte Paare

- (define-struct cons-cell (car cdr))

Mittels cons-cell lassen sich beliebige Listen von beliebigen Werten bilden. Aber: kein Strukturprädikat.

Erste Komponente: Wert

Zweite Komponente: Weitere cons-cell oder “nil“

Entwurfsrezept mit Strukturen

- 1. Informationsrepräsentation
- Enthält die Informationsbeschreibung zusammengehörige Daten, müssen diese als Struktur gruppiert werden.
- Feld
  - Relevante Eigenschaft
  - Sinnvoll für alle Instanzen der Struktur
  - Interpretation der Daten
  - Beschreibung welche Daten erlaubt sind
- Erstellen von Beispielen
  - Bei Komponenten mit Summentyp:
    - Abdecken aller Werte (Enumerationstyp)
    - Abdecken aller Grenzen (Intervalltyp)

Entwurfsrezept mit Strukturen

- 3. Tests
- Bei einer Funktion mit Produkttyp als Argument
- Verwendung von Beispielwerten aus Schritt 1
- Bei Feldern mit Summentyp: Menge von Testdaten abhängig von enumerierten Werten/Intervallen
- Für Produkttyp: idealerweise Test für jede mögliche Kombination

Entwurfsrezept mit Strukturen

- Entwurfsrezept Schritt 4: Schablone
  - Verwendung der Funktionen zur Extraktion von Teilwerten

```
(define (distance-to-0 a-posn)
  (... (posn-x a-posn) ...
   ... (posn-y a-posn) ...))
```

Strukturen sind Abstraktionen. Greifen Sie daher nicht auf Eigenschaften der Eigenschaften zu! Definieren Sie ggf. Hilfsfunktionen.

- Implementierung der Body-Funktion

```
(define (distance-to-0 a-posn)
  (sqrt
    (+ (sqr (posn-x a-posn))
       (sqr (posn-y a-posn)))))
```

Entwurfsrezept mit Strukturen

- 6. Tests ausführen
- Testen direkt nachdem alle Header geschrieben sind

- Tests müssen jetzt Fehlschlagen (es sei denn ein Dummy-Wert entspricht zufällig dem erwarteten Wert)

- So kann überprüft werden dass Tests nicht zu schwach formuliert sind

- Der Code muss vollständig während der Tests abgedeckt sein (siehe Codefärbung in DrRacket)

Kombination von Summen- und Produkttypen

```
(define-struct gcircle (center radius))
; A GCircle is (make-gcircle Posn Number)
; interp. the geometrical representation of a circle

(define-struct grectangle (corner-ul corner-dr))
; A GRrectangle is (make-grectangle Posn Posn)
; interp. the geometrical representation of a rectangle
; where corner-ul is the upper left corner
; and corner-dr the down right corner

; A Shape is either:
; - a GCircle
; - a GRectangle
; interp. a geometrical shape representing a circle or a rectangle
```

Funktionen über Summen- und Produkttypen

- Hauptfunktion overlaps/3: überlappen drei Shapes?

```
; Shape Shape Shape -> Boolean
; determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
 (cond [(and (gcircle? shape1) (gcircle? shape2) (gcircle? shape3))
        ... overlaps-circle-circle-circle ...]
       [(and (gcircle? shape1) (gcircle? shape2) (grectangle? shape3))
        ... overlaps-circle-circle-rectangle ...]
       [(and (gcircle? shape1) (grectangle? shape2) (gcircle? shape3))
        ... overlaps-circle-rectangle-circle ...]
       ...)))
```

Schablone laut Entwurfsrezept.

Sinnvoll oder nötig?

Weder noch ...

Algebraische Datentypen

- Datentypen mit ähnlicher Struktur führen zu ähnlichen Hilfsfunktionen
- → DRY: don‘t repeat yourself!
- Abstraktion durch “algebraische Datentypen“

Algebraische Datentypen

- Kreis und Rechteck haben gemeinsame Eigenschaften und Operationen
  - Eingeschlossene Fläche
  - Position veränderbar
  - Größe änderbar
- Übergeordnetes Konzept: Form (oder Shape)
- Operationen können allgemein für übergeordneten Typ definiert werden

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2)  ...)
```

Algebraische Datentypen

- Aufbauend auf Funktionen des algebraischen Datentyps:
  - Entwicklung weiterer Funktionen
  - Unabhängig von konkretem Typ

```
; Shape Shape Shape -> Boolean
; determines whether the shapes overlap pairwise
(define (overlaps/3 shape1 shape2 shape3)
  (and
    (overlaps shape1 shape2)
    (overlaps shape1 shape3)
    (overlaps shape2 shape3)))
```

"Abstrakter Algorithmus"

Abstrakte Algorithmen

- Konkrete Algorithmen hängen von konkreten Datentypen ab
- Abstrakte Algorithmen sind unabhängig von konkretem Datentyp
- Sehr mächtige Code-Wiederverwendung
  - Abstrakte Algorithmen sind sofort auf alle Varianten (auch neue) von algebraischen Datentypen anwendbar …
  - ... solange die entsprechenden konkreten Algorithmen implementiert sind

Konkrete Algorithmen auf algebraischem Datentyp

- Abprüfen aller Varianten des Summentyps

```
; Shape Posn -> Boolean
; Determines whether a point is inside a shape
(define (point-inside shape point)
  (cond [(gcircle? shape) (point-inside-circle shape point)]
        [(grectangle? shape) (point-inside-rectangle shape point)]))
```

Warum nicht Implementierung direkt hier schreiben?

Konkrete Algorithmen auf algebraischem Datentyp

- Varianten von algebraischem Datentyp oft Produkttyp
- Implementierung der einzelnen Fälle wird dann schnell komplex
- Auslagerung in Hilfsfunktionen fördert Wiederverwendung
- Zwei Typen von konkreten Algorithmen
  1. Bei Parameter von algebraischem Datentyp:
     Erkennen der Variante und Weiterleitung (“Dispatch“) an Hilfsfunktion
  2. Bei Parametern von konkreten Typen (Varianten des Summentyps):
     Implementierung der Funktionalität für diesen Typ

Konkrete Algorithmen auf algebraischem Datentyp

- “Dispatch“

```
; Shape Posn -> Boolean
; Determines whether a point is inside a shape
(define (point-inside shape point)
   (cond [(gcircle? shape) (point-inside-circle shape point)]
         [(grectangle? shape) (point-inside-rectangle shape point)]))
```

- Funktionalität

```
; GCircle Posn -> Boolean
; Determines whether a point is inside a circle
(define (point-inside-circle circle point)
   (<= (vector-length (posn- (gcircle-center circle) point))
      (gcircle-radius circle)))
```

Dispatch bei mehreren Parametern vom algebraischen Datentyp

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2)
  (cond [( and (gcircle? shape1) (gcircle? shape2))
         (overlaps-circle-circle shape1 shape2)]
        [( and (grectangle? shape1) (grectangle? shape2))
         (overlaps-rectangle-rectangle shape1 shape2)]
        [( and (grectangle? shape1) (gcircle? shape2))
         (overlaps-rectangle-circle shape1 shape2)]
        [( and (gcircle? shape1) (grectangle? shape2))
         (overlaps-rectangle-circle shape2 shape1)]))
```

Dispatch bei mehreren Parametern vom algebraischen Datentyp

```
; Shape Shape -> Boolean
; determines whether shape1 overlaps with shape2
(define (overlaps shape1 shape2)
  (cond [( and (gcircle? shape1) (gcircle? shape2))
         (over
         [( and                                  ? shape2))
         (over                                   hape2)]
         [( and (grec          e? shape1) (gcircle? shape2))
         (overlaps-r ctangle-circle shape1 shape2)]
         [( and (gcircle? shape1) (grectangle? shape2))
         (overlaps-rectangle-circle shape2 shape1)]))
```

Z.B. bei Kommutativität muss nicht für jede Kombination eine eigene konkrete Funktion implementiert werden.

Konkrete Funktionalität für konkrete Datentypen

```
; GCircle GCircle -> Boolean
; determines whether c1 overlaps with c2
(define (overlaps-circle-circle c1 c2)
   ; Two circles overlap if and only if the distance of their
   ; centers is smaller than the sum of their radii
   (<= (vector-length (posn- (gcircle-center c1)
                             (gcircle-center c2)))
       (+ (gcircle-radius c1) (gcircle-radius c2))))

; GRectangle GRectangle -> Boolean
; determines whether r1 overlaps with r2
( define (overlaps-rectangle-rectangle r1 r2)  ...)

; GRectangle GCircle -> Boolean
; determines whether r overlaps with c
( define (overlaps-rectangle-circle r c)  ...)
```

Entwurfsrezept mit algebraischen Datentypen

- 1. Informationsrepräsentation
- Verschiedene Informationen werden als Produkttyp definiert
- Repräsentieren gemeinsames Konzept
- → Kombinieren zu algebraischem Datentyp mittels Summentyp

- Algebraische Datentypen können überall verwendet werden wo ein Typ verlangt wird
  - Also auch als Variante eines Summentyps oder im Feld eines Produkttyps
  - → Hierarchische Organisation von algebraischen Datentypen

Hierarchische Organisation von algebraischen Datentypen

- Verschachtelte Verwendung von algebraischen Datentypen ist gut!
  - Hoher Abstraktionsgrad
  - Mehr Wiederverwendung
- Probieren Sie immer gemeinsame Konzepte zu algebraischen Datentypen zusammenzufassen
  - → Gruppierung der Alternativen von Summentypen: kleinere Anzahl an Alternativen
  - → Gruppierung der Felder von Produkttypen: kleinere Anzahl an Feldern
  - Verbesserte Lesbarkeit

Entwurfsrezept mit algebraischen Datentypen

- 3. Tests
- Algebraische Datentypen sind auf oberster Ebene Summentypen
- Daher: zumindest ein Test pro Alternative
- Herausforderung: verschachtelte Nutzung von algebraischen Datentypen
  - → im Allgemeinen exponentieller Anstieg der möglichen Kombinationen mit Tiefe der Hierarchie

Entwurfsrezept mit algebraischen Datentypen

- 4. Schablone
- Algebraischer Datentyp als Parametertyp, allgemein: Summentyp von Produkttypen
- Ist es möglich die Funktion abstrakt zu formulieren?
  - Implementierung durch Aufruf existierender Funktionen
- Anderenfalls
  - Fallunterscheidung der Alternativen mit cond
  - Pro Fall: Aufruf einer Hilfsfunktion für konkreten Typ
  - Bei mehreren Parametern mit algebraischem Datentyp: **Hilfsfunktion pro Kombination**
  - Zumindest bei Produkttypen: Implementierung niemals direkt!

Entwurfsrezept mit algebraischen Datentypen

- 4. Schablone
- Algebraischer Datentyp als Pa
  Summentyp von Produkttypen
- Ist es möglich die Funktion abstrakt zu formulieren?
  - Implementierung durch Aufruf existierender Funktionen
- Anderenfalls
  - Fallunterscheidung der Alternativen mit cond
  - Pro Fall: Aufruf einer Hilfsfunktion für konkreten Typ
  - Bei mehreren Parametern mit algebraischem Datentyp: **Hilfsfunktion pro Kombination**
  - Zumindest bei Produkttypen: Implementierung niemals direkt!

D.h. ausschließlich durch Aufrufe von Funktionen, die auf allen Ausprägungen des algebraischen Datentyps sinnvoll sind.

Entwurfsrezept mit algebraischen Datentypen

- 7. Nachbearbeitung
- Vereinheitlichen Sie Gemeinsamkeiten in den Varianten
  - Bspw. gleiche Darstellung gleicher Daten
  - So können Hilfsfunktionen geteilt werden
- Vermeiden Sie breite, flache algebraische Datentypen
  - Gruppieren von Alternativen in Summentypen
  - Gruppieren von Feldern in Produkttypen
- Logische Gruppierung
  - Steigert Lesbarkeit
  - Steigert Wiederverwendbarkeit