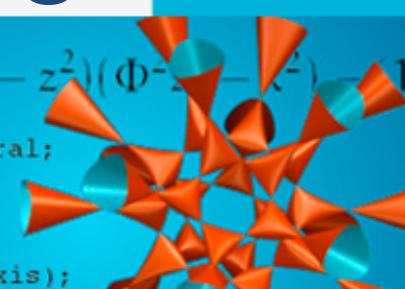


# Deklarative Programmierung

Sommersemester 2018

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



```
4(\Phi^2x^2 - y^2)(\Phi^2y^2 - z^2)(\Phi^2z^2 - x^2) - (1  
perspective=central;  
spec_p=150.0;  
radius=10.0;  
sextic=rotate(  
    sextic,-0.1,xAxis);
```

[Skript 16,17]

# Mächtigkeit von Programmiersprachen

- Jedes Problem lässt sich in jeder (general-purpose) Programmiersprache lösen
- Unterscheidung der Sprachen in Unterstützung für Entwurfstechniken
- Fallstudie: Unterstützung für Datendefinitionen und Signaturen

# Mächtigkeit von Programmiersprachen

- Vergleichskriterien

1. Sicherstellen, dass Verwendung von Werten/Funktionen zu deren Repräsentation passt
2. Zeitpunkt, zu dem Fehler gefunden werden
3. Bezug von Fehlermeldung auf Fehlerverursacher
4. Ausdrucksstärke der Signatursprache
5. Einschränkung der möglichen Programme
6. Konsistenz zwischen Datendefinitionen/Funktionen und dem Programmverhalten

# Klassifikation von Programmiersprachen

- Typ
  - Werte haben Typ
  - Funktionen erwarten Typen von Argumenten
- Klassifikation nach Umgang mit Typen
  - Ungetypte Sprachen
  - Dynamisch getypte Sprachen
  - Dynamisch überprüfte Signaturen und Kontrakte
  - Statisch getypte Sprachen

# Ungetypte Sprachen

- Intern werden alle Werte durch Zahlen bzw. Blöcke von Zahlen repräsentiert
- Daher ist bspw. die Addition von Zahl und String möglich, wenn auch nicht sinnvoll
- Beispiel: Assembler Sprachen

# Ungetypte Sprachen

- Bewertung

1. Sicherstellen passender Verwendung von Werten/Funktionen
  - Nicht durch die Sprache. Nur durch Coding-Disziplin
2. Zeitpunkt, zu dem Fehler gefunden werden
  - Sehr spät, also bei Verwendung falsch berechneter Werte
3. Bezug von Fehlermeldung auf Fehlerverursacher
  - Fehler tritt typischerweise in Programmteilen auf, die weit entfernt von der fehlerhaften Berechnung sind
4. Ausdrucksstärke der Signatursprache
  - Nicht vorhanden
5. Einschränkung der möglichen Programme
  - Keine Einschränkungen
6. Konsistenz zwischen Datendefinitionen/Funktionen und dem Programmverhalten
  - Keine

# Dynamisch getypte Sprachen

- Werte haben einen Typ
- Zuordnung von Wert zu Typ existiert zu Laufzeit, kann abgefragt werden
- Arten von Typen
  - Eingebaute (primitive) Typen
  - Benutzer-definierte Typen
- Beispiel: Racket
  - Abfrage des Typs bspw.: `boolean?`, `number?`, `string?`, `symbol?` oder Strukturprädikate

# Dynamisch getypte Sprachen

- Bei primitiven Operationen
  - Laufzeitsystem überprüft Anwendbarkeit auf Werte der gegebenen Typen
- Beispiel:  $(+ \ x \ y)$ 
  - Überprüfen, dass  $x$  und  $y$  Zahlen sind
  - Bei Verletzung Fehlermeldung anstatt z.B. Boolean Wert als Zahl zu interpretieren
- Gilt nicht (automatisch) für Benutzer-definierte Datentypen und Funktionen
  - Wir haben Signaturen bisher nur als Kommentar angegeben



# Dynamisch getypte Sprachen

- Typinformation nur für eingebaute Arten von Typen vorhanden
  - Bspw. primitive Typen oder Structs
- Keine Typinformation für nur logisch definierte Typen
  - Bspw. keine Unterscheidung, ob Zahl als Temperatur oder Länge verwendet wird

# Dynamisch getypte Sprachen

; Number (list-of Number) -> (list-of Number)

; returns the remainder of xs after first occurrence of x, or empty otherwise

```
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs))))))
```

```
> (rest-after 5 (list 1 2 3 4))
```

```
'()
```

```
> (rest-after 2 (list 1 2 3 4))
```

```
'(3 4)
```

# Dynamisch getypte Sprachen

; Number (list-of Number) -> (list-of Number)  
 ; returns the remainder of xs after first occurrence of x, or empty otherwise

```
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs))))))
```

```
> (rest-after 5 (list 1 2 3 4))
```

```
'()
```

```
> (rest-after 2 (list 1 2 3 4))
```

```
'(3 4)
```

```
> (rest-after 2 (list "eins" "zwei" "drei"))
```

**=: expects a number as 2nd argument, given "eins"**

```
> (rest-after 2 (list 1 2 "drei" "vier"))
```

```
'("drei" "vier")
```

Definition der Signatur

Verletzung der Signatur  
während Auswertung  
festgestellt.  
Laufzeitfehler

Nicht alle Verletzungen der  
Signatur werden während  
Funktionsausführung entdeckt.

# Dynamisch getypte Sprachen

; Number (list-of Number) -> (list-of Number)

; returns the remainder of xs after first occurrence of x, or empty otherwise

```
(define (rest-after x xs)
  (if (empty? xs)
      empty
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs))))))
```

> (rest-after 5 (list 1 2 3 4))

'()

> (rest-after 2 (list 1 2 3 4))

'(3 4)

> (rest-after 2 (list "eins" "zwei" "drei" "vier"))

**=: expects a number as 2nd argument**

> (rest-after 2 (list 1 2 "drei" "vier"))

'("drei" "vier")

Verletzung der Signatur  
durch den Aufrufer

# Dynamisch getypte Sprachen

; Number (list-of Number) -> (list-of Number)

; returns the remainder of xs after first occurrence of x, or empty otherwise

(define (rest-after x xs)

(if (empty? xs)

"not a list"

(if (= x (first xs))

(rest xs)

(rest-after x (rest xs))))

> (rest-after 5 (list 1 2 3 4))

"not a list"

> (cons 6 (rest-after 5 (list 1 2 3 4)))

cons: second argument must be a list, but received 6 and "not a list"

Laufzeitfehler bei Verwendung  
des Ergebniswerts

Fehler kann sehr viel später  
auftreten. Daher: Fehler  
kann nicht zu rest-after  
zurückverfolgt werden

# Dynamisch getypte Sprachen

- Bewertung

1. Sicherstellen passender Verwendung von Werten/Funktionen
  - Nur für primitive (und Struct-) Typen und Funktionen
2. Zeitpunkt, zu dem Fehler gefunden werden
  - Laufzeit
3. Bezug von Fehlermeldung auf Fehlerverursacher
  - Fehler tritt unter Umständen in Programmteilen auf, die weit entfernt von der fehlerhaften Berechnung sind
4. Ausdrucksstärke der Signatursprache
  - Nicht vorhanden
5. Einschränkung der möglichen Programme
  - So gut wie keine
6. Konsistenz zwischen Datendefinitionen/Funktionen und dem Programmverhalten
  - Eingeschränkt

# Dynamisch überprüfte Signaturen und Contracts

- Definition von Signaturen und Datendefinition als Programm
- Überprüfung der Signatur zur Laufzeit

; a list-of-numbers is either:

; - empty

; - (cons Number list-of numbers)

; [X] (list-of X) -> Boolean

; checks whether xs contains only numbers

(define (list-of-numbers? xs)

(if (empty? xs)

true

(and (number? (first xs))

(list-of-numbers? (rest xs))))))

Benutzer-definierte Prädikate

# Dynamisch überprüfte Signaturen und Contracts

- Überprüfung der korrekten Argument-Typen
  - Verwendung von primitiven Prädikaten zusammen mit Benutzerdefinierten Prädikaten

; Number (list-of Number) -> (list-of Number)

; dynamically checked version of rest-after

```
(define (rest-after/checked x xs)
```

```
  (if (number? x)
```

```
      (if (and (list? xs)
```

```
          (list-of-numbers? xs)))
```

```
      (if (list-of-numbers? (rest-after x xs))
```

```
          (rest-after x xs)
```

```
          (error "function must return list-of-numbers")))
```

```
      (error "second arg must be list-of-numbers"))
```

```
      (error "first arg must be a number"))))
```

Überprüfung der  
Argument- und  
Rückgabetypen

Bei Erfolg Aufruf  
der Funktion

Fehler-  
meldungen



# Dynamisch überprüfte Signaturen und Contracts

> (rest-after/checked 2 (list 1 2 3 4))  
'(3 4)

Bei korrekter Verwendung:  
Verhalten wie zuvor

> (rest-after/checked "x" (list 1 2 3 4))  
first arg must be a number

Bei falscher Verwendung:  
Fehlermeldung bei  
Funktionsaufruf

> (rest-after/checked 2 (list 1 2 "drei" 4))  
second arg must be list-of-numbers

Einige zuvor  
erfolgreiche  
Funktionsaufrufe  
werden nun  
unterbunden

# Dynamisch überprüfte Signaturen und Contracts

```
> (rest-after/checked 2 (list 1 2 "drei" 4))
```

second arg must be list-of-numbers

Einige zuvor erfolgreiche Funktionsaufrufe  
werden nun unterbunden

Schlecht oder gut?

Verhindert späteren Fehler bei  
Verwendung der Ergebniswerts

# Dynamisch überprüfte Signaturen und Contracts

- Überprüfung von Kontrakten mühselig
- Sprache für Definition von Kontrakten
  - Anstelle von Implementierung in Funktionskörper
  - Spezifikation von Kontrakten an Schnittstelle

```

(provide
  (contract-out
    [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))

```

Bedingung für 1. Argument

Bedingung für 2. Argument

Bedingung für Ergebniswert

# Dynamisch überprüfte Signaturen und Contracts

- Modulkonzept von Racket
  - Datei als Modul-Granularität
  - Modulname: Dateiname

Spracheinstellung!

```
#lang racket
(provide
  (contract-out
    [rest-after (-> number? (listof number?) (listof number?))]))
(define (rest-after x xs)
  (if (empty? xs)
      "not a list"
      (if (= x (first xs))
          (rest xs)
          (rest-after x (rest xs)))))
```

heinz.rkt

# Dynamisch überprüfte Signaturen und Contracts

- Modulkonzept von Racket
  - Zur Verwendung eines Moduls muss dieses "importiert" werden (require)

```
#lang racket
```

elke.rkt

```
(require "heinz.rkt")
```

```
(rest-after "x" (list 1 2 3 4))
```

Verletzung des Kontrakts

# Dynamisch überprüfte Signaturen und Contracts

(rest-after "x" (list 1 2 3 4))

rest-after: contract violation

expected: number?

given: "x"

in: the 1st argument of

(->

number?

(listof number?)

(listof number?))

contract from: /Users/klaus/heinz.rkt

blaming: /Users/klaus/elke.rkt

(assuming the contract is correct)

at: /Users/klaus/heinz.rkt:3.24

Ausführliche Erläuterung  
der Kontraktverletzung

Verantwortlich für  
Kontraktverletzung

# Dynamisch überprüfte Signaturen und Contracts

(rest-after 5 (list 1 2 3 4))

rest-after: broke its contract

promised: "list?"

produced: "not a list"

in: the range of

(->

number?

(listof number?)

(listof number?))

contract from: /Users/klaus/heinz.rkt

blaming: /Users/klaus/heinz.rkt

(assuming the contract is correct)

at: /Users/klaus/heinz.rkt:3.24

Auch Verletzung des  
Kontrakts für Rückgabewert  
wird überprüft

Verantwortlich für  
Kontraktverletzung

# Dynamisch überprüfte Signaturen und Contracts

- Kontrakte werden nur bei Programmausführung überprüft
- Keine Sicherheit, dass alle Kontraktverletzungen gefunden werden
  - Neue Programmausführung kann neue Kontraktverletzungen hervorbringen
- Kontrakte können nur Bedingungen überprüfen, die berechnet werden können



# Dynamisch überprüfte Signaturen und Contracts

- Bewertung

1. Sicherstellen passender Verwendung von Werten/Funktionen
  - Ja
2. Zeitpunkt, zu dem Fehler gefunden werden
  - Laufzeit
3. Bezug von Fehlermeldung auf Fehlerverursacher
  - Starker Bezug
4. Ausdrucksstärke der Signatursprache
  - Beschränkt auf berechenbare Bedingungen
5. Einschränkung der möglichen Programme
  - Stärker als bei dynamisch getypten Signaturen
6. Konsistenz zwischen Datendefinitionen/Funktionen und dem Programmverhalten
  - Sofern Kontrakt korrekt spezifiziert ist

# Statisch getypte Sprachen

- Typsystem
  - Überprüfung der Signaturen und Funktionsaufrufe vor Laufzeit
  - Überprüfung aller möglichen Ausführungen
- Kompositionalität
  - Überprüfung nur abhängig von Modul selbst und den Typen/Signaturen der direkt verwendeten Module
- Eigenschaften
  1. Bei erfolgreichem Typcheck ("wohlgetyptes" Programm): Bei keiner Ausführung kann ein Typfehler auftreten
  2. Es gibt Programme, die abgelehnt werden, obwohl es Ausführungen ohne Typfehler geben kann
    - Theorem von Rice: nicht-triviale Verhaltenseigenschaften sind nicht entscheidbar

# Statisch getypte Sprachen

- Statisch getypte Variante von Racket
  - Sprachlevel: Typed Racket

#lang typed/racket

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))  
(define (rest-after x xs)  
  (if (empty? xs)  
      empty  
      (if (= x (first xs))  
          (rest xs)  
          (rest-after x (rest xs)))))
```

Statische Deklaration  
der Funktionssignatur

# Statisch getypte Sprachen

```
> (rest-after 2 (list 1 2 3 4))
```

```
- : (Listof Integer)
```

```
'(3 4)
```

```
> (rest-after "x" (list 1 2 3 4))
```

```
eval:5:0: Type Checker: type mismatch  
expected: Integer
```

```
given: String
```

```
in: "x"
```

```
> (rest-after 2 (list 1 2 "drei" 4))
```

```
eval:6:0: Type Checker: type mismatch  
expected: (Listof Integer)
```

```
given: (List One Positive-Byte String Positive-Byte)
```

```
in: (list 1 2 "drei" 4)
```

# Statisch getypte Sprachen

- Typprüfung auch möglich, wenn Programm Laufzeitfehler enthält

```
> (:print-type (rest-after (/ 1 0) (list 1 2 3 4)))  
(Listof Integer)
```

# Statisch getypte Sprachen

- Typprüfung der Funktionsdefinition ohne Aufruf

```
(: rest-after (-> Integer (Listof Integer) (Listof Integer)))  
(define (rest-after x xs)  
  (if (empty? xs)  
      "not a list"  
      (if (= x (first xs))  
          (rest xs)  
          (rest-after x (rest xs))))))
```

eval:4:9: Type Checker: type mismatch  
expected: (Listof Integer)  
given: String  
in: "not a list"

# Statisch getypte Sprachen

- Statische Typprüfung auch möglich, wenn nicht alle Anwendungen bekannt sind
- Reduktionssemantik bei statisch getypten Sprachen
  - Reduktion erhält Wohlgetyptheit ("Preservation" oder "Subject Reduction" Theorem)
  - Wohlgetypte Programme
    - Sind entweder Werte oder
    - Lassen sich stets reduzieren

# Statisch getypte Sprachen

- Es gibt stets Programme, die abgelehnt werden, obwohl es Ausführungen ohne Typfehler geben kann

- Beispiel

- Dynamisch getypt: Ausführung ohne Typfehler

```
> (+ 1 ( if (> 5 2) 1 "a"))
```

```
2
```

- Statisch getypt: Typfehler

```
> (+ 1 ( if (> 5 2) 1 "a"))
```

eval:2:5: Type Checker: type mismatch

expected: Number

given: (U String One)

in: (if (> 5 2) 1 "a")

U: "Union-Type"  
bzw. Summentyp



# Statisch getypte Sprachen

- Bewertung

1. Sicherstellen passender Verwendung von Werten/Funktionen
  - Ja
2. Zeitpunkt, zu dem Fehler gefunden werden
  - Entwicklungszeit
3. Bezug von Fehlermeldung auf Fehlerverursacher
  - Starker Bezug
4. Ausdrucksstärke der Signatursprache
  - Formale Syntax für Signaturen
5. Einschränkung der möglichen Programme
  - Stärkere Einschränkung als bei dynamisch getypten Sprachen
6. Konsistenz zwischen Datendefinitionen/Funktionen und dem Programmverhalten
  - Laufzeit-Typfehler können ausgeschlossen werden

# Statisch getypte Sprachen

- Bewertung

1. Sicherstellen passender Verwendung von Werten/Funktionen
  - Ja
2. Zeitpunkt, zu dem Fehler gefunden werden
  - Entwicklungszeit
3. Bezug von Fehlermeldung auf Fehlerverursacher
  - Starker Bezug
4. Ausdrucksstärke der Signatursprache
  - Formale Syntax für Signaturen
5. Einschränkung der möglichen Programme
  - Stärkere Einschränkung als bei dynamisch getypten Sprachen
6. Konsistenz zwischen Datendefinitionen/Funktionen und dem Programmverhalten
  - Laufzeit-Typfehler können ausgeschlossen werden

Trade-off zwischen  
Typsicherheit und  
Flexibilität

# Sprachunterstützung für Algebraische Datentypen

- Verschiedene Arten der Definition von algebraischen Datentypen
  - Unterscheiden sich in Eigenschaften, ähnlich wie Unterstützung für Signaturen
- Algebraische Datentypen haben
  - Datendefinition
  - Schnittstelle (“Interface”): Menge von Konstruktoren, Selektoren und Prädikaten

# Algebraischer Datentyp - Beispiel

```

; An Expression is one of:
; - (make-literal Number)
; - (make-addition Expression Expression)
; interp. abstract syntax of arithmetic expr.
; Expression -> Expression

; Number -> Expression
; constructs a literal expression
(define (make-literal value) ...)
; Expression -> Number
; returns the number of a literal
; throws an error if lit is not a literal
(define (literal-value lit) ...)
; [X] X -> Bool
; returns true iff x is a literal
(define (literal? x) ...)
; Expression Expression -> Expression
; constructs an addition expression
(define (make-addition lhs rhs) ...)

; [X] X -> Bool
; returns true iff x is an addition expr.
(define (addition? x) ...)
; Expression -> Expression
; returns left hand side of an addition
expression
; throws an error if e is not an addition
expression
(define (addition-lhs e) ...)
; Expression -> Expression
; returns right hand side of an addition
expression
; throws an error if e is not an addition
expression
(define (addition-rhs e) ...)

```

# Algebraischer Datentyp - Beispiel

- Interface des algebraischen Datentypen kann in Programm verwendet werden
  - Unabhängig von der Repräsentation

```
(define (calc e)
  (cond [(addition? e) (+ (calc (addition-lhs e))
                          (calc (addition-rhs e)))]
        [(literal? e) (literal-value e)]))
```

```
> (make-addition
   (make-addition (make-literal 0) (make-literal 1))
   (make-literal 2))
'(addition (addition (literal 0) (literal 1)) (literal 2))
```

Repräsentation, z.B.:  
S-Expression

```
> (calc (make-addition
         (make-addition (make-literal 0) (make-literal 1))
         (make-literal 2)))
```

3

# Algebraische Datentypen mit Listen und S-Expressions

- S-Expressions
  - Verschachtelte Listen
  - Universelle Datenstruktur

```
(define (make-literal n)
  (list 'literal n))
(define (literal-value l)
  (if (literal? l)
      (second l)
      (error 'not-a-literal)))
(define (literal? l)
  (and
    (cons? l)
    (symbol? (first l))
    (symbol=? (first l) 'literal)))
```

...

Alle Konstruktoren, Prädikate und Selektoren müssen vom Entwickler implementiert werden

# Algebraische Datentypen mit Strukturdefinitionen

- Das define-struct Konstrukt generiert automatisch Funktionen
  - Konstruktoren
  - Selektoren
  - Prädikate
- Im Beispiel: Funktionen im Interface stimmen mit Namenskonventionen von structs überein
  - Sprache nimmt uns den Aufwand der Implementierung ab  
(define-struct *literal* (*value*))  
(define-struct *addition* (*lhs rhs*))
  - Funktion calc ist unabhängig von der Datenrepräsentation und funktioniert unverändert

# Algebraische Datentypen mit Strukturdefinitionen

- Zusätzliche Möglichkeit
  - Durch Verwendung von Structs ist auch Pattern Matching möglich
  - D.h.: das Interface wird implizit mächtiger

```
(define (calc e)
  (match e
    [(addition e1 e2) (+ (calc e1) (calc e2))]
    [(literal x) x]))
```



# Algebraische Datentypen mit define-type

- Strukturdefinitionen nur für Produkttypen
- Algebraische Datentypen haben aber Alternativen (Summentyp)
- define-type Konstrukt
  - Alternativen von Produkttypen
  - Für jede Alternative
    - Felder
    - Zusammen mit Prädikat: Welche Werte sind zulässig
  - Automatische Generierung von Konstruktoren, Selektoren, Prädikaten
  - Prädikate von Feldern werden zu dynamischen Kontrakten
  - (Ursprüngliche Implementierung von calc funktioniert unverändert)

#lang racket  
(require 2htdp/abstraction)

# Algebraische Datentypen mit define-type

```
(define-type Expression  
  (literal (value number?))  
  (addition (left Expression?) (right Expression?)))
```

```
> (make-addition  
   (make-addition (make-literal 0) (make-literal 1))  
   (make-literal 2))
```

```
(addition (addition (literal 0) (literal 1)) (literal 2))
```

# Algebraische Datentypen mit define-type

- Zusätzliche (Typ-)Prüfungen gegenüber define-struct
  - Feld-Werte müssen Prädikat erfüllen

> (make-addition 0 1)

make-addition: contract violation

expected: (or/c undefined? Expression?)

given: 0

in: the 1st argument of

(->

(or/c undefined? Expression?)

(or/c undefined? Expression?)

addition?)

contract from: make-addition

blaming: use

(assuming the contract is correct)

at: eval:2:14

# Algebraische Datentypen mit define-type

- Zusätzliche (Typ-)Prüfungen gegenüber define-struct

- Feld-W

or/c bedeutet: einer der folgenden Kontrakte.

> (make-addition)

make-addition: contract violation

expected: (or/c undefined? Expression?)

given: 0

in: the 1st argument of

(->

(or/c undefined? Expression?)

(or/c undefined? Expression?)

addition?)

contract from: make-addition

blaming: use

(assuming the contract is correct)

at: eval:2:14

undefined? ist der Kontrakt für einen Wert, der jeden Typ erfüllt.

# Algebraische Datentypen mit define-type

- Erweiterung zu Pattern-Matching: type-case
  - Angabe des Typs für den Wert der “gematcht” wird
  - Unterscheidung der Alternativen
- Ermöglicht statische Typprüfung
  - Sind alle Alternativen des gematchten Typs abgedeckt?



# Algebraische Datentypen mit define-type

```
(define (calc e)
  (type-case Expression e
    [literal (value) value]
    [addition (e1 e2) (+ (calc e1) (calc e2))]))
> (calc (make-addition
  (make-addition (make-literal 0) (make-literal 1))
  (make-literal 2)))
```

3

```
> (define (calc2 e)
  (type-case Expression e
    [addition (e1 e2) (- (calc2 e1) (calc2 e2))]))
```

type-case: syntax error; probable cause: you did not include a case for the literal variant, or no else-branch was present

# Algebraische Datentypen mit define-type

- type-case
  - Vollständigkeitsüberprüfung
  - Aber: Einschränkung der unterstützten Patterns
    - Variantenname und Felder
    - Keine Literale, verschachtelte Patterns, etc.

- Allgemeine Form

(type-case *type e*

[*variant*<sub>1</sub> (*name*<sub>1\_1</sub> ... *name*<sub>1\_n1</sub>) *body-expression*<sub>1</sub>]

...

[*variant*<sub>*m*</sub> (*name*<sub>*m*\_1</sub> ... *name*<sub>*m*\_nm</sub>) *body-expression*<sub>*m*</sub>])

*name*<sub>*i*\_1</sub> ... *name*<sub>*i*\_ni</sub> dürfen  
in *body-expression*<sub>*i*</sub>  
verwendet werden.