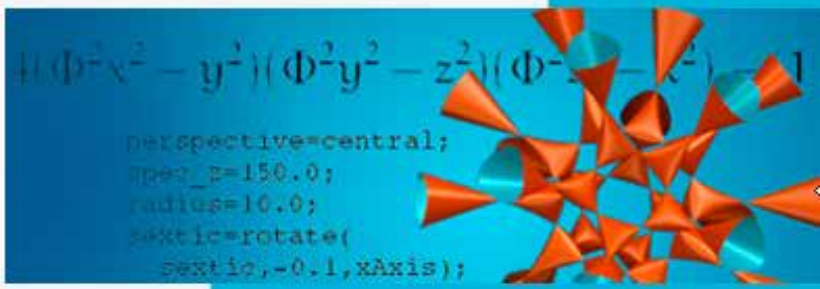
Philipps Universität Marburg

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 13, 14]

Algebraische Datentypen

- Werden häufig verwendet
  - Flexibel
  - Ausdrucksmächtig
- Aber: Funktionen über algebraischen Datentypen werden leicht komplex

Funktionen über algebraische Datentypen

- Entwurfsrezept
  - Fallunterscheidung über Form der Daten
  - Zugriff auf Komponenten über Selektoren
  - Aufruf von Hilfsfunktionen

- Beispiel: Liste als Argument

```
(define (f l)
   (cond [(cons? l) (... (first l) ... (sum (rest l))...)]
         [(empty? l) ...]))
```

Pattern Matching

- Vereinfacht Fallunterscheidungen
  - Deklarative Beschreibung der Fälle
  - Benennung der Strukturelemente zur weiteren Verwendung

- Beispiele

```
(define (f x)
   (match x
      [7 8]
      [else 9]))
>(f 7)
8
```

"Match" abhängig von der Form von x

x ist das Literal 7, dann ist das Ergebnis 8

Sonst ist das Ergebnis 9

Sprache umschalten auf "Fortgeschrittene"

Pattern Matching

- Beispiele

```
(define (f x)
  (match x
    [(list 1 y 3) y]))
```

Platzhalter stehen für beliebige Werte und werden gebunden.

Gebundene Platzhalter können im Ergebnisausdruck verwendet werden.

Nicht nur Literale, auch zusammengesetzte Werte können als Bedingung angegeben werden.

```
>(f (list 1 2 3))
2
```

Pattern Matching

- Beispiele

```
(define (f x)
  (match x
    [(struct posn (y y)) y]))
```

Platzhalter können im Pattern mehrfach vorkommen. Das Pattern matcht dann nur, wenn an allen Positionen derselbe Wert auftaucht.

Durch das Binden der Werte entfällt der Gebrauch von Selektoren.

Pattern Matching

- Beispiele

```
(define (f x)
  (match x
    [(cons (struct posn (1 z)) y) z]))
```

Datenstrukturen auf die gematcht wird können auch beliebig tief verschachtelt sein.

Pattern Matching

- Beispiele

```
(define (f x)
  (match x
    [7 8]
    ["hey" "joe"]
    [(list 1 y 3) y]
    [(cons a (list 5 6)) (add1 a)]
    [(struct posn (5 5)) 42]
    [(struct posn (y y)) y]
    [(struct posn (y z)) (+ y z)]
    [(cons (struct posn (1 z)) y) z]))
```

Beliebig viele match-Klauseln

Nur der Resultatausdrucks des ersten erfolgreichen Match wird ausgewertet. Bspw.:

```
>(f (make-posn 5 5))
42
```

Matcht kein Ausdruck, dann Fehlermeldung:
match: no matching clause for ...

Analog zu cond.

Pattern Matching

- Anstatt

```
(define (person-has-ancestor p a)
   (cond [(person? p)
          (or
             (string=? (person-name p) a)
             (person-has-ancestor (person-father p) a)
             (person-has-ancestor (person-mother p) a))]
         [else false]))
```

- Mit Pattern Matching

```
(define (person-has-ancestor p a)
   (match p
      [(struct person (name father mother))
         (or
            (string=? name a)
            (person-has-ancestor father a)
            (person-has-ancestor mother a))]
      [else false]))
```

Bedeutung von Pattern Matching

- (match ...) ist ein Ausdruck und kann überall verwendet werden, wo ein Ausdruck erwartet wird

- Allgemeine Form:

(match v [(p_1 e_1)] ... [(p_n e_n)])

- Bedeutung informell:
  - Finde das erste p_i, das auf v matcht
  - Binde dabei die Variablen, die in p_i vorkommen mit Werten aus v
  - Ersetze die Vorkommen der Variablen e_i durch die gebundenen Werte

Bedeutung von Pattern Matching

- Betrachten wir Matching als Funktion
  - Eingabe: Pattern und Wert
  - Ausgabe: “no match“ oder Substitution

- Substitution:
  - $[x_1 := v_1, ..., x_n := v_n]$
  - x_i: Bezeichner
  - v_i: Werte

Bedeutung von Pattern Matching

- match(***v***, ***v***) = [] (die leere Substitution)
- match(***x***, ***v***) = [***x*** := ***v***]
- match((**struct *id* (*p_1* ... *p_n*)) , (make-*id* *v_1* ... *v_n*)**) = match(*p_1*, *v_1*) + ... + match(*p_n*, *v_n*)
- Spezialfall für Listen analog
- match(...,...) = ”no match” in allen anderen Fällen

Operator “+“: Kombination von Substitutionen.

Bedeutung von Pattern Matching

- Kombination von Substitutionen
  - Eine der beiden Substitutionen ist “no match“ → “no match“
  - Beide Substitutionen enthalten Mapping für denselben Namen, aber mit unterschiedlichen Werten → “no match“
  - Ansonsten Vereinigung der Mappings aus beiden Substitutionen

Bedeutung von Pattern Matching

- Hat ein Ausdruck *e* de Form
  (match v [(p_1 e_1)] ... [(p_n e_n)])

- Ist match(p_1, v) = [x_1 := v_1, ..., x_n := v_n]
  - Dann gilt: e → e_1**[**x_1 **:=** v_1**, ...,** x_n **:=** v_n**]**
- Ist match(p_1, v) = “no match“
  - Dan gilt: e → (match v [(p_2 e_2)] ... [(p_n e_n)])

Generative Rekursion

- Entwurfsrezept für Funktionen über algebraische Datentypen
  - Eine Hilfsfunktion pro Alternative
- Entwurfsrezept für Funktionen über rekursiven Datentypen
  - Strukturelle Rekursion
- Entwurfsrezept kann nicht immer angewandt werden
  - Struktur der Daten passt nicht zur Aufteilung des Problems

1. Fallstudie

- Basierend auf Funktion move-ball
- Berechnen, wo der Ball kollidiert

- Simulation der Bewegung durch rekursives Einsetzen der Funktion
  - (move-ball ball)
  - (move-ball (move-ball ball))
  - Etc.
- Rekursionsabbruch sobald (collision current-ball) nicht "none"

1. Fallstudie

Entspricht diese Funktion einem unserer Entwurfsmuster?

```
; Ball -> Posn
; computes the position where the first
; collision of the ball occurs
(define (first-collision ball)
  (cond [(string=? (collision (ball-loc ball)) "none")
         (first-collision (move-ball ball))]
        [else (ball-loc ball)]))
```

Fallunterscheidung und Rekursion sind unabhängig von Struktur des Eingabetyps.

Argument für rekursiven Aufruf nicht durch Selektor auf dem Eingabewert, sondern neu generiert.

2. Fallstudie

- Sortieren einer Liste von Zahlen
- Liste ist ein rekursiver Datentyp
- Laut Entwurfsrezept:

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
    (match l
        [empty  ...]
        [(cons x xs)  ... x  ... (sort xs)  ...]))
```

2. Fallstudie

- Sortieren einer Liste von Zahlen
- Liste ist ein rekursiver Datentyp
- Laut Entwurfsrezept:

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
  (match l
    [empty  empty]]
    [(cons x xs) (insert x (sort xs))]))
```

Basisfall: Die leere Liste ist sortiert.

Rekursiver Fall: Sortiertes Einfügen von x in sortierte Liste (sort xs)

2. Fallstudie

- Sortieren einer Liste von Zahlen
- Liste ist ein rekursiver Datentyp
- Laut Entwurfsrezept:

Insertion Sort:
- Sortieralgorithmus mit struktureller Rekursion.
- Nicht effizient

```
; (listof number) -> (listof number)
; to create a list of numbers with the same numbers as
; l sorted in ascending order
(define (sort l)
  (match l
    [empty  empty]]
    [(cons x xs) (insert x (sort xs))]))
```

2. Fallstudie

- Effizienz von Insertion Sort

- Gegeben ein Funktionsaufruf
  (sort (list x-1 ... x-n))
- Dann ist die Expansion davon
  (insert x-1 (insert x-2 ... (insert x-n empty) ...))

Max. n Berechnungsschritte

Max. n - 1 Berechnungsschritte

Max. 1 Berechnungsschritt

- Effizienz von insert
  - Durchläuft die Liste bis Einfüge-Position gefunden
  - Bei rückwärts sortierter Eingabe: jedes Mal vollständig durchlaufen

2. Fallstudie

- Effizienz von Insertion Sort

- Berechnungsschritte:
  $n + (n - 1) + ... 1 = n * (n + 1) / 2 = (n^2 + n) / 2$

- Effizient:
  - Im schlechtesten Fall hängt die Anzahl der Berechnungsschritte vom Quadrat der Länge der Eingabe Liste ab.

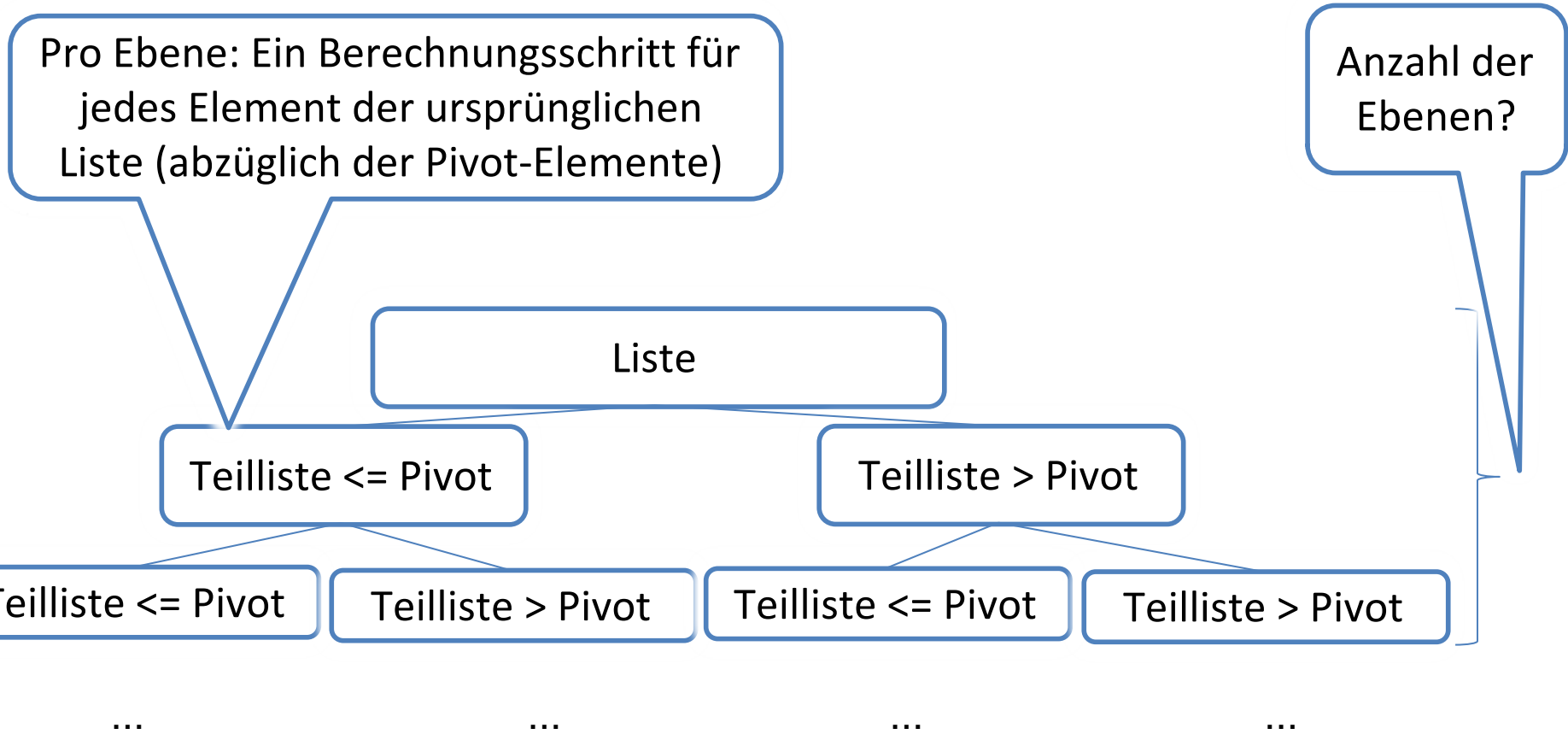
2. Fallstudie

- Bessere Effizienz möglich
- Dazu: bessere Aufteilung in Teilprobleme
- Bekannter Algorithmus: Quick Sort

2. Fallstudie

- Quick Sort
  - Auswahl eines beliebigen Elements der Liste (genannt “Pivot-Element“)
  - Generieren:
    - Eine Liste mit Elementen <= Pivot-Element
    - Eine Liste mit Elementen > Pivot-Element
  - Rekursiver Aufruf von Quick Sort mit beiden Listen
  - Zusammenfügen beider sortierter Listen mit Pivot-Element in der Mitte

2. Fallstudie



2. Fallstudie

- Anzahl der Ebenen (Rekursionstiefe)
  - Anhängig von der Wahl der Pivot-Elemente
  - Teilung in zwei gleich große Teillisten: log(n)
- Effizienz im Besten Fall: n*log(n) Berechnungsschritte
- Es lässt sich zeigen:
  - Im Durchschnitt benötigt Quick Sort
    ”in der Größenordnung von n*log(n)” Berechnungsschritte

2. Fallstudie

```
; (listof number) -> (listof number)
; to create a list with the numbers of l in ascending order
(define (qsort l)
   (match l
      [empty empty]
      [(cons x xs)
       (append
       (qsort (smaller-or-equal-than x xs))
       (list x)
       (qsort (greater-than x xs)))]))
; Number (listof Number) -> (listof Number)
; generates a list of all elements of xs that are smaller or equal than x
(define (smaller-or-equal-than x xs)
   (filter (lambda (y) (<= y x)) xs))
; Number (listof Number) -> (listof Number)
; generates a list of all elements of xs that are greater than x
(define (greater-than x xs)
   (filter (lambda (y) (> y x)) xs))
```

Folgt der Struktur von l

Aber: rekursiver Aufruf mit generiertem Wert.

Entwurf von generativ rekursiven Funktionen

- Rekursion nicht auf Ergebnis von Selektor sondern auf generiertem Wert
- Fallunterscheidung folgt nicht notwendigerweise Struktur der Daten
- Allgemein

```
(define (generative-recursive-fun problem)
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
     ... problem ...
     (generative-recursive-fun (generate-problem-1 problem))
     ...
     (generative-recursive-fun (generate-problem-n problem)))]))
```

Rekursionsabbruch

Rekursiver Aufruf

Generierung von Teilproblemen

Entwurf von generativ rekursiven Funktionen

1. Was ist ein trivial lösbares Problem?
2. Was ist die Lösung für ein trivial lösbares Problem?
3. Wie generieren wir neue Probleme, die leichter lösbar sind als das Originalproblem? Wie viele neue Probleme sollen wir generieren?
4. Wie berechnen wir aus den Lösungen der generierten Probleme die Lösung des Originalproblems? Benötigen wir hierzu nochmal das Originalproblem (oder einen Teil davon)?
5. Terminiert der Algorithmus?

Entwurf von generativ rekursiven Funktionen – 1. Beispiel

- Problem: Wann kollidiert der Ball?

1. Triviales Problem:	
2. Triviale Lösung	
3. Generierung neues Problem	
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 1. Beispiel

- Problem: Wann kollidiert der Ball?

1. Triviales Problem:	Ball bereits kollidiert.
2. Triviale Lösung	
3. Generierung neues Problem	
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 1. Beispiel

- Problem: Wann kollidiert der Ball?

1. Triviales Problem:	Ball bereits kollidiert.
2. Triviale Lösung	Aktuelle Position
3. Generierung neues Problem	
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 1. Beispiel

- Problem: Wann kollidiert der Ball?

1. Triviales Problem:	Ball bereits kollidiert.
2. Triviale Lösung	Aktuelle Position
3. Generierung neues Problem	Bewegungsschritt berechnen
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 1. Beispiel

- Problem: Wann kollidiert der Ball?

1. Triviales Problem:	Ball bereits kollidiert.
2. Triviale Lösung	Aktuelle Position
3. Generierung neues Problem	Bewegungsschritt berechnen
4. Berechnen der Lösung	Teil-Lösung ist bereits Gesamtlösung
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 1. Beispiel

- Problem: Wann kollidiert der Ball?

1. Triviales Problem:	Ball bereits kollidiert.
2. Triviale Lösung	Aktuelle Position
3. Generierung neues Problem	Bewegungsschritt berechnen
4. Berechnen der Lösung	Teil-Lösung ist bereits Gesamtlösung
5. Terminierung	*(siehe später)*

Entwurf von generativ rekursiven Funktionen – 2. Beispiel

- Problem: Quick Sort

1. Triviales Problem:	
2. Triviale Lösung	
3. Generierung neues Problem	
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 2. Beispiel

- Problem: Quick Sort

1. Triviales Problem:	Sortieren einer leeren Liste
2. Triviale Lösung	
3. Generierung neues Problem	
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 2. Beispiel

- Problem: Quick Sort

1. Triviales Problem:	Sortieren einer leeren Liste
2. Triviale Lösung	Die leere Liste
3. Generierung neues Problem	
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 2. Beispiel

- Problem: Quick Sort

1. Triviales Problem:	Sortieren einer leeren Liste
2. Triviale Lösung	Die leere Liste
3. Generierung neues Problem	Auswählen eines Pivot-Elements, generieren von zwei Teillisten
4. Berechnen der Lösung	
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 2. Beispiel

- Problem: Quick Sort

1. Triviales Problem:	Sortieren einer leeren Liste
2. Triviale Lösung	Die leere Liste
3. Generierung neues Problem	Auswählen eines Pivot-Elements, generieren von zwei Teillisten
4. Berechnen der Lösung	Zusammenfügen der Teil-Listen mit Pivot-Element in der Mitte
5. Terminierung	

Entwurf von generativ rekursiven Funktionen – 2. Beispiel

- Problem: Quick Sort

1. Triviales Problem:	Sortieren einer leeren Liste
2. Triviale Lösung	Die leere Liste
3. Generierung neues Problem	Auswählen eines Pivot-Elements, generieren von zwei Teillisten
4. Berechnen der Lösung	Zusammenfügen der Teil-Listen mit Pivot-Element in der Mitte
5. Terminierung	*(siehe später)*

Verwendung von generativer Rekursion

- Typische Situationen
- Eingabe hat rekursive Datenstruktur
  - Strukturelle Rekursion ist ...
    - ... nicht möglich
      bspw. lässt sich Lösung zu original-Problem nicht aus Ergebnis der Rekursion berechnen
    - ... zu kompliziert
    - ... zu ineffizient
- Eingabe ist nicht rekursiv
  - Anzahl der Berechnungsschritte nicht proportional zur Größe der Eingabe

Verwendung von generativer Rekursion

```
(define (generative-recursive-fun
  (cond
    [(trivially-solvable? problem)
     (determine-solution problem)]
    [else
     (combine-solutions
     ... problem ...
     (generative-recursive-fun (generate-problem-1 problem))
     ...
     (generative-recursive-fun (generate-problem-n problem)))]))
```

Frage 1

Frage 2

Frage 4

Frage 3

Verwendung von generativer Rekursion

- Tests für
  - Trivialen Fall
  - Rekursiven Fall
- Strukturelle Rekursion
  - Antworten auf Fragen 1 und 3 bereits im Template verankert
  - Antworten auf Fragen 2 und 4 ergeben sich fast automatisch
- Generative Rekursion
  - Mehr Kreativität erforderlich

1. Triviales Problem:
2. Triviale Lösung
3. Generierung neues Problem
4. Berechnen der Lösung

Terminierung

- Strukturelle Rekursion terminiert immer:
  - Datenstruktur ist endlich
  - Eingabe wird bei jedem rekursiven Aufruf kleiner
  - Basisfall muss erreicht werden

- Bei generativer Rekursion
  - Keine feste Beziehung zwischen Größe der Eingabe und Größe der Teilprobleme
  - Terminierung muss individuell gezeigt werden

Terminierung - Beispiel

```
(define (qsort l)
   (match l
      [empty empty]
      [(cons x xs)
       (append
       (qsort (smaller-or-equal-than x l))
       (list x)
       (qsort (greater-than x xs)))]))
```

Fehler: Anstatt Suche in "Rest" der Eingabe, Suche in ganzer Liste (enthält auch Pivot-Element)

```
>(qsort (list 5))
```

Ergebnis?

(smaller-or-equal-than 5 (list 5))
→ (list 5)
Identisch mit Eingabe, daher endlose Rekursion.

Zeigen der Terminierung

1. Definition einer Abbildung
   - Satz der Funktionsargumente → Natürliche Zahl
   - Ergebnis-Wert entspricht Größe des Problems aus Sicht des Algorithmus
2. Zeigen, dass Größe der Eingabe mit rekursivem Aufruf strikt abnimmt

- Größe entspricht der (maximalen) Rekursionstiefe

Zeigen der Terminierung

- Quick Sort
  - Größe der Eingabe: Länge der Liste (Eingabeargument)
  - Originaleingabe
    - (cons x xs)
    - Größe: n
    - Dann ist Größe von xs: n - 1
  - Hilfsfunktionen geben eine Teilliste zurück
    - (smaller-or-equal-than x xs)
    - (greater-than x xs)
    - → Eingabegröße in allen rekursiven Aufrufen < n – 1
  - Eingabegröße wird strikt kleiner

Zeigen der Terminierung

- First Collision
  - Größe der Eingabe: ?

Wir können keine Abbildung finden.

Vielleicht gibt es aber eine. Wir können so nicht die “nicht-terminierung“ beweisen.

In diesem Fall terminiert first-collision aber tatsächlich nicht immer.