18 Generative Rekursion

Bisher: Funktionsdefinition durch Muster für induktive Datentypen

strukturelle Rekursion

Jetzt: Verallgemeinerung zu generativer (allgemeiner) Rekursion

- keine direkte Anlehnung an einen Datentyp
- keine Terminationsgarantie

18.1 Beispiel: Sortieren von Listen

```
(: list-sort ((%X %X -> bool) (list %X) -> (list %X)))
```

Erklärung: `(list-sort leq l)` liefert eine Liste mit den gleichen Elementen wie `l`, aber aufsteigend gemäß der kleiner–gleich Relation `leq` sortiert.

Beispiele:

```
(list-sort <= (list 32 16 8))     ; == (list 8 16 32)
(list-sort string<=? empty)       ; == empty
```

Struktureller Ansatz: führt zum Sortieren durch Einfügen

Generativer Ansatz

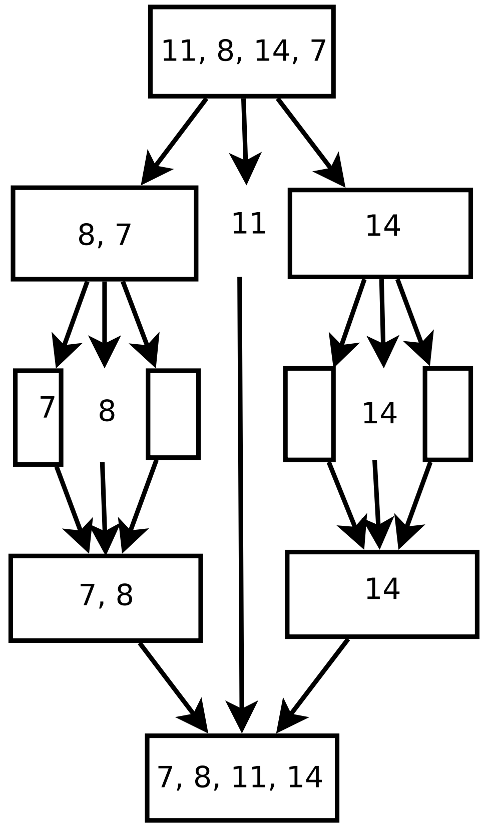
- Trivial lösbare Fälle erkennen und lösen
- Restliche Fälle (geschickt) in Teilprobleme aufteilen, die Teilprobleme lösen und die Lösungen zusammensetzen

$\Rightarrow$ Strategie *divide and conquer*

Anwendung aufs Sortieren: Quicksort (Hoare)

- Triviale Instanzen: Eingabeliste hat Länge 0 (oder 1)
- Restliche Instanzen
  - Entferne ein beliebiges Listenelement (*Pivotelement* p)
  - Teile die verbleibende Liste in zwei Teile,
    die Liste der Elemente $< p$ und die Liste der Elemente $\geq p$
  - Sortiere die beiden Teile
  - Verkette die Teillösungen und das Pivotelement zur Gesamtlösung

Beispiel



Quicksort

```
(: qsort ((%X %X -> bool) (list %X) -> (list %X)))
```

Erklärung: `(qsort leq l)` liefert eine Liste mit den gleichen Elementen wie `l`, aber aufsteigend gemäß der kleiner–gleich Relation `leq` sortiert.

Implementierung:

```
(define qsort
  (lambda (<= l)
    (if (empty? l)
        l
        (let ((p (first l))
              (l (rest l)))
          (let ((l-smaller (qsort <= (filter (lambda (x) (< x p)) l)))
                (l-larger  (qsort <= (filter (lambda (x) (>= x p)) l))))
            (append l-smaller (make-pair p l-larger)))))))
```

18.2 Beispiel: Größter gemeinsamer Teiler

```
(define posnat
  (contract (combined natural (predicate positive?))))
```

```
(: ggt (posnat posnat -> posnat))
```

Erklärung: `(ggt m n)` liefert den größten gemeinsamen Teiler von `m` und n.

Implementierung: Durch strukturelle Rekursion oder durch generative Rekursion?

GGT mit struktureller Rekursion

```
(: ggt0 (posnat posnat -> posnat))
(define ggt0
  (lambda (m n)
    (letrec ((greatest-divisor-<=
              (lambda (i)
                (if (= i 1) 1
                    (if (and (= 0 (remainder m i))
                             (= 0 (remainder n i)))
                        i
                        (greatest-divisor-<= (- i 1)))))))
      (greatest-divisor-<= (min m n)))))
```

GGT mit generativer Rekursion

Beobachtung: Es gilt, dass `(ggt a b)` $=$ `(ggt b (remainder a b))`, falls `a` $\geq$ `b` > 0

GGT mit generativer Rekursion

```
; größter gemeinsamer Teiler, Euklids Algorithmus
(: ggt (posnat posnat -> posnat))
(define ggt
  (letrec ((loop
            (lambda (m n) ; m >= n > 0
              (let ((r (remainder m n)))
                (if (zero? r)
                    n
                    (loop n r))))))
    (lambda (m n)
      (if (>= m n)
          (loop m n)
          (loop n m)))))
```

Vergleich der GGT Implementierungen

- Lösung mit struktureller Rekursion verwendet nur die Definitionen und setzt kein spezielles Wissen voraus.
  Dieser Algorithmus führt maximal $2\min(m,n)$ Divisionen durch.

- Der Euklidische Algorithmus verwendet spezielles mathematisches Wissen.
  Hier tritt die schlechteste Laufzeit bei Eingabe von aufeinanderfolgenden Fibonacci-Zahlen ein.
  Es gilt nämlich $\mathit{fib}(k+2) \bmod \mathit{fib}(k+1) = \mathit{fib}(k)$.
  Hierfür werden $k+1$ Divisionen benötigt.
  Der kleinere Operand ist $\approx \frac{1}{\sqrt{5}}(\frac{1+\sqrt{5}}{2})^{k+1}$
  Also ist bei allgemeinem n die maximale Anzahl der Divisionen proportional zu $\log(\min(m,n))$.

- Bemerkung: In beiden Fällen wird ignoriert, dass der Aufwand für eine Division von der Länge der Operanden abhängt.

18.3 Nullstellen von Funktionen

- Gegeben: Ein Intervall $[a, b] \subseteq \mathbf{R}$ und eine stetige Funktion $f : [a, b] \to \mathbf{R}$ auf dem Intervall, für die die Vorzeichen von $f(a)$ und $f(b)$ unterschiedlich sind.
- Aufgabe: bestimme eine Nullstelle von f in diesem Intervall
- Verwende dabei den Mittelwertsatz:
  Wenn $f(a) \leq 0 \wedge f(b) \geq 0$ oder $f(a) \geq 0 \wedge f(b) \leq 0$,
  dann existiert ein $x \in [a, b]$ mit $f(x) = 0$.

Verbesserte Aufgabenstellung

- Der Test $f(x) = 0$ ist mit Gleitkommazahlen nicht exakt durchführbar.
- Aufgabe: bestimme $x \in [a, b]$, so dass $[x, x + \epsilon]$ eine Nullstelle von f enthält. $\epsilon > 0$ ist dabei eine Konstante, die von der gewünschten (und möglichen) Rechengenauigkeit abhängt.

Lösungsansatz mit generativer Rekursion

```
(: find-root ((real -> real) real real -> real))
```

Erklärung: `(find-root f a b)` liefert eine Zahl x zwischen a und b, so dass f eine Nullstelle zwischen x und $x + \epsilon$ besitzt.

Muster:

```
(define find-root
  (lambda (f a b)
    (if (trivially-solvable f a b)
        (solution-for f a b)
        (... f a b ... (find-root f ...)))))
```

Tests:

```
(define poly
  (lambda (x) (* (- x 1) (- x 2) (- x 3))))
(check-within (find-root poly 0 1.5) 1 epsilon)
(check-within (find-root poly 1.5 2.5) 2 epsilon)
(check-within (find-root poly 2.5 1000) 3 epsilon)
```

Lösung

```
; finde Nullstelle einer stetigen Funktion in einem Intervall
(: find-root ((number -> real) number number -> number))
(define find-root
  (lambda (f a b)
    (if (< (abs (- a b)) epsilon)
        a
        (let* ((m (/ (+ a b) 2))
               (fa (f a))
               (fb (f b))
               (fm (f m)))
          (if (mean-value-condition fa fm)
              (find-root f a m)
              (find-root f m b))))))
```

Hilfsfunktion

```
(: epsilon real)
(define epsilon 10e-10)

; Bedingung für den Mittelwertsatz
(: mean-value-condition (real real -> boolean))
(define mean-value-condition
  (lambda (fv1 fv2)
    (or (<= fv1 0 fv2)
        (<= fv2 0 fv1))))
```

18.4 Allgemeines Muster: Generative Rekursion

```
(define generative-recursive-fun
  (lambda (instance ...)
    (if (trivially-solvable instance)
        (solution-for instance)
        (combine-solutions
         ... instance ...
         (generative-recursive-fun (subproblem-1 instance) ...)
         ...
         (generative-recursive-fun (subproblem-n instance) ...)))))
```

18.5 Schritte zur generativen Rekursion

1. Was sind die trivial lösbaren Instanzen?
2. Was ist die Lösung dieser Instanzen?
3. Wie kann eine Probleminstanz in Teilprobleme zerlegt werden? Die Teilprobleme können anders geartet sein als das ursprüngliche Problem.
4. Wie können Lösungen der Teilprobleme zur Lösung der ursprünglichen Instanz zusammengesetzt werden?

Neuer Schritt: Terminationsargument

- Bei generativer Rekursion ist die Termination nicht mehr “eingebaut”
- Termination muss separat bewiesen werden:
  - Ordne jedem Funktionsaufruf einen Terminationswert zu
  - Der Terminationswert jedes rekursiven Aufrufs muss kleiner (in einer wohlfundierten Ordnung) sein, als der Terminationswert des Funktionsaufrufs
- Beispiel
  - Funktion auf natürlichen Zahlen: Ordnungsrelation ist $<$
  - Funktion auf Listen: Ordnungsrelation ist Teillistenrelation
  - Funktion auf Listen: alternative Ordnungsrelation ist $<$ auf den Längen der Listen (in `qsort`)
  - Funktion auf Paaren von natürlichen Zahlen: Ordnungsrelation ist $<$ auf der Summe der Komponenten (in `ggt`)
  - Nullstellensuche: Intervallgröße halbiert sich

18.6 Strukturelle Rekursion ist ein Spezialfall

```
(define generative-recursive-fun
  (lambda (instance ...)
    (if (trivially-solvable instance)
        (solution-for instance)
        (combine-solutions
         instance
         (generative-recursive-fun (subproblem instance) ...)))))
```

Setze ein: `trivially-solvable` $\mapsto$ `empty?` und `subproblem` $\mapsto$ `rest`

```
(define generative-recursive-fun
  (lambda (instance ...)
    (if (empty? instance)
        (solution-for instance)
        (combine-solutions
         instance
         (generative-recursive-fun (rest instance) ...)))))
```

18.7 Backtracking

Problemstellungen

1. Suche einen Weg durch ein Labyrinth von Einbahnstrassen
2. Stelle fest, ob Onkel Rudi von Alexander dem Großen abstammt
3. Kann ich die Krawatte vor der Unterhose anziehen?

Allen drei Problemen liegt das gleiche Prinzip zugrunde ...

Graphen

Ein *gerichteter Graph* $G = (N, E)$ besteht aus einer Menge von *Knoten* N und einer Menge von *gerichteten Kanten* $E \subseteq N \times N$.

Wenn $(a, b) \in E$, dann ist b Nachfolger (Nachbar) von a bzw. a ist Vorgänger von b.

1. Knoten: Kreuzungen;
   Kanten: Verbindungen zwischen Kreuzungen

2. Knoten: Personen;
   Kante jeweils zwischen Kind und Elternteil

3. Knoten: Kleidungsstücke;
   Kante zwischen a und b, falls a vor b angezogen werden muss

Pfade in Graphen

Seinen $a, b \in N$ Knoten in einem Graph G.

Ein *gerichteter Pfad* von a nach b in einem Graphen G ist eine Folge $a_0, \ldots, a_k$ von Knoten, so dass für alle $0 \leq i < k$ eine Kante $(a_i, a_{i+1}) \in E$ existiert.

1. Abfolge von Kreuzungen, die besucht werden muss
2. Ahnenkette von Onkel Rudi bis Alexander
3. Anziehreihenfolge der Kleidungsstücke

Aufgabe: Finde einen Pfad in einem Graphen!

Repräsentation von Graphen

- Knoten: Symbole
- Kanten: Repräsentiert durch Abbildung von Knoten auf Liste der Nachfolger

```
(define node
  (contract symbol))
; erster Knoten jeder Liste ist Vorgänger aller restlichen Knoten
(define graph
  (contract (list (list node))))
```

Beispiel: Kleidungsstücke

```
; Kleidungsstücke
(: clothes (list node))
(define clothes '(unterhose socken schuhe hose guertel
                  unterhemd oberhemd krawatte sacko mantel))
; vorher anziehen
(: put-on-before graph)
(define put-on-before
  (list '(unterhose ...)
        '(socken ...)
        '(schuhe ...)
        '(hose ...)
        '(guertel ...)
        '(unterhemd ...)
        '(oberhemd ...)
        '(krawatte ...)
        '(sacko ...)
        '(mantel ...)))
```

Nachfolger eines Knotens

```
; Liste der Nachfolger eines Knotens
(: succ (graph node -> (list node)))
(define succ
  (lambda (g n)
    (if (empty? g)
        (violation "illegal node")
        (let ((nodelist (first g)))
          (if (equal? (first nodelist) n)
              (rest nodelist)
              (succ (rest g) n))))))
```

Pfadsuche

```
; finde einen Pfag zwischen zwei Knoten oder #f, falls keiner existiert
(: find-path (graph node node -> (mixed (one-of #f) (list node))))
(define find-path
  (lambda (g orig dest)
    (if (trivially-solvable g orig dest)
        (solution-for g orig dest)
        (combine-solutions
         g orig dest
         (find-path g ...)))))
```