



INFORMATIK I

Tutorium 06.12.2016

BESPRECHUNG

Blatt 6



WIEDERHOLUNG

Vorlesung & Für Blatt 7



NEUE SPRACHEBENE

- Wechsel in DMdA: „Die Macht der Abstraktion“
- Signatur (list-of %a) nun direkt eingebaut
- Neuer syntaktischer Zucker (list ...) eingebaut:

```
(list <e1> <e2> ... <e209>))
```

≡

```
(make-pair <e1>  
  (make-pair <e2>  
    (make-pair ...  
      (make-pair <e209>  
        empty)...)))
```

LISTEN ZUSAMMENFÜGEN

- Zwei Listen zusammenfügen über append bzw. cat

```
(define cat
  (lambda (xs ys)
    (cond ((empty? xs)
           ys)
          ((pair? xs)
           (make-pair (first xs) ; <- cat dennoch param. polymorph
                       (cat (rest xs) ys))))))
```

; Hinweis: Verfügbar als eingebaute Funktion 'append'

BEDINGTE ALGEBRAISCHE EIGENSCHAFTEN

- Bei check-property besteht die Möglichkeit, nur dann zu testen, wenn eine bestimmte Bedingung zutrifft.
- Syntax: $(==> \langle p \rangle \langle e \rangle)$
- Nur wenn $\langle p \rangle$ zu $\#t$ ausgewertet, wird $\langle e \rangle$ ausgewertet und getestet

PROBLEM: SPEICHER

.....

- Betrachte folgende Prozedur:

```
; Berechne n!  
(: factorial (natural -> natural))  
  
(check-expect (factorial 0) 1)  
(check-expect (factorial 3) 6)  
(check-expect (factorial 10) 3628800)  
  
(define factorial  
  (lambda (n)  
    (cond ((= n 0) 1)  
          (else (* n (factorial (- n 1)))))))
```

- Was für ein Problem kann beobachtet werden?

PROBLEM: SPEICHER

- Wir wollen Prozeduren schreiben, die konstanten Platzverbrauch haben
- Idee: Endrekursion
- Führe Berechnung sofort aus und führe dieses Zwischenergebnis in neuen Rekursionsaufruf mit (akkumulierendes Argument)
- Schreibe dazu einen „Worker“

PROBLEM: SPEICHER

► Beispiel: Fakultät

```
; Berechne n!  
; (Wrapper)  
(: fac (natural -> natural))  
  
(define fac  
  (lambda (n)  
    (fac-worker n 1)))  
;  
; Ergebnis von 0!, neutrales Element für *  
  
; Berechne n!,  
; bisheriges Zwischenergebnis (Akkumulator): acc  
; (Worker)  
(: fac-worker (natural natural -> natural))  
  
(define fac-worker  
  (lambda (n acc)  
    (cond ((= n 0) acc)  
          ((> n 0) (fac-worker (- n 1) (* n acc))))))  
;  
;  
;  
;                                     ↑  
;                                neuer Wert des Akkumulators
```

PROBLEM SPEICHER

- Dadurch konstanter Speicherverbrauch
- Es wird kein Kontext aufgebaut
- Beispiel: Liste umdrehen

```
; Liste xs umdrehen
(: backwards ((list-of %a) -> (list-of %a)))
(check-expect (backwards empty) empty)
(check-expect (backwards (list 1 2 3)) (list 3 2 1))
(define backwards
  (lambda (xs)
    (backwards-worker xs empty)))

(: backwards-worker ((list-of %a) (list-of %a) -> (list-of %a)))

(define backwards-worker
  (lambda (xs acc)
    (cond ((empty? xs) acc)
          ((pair? xs)
           (backwards-worker (rest xs) (make-pair (first xs) acc))))))

;           ↑           ↑
;           tail call   ein make-pair-Aufruf je Rekursion
```

KOMPAKTE ENDREKURSION

- Letrec als neues Schlüsselwort
- Arbeitet ähnlich wie let, erlaubt das verwenden von gebunden Werten in anderen gebundenen Werten
- Wir benutzen letrec um Worker-Prozeduren in die Hauptprozedur zu „holen“

```
(: backwards ((list-of %a) -> (list-of %a)))  
(define backwards  
  (lambda (xs)  
    (letrec  
      ((backwards-worker  
        (lambda (xs acc)  
          (cond ((empty? xs) acc)  
                ((pair? xs)  
                 (backwards-worker (rest xs)  
                                   (make-pair (first xs) acc))))))  
      (backwards-worker xs empty))))
```

ÜBUNGSAUFGABEN

- Schreibe eine endrekursive Prozedur, die die Länge einer Liste bestimmt
- Schreibe eine endrekursive Prozedur, die die Summe einer Liste berechnet
- Schreibe eine endrekursive Prozedur, die jedes zweite Element zurückgibt
- Schreibe eine endrekursive Prozedur, die alle natürlichen Zahlen zwischen a und b (inklusive a und b) zurückgibt
- Schreibe eine endrekursive Prozedur, die die fehlenden Elemente in einer (aufsteigend sortierten) Liste hinzufügt