Informatik II Skript

Steffen Lindner

June 19, 2015

Contents

Chapter 1

Einführung - 14.04.15

Scheme: Ausdrücke, Auswertung und Abstraktion

Dr.Racket: Definitionsfenster (oberer Bereich), Interaktionsfenster (unterer Bereich)

Die Anwendung von Funktionen wird in Scheme ausschließlich in Präfixnotation durchgeführt.

Beispiele

Mathematik	Scheme
44-2	(- 44 2)
f(x,y)	(f x y)
$\sqrt{81}$	(sqrt 81)
9^2	(expt 9 2)
3!	(! 3)

Allgemein: $(< function >< arg1 >< arg2 > ...)$

(+ 40 2) und (odd? 42) sind Beispiele für Ausdrücke, die bei Auswertung einen Wert liefern. (Notation: $\rightsquigarrow$)

(+ 40 2) $\rightsquigarrow$ 42 ($\rightsquigarrow$ = Auswertng / Reduktion / Evalutation)

(odd? 42) $\rightsquigarrow$ #f

Interaktionsfenster: Read → Eval → Print → Read ... (Read-Eval-Print-Loop aka. REPL)

Literale stehen für einen konstanten Wert (auch konstante) und sind nicht weiter reduzierbar.

Literal:

#t, #f (true, false, Wahrheitswerte) (boolean)

"abc", "x", " " (Zeichenkette) (String)

0 1904 42 -2 (ganze Zahlen) (Integer)

0.42 3.1415 (Fließkommazahl) (Reel)

1/2, 3/4 (rationale Zahl) (Rational)

_(")_/" (Bilder) (Image)

Chapter 2

Ausdrücke, Defines, usw. - 16.04.2015

Auswertung zusammengesetzter Ausdrücke in mehreren Schritten (steps), von "innen nach außen" bis keine Reduktion mehr möglich ist.

$$(+\ (+\ 20\ 20)\ (+\ 1\ 1)) \rightsquigarrow (+\ 40\ (+\ 1\ 1) \rightsquigarrow (+\ 40\ 2) \rightsquigarrow 42$$

Achtung: Scheme rundet bei Arithmetik mit Fließkommazahlen (interne Darstellung ist binär).

Bsp.: Auswertung des zusammengesetzten Ausdrucks 0.7 + (1/2)/0.25 - 0.6/0.3

Arithmetik mit rationalen Zahlen ist exakt.

Ein Wert kann an einen Namen (auch Identifier) gebunden werden , durch

$$(\text{define} < id > < e >)\ (< id > \text{Identifier}, < e > \text{Expression})$$

Erlaubt konsistente Wiederverwendung und dient der Selbstdokumentation von Programmen.

Achtung: Dies ist eine sogenannte Spezifikation und kein Ausdruck. Insbesodnere besitzt diese Spezialform keinen Wert, sondern einen Effekt: Name $< id >$ wird an den Wert von $< e >$ gebunden.

Namen können in Scheme fast beliebig gewählt werden, solange:

1. die Zeichen (kommt noch) nicht vorkommen

2. der Name nicht einem numerischen Literal gleicht

3. kein whitespace (Leerzeichen, Tabulatoren, Return) enthalten ist.

Bsp.: euro $\rightarrow$ us$

Achtung: Groß-/Kleinschreibung ist in Identifiern nicht relevant.

Eine Lambda-Abstraktion (auch: Funktion, Prozedur) erlaubt die Formulierung von Ausdrücken, die mittels Parametern konkreten Werten abstrahieren:

$$(\text{lambda}\ (< p1 >< p2 > ...)\ < e >),\ < e > \text{Rumpf}$$

$< e >$ enthälft Vorkommen der Parameter $< p1 >, < p2 >$...

(lambda ...) ist eine Spezialform. Wert der Lambda-Abstraktion ist $\# < procedure >$

Anwendung (auch: Applikation/Aufruf) der Lambda-Abstraktion führt zur Ersetzung der vorkommenden Parameter im Rumpf durch die angegebenen Argumente:

(lambda (days) (* days (* 155 min-in-a-day))) $\rightsquigarrow$ (* 365 (* 155 min-in-a-day)) $\rightsquigarrow$ 81468000

In Scheme leitet ein Semikolon einen Kommentar, der bis zum Zeilenende reicht, ein und wird vom System bei der Auswertung ignoriert.

Prozeduren sollten im Programm eine ein-bis zweizeiliger Kurzberschreibung direkt voran gestellt werden.

Chapter 3

Signaturen, Testfälle - 21.04.15

Eine Signatur prüft, ob ein Name an einen Wert einer angegebenen Sorte (Typ) gebunden wird. Signaturverletzungen werden protokolliert.

$$(: < id > < signatur >)$$

Bereits eingebaute Signaturen:

- natural $\mathbb{N}$
- integer $\mathbb{Z}$
- rational $\mathbb{Q}$
- real $\mathbb{R}$
- number $\mathbb{C}$
- boolean
- string
- image

(: ...) ist eine Spezialform ohne Wert, aber Effekt: Signaturprüfung

Prozedur-Signaturen spezifizieren sowohl Signaturen für die Parameter p1, p2, ... , pn als auch den Ergebniswert der Prozedur:

$$(< signaturp_1 > ... < signaturp_n > - >< signatur - ergebnis >)$$

Prozedur-Signaturen werden bei jeder Anwendung eine Prozedur auf Verletzung geprüft.

Testfälle dokumentieren das erwartete Ergebnis einer Prozedur für ausgewählte Argumente:

$$(check - expect < e_1 >< e_2 >)$$

Werte Ausruck $< e_1 >$ aus und teste, ob der erhaltene Wert der Erwarung (= der Wert von $< e_2 >$) entspricht.

Einer Prozedurdefinition sollten Testfälle direkt vorangestellt werden.

Spezialform: Kein Wert, aber Effekt: Testverletzung protokollieren.

Konstruktionsanleitung für Prozeduren

- ; ... (1) Kurzbeschreibung (1-2 zeiliger Kommentar mit Bezug auf Parameter)
- (: ...) (2) Signatur
- (check-expect ...) (3) Testfälle
- (define (lambda (...) ...) (4) Prozedur + Rumpf

<u>Top-Down-Entwurf</u> (Programmieren durch "Wunschdenken")

Bsp.: Zeichen Ziffernblatt (Stunden- und Minutenzeiger) zur Uhrzeit H:m auf einer analogen 24h-Uhr

- Minutenzeiger legt 360°/60 pro Minute zurück (360/60 * m)
- Stundenzeiger legt 360°/12 pro Stunde zurück (360/12 * h + 360/12 * m/60)

Chapter 4

Substitutionsmodell, Fallunterscheidung - 23.04

Reduktionsregeln für Scheme (Fallunterscheidug je nach Ausrucksart)

Wiederhole, bis keine Reduktion mehr möglich:

- Literal (1, "abc", #t, ...) [eval_{lit}]

  l ⇝ l

- Identifier id (pi, clock-face, ...) [eval_{id}]

  id ⇝ gebundener Wert

- Lambd-Abstraktion

  (lambda ()) ⇝ (lambda ()) [eval_{λ}]

- Applikation (f, e1, e2)

  – (1) f, e1, e2 reduziere, erhalte f', e1', e2'

  – (2)

    ∗ Operation f' auf e1', e2', ... falls f' primitive Operation (+, *, ...) [apply_{prim}]

    ∗ Argumentenwert e1', e2', ... Rumpf von f' einsetzen, dann Rumpf reduzieren , falls f' Lambdaabstraktion [apply_{λ}]

Beispiel: Applikation

(+ 40 2)

⇝ (#< procedure + > 40 2) ⇝ 42

eval_{lit} (+)

eval_{lit} (40)

eval_{lit} (2)•

(position-minute-hand 30)

⇝ ((lambda (m) (* degrees-per-minute m)) 30)

⇝ (* degrees-per-minute 30)

⇝ (* degress-per-minute 30)

⇝ (#<procedure * > 360/60 30)

Bezeichnen (lambda (x) (* x x)) und (lambda (r) (* r r)) die gleiche Prozedur? ⇒ Ja!

Achtung: Das hat Einfluss auf das korrekte Einsetzen von Argumenten für Parameter! (s. apply_{λ})

Das bindenen Vorkommen eines Identifiers x kann im Programmtext systematisch bestimmt werden: suche strik von "innen nach außen" bis zum ersten

- (lambda (x))
- (define x)

(Prinzip der lexikalischen Bindung)

Übliche Notation in der Mathematik: Fallunterscheidung

$$maximum(x_1, x_2) = \begin{cases} x_1, & \text{falls } x_1 \geq x_2 \\ x_2, & \text{sonst} \end{cases}$$

Tests auch (Prädikate) sind Funktionen, die einen Wert der Signatur boolean liefern. Typische primitive Tests:

- (: = (number number → boolean))
- (: < (real real → boolean)), auch $>, \leq, \geq$
- (: string=? (string string → boolean)), auch string>?, string≤?
- (: boolean? (boolean boolean → boolean))
- (: zero? (number → boolean))
- odd?, even?, positive?, negative?, ...

Binäre Fallunterscheidung: if

(if $< t_1 >< e_1 >< e_2 >$)

$$\text{Mathematisch: } \begin{cases} e_1, & \text{falls } t_1 \\ e_2, & \text{sonst} \end{cases}$$

Chapter 5

One-of Signatur - 28.04.15

Die Signatur <u>one-of</u> lässt genau einen der aufgezählten n Werte zu:

$$\text{(one-of} < e_1 > ... < e_n >)$$

Reduktion von if:

$$(\text{if } t_1\ e_1\ e_2) \rightsquigarrow \begin{cases} < e_1 >, & \text{falls t1' = \#t ; e2 wird niemals ausgewertet} \\ < e_2 >, & \text{sonst; e1 wird niemals ausgewertet} \end{cases}$$

(1) Reduziere t_1, erhalte t_1'

Spezialform Fallunterscheidung (conditional expression):

$$\text{(cond (} < t_1 >< e_1 >) \text{ ... } (< t_n >< e_n >) \text{ (else } < e_{n+1} >)) \text{ (else optional)}$$

Werte die Tests in der Reihenfolge $t_1, t_2, ..., t_n$ aus. Sobald t_i # t ergibt werte Zweig e_i aus. e_i ist das Ergebnis der Fallunterscheidung. Wenn t_n # f liefert, dann liefere

$$\begin{cases} \mathit{Fehlermeldung}, & \text{"cond: alle Tests ergaben \#f", falls kein else-Zweig} \\ < e_{n+1} >, & \text{sonst} \end{cases}$$

Reduktion von cond [eval$_{cond}$]

$$\text{(cond (} < t_1 >< e_1 >)(< t_2 >< e_2 >)...) \rightsquigarrow \begin{cases} < e_1 >, & \text{falls t1' = \#f} \\ (cond(< t_2 >< e_2 >)(...)), & \text{sonst} \end{cases}$$

Reduziere t_1, erhalte t_1'.

(cond) $\rightsquigarrow$ Fehlermeldung "Alle Tests..."

(cond (else $< e_{n+1} >$)) $\rightsquigarrow e_{n+1}$

cond ist "systematischer Zucker"

(auch: abgleitete Form) für eine verschachtelte Anwendung von 'if':

(cond ($< t_1 >< e_1 >$) ($< t_1 >< e_1 >$) ...))) entspricht (if $< t_1 >< e_1 > (if < t_1 >< e_1 > (if...))$

Spezialformen 'and' und 'or':

$$(or < t_! >< t_2 > ... < t_n >) \text{ entspricht } (if < t_1 > \#t(or < t_2 > ...))$$

$$\text{(or)} \rightsquigarrow \text{\#f}$$

$$(and < t_1 > ... < t_n > \rightsquigarrow (if < t_1 > (and < t_2 > ... < t_n >)\#f)$$

$$\text{(and)} \rightsquigarrow \#t$$

Chapter 6

Zusammengesetzte Daten - 30.04.15

Ein Charakter besteht aus drei Komponenten.

- Name des Charakters (name)
- Handelt es sich um einen Jedi? (jedi?)
- Stärke der Macht (force)

→ Datendefinition für zusammengesetzte Daten.

Konkreter Charakter:

Name	"Luke Skywalker"
jedi?	#f
force	25

; Ein Charakter (character) besteht aus

; - Name (name)

; - Jedi-Status (jedi?)

; - Stärke der Macht (force)

(define-records-procedures charakter

make-character

character?

(character-name

character-jedi

character-force))

(make-character n j f) $\rightsquigarrow < records >$ (konstruktion)

(character-name $< record > \rightsquigarrow$ n (Komponentenzugriff)

(character-jedi? $< record >$) $\rightsquigarrow$ j (Komponentenzugriff)

(character-force $< record >$) $\rightsquigarrow$ f (Komponentenzugriff)

Zusammengesetzte Daten = Records in Scheme.

Record-Definition legt fest:

- Record-Signatur
- Konstruktor (Baut aus Komponenten einen Record)

- Prädikat (liegt Record vor?)
- Liste von Selektoren (lesen jewils eine Komponenten des Records)

$$(\text{define-records-procedures} < t >$$
$$\text{make-}< t > \text{ ; Konstruktor}$$
$$< t >?$$
$$(< t > - < comp_1 > ... < t > - < comp_n > 4)) \text{ ; Liste der n-Seleketoren}$$

Verträge des Konstruktors / der Selektoren für Record-Signatur $< t >$ mit n Komponenten namens $< comp_1 > ... < comp_n >$:

- (: make-$< t >$ $(< t_1 > ... < t_n > \rightarrow < t >)$)
- (: $< t > - < comp_1 >$ $(< t > \rightarrow < t_1 >)$)
- ...
- (: $< t > - < comp_n >$ $(< t > \rightarrow < t_n >)$)

Es gilt für die Strings n, Booleans j und Integer f:

(character-name (make-character n j f)) = n

(analog für den Rest)

Interaktion von Funktionen (algebraische Eigenschaften).

Spezialform check-property:

$$(\text{check-property}$$
$$(\text{for-all } ((< id_1 > < sig_1 > ... < id_n > < sig_n >))$$
$$< e >))$$

$< e >$ bezieht sich auf $< id_1 > ... < id_n >$.

Test erfolgreich, falls $< e >$ für bel. gewählte Bindungen für $< id_1 > ... < id_n >$ immer #t ergibt.

Interaktion von Selektor und Konstruktor:

$$(\text{check-property}$$
$$(\text{for-all } ((\text{n string})$$
$$(\text{j booleans})$$
$$(\text{f integer}))$$
$$(\text{string=? } (\text{character-name } (\text{make-character n j f})) \text{ n })))$$

Beispiel: Die Summe zweier natürlicher Zahlen ist mindestens so groß wie jede dieser Zahlen: $\forall x_1, x_2 \in \mathbb{N} : x_1 + x_2 \geq max\ x_1, x_2$

$$(\text{check-property}$$
$$(\text{for-all } ((x_1 \text{ natural})$$
$$(x_2 \text{ natural}))$$
$$(\geq (+\ x_1\ x_2)\ (\text{max } x_1\ x_2))))$$

Konstruktion von Funktionen, die zusammengesetzte Daten <u>konsumieren</u>:

- Welche Record-Komponenten sind relevant für Funktionen?

  → Schablone:

  ; könnte Charakter e ein Sith-Lord sein?

  (: sith? (character → boolean))

  (define sith?

  (lambda (e) ... (character-jedi? c) ... (character-force c) ...))

Konstrukton von Funktionen, die zusammengesetzte Daten <u>konstruieren</u>:

- Der Konstruktor <u>muss</u> aufgerufen werden.

  → Schablone:

  (define

  (lambda (...)

  ... (make-$< t >$ ...) ...))

Chapter 7

Fortsetzung zusammengesetzte Daten - 05.05.1

- lego-character
  - figure
  - character
    - Name
    - jedi?
    - force

Sei $< p >$ ein Prädikat mit Signatur $(< t >\rightarrow boolean)$. Eine Signatur der Form (predicate $< p >$) gilt für jeden Wert x der Signatur $< t >$ sofern ($< p >$ x) $\rightsquigarrow \#t$. Signatur (predicate $< p >$) ist somit spezifischer (restriktiver) als die Signatur $< t >$ selbst.
Einfürhung eines neuen Signaturnamens $< new - t >$ für die Signatur $< t >$:

$$\text{(define} < new - t > \text{(signatur} < t >\text{))}$$

Bsp.:

(define farbe (signatur (one-of "Karo" "Herz" "Pik" "Kreuz")))

(define latitude (signature (predicate latitude?)))

Chapter 8

Gemischte Daten - 07.05.15

Geocoding: Übersetzte eine Ortsangabe mittels des Google Maps Geocoding API (Application Programming Interface) in eine Position auf der Erdkugel.

$$(\text{: geocoder (string} \rightarrow \text{(mixed geocode geocode-error)))}$$

Ein Geocode besteht aus:

- Adresse (address) (string)
- Ortsangabe (loc) (location)
- Nordostecke (northeast) (location)
- Südwestecke (southwest) (location)
- Typ (type) (string)
- Genauigkeit (accuracy) (string)

(: geocode-address (geocode → string)) ...

Ein geocode-error besteht aus:

- Fehlerart (level) (one-of ”TCP” ”HTTP” ”JSON” ”API”)
- Fehlermeldung (message) (string)

Teachpack: geocoder.rkt

Gemischte Daten Die Signatur

$$(\text{mixed } < t1 > \ldots < t_n >)$$

ist gültig für jeden Wert, der mindestens eine der Signatur $< t_1 >$ ... $< t_n >$ erfüllt.

Beispiel: Datendefinition:

Eine Antwort des Geocoders ist entweder

- ein Geocode (geocode) oder
- eine Fehlermeldung (geocode-error)

Beispiel (eingebaute Funktion string → number):

(: string→number (string → (mixed number (onfe of #f))))

(string→number ”42”) $\leadsto$ 42

(string→number "foo") ⇝ #f

Erinnerung:

Das Prädikat $< t >$? einer Signatur $< t >$ unterscheidet Werte der Signatur $< t >$ von <u>allen anderen</u> Werten:

(: $< t >$? (any → boolean))

Auch Prädikafür eingebaute Signaturen:

- number?
- complex?
- real?
- rational?
- integer?
- natural?
- string?
- boolean?

Prozeduren die gemische Daten der Signaturen $< t_1 > ... < t_n >$ konsumieren:

<u>Konstruktionsanleitung</u>:

(: $< f >$ ((mixed $< t_1 > ... < t_n > \rightarrow$ ...))

(define $< f >$ (lambda (x) (cond (($< t_1 >$? x) ...) ... ($< t_n >$? x)...)))))

Mittels <u>let</u> lassen sich Werte an <u>lokale Namen</u> binden:

$$(\text{let } ((< id_1 >< e_1 >) \ ... \ (< id_n >< e_n >)) < e >)$$

Die Ausdrücke $< e_1 > ... < e_n >$ werden <u>parallel</u> ausgewertet → $< id_1 > ... < id_n >$ können in $< e >$ (<u>und nur hier</u>) verwendet werden. Der Wert let-Ausdruck ist der Wert von $< e >$.

Achtung: 'let' ist verfügbar ab Sprachebene "DMdA".

'let' ist syntaktischer Zucker.

(let (($< id_1 >< e_1 >$) ... ($< id_n >< e_n >$)) $< e >$)

↔

((lambda ($< id_1 > ... < id_n >$) $< e >$) $< e_1 > ... < e_n >$)

Chapter 9

Parametrisch polymorphe Funktionen - 12.05.1

Abstand zwier geografischer Positionen l_1, l_2 auf der Erdkugel in km (lat, lng jeweils in Radian):

dist(l_1, l_2) = Erdradius in km $\cdot$ acos(cos(l_1.lat) $\cdot$ cos(l_1.lng) $\cdot$ cos(l_2.lat) $\cdot$ cos(l_2.lng) + cos(l_1.lat) $\cdot$ sin(l_1.lng) $\cdot$ cos(l_2.lat) $\cdot$ sin(l_2.lng) + sin(l_1.lat) $\cdot$ sin(l_2.lat))

Parametrisch polymorphe Funktionen

Beobachtung: Manche Prozeduren arbeiten unabhängig von den Signaturen ihrer Argumente: parametrisch polymophe Prozduren (gr.: vielgestaltig). Nutze Signaturvariablen: %a, %b, ...

Beispiel:

; Identität

(: id (%a → %a))

(define id (lambda (x) x))

; Konstante Funktion (ignoriert zweites Argument)

(: const (%a %b → %a))

(define cost (lambda (x y) x))

; Projection (ein Argument auswählen)

(: proj ((one-of 1 2) %a %b → (mixed %a %b)))

(define proj (lambda (i $x1$ x_2) (cond ((= i 1) x_1) ((= i 2) x_2))))

Eine polymorphe Signatur steht für alle Signaturen in denen die Signaturvariablen durch konkrete Signaturen ersetz werden.

Beispiel:

Wenn eine Prozedur (number %a %b → %a) erfüllt, dann auch :

(number string boolean → string)

(number boolean natural → boolean)

(number number number → number)

; Ein polymorphes Paar (pair) besteht aus

; - erster Komponente (first)

; - zweiter Komponente (rest)

; wobei die Komponenten bel. Signaturen besitzen

(define-record-procedures-pair pair-of

make-pair

pair?

(first rest))

(pair-of $< t_1 >$ $< t_2 >$) ist eine Signatur für Paare, deren erste bzw. zweite Komponente die Signaturen $< t_1 >$ bzw. $< t_2 >$ erfüllen

→ pair-of: Signatur mit (zwei) Signaturparametern

(: make-par (%a %b → (pair-of %a %b)))

(: pair? (any → boolean))

(: first ((pair-of %a %b) → %a))

(: rest ((pair-of %a %b) → %b))

Chapter 10

Listen - 12.05.15

Eine Liste von werten der Signatur $< t >$ (list-of $< t >$) ist entweder:

- leer (Signatur empty-list) oder
- ein Paar (Signatur pair-of) aus einem Wert der Signatur $< t >$ und einer Liste von Werten der Signatur $< t >$

(define list-of (lambda (t) (signature (mixed empty-list (pair-of t (list-of t))))))

Signatur empty-list bereits in Racket vordefiniert. Ebenfalls vordefiniert:

- (: empty empty-list)
- (: empty? (any → boolean))

Operationen auf Listen

- Konstruktoren:

  (: empty empty-list) ; leere Liste

  (: make-pair (%a (list-of %a) → (list-of %a)))

- Prädikate:

  (: empty? (any → boolean)) ; leer Liste?

  (: pair? (any → boolean)) ; nicht-leere Liste?

- Selektoren:

  (: first ((list-of %a) → %a)) ; Kopfelement

  (: rest (list-of %a) → (list-of %a))) ; Restliste

Chapter 11

Listenprozeduren - 19.05.2015

Prozeduren, die Listen konsumieren

Konstruktionsanleitung befolgen!

Beispiel:

; Summe der Zeichen der Liste xs

(: list-sum (list-of number) → number)

(check-expect (list-sum empty) 0)

(check-expect (list-sum one-to-four) 10)

Schablone (gemische + zusammengesetzte Daten)

(define list-sum

(lambda (xs) (cond ((empty? xs) 0)

(pair? xs) (+ (first xs) (list-sum (rest xs)))))))

(rest xs) mit Signatur (list-of number) ist selbst wieder eine kürzere Liste von Zahlen.

(list-sum (rest xs)) erzielt Fortschritt !.

Konstruktionsanleitung für Prozeduren ¡f¿, die Liste xs konsumiert.

(: $< f >$ ((list-of $< t_1 >$) → $< t_2 >$))

(define $< f >$

(lambda (xs) (cond ((empty? xs) ...)

(pair? xs) ... (first xs) ... ($< f >$ (rest xs)) ...)))

Neue Sprachebene "Macht der Abstraktion"

- Signatur (list-of % a) eingebaut
- Neuer syntaktischer Zucker:

  (list $< e_1 >< e_2 >$ ... $< e_n >$)
- Ausgabeformat für nicht leere Liste:

  #$< list\ x_1\ x_2 ...\ x_n >$

Füge Listen xs, ys zusammen (concatenation):

Beobachtung:

- Die Länge von xs bestimmt die Anzahl der rekursiven Aufrufe von cat

- Auf ys werden niemals Selektoren angewandt

Chapter 12

Rekursion auf Listen - 21.05.15

Generierung aller natürlichen Zahlen (vgl. gemische Daten). Eine natürliche Zahl (natural) ist entweder:

- die 0 (zero)
- der Nachfolger (succ) einer natürlichen Zahl.

<u>Konstruktoren</u>

(: zero natural)

(define zero 0)

(: succ (natural → natural))

(define succ (lambda (n) (+ n 1)))

Vorgängerfunktion (pred), definiert für $n > 0$:

(: pred (natural → natural))

(define pred

(lambda (n) (- n 1)))

Bedinge algebraische Eigenschaften (s. check-property):

$(\Rightarrow < p > < t >)$

Nur wenn $< p > \not\Rightarrow \#t$, wird Ausdruck $< t >$ ausgewertet und getestet ob $< t > \not\Rightarrow \ \#t$.

<u>Beispiel:</u>

Fakultätsfunktion n! ($n \in \mathbb{N}$)

0! = 1

n! = n · (n-1)!

Konstruktionsanleitung für gem. Daten:

; Berechne n!

(: factorial (natural → natural))

(define factorial

(lambda (n)

(cond ((= n 0) 1) ((> n 0) (· n (factorial (- n 1))))))

<u>Beobachtung:</u>

- Im letzten Zweig ist $n > 0 \Rightarrow$ pred anwendbar
- ($< f >$ (- n 1)) hat die Signatur $< t >$

Satz

Eine Prozedur, die nach der Konstruktionsanleitung für Listen oder natürliche Zahlen konstruiert ist, terminiert immer (= liefert immer ein Ergebnis).

Beweis: in Kürze

Die Größe eines Ausdrucks ist proportional zum Platzverbrauch des Reduktionsprozesses im Rechner

$\Rightarrow$ Wenn möglich, erzeugte Reduktionsprozesse, die konstanten Platzverbrauch - unabhängig vom Eingabeparametern - benötigen.

Chapter 13

Induktive Definitionen 11.6.15

Konstruktive Definition der natürlichen Zahlen: Definition: (Paeno-Axiome)

- (P1) $0 \in \mathbb{N}$ (Null)
- (P2) $\forall n \in \mathbb{N}$: $succ(n) \in \mathbb{N}$ (Nachfolger)
- (P3) $\forall n \in \mathbb{N}$: $succ(n) \neq 0$
- (P4) $\forall m, n \in \mathbb{N}$: $succ(m) = succ(n) \Rightarrow m = n$ Durch P3 und P4 ist $\mathbb{N}$ induktiv definiert.
- (P5) Induktionsaxiom: Für jede Menge $M \subseteq \mathbb{N}$ mit $0 \in M$ und $\forall n : (n \in M \Rightarrow succ(n) \in M)$ gilt $M = N$
  "Nichts sonst ist in $\mathbb{N}$"

13.0.1 Beweisschema vollständige Induktion

Sei $P(n)$ eine Eigenschaft einer Zahl $n \in \mathbb{N}$ (: P (natural $\rightarrow$ boolean))

Ziel: $\forall n \in \mathbb{N} : P(n)$

Definiere $M = \{n \in \mathbb{N} | P(n)\} \subseteq \mathbb{N}$ Induktionsaxiom: Falls $0 \in M$ und $\forall n(n \in M \Rightarrow succ(n) \in M)$ dann $M = N$

Induktionsanfang(I.A.): Falls $P(0)$, Induktionsschritt (I.S.) und $\forall n(P(n) = P(succ(n)))$ dann $\forall n \in \mathbb{N} P(n)$

Beispiel 1

$1 = 1$

$1 + 3 = 4$

$1 + 3 + 5 = 9$

$1 + 3 + 5 + 7 = 16$

...

$P(n) \equiv \sum\limits_{i=0}^{n}(2i + 1) = (n + 1)^2$ Zeige : $\forall n \in \mathbb{N} : P(n)$

I.A. : $P(0) \Rightarrow P(n + 1))$

$$\sum_{i=0}^{n+1}(2i+1) = \sum_{i=0}^{n}(2i+1) + (2(n+1)+1) = (n+1)^2 + (2(n+1)+1) = n^2+2n+1+2n+3 = n^2+4n+4 = (n+2)^2\Box \quad (13.1)$$

Beispiel 2

(define factorial (lambda (k)

(if (= k 0) 1 (* k (facotrial (- k 1))))))

$P(n) \equiv$ (factorial n) $= \underline{n!}$

Bemerkung: $\underline{x}$: Racket-Repräsentation für $x \in \mathbb{N}$

Zeige: $\forall n \in \mathbb{N} : P(n)$

I.A. P(0) factorial ($\underline{0}$) $*\rightsquigarrow$ ((lambda(k) ...) 0) $\rightsquigarrow$ (if(= $\underline{0}$ 0) 1 ...) $\rightsquigarrow$ (if #t 1 ...) $\rightsquigarrow 1 = 0!\checkmark$

I.S. : $\forall n$ (P(n) $\Rightarrow$ P(n + 1)) factorial (n + 1) $*\rightsquigarrow$ ((lambda (n) ...) n + 1) $\rightsquigarrow$ (if (= $\underline{(n+1)}$ 0) ... (* ...)) $\rightsquigarrow$ (if #f ... (* ...))

$\rightsquigarrow$ (* $\underline{n+1}$ (factorial (- $\underline{n+1}$ 1))) $\rightsquigarrow$ (* n + 1 (factorial n)) = (* $\underline{n+1}$ $\underline{n!}$) = $\underline{(n+1)!}$

Beispiel 3

Nach Konstuktionsanleitung über $\mathbb{N}$ konstruierte f terminieren immer.

Sei (: f (natural natural $\rightarrow$ natural)) definiert durch

(define f (lambda (n) (if (= n 0) base (step (f (- n 1)) n))))

- (: base natural)
- (: step (natural natural $\rightarrow$ natural))

Bsp: step $\equiv$ (lambda (x y) (* y x))

Dann gilt (Pn) $\equiv$ (f n) terminiert (mit Ergebnis der Signatur natural)

Zeige: $\forall n \in \mathbb{N} : P(n)$

I.A. P(0): (f 0) $*\rightsquigarrow$ (if (= $\underline{0}$ 0) base ...) $\rightsquigarrow$ (if #t base ...) $\rightsquigarrow$ base $\checkmark$

I.S. $\forall n(P(n) \Rightarrow P(n+1)) : (f\ \underline{n+1})$

$\rightsquigarrow$ (if (= $\underline{n+1}$ 0) ... (step ...)) $\rightsquigarrow$ (if #f ... (step ...)) $\rightsquigarrow$ (step (f (- $\underline{n+1}$ 1) n + 1))

$\rightsquigarrow$ (step $\underbrace{(f\underline{n})}$ $\underline{n+1}$) $\Rightarrow$ (step (f n) n+ 1) terminiert. $\checkmark$

- - - - terminiert

Definition: Listen, endliche Folgen

Die Menge M* (= Liste mit Elementen aus M, (list-of M)) ist induktiv definiert.

- (L1) empty $\in$ M* - (leere Liste)
- (L2) $\forall x \in M, xs \in M*$ (make-pair x xs) $\in$ M* (make-pair x xs) $\in$ M* - nicht-leere Liste
- (L3) Nichts sonst ist in M* (P5)

Beweisschema der Listeninduktion:

Sei P(xs) eine Eigenschaft von Listen über M.

(: P ((list-of M) $\rightarrow$ boolean)

I.A.: Falls P(empty) I.S.: und $\forall x \in$ M, xs : (P(xs) $\Rightarrow$ P((make-pair x xs)))

dann $\forall xs \in$ M* : P(xs) $\checkmark$