

Signatur sig	akzeptierte Werte
natural	$\mathbb{N}_0$ (natürliche Zahlen mit 0)
integer	$\mathbb{Z}$ (ganze Zahlen)
rational	$\mathbb{Q}$ (rationale Zahlen, Brüche)
real	$\mathbb{R}$ (reelle Zahlen)
number	$\mathbb{C}$ (komplexe Zahlen, umfasst $\mathbb{N}_0, \mathbb{Z}, \mathbb{Q}, \mathbb{R}$)
boolean	$\{ \#t, \#f \}$ (Wahrheitswerte, Booleans)
strings	Zeichenketten
image	Bilder
(enum "Mo" "Di" "Mi" "Do" "Fr" "Sa" "So")	

(define new-t (signature t))

z.B. (mixed — —)
 mind. 1 erfüllt
 (predicate —)
 sig: $(t \rightarrow \text{boolean})$, jeder Wert der $\in t$ und Pred. erfüllt
 (combind — —) als and

Predikat $t?$ zu
 Signatur t

(cond (t₁ e₁)
 (t_n e_n)
 (else e_{n+1}))

(if t₁ e₁ e₂)

Fallunterscheidung: Spezialform match vergleicht Wert mit gegebenen Patterns pat₁, pat₂, ..., pat_n:

(match e
 (pat₁ e₁)
 (pat₂ e₂)
 ...
 (pat_n e_n))
 Patterns pat_i werden von oben nach unten gegen e gematched

Falls pat_i das erste auf e matchende Pattern ist, ist Zweig e_i das Ergebnis des match. Sonst Programmabbruch.

let:

(let ((id₁ e₁)
 (id₂ e₂)
 ...
 (id_n e_n))
 e)
 ; der Wert von e ist Wert
 ; des gesamten let-Ausdrucks

id₁, ..., id_n können in e (nur hier) verwendet werden:
 1. Außerhalb des let sind die id_i unbekannt.
 2. In den e_i sind die id_i unbekannt (alle e_i werden [potentiell] parallel ausgewertet).

(let* ((id₁ e₁)
 (id₂ e₂)
 ...
 (id_n e_n))
 e)
 Reihenfolge der Bindungen
 ; id_i sichtbar

In e_{i+1} sind die Identifier id₁, ..., id_i sichtbar.
 Zirkuläre Abhängigkeiten sind so nicht formulierbar.

(let ((id₁ e₁)
 ...
 (id_n e_n))
 e) = ((lambda (id₁ ... id_n) e)
 e₁ ... e_n)

```

(define a 39)
(define b 3)

(let ((a 40)
      (b 2))
  (+ a b))
  
```

Polymorphe Funktionen:

wissen nichts über Argumente
 können nur reproduzieren &
 weitergeben

• leer oder
 • ein Paar, bestehend aus
 • einem Listenkopf der Signatur t und
 • einer Restliste der Signatur (list-of t):

(list-of t): Listen von Werten der Signatur t
 (define list-of (lambda (t)
 (signature (mixed empty-list
 (cons-of t (list-of t))))))

Literal l (1, #t, "abc", #<function:0>, ...): [eval_lit]

$[[l]]^k \equiv l$

Lambda-Abstraktion (lambda (v₁ ... v_n) e): [eval_λ]

$[[(\text{lambda } (v_1 \dots v_n) e)]]^k \equiv (\text{lambda } (v_1 \dots v_n) e)$

Identifier id: [eval_id]

$[[id]]^k \equiv [[e]]^k$, falls (define id e) gilt

Applikation (f e₁ ... e_n)

$[[f e_1 \dots e_n]]^k \equiv \begin{cases} [[e_1]]^{k+1} \dots [[e_n]]^{k+1}, & \text{falls } f = \#<\text{function:0}> \\ [[e(v_1 \mapsto [[e_1]]^{k+2}, \dots, v_n \mapsto [[e_n]]^{k+2})]]^k, & \text{falls } f = (\text{lambda } (v_1 \dots v_n) e) \\ [[[[f]]^{k+1} e_1 \dots e_n]]^k, & \text{sonst} \end{cases}$

e(v₁→e₁, ..., v_n→e_n) bedeutet: In Ausdruck e wird Identifier v_i durch Ausdruck e_i ersetzt.

Beispiel: (+ (* 4 y) x) (x→2, y→10) = (+ (* 4 10) 2).

Fallunterscheidung (cond (t₁ e₁) (t₂ e₂) ...)

$[[(\text{cond } (e_1 e_1) \dots (e_n e_n))]^k \equiv \begin{cases} [[e_1]]^k, & \text{falls } e_1 = \#<\text{function:0}> \\ [[(\text{cond } (\#f e_1) (t_2 e_2) \dots))]^k, & \text{falls } e_1 = (\text{lambda } (v_1 \dots v_n) e) \\ [[[[f]]^{k+1} e_1 \dots e_n]]^k, & \text{sonst} \end{cases}$

$[[(\text{cond})]]^k \equiv \text{Abbruch } \perp$

$[[(\text{cond } (t_1 e_1) (t_2 e_2) \dots))]^k \equiv [[(\text{cond } ([[t_1]]^{k+1} e_1) ([[t_2]]^{k+1} e_2) \dots))]^k$

(or — —) (or) → #f
 (and — —) (and) → #t

match:

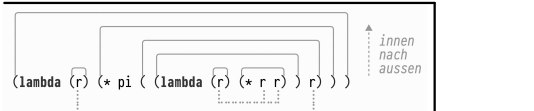
Jedes Pattern pat_i kann eine von vier Formen annehmen:

- l (Literal):**
e matched, falls e → l.
- _ (Don't care, Unterstrich):**
e matched immer.
- id (Identifier):**
e matched immer, im Zweig e_i ist id an e gebunden.
- (make-t pat₁₁ ... pat_{1k}) (Record-Konstruktor, k ≥ 0):**
e matched, falls es via (make-t x₁ ... x_k) konstruiert wurde und $\forall$ alle x_j auf pat_{1j} (j = 1...k) matchen.

(match (+ 40 2)
 (4 1 —)
 (4 2 —)
 (x (number → string x))
 (— ...))

apply → eval-id —
 → eval-λ → apply-λ

David Kreidmann



Tests: equal? / = > <
 string =? string >? → alphabet.
 boolean=?
 zero? odd? even?

(cons y xs) y xs
 (list y xs) y xs

records

(: make-demo (string natural → demo))
 (: demo? (any → boolean))
 (: demo-exists (demo → string))
 (define-record demo
 make-demo
 demo?
 (demo-exists string) (demo-zweites nat))

(check-property (for-all ((id₁ t₁)
 (id_n t_n))
 * (produkt))) (⇒ p 'produkt')

(check-property (for-all ((n string)
 (j boolean)
 (f natural))
 (string=? (character-name (make-character n j f)) n)))

Konstruktorableitung gemischte Daten

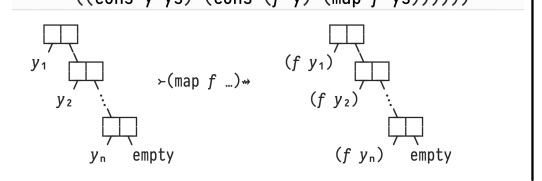
(: f (... (mixed t₁ ... t_n) ... → ...))
 (define f (lambda (... x ...) ...))
 (cond ((t₁? x) ...) ; 1 Signatur testen
 ...
 ((t_n? x) ...) ; hier hat x definitiv Signatur t_n

rekursive Listen:

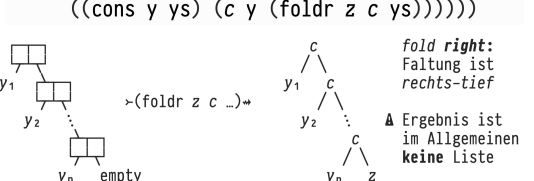
(: f ((list-of t₁) → t₂))
 (define f (lambda (xs)
 (cond ((empty? xs) ...)
 ((cons? xs) ... (first xs) ... (f (rest xs)) ...))))
 (cond
 ((cons y empty) ...) ggf.
 ((make-demo-record — —) ...))

HOF:

(filter (lambda (x) ...) → boolean (list))
 (: map ((%a → %b) (list-of %a) → (list-of %b)))
 (define map (lambda (f xs)
 (match xs
 (empty empty)
 ((cons y ys) (cons (f y) (map f ys))))))



(: foldr ((%b (%a %b → %b) (list-of %a) → %b))
 (define foldr (lambda (f z xs)
 (match xs
 (empty z)
 ((cons y ys) (c y (foldr z c ys))))))



fold right:
 Faltung ist rechts-tief
 Ergebnis ist im Allgemeinen keine Liste

```

;Signatur list-length-13 als Kombination aus (list-of t) und listlength13
(define list-of-13
  (lambda (t)
    (signature (combined (list-of t) listlength13))))

;Aus listlength13-pred eine Signatur erstellen
(define listlength13 (signature (predicate listlength13-pred)))

;Funktion die testet ob eine Liste die Länge 13 hat
(: listlength13-pred ((list-of %a) -> boolean)) ;Signatur
(define listlength13-pred ;Erstellen der Funktion und überprüfen der Länge
  (lambda (xs)
    (= (length xs) 13)))

```

```

(: make-tuple (%a %b -> (tuple-of %a %b)))
(: tuple? (any -> boolean))
(: tuple-fst ((tuple-of %a %b) -> %a))
(: tuple-snd ((tuple-of %a %b) -> %b))
(define-record (tuple-of t1 t2)
  make-tuple ;Konstruktor
  tuple? ;Prädikat
  (tuple-fst t1) ;Selektor
  (tuple-snd t2)) ;Selektor

```

TUPLE
↓ split, wenn

MENGEN: neue Menge

```

(define set-of
  (lambda (t)
    (signature (t -> boolean))))

(: list->set ((list-of %a) -> (set-of %a)))
(define list->set
  (lambda (xs)
    (fold empty-set set-insert xs)))

(: empty-set (set-of %a))
(define empty-set
  (lambda (x)
    #f))

(: set-insert (%a (set-of %a) -> (set-of %a)))
(define set-insert
  (lambda (x S)
    (lambda (y)
      (or (equal? y x)
          (S y)))))

(define set-member?
  (lambda (x S)
    (S x)))

(: set-delete (%a (set-of %a) -> (set-of %a)))
(define set-delete
  (lambda (x S)
    (lambda (z)
      (if (equal? z x)
          #f
          (S z)))))

; Differenzquotient von f mit Differenz h
; ((derive f) x) = f'(x)
(define diff-quot
  (lambda (h f)
    (/ (- (f (+ x h)) (f x)) h)))

; Ableitung (Differenzquotient mit Differenz h nahe 0)
; ((derive f) x) = f'(x)
(define derive
  (lambda (f x)
    (let ((diff-quot (diff-quot 0.00001 f)))
      (diff-quot x f))))

```

Fällfunktionen

```

(define snoc
  (lambda (xs n)
    (match xs
      (empty (cons n empty))
      ((cons y ys) (cons y (snoc ys n))))))

(define reversed
  (lambda (xs)
    (match xs
      (empty empty)
      ((cons y empty) (cons y empty))
      ((cons y ys) (snoc (reverse ys) y))))

(define list-max
  (lambda (xs)
    (match xs
      (empty (violation "Liste leer"))
      ((cons y empty) y)
      ((cons y ys) (max y (list-max ys)))))

(define all-equal?
  (lambda (xs)
    (match xs
      (empty #t)
      ((cons y empty) #t)
      ((cons y (cons z zs))
       (and (equal? y z) (all-equal? (cons z zs))))))

```

```

(: curry ((%a %b -> %c) -> (%a -> (%b -> %c))))
(define curry
  (lambda (f)
    (lambda (x)
      (lambda (y)
        (f x y))))))

(: uncurry ((%a -> (%b -> %c)) -> (%a %b -> %c)))
(define uncurry
  (lambda (f)
    (lambda (x y)
      (f x y))))

```

FOLDL

```

(: foldl (%b (%a %b -> %b) (list-of %a) -> %b))
(define foldl
  (lambda (z c xs)
    (match (reverse xs)
      (empty z)
      ((cons y ys) (c (foldl z c (reverse ys)) y)))))

```

$(c (c (c z 1) 2) 3)$

```

(: indexed-foldr (%b (natural %a %b -> %b) (list-of %a) -> %b))
(define indexed-foldr
  (lambda (z c xs)
    (letrec ((ifolder-worker (lambda (z c xs n)
      (match xs
        (empty z)
        ((cons y ys) (c n y (ifolder-worker z c ys (+ n 1))))
        (ifolder-worker z c xs 0))))))
    (ifolder-worker z c xs 0)))

```

LETREC

```

(check-expect (...) value)
(check-error (...) „message“)
(check-within (...) value +/-...)
(property)

```

TESTS

PRIMITIVES:

floor (abrunden)
ceiling (aufrunden)
(max --) (größere Zahl)
(violation "...") ERROR
(list-ref liste [0, ..., n])
(append ys (list x))

```

(let ((id1 e1)
      (id2 e2)
      ...
      (idn en))
  e)

```

$((\lambda (id_1 \dots id_n) e) e_1 \dots e_n)$

```

((backwards-worker |
  (lambda (xs acc)
    (match xs
      (empty acc)
      ((cons y ys) (backwards-worker ys (cons y acc))))

```

WORKER

```

(: cat ((list-of %a) (list-of %a) -> (list-of %a)))
(define cat
  (lambda (xs ys)
    (match xs
      (empty ys)
      ((cons z zs)
       (cons z (cat zs ys))))))

```

REKURSION
dog
(foldr ys cons xs)

```

(define nth
  (lambda (n xs)
    ((compose first (repeat (- n 1) rest)) xs)))

```

nat. Zahl Rekursion

```

(: f (natural -> t))
(define f
  (lambda (n)
    (cond ((= n 0) ...)
          (> n 0) ... (f (- n 1)) ...))))

```

```

(: f (natural -> t))
(define f
  (lambda (n)
    (cond ((= n 0) ...)
          ((even? n) ... (f (/ n 2)) ...)
          ((odd? n) ... (f (/ (- n 1) 2)) ...))))

```

Halbierung

```

(: f (natural -> t))
(define f
  (lambda (n)
    (cond ((= n 0) ...)
          (> n 1) ... (f (- n 1)) ... (f (- n 2)) ...))))

```

zweifach

```

(define every-nth
  (lambda (n xs)
    (if (= 0 n) xs
        (match xs
          (empty empty)
          ((cons y ys) (cons y (every-nth n (drop (- n 1) ys)))))))

```

```

(define split-list
  (lambda (xs)
    (match xs
      (empty (make-tuple empty empty))
      ((cons y empty) (make-tuple (cons y empty) empty))
      ((cons y (cons z zs))
       (make-tuple (cons y (tuple-fst (split-list zs)))
                    (cons z (tuple-snd (split-list zs))))))

```

```

(define weave-lists
  (lambda (tpl)
    (match tpl
      ((make-tuple empty empty) empty)
      ((make-tuple empty Liste) Liste)
      ((make-tuple Liste empty) Liste)
      ((make-tuple (cons y ys) (cons z zs))
       (cons y (cons z (weave-lists (make-tuple ys zs)))))))

```

```

(define weave-lists2
  (lambda (tpl)
    (let ((a (tuple-fst tpl))
          (b (tuple-snd tpl)))
      (match a
        (empty b)
        ((cons y ys) (cons y (weave-lists2 (make-tuple b ys))))))

```