INLEIDING TOT DE λ-CALCULUS

VRIJE
UNIVERSITEIT
BRUSSEL

OVERZICHT

Geïntroduceerd door Alonzo Church en Stephen Kleene in de jaren 1930s om het begrip functie abstract te kunnen bestuderen

Ook gebruikt om het begrip berekenbaarheid te formaliseren

- Om na te gaan welke functies al dan niet berekenbaar zijn, of nog, welke (wiskundige) problemen oplosbaar zijn, en welke niet

(Een functie *f* is berekenbaar desda er bestaat een programma P dat de functie f berekent, m.a.w. Als P stop voor input *a* dan geeft die de correcte output en anders stop P nooit).

λ-CALCULUS & PROGRAMMEERTALEN

Hoewel ontwikkeld vóór het bestaan van concrete computertalen vormt het formalisme de basis van functionele programmeertalen zoals Lisp en Scheme

Basisconcept van deze programmeertalen zijn nl functies

Invloed op programmeertalen:

- "Call by name" mechanism
  - Parameters worden pas geëvalueerd indien nodig
- Hogere orde functies
  - functie met functies als parameters of
  - Output is een functie.

FUNCTIES

Een numerieke functie is op twee manieren te definiëren

▶ Extensioneel (verzameling theoretisch)

{…, (-1,1),(0, 2), (1, 3), … }   extensie

▶ Intentioneel (functievoorschrift)

Voorbeeld:

$f(x) = 2 + x$ functievoorschrift

FUNCTIES

$f(x) = 2 + x$

In λ-achtige notatie:

parameter λx.((+)2)x voorschrift

- Te lezen als:

  Om de waarde van f(x) te berekenen voor de parameter x:

  Pas de functie + toe op 2: dit resulteert in de functie '+2'

  +2 (x) = x + 2

  Pas de functie '+2' toe op de parameter x

- Merk prefix notatie en geen naam nodig is voor de functie.

VOORBEELD IN SCHEME

```
(define (add-three number) (+ number 3))

(define (add-three-to-each list)
  (every add-three list))

> (add-three-to-each '(1 9 9 2)) (4 12 12 5)


(define (add-three-to-each list)
  (every (lambda (number) (+ number 3)) list))

> (add-three-to-each '(1 9 9 2)) (4 12 12 5)
```

λ-CALCULUS - BASIS

2 fundamentele bewerkingen:

- **Abstractie**: het maken van een functievoorschrift
- **Toepassen** (of aanroepen) van het functievoorschrift op een parameter.

SYNTAX OF λ-CALCULUS

Definitie λ-expressies

Laat V een verzameling variabelen zijn V = {x, y, z, …}

De verzameling van λ-expressies wordt als volgt gedefinieerd:

1. alle variabelen (elementen van V) zijn λ-expressies
2. Als M en N λ-expressies zijn dan is
   (M)N een λ-expressie (toepassing)
3. Als x ∈ V en M λ-expressie dan is
   λx.M een λ-expressie (abstractie)
4. Niets anders is een λ-expressie

M.a.w.: definitie van functie in λ-calculus: λformele-parameter.functievoorschrift

aanroep van een functie in λ-calculus: (functie) actuele-parameter

VRIJE UNIVERSITEIT BRUSSEL

De verzameling van λ-expressies wordt genoteerd als *Λ*

VOORBEELDEN λ-EXPRESSIES

$\lambda x.x$

$\lambda x.\lambda y.(y)x$

$(\lambda y.(x)y)\ \lambda x.(u)x$

$\lambda x.\ \lambda y.x$

$\lambda f.\ \lambda x.(f)x$

$\lambda f.\ \lambda x.(f)(f)x$

SYNTAX OF λ-CALCULUS

Toepassing van rechts naar links:

(P)(Q)x   komt overeen met P(Q(x))

(in klassieke functienotatie)

$\underbrace{((P)Q)}_{\text{functie}}x$ komt overeen met eerst P toepassen op Q en daarna het resultaat op x

CURRYING

Functies hebben slechts 1 parameter, is dit een fundamentele beperking?

Functie met meerdere parameters kan steeds herleid worden tot functies met 1 parameter

f: domein $\rightarrow$ co-domein

waarbij co-domein zelf weer functies kan bevatten

m.a.w: $f: A \times B \rightarrow C$ wordt

$g: A \rightarrow (B \rightarrow C)$ functie

$g(a) = f_a$ en $f_a(b) = f(a,b)$

<u>Voorbeeld</u>:

plus(2,3) wordt $+(2) = +_2$ en $+_2(3) = 5$

CURRYING

In het algemeen:

$$A_1 \times \ldots \times A_n \to B \text{ wordt } A_1 \to (A_2 \to \ldots (A_n \to B)$$

Deze techniek wordt **currying** genoemd, genoemd naar een Amerikaanse wiskundige Curry.

Voordeel:

- ▶Theorie te beperken tot functies met 1 argument

Nadeel:

- ▶Leesbaarheid

Voorbeeld: $\lambda x.\ \lambda y.(y)x$, is functie met 2 argumenten die zijn 2de argument toepast op zijn eerste argument of m.a.w. g(x,f) = f(x)

CURRYING IN SCHEME

```
(lambda (x y)
  (+ x y))
```

Currying →

```
(lambda (x)
  (lambda (y)
    (+ x y)))
```

```
((lambda (x y)
   (+ x y)) 3 4)
```

```
(((lambda (x)
    (lambda (y)
      (+ x y))) 3) 4)
```

geeft bij evaluatie (x gebonden aan 3):

```
((lambda (y)
   (+ 3 y)) 4)
```