

# Programowanie Funkcyjne

lato 2013/2014

Jakub Kozik

Informatyka Analityczna  
tcs@jagiellonian

## Punkty

- zadania programistyczne submitowane przez Satori (zadania różnie punktowane, w sumie 80 punktów)
- aby uzyskać pozytywną ocenę należy zaliczyć wszystkie zadania oznaczone jako obowiązkowe
- egzamin (20 punktów)
- aby uzyskać pozytywną ocenę należy z egzaminu uzyskać przynajmniej 10 punktów

## Ostateczna ocena

0-50	ndst
50-60	dst
60-70	+dst
70-80	db
80-90	+db
90-100	++db

## Część 1: Lambda rachunek i podstawy SML

- podstawowe konstrukcje SML'a i ich odpowiedniki w lambda rachunku
- strategie ewaluacji termów/programów
- system typów (Hindley-Milner)
- podstawowe techniki programowania funkcyjnego

## Część 2: Funkcyjne struktury danych

- ocena wydajności programów funkcyjnych
- amortyzowana analiza persystentnych struktur danych
- eliminacja amortyzacji
- implementacje z uleniwianiem/wymuszaniem obliczeń

## Część 3: Haskell

- leniwa ewaluacja
- monady
- ...

Paweł Urzyczyn, Rachunek Lambda, skrypt dostępny na stronie autora

Robert Harper, Programming in Standard ML, Working Draft dostępny na stronie autora

Chris Okasaki, Purely Functional Data Structures, Cambridge University Press 1999 (wczesna wersja książki (rozprawa doktorska) dostępna na stronie autora)

Simon L. Peyton Jones, The Implementation of Functional Programming Languages, Prentice Hall 1987 (książka w wersji elektronicznej jest udostępniana przez autora)

Bryan O'Sullivan, Don Stewart, John Goerzen, Real World Haskell, O'Reilly Media, 2008 (książka w wersji elektronicznej jest udostępniana przez autorów)

## Składnia

- zmienne są termami (przeliczalny zbiór zmiennych  $Var$ )
- jeśli  $T$  jest termem a  $x$  jest zmienną to  $\lambda x. T$  jest termem (abstrakcja)
- jeśli  $T$  oraz  $S$  są termami to  $(T \cdot S)$  jest termem (aplikacja)

## Konwencje

- pomijanie  $\cdot$  przy aplikacji  $(T \cdot S) \equiv (TS)$
- pomijanie nawiasów – domyślne nawiasowanie do lewej  
 $RST \equiv ((RS)T)$
- grupowanie abstrahowanych zmiennych  $\lambda xy. T \equiv \lambda x. (\lambda y. T)$

## $\beta$ redukcja

$$(\lambda x. T)S \rightarrow_{\beta} T[x \leftarrow S]$$

(pod warunkiem że żadne wolne wystąpienie zmiennej w  $S$  nie zostaje związane w  $T[x \leftarrow S]$ )

## $\alpha$ równoważność

Termy, które różnią się tylko nazwami zmiennych związanych, są równoważne i można je sobą zastępować.

$$(\lambda s\ z. s(s(z)))(\lambda s\ z. s(s(z)))$$

# Obliczenia w lambda rachunku

## Postać normalna

Term jest w *postaci (beta) normalnej* jeśli nie zawiera  $\beta$ -redex'u.

program	$\leftrightarrow$	term
ewaluacja programu	$\leftrightarrow$	wykonywanie $\beta$ redukcji
wynik obliczenia	$\leftrightarrow$	term w postaci normalnej

## Theorem (Church-Rosser)

Jeśli  $P \beta\leftarrow M \twoheadrightarrow_{\beta} Q$  to istnieje  $M'$  taki że  $P \twoheadrightarrow_{\beta} M' \beta\leftarrow Q$ .

## Wniosek

Każdy term ma co najwyżej jedną postać normalną.

Wzbogacamy  $\lambda$  rachunek o:

- 1 wyrażenia let i letrec
- 2 pattern-matching lambda abstractions
- 3 operator []
- 4 wyrażenia case
- 5 stałe: małe liczby, znaki, funkcje na małych liczbach itp.



# let

## Składnia (bez pattern-matchingu)

**let**  $v = B$  **in**  $E$

**let**

$w = A$

$v = B$

**in**  $E$

$=$

**let**  $w = A$  **in**

**(let**  $v = B$  **in**  $E$ **)**

## Tłumaczenie

**(let**  $v = B$  **in**  $E$ **)**  $\equiv ((\lambda v. E) B)$

## Składnia (bez pattern-matchingu)

**letrec**

a = A

b = B

...

n = N

**in** E

## Tłumaczenie (dla jednej zmiennej)

$$(\text{letrec } v = B \text{ in } E) \equiv (\text{let } v = Y (\lambda v. B) \text{ in } E)$$

(Y jest kombinatorem punktu stałego)

letrec f = \n. IF (n=1) THEN 1 ELSE (n \* (f (n-1))) in f 4

$$F = \lambda f\ n. \mathbf{if}(n = 1)\mathbf{then}\ 1\ \mathbf{else}(n * f(n - 1))$$

$$(Y\ F)\ 4 \twoheadrightarrow F\ (Y\ F)\ 4$$

$$= (\lambda f\ n. \mathbf{if}(n = 1)\mathbf{then}\ 1\ \mathbf{else}(n * f(n - 1)))\ (Y\ F)\ 4$$

$$\twoheadrightarrow \mathbf{if}(4 = 1)\mathbf{then}\ 1\ \mathbf{else}(4 * ((Y\ F)(4 - 1)))$$

$$\twoheadrightarrow 4 * ((Y\ F)(4 - 1)) \twoheadrightarrow 4 * (F(Y\ F)(4 - 1))$$

$$\twoheadrightarrow 4 * (\mathbf{if}(4 - 1 = 1)\mathbf{then}\ 1\ \mathbf{else}((4 - 1) * ((Y\ F)(4 - 1 - 1))))$$

$$\twoheadrightarrow 4 * (\mathbf{if}(3 = 1)\mathbf{then}\ 1\ \mathbf{else}(3 * ((Y\ F)(3 - 1))))$$

$$\twoheadrightarrow 4 * (3 * ((Y\ F)(3 - 1))) \twoheadrightarrow 4 * (3 * (F(Y\ F)(3 - 1)))$$

$$\twoheadrightarrow 4 * (3 * (2 * (F(Y\ F)(2 - 1)))) \twoheadrightarrow 4 * (3 * (2 * 1))$$

$$\twoheadrightarrow 4 * (3 * 2) \twoheadrightarrow 4 * 6 \twoheadrightarrow 24$$

## Tłumaczenie

$$(\text{let } v = B \text{ in } E) \equiv ((\lambda v. E)B)$$

- ❶ polimorfizm - system typowania inaczej traktuje konstrukcje `let` i `λ` abstrakcje
- ❷ wydajność - specyficzna aplikacja  $(\lambda v. E)$  jest aplikowana tylko do konkretnego argumentu  $B$
- ❸ `letrec` generuje  $Y$ 
  - ❶ można wydajniej obliczać bezpośrednio na termach z `letrec`
  - ❷ termy z  $Y$  nie mogą być ewaluowane gorliwie

Wzbogacamy  $\lambda$  rachunek o:

- 1 wyrażenia `let` i `letrec`
- 2 pattern-matching lambda abstractions
- 3 operator `[]`
- 4 wyrażenia `case`

# Structured Types (algebraic types)

## Typy wyliczeniowe

```
datatype color = RED | GREEN | BLUE  
datatype bool = TRUE | FALSE
```

## Tuples/Typy produktowe

```
datatype pair 'a 'b = PAIR 'a * 'b  
datatype triple 'a 'b 'c = TRIPLE 'a * 'b * 'c
```

## Union types

```
datatype shape = CIRCLE color * int  
              | RECTANGLE color * int * int  
datatype intList = INIL | ICONS int * intList  
datatype list 'a = NIL | CONS 'a * ('a list)
```

# Pattern matching - przykłady

## Overlapping patterns & Constant patterns

```
factorial 0 = 1  
factorial n = n * factorial (n-1)
```

## Nested patterns

```
getEven [] = []  
getEven [x] = []  
getEven (x:(y:ys)) = y: (getEven ys)
```

## Multiple arguments

```
xor FALSE y = y  
xor TRUE FALSE = TRUE  
xor TRUE TRUE = FALSE
```

# Pattern matching - przykłady

## Non-exhaustive sets of equations

```
head (x:xs) = x
```

## Conditional equations

```
fibb n = 1,  n<2  
fibb n = fibb (n-1) + (fibb (n-2))
```

## Repeated variables (?)

```
noDups [] = []  
noDups [x] = [x]  
noDups (x:x:xs) = noDups (x:xs)  
noDups (x:y:ys) = x: (noDups (y:ys))
```



## Definicja

### Wzorzec (*Pattern*)

- 1 *zmienna jest wzorcem*
- 2 *stała jest wzorcem (*int*, *char*, *bool* itp.)*
- 3 *jeśli  $c$  jest konstruktorem arności  $r$ , oraz  $p_1, \dots, p_r$  są wzorcami, to wzorcem jest*

$$(c \ p_1 \ \dots \ p_r)$$

*Dodatkowo wszystkie nazwy zmiennych we wzorcu muszą być różne.*

Wzorce postaci  $(c \ p_1 \ \dots \ p_r)$  nazywamy *sum construction patterns* jeśli  $c$  jest konstruktorem dla typu będącego sumą (*union types*).

Wzorce postaci  $(c \ p_1 \ \dots \ p_r)$  nazywamy *product construction patterns* jeśli  $c$  jest konstruktorem dla typu produktowego (*product types*).

## Rozszerzenie składni - term:

- 1 wzorzec (!)
- 2 aplikacja ( $T_1 \cdot T_2$ )
- 3 abstrakcja ( $\lambda p.T$ ) (gdzie  $p$  jest wzorcem a  $T$  jest termem)
- 4 ...
- 5 let, letrec ...

## Przykład

$$\text{fst } (x,y) = x \quad \rightarrow \quad \text{fst} = \lambda(PAIR\ x\ y).x$$

# Pattern matching w rozszerzonym lambda rachunku

## Problem

```
null NIL = true  
null (CONS x xs) = false
```

## Rozszerzenie składni - term:

- 1 wzorzec (!) (w tym stała FAIL)
- 2 aplikacja ( $T_1 \cdot T_2$ )
- 3 abstrakcja ( $\lambda p. T$ ) (gdzie  $p$  jest wzorcem a  $T$  jest termem)
- 4 operator binarny  $[]$  (będziemy zapisywać infiksowo)
- 5 let, letrec ...

## operator $[]$

$$\begin{aligned}a [] b &= a && \text{jeśli } a \neq \text{FAIL} \text{ oraz } a \neq \perp \\ \text{FAIL} [] b &= b \\ \perp [] b &= \perp\end{aligned}$$

# Pattern matching w rozszerzonym lambda rachunku

## Przykład

```
null NIL = true  
null (CONS x xs) = false
```

$$\lambda v. ( ((\lambda NIL.true)v) \ [] \ ((\lambda (CONS\ x\ xs).false)v) )$$

## Przykład

```
reflect (LEAF n) = LEAF n  
reflect (BRANCH t1 t2) = BRANCH (reflect t2) (reflect t1)
```

```
letrec  
  reflect =  $\lambda t. ($   
     $\ [] \ ((\lambda (BRANCH\ t1\ t2).BRANCH\ (reflect\ t2)(reflect\ t1))t)$   
     $\ [] \ ERROR)$   
  in  $E$ 
```

# Pattern matching - wiele argumentów

$$\begin{aligned} & f \ p_1 \ \dots \ p_m = E \\ & \quad \lambda v_1 \dots \lambda v_n. ((\lambda p_1 \dots \lambda p_m. E) v_1 \dots v_m \ [] \ \text{ERROR}) \end{aligned}$$

## Dodatkowa reguła dla FAIL

$$(\text{FAIL } A) \rightarrow \text{FAIL}$$

## Przykład

```
xor False y = y
xor True False = True
xor True True = False
```

$$\begin{aligned} \text{xor} = \quad & \lambda x. \lambda y. ( \\ & \quad [] \quad ((\lambda \text{TRUE}. \lambda \text{FALSE}. \text{TRUE}) \times y) \\ & \quad [] \quad ((\lambda \text{TRUE}. \lambda \text{TRUE}. \text{FALSE}) \times y) \\ & \quad [] \quad \text{ERROR}) \end{aligned}$$

# Pattern matching - guards

## Przykład

```
foo (x:xs) = x, x<0
```

```
foo (x:[]) = x
```

```
foo (x:xs) = foo xs
```

```
foo =      λv.(((λ(CONS x xs).IF (x < 0) THEN x ELSE FAIL) v)
           []((λ(CONS x NIL).x) v)
           []((λ(CONS x xs).foo xs) v)
           []ERROR)
```

## Przykład - guard TRUE

```
fac n = 1,  n=0
```

```
fac n = n * (factorial (n-1))
```

```
fac =      λv.((λn.(IF (n = 0) THEN 1 ELSE (n * (fac(n - 1))))) v
              []ERROR)
```

# Powtórzone zmienne

```
nasty x x True = 1
```

```
nasty x y z    = 2
```

```
nasty' x y True = 1, x=y
```

```
nasty' x y z    = 2
```

$$(nasty \perp 3 \text{ False}) \neq (nasty' \perp 3 \text{ False})$$

```
multi p q q p = 1
```

```
multi p q r s = 2
```

$$multi \perp 1 2 3 = ?$$

# Dopasowania typów algebraicznych

$$\begin{array}{ll} (\lambda(s \ p_1 \dots p_r).E) (s \ a_1 \dots a_r) \rightarrow & (\lambda p_1 \dots p_r.E) \ a_1 \dots a_r \\ (\lambda(s \ p_1 \dots p_r).E) (s' \ a_1 \dots a_r) \rightarrow & \text{FAIL} \\ \text{Eval}[[ (\lambda(s \ p_1 \dots p_r).E) ]] \perp = & \perp \end{array}$$



# (Leniwe) Dopasowywanie typów produktowych

## Przykład

`zeroAny`     `x`     `= 0`

`zeroList`    `[]`    `= 0`

`zeroPair`    `(x,y)` `= 0`

*zeroAny*  $\perp = 0$

*zeroList*  $\perp = \perp$

*zeroPair*  $\perp = ?$

## Destruktory par

`addPair`     `(x,y)` `= x+y`

*addPair*  $= \lambda p.((\text{SEL-PAIR-1 } p) + (\text{SEL-PAIR-2 } p))$

## Destruktory par

`addPair`       $(x,y) = x+y$

$addPair = \lambda p. ((SEL-PAIR-1\ p) + (SEL-PAIR-2\ p))$

## Ogólnie

$(\lambda(t\ p_1 \dots p_r).E)\ a \rightarrow (\lambda p_1 \dots p_r.E)\ (SEL-t-1\ a) \dots (SEL-t-r\ a)$

Wzbogacamy  $\lambda$  rachunek o:

- 1 wyrażenia `let` i `letrec`
- 2 pattern-matching lambda abstractions
- 3 operator `[]`
- 4 wyrażenia `case`

# Case expressions

case  $v$  of

$c_1 \ v_{1,1} \dots v_{1,r_1} \quad \Rightarrow E_1$

$\dots$

$c_n \ v_{n,1} \dots v_{n,r_n} \quad \Rightarrow E_n$

*reflect* =  $\lambda t.$ case  $t$  of

*LEAF*  $n \quad \Rightarrow$  *LEAF*  $n$

*BRANCH*  $t_1 \ t_2 \quad \Rightarrow$  *BRANCH* (*reflect*  $t_1$ ) (*reflect*  $t_2$ )

$(\lambda(c_1 \ v_{1,1} \dots v_{1,r_1}).E_1)v$

$\square \dots$

$\square(\lambda(c_n \ v_{n,1} \dots v_{n,r_n}).E_n)v$

# Powrót do zwykłego $\lambda$ rachunku

...

# Twierdzenie Churcha-Rossera

## Twierdzenie (Church-Rosser)

*Jeśli  $P \beta\leftarrow M \twoheadrightarrow_\beta Q$  to istnieje  $M'$  taki że  $P \twoheadrightarrow_\beta M' \beta\leftarrow Q$ .*

## Wniosek

Każdy  $\lambda$ -term ma co najwyżej jedną postać normalną.

## Słaba własność Churcha-Rossera (WCR)

Relacja  $\rightarrow$  ma *słabą własność Churcha-Rossera* jeśli  $b \leftarrow a \rightarrow c$  implikuje istnienie  $d$  takiego że  $b \twoheadrightarrow d \leftarrow c$ .

$$\text{WCR} \not\Rightarrow \text{CR}$$

## Własność silnej normalizacji (SN)

Relacja  $\rightarrow$  ma *własność silnej normalizacji* jeśli nie istnieją nieskończone ciągi  $(a_n)_{n \in \mathbb{N}}$  takie że  $a_n \rightarrow a_{n+1}$ , dla każdego  $n \in \mathbb{N}$ .

## Lemat Newmanna

$$\text{WCR} + \text{SN} \Rightarrow \text{CR}$$

## Definicja (1)

- ❶  $x \xrightarrow{1} x$  gdy  $x$  jest zmienną,
- ❷ jeśli  $M \xrightarrow{1} M'$ , to  $\lambda x.M \xrightarrow{1} \lambda x.M'$ ,
- ❸ jeśli  $M \xrightarrow{1} M'$  oraz  $N \xrightarrow{1} N'$  to:
  - ❶  $MN \xrightarrow{1} M'N'$ ,
  - ❷ oraz  $(\lambda x.M)N \xrightarrow{1} M'[x := N']$ .

## Definicja (•)

- ❶  $x^\bullet = x$  gdy  $x$  jest zmienną,
- ❷  $(\lambda x.M)^\bullet = \lambda x.M^\bullet$ ,
- ❸  $(MN)^\bullet = M^\bullet N^\bullet$ , gdy  $MN$  nie jest redeksem,
- ❹  $((\lambda x.M)N)^\bullet = M^\bullet[x := N^\bullet]$ .



## Lemat

- 1 Dla dowolnego  $M$  mamy  $M \xrightarrow{1} M$  oraz  $M \xrightarrow{1} M^\bullet$ .
- 2 Jeśli  $M \xrightarrow{1} M'$  oraz  $N \xrightarrow{1} N'$  to  $M[x := N] \xrightarrow{1} M'[x := N']$ .
- 3 Jeśli  $M \xrightarrow{1} M'$  to  $M' \xrightarrow{1} M^\bullet$ .

## Wnioski

- 1 Relacja  $\xrightarrow{1}$  ma własność rombu.
- 2 Relacja  $\rightarrow_\beta$  ma własność C-R.

# Strategie redukcji termów

Eager evaluation - najpierw argumenty

```
red((\x.A) B) = let
  B' = red B
in red( A [x:=B'] )
```

## Eager

$$\begin{aligned} & (\lambda xy.x)(\lambda v.v)((\lambda z.f(zz))(\lambda z.f(zz))) && \equiv KI(Yf) \\ \rightarrow_{\beta} & (\lambda xy.x)(\lambda v.v)(f((\lambda z.f(zz))(\lambda z.f(zz)))) && \equiv KI(f(Yf)) \\ \rightarrow_{\beta} & (\lambda xy.x)(\lambda v.v)(f(f((\lambda z.f(zz))(\lambda z.f(zz)))))) && \equiv KI(f(f(Yf))) \\ \rightarrow_{\beta} & \dots \end{aligned}$$

$$\begin{aligned} & (\lambda xy.x)(\lambda v.v)((\lambda z.f(zz))(\lambda z.f(zz))) && \equiv KI(Yf) \\ \rightarrow_{\beta} & (\lambda y.\lambda v.v)((\lambda z.f(zz))(\lambda z.f(zz))) && \equiv .. \\ \rightarrow_{\beta} & \lambda v.v && \equiv I \end{aligned}$$

# Strategie redukcji termów

## Normal order reduction

### Definicja

*Ciąg redukcji:*

$$M_0 \rightarrow M_1 \rightarrow \dots \rightarrow M_n$$

*nazywamy standardowym jeśli w kroku  $M_i \rightarrow M_{i+1}$  redukujemy redeks zaczynający się nie dalej od początku termu niż redeks redukowany w kroku  $M_{i+1} \rightarrow M_{i+2}$ .*

### Twierdzenie

*Jeśli  $M \twoheadrightarrow_{\beta} N$  to istnieje standardowa redukcja z  $M$  do  $N$ .*

# Strategie redukcji termów

Normal order reduction - leftmost outermost

## normal order

$\text{red}((\lambda x.A)B) = \text{red}(A[x:=B])$

## eager

$\text{red}((\lambda x.A) B) = \text{let}$   
     $B' = \text{red } B$   
in  $\text{red}(A[x:=B'])$

$$\begin{array}{lll} & \frac{(\lambda xy.x)(\lambda v.v)((\lambda z.f(zz))(\lambda z.f(zz)))}{\rightarrow_{\beta} (\lambda y.\lambda v.v)((\lambda z.f(zz))(\lambda z.f(zz)))} & \equiv KI(Yf) \\ \rightarrow_{\beta} & & \equiv .. \\ \rightarrow_{\beta} & \lambda v.v & \equiv I \end{array}$$

# Strategie redukcji termów

Normal order reduction - leftmost outermost

## Definicja (Czołowa postać normalna)

Każdy term ma jedną z poniższych postaci:

- 1  $\lambda \vec{x}. x \vec{R}$
- 2  $\lambda \vec{x}. (\lambda y. \underline{P}) Q \vec{R}$ .

Termy postaci (1) są w czołowej postaci normalnej (head normal form).  
Dla termów postaci (2), podkreślony redeks nazywamy redeksem czołowym.

Redukcję redeksu czołowego nazywamy *redukcją czołową* i oznaczamy

$$M \xrightarrow{h} N.$$

Pozostałe redukcje nazywamy wewnętrznymi i oznaczamy

$$M \xrightarrow{i} N.$$

# Strategie redukcji termów

Normal order reduction - leftmost outermost

## Leftmost outermost

$$\begin{aligned} \underline{4}(\lambda x. SKKx) &\equiv (\lambda sz. s(s(s(s(z)))))(\lambda x. SKKx) \\ &\rightarrow_{\beta} \lambda z. (\lambda x. SKKx)((\lambda x. SKKx)((\lambda x. SKKx)((\lambda x. SKKx)(z)))) \\ &\twoheadrightarrow_{\beta} \lambda z. (\lambda x. x)((\lambda x. SKKx)((\lambda x. SKKx)((\lambda x. SKKx)(z)))) \\ &\rightarrow_{\beta} \lambda z. (\lambda x. SKKx)((\lambda x. SKKx)((\lambda x. SKKx)(z))) \\ &\twoheadrightarrow_{\beta} \lambda z. (\lambda x. x)((\lambda x. SKKx)((\lambda x. SKKx)(z))) \\ &\rightarrow_{\beta} \lambda z. (\lambda x. SKKx)((\lambda x. SKKx)(z)) \\ &\twoheadrightarrow_{\beta} \lambda z. (\lambda x. x)((\lambda x. SKKx)(z)) \\ &\rightarrow_{\beta} \lambda z. (\lambda x. SKKx)(z) \\ &\twoheadrightarrow_{\beta} \lambda z. (\lambda x. x)(z) \\ &\rightarrow_{\beta} \lambda z. z \end{aligned}$$

# Strategie redukcji termów

## Lazy evaluation

### Lazy

$$\begin{aligned} \underline{4}(\lambda x.SKKx) &\equiv (\lambda sz.s(s(s(s(z)))))(\lambda x.SKKx) \\ &\rightarrow_{\beta} (\lambda z.r1(r1(r1(r1(z))))) && \text{where } r1 = (\lambda x.SKKx) \\ &\twoheadrightarrow_{\beta} (\lambda z.r1(r1(r1(r1(z))))) && \text{where } r1 = \lambda x.x \\ &\rightarrow_{\beta} (\lambda z.r1(r1(r1(z)))) && \text{where } r1 = \lambda x.x \\ &\rightarrow_{\beta} (\lambda z.r1(r1(z))) && \text{where } r1 = \lambda x.x \\ &\rightarrow_{\beta} (\lambda z.r1(z)) && \text{where } r1 = \lambda x.x \\ &\rightarrow_{\beta} (\lambda z.z) \end{aligned}$$

# Typy proste

## Składnia

- typy atomowe  $p, q, r, \dots$  są typami,
- jeśli  $\sigma$  i  $\tau$  są typami to  $\sigma \rightarrow \tau$  jest typem.

## Reguły wnioskowania

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad [\text{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 e_1 : \tau'} \quad [\text{App}]$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x . e : \tau \rightarrow \tau'} \quad [\text{Abs}]$$

## Przykład

$SKK : \text{bool} \rightarrow \text{bool}$

gdzie  $(S = \lambda abc.(ac)(bc), K = \lambda ab.a)$



## Wariant Curry'ego

$$\lambda a\ b\ c.(ac)(bc) : (p \rightarrow q \rightarrow r) \rightarrow (p \rightarrow q) \rightarrow p \rightarrow r$$

## Wariant Churcha

$$\lambda a^{p \rightarrow q \rightarrow r} b^{p \rightarrow q} c^p.((ac)^{q \rightarrow r} (bc)^q)^r : (p \rightarrow q \rightarrow r) \rightarrow (p \rightarrow q) \rightarrow p \rightarrow r$$

## Zgodność z redukcją

Jeśli  $\Gamma \vdash M : \tau$  oraz  $M \rightarrow_{\beta} N$  to  $\Gamma \vdash N : \tau$ .

$$(\lambda id\ a\ b.K(id\ a)(id\ b))(\lambda x.x)$$

## Normalizacja

Jeśli  $\Gamma \vdash M : \tau$  to  $M$  ma postać normalną.

Ranga redeksu  $(\lambda x^{\sigma}.P)Q$  to długość  $\sigma$ . Indukcja ze względu na  $(n, m)$  gdzie  $n$  jest maksymalną rangą redeksu w termie, a  $m$  liczbą takich redeksów.

## Normalizacja

Jeśli  $\Gamma \vdash M : \tau$  to  $M$  jest silnie normalizowalny.

# Izomorfizm Curry-Howard(-de Bruijn-Lambek)

## Reguły wnioskowania dla logiki minimalnej

$$\frac{\tau \in \Gamma}{\Gamma \vdash \tau} \quad [\text{Ax}]$$

$$\frac{\Gamma \vdash \tau \rightarrow \tau' \quad \Gamma \vdash \tau}{\Gamma \vdash \tau'} \quad [\text{E} \rightarrow]$$

$$\frac{\Gamma, \tau \vdash \tau'}{\Gamma \vdash \tau \rightarrow \tau'} \quad [\text{I} \rightarrow]$$

## Reguły wnioskowania dla typów prostych

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \quad [\text{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 \ e_1 : \tau'} \quad [\text{App}]$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x . e : \tau \rightarrow \tau'} \quad [\text{Abs}]$$

# Type inhabitation

## Problem

Dla danych  $\Gamma, \sigma$  czy istnieje term  $M$  taki że  $\Gamma \vdash M : \sigma$ .

(algorytm Ben-Yellesa)

## Twierdzenie (Statman)

*'Type inhabitation' jest PSPACE-zupełny.*

## Problem

Dla danych  $\Gamma, M$  czy istnieje typ  $\sigma$  taki że  $\Gamma \vdash M : \sigma$ .

(unifikacja pierwszego rzędu)

## Twierdzenie

*'Type reconstruction' jest P-zupełny.*

# Hindley-Milner type system

## Typy (formuły typów)

- typy atomowe  $p, q, r, \dots$  są typami,
- zmienne typowe  $\alpha, \beta, \dots$  są typami,
- jeśli  $\tau$  i  $\tau'$  są typami to  $\tau \rightarrow \tau'$  jest typem.

## Schematy typów

- formuła typu jest schematem typu,
- jeśli  $\sigma$  jest schematem typu a  $\alpha$  zmienną to  $\forall \alpha. \sigma$  jest schematem typu.

# Hindley-Milner type system

$$\frac{x : \sigma \in \Gamma}{\Gamma \vdash x : \sigma} \quad [\text{Var}]$$

$$\frac{\Gamma \vdash e_0 : \tau \rightarrow \tau' \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash e_0 \ e_1 : \tau'} \quad [\text{App}]$$

$$\frac{\Gamma, x : \tau \vdash e : \tau'}{\Gamma \vdash \lambda x . e : \tau \rightarrow \tau'} \quad [\text{Abs}]$$

$$\frac{\Gamma \vdash e_0 : \sigma \quad \Gamma, x : \sigma \vdash e_1 : \tau}{\Gamma \vdash \text{let } x = e_0 \text{ in } e_1 : \tau} \quad [\text{Let}]$$

$$\frac{\Gamma \vdash e : \sigma' \quad \sigma' \sqsubseteq \sigma}{\Gamma \vdash e : \sigma} \quad [\text{Inst}]$$

$$\frac{\Gamma \vdash e : \sigma \quad \alpha \notin \text{free}(\Gamma)}{\Gamma \vdash e : \forall \alpha . \sigma} \quad [\text{Gen}]$$

(w powyższych regułach  $\tau, \tau'$  muszą być formułami typu,  $\sigma$  może być schematem)

## Przykład

**let**  $id = (\lambda x.x)$  **in**  $(\lambda a b.K(id\ a)(id\ b))$



## Dodatkowo:

- Typy dla stałych np.

`PAIR: x->y->Pair x y, fst:Pair x y -> x ,...`

- Rekurencyjne definicje w `let`:

`let`

```
length ls = if (empty l)
  then 0
  else (+ 1 (length (tail ls)))
```

`in ...`

- Pattern matching...

# Rachunek kombinatorów S,K

## Składnia

- stałe S,K są kombinatorami,
- jeśli  $\alpha$  oraz  $\beta$  są kombinatorami to  $(\alpha \cdot \beta)$  jest kombinatorem

## Reguły

$$K\alpha\beta \Rightarrow \alpha$$

$$S\alpha\beta\gamma \Rightarrow ((\alpha \cdot \gamma) \cdot (\beta \cdot \gamma))$$

## Własności

- Równoważny ekstensjonalnemu rachunkowi lambda.
- Turing-complete.

# Translacja $\lambda$ -termów do rachunku kombinatorów.

$$\begin{cases} Tr[A \cdot B] &= Tr[A] \cdot Tr[B] \\ Tr[\lambda x. A] &= Pr_x[A] \end{cases}$$

gdzie

$$\begin{cases} Pr_x[A' \cdot B'] &= S(Pr_x[A'])(Pr_x[B']) \\ Pr_x[x] &= I \quad (= SKK) \\ Pr_x[y] &= K \ y \end{cases}$$

# Problem 1.

Niepotrzebne propagacje w poddrzewach, które nie zawierają zmiennej.

## Rozwiązanie 1

Wcześniej zapplikować K.

## Rozwiązanie 2

Dodać kombinatory:

$$Bfgx \Rightarrow f(gx)$$

$$Cfgx \Rightarrow (fx)g$$

## Problem 2.

$$\begin{aligned} & \lambda x_4 x_3 x_2 x_1. p \ q \\ \rightarrow & \lambda x_4 x_3 x_2. S \ p^{(1)} \ q^{(1)} \\ \rightarrow & \lambda x_4 x_3. S \ (B \ S \ p^{(2)}) \ q^{(2)} \\ \rightarrow & \lambda x_4. S \ (B \ S \ (B \ (B \ S) \ p^{(3)})) \ q^{(3)} \\ \rightarrow & S \ (B \ S \ (B \ (B \ S) \ (B \ (B \ (B \ S)) \ p^{(4)}))) \ q^{(4)} \end{aligned}$$

## Problem 2 - c.d.

### Rozwiązanie

$$S' c f g x \Rightarrow c(fx)(gx)$$

$$\begin{aligned} & \lambda x_4 x_3 x_2 x_1. p \ q \\ \rightarrow & \lambda x_4 x_3 x_2. S \ p^{(1)} \ q^{(1)} \\ \rightarrow & \lambda x_4 x_3. S' \ S \ p^{(2)} \ q^{(2)} \\ \rightarrow & \lambda x_4. S' \ (S' \ S) \ p^{(3)} \ q^{(3)} \\ \rightarrow & S' \ (S' \ (S' \ S)) \ p^{(4)} \ q^{(4)} \end{aligned}$$

Analogicznie definiujemy  $C', B'$ .

# Persistent vs Ephemeral data structures

## Persistent

Struktura danych jest *persistent* jeśli update struktury tworzą nową uaktualnioną strukturę nie zmieniając jej poprzednich wersji. W programowaniu czysto funkcyjnym wszystkie struktury danych są *persistent*.

## Ephemeral

wpp  
(typowe dla programowania imperatywnego)

# Analiza amortyzowana - ephemeral structures

R. E. Tarjan, Amortized Computational Complexity, SIAM J. Alg. Disc. Meth. 1985

## Metoda bankiera (księgowania)

- amortyzacja kosztu jest reprezentowana przez kredyty
- każda operacja może pozostawić pewną liczbę kredytów wiążąc ją z pewnym miejscem w strukturze
- operacja może wykorzystać kredyty pozostawione w strukturze
- $a_i$  (amortyzowany koszt) =  $t_i$  (realny koszt czasowy)  
-  $\bar{c}_i$  (liczba zużytych kredytów) +  $c_i$  (liczba pozostawionych kredytów)

Każdy wykorzystany kredyt musi być wcześniej pozostawiony więc:

$$\sum c_i \geq \sum \bar{c}_i \quad \text{stąd} \quad \sum a_i \geq \sum t_i$$



## Metoda fizyka (potencjału)

- funkcja potencjału  $\Phi$  przypisuje strukturze nieujemną liczbę rzeczywistą
- $a_i$  (amortyzowany koszt),  $t_i$  (realny koszt czasowy)

$$a_i = t_i + \Phi(d_i) - \Phi(d_{i-1})$$

Wtedy:

$$\sum_{i=1}^j t_i = \sum_{i=1}^j a_i + \Phi(d_0) - \Phi(d_j)$$

więc suma kosztów amortyzowanych przewyższa sumę realnych kosztów jeśli tylko

$$\Phi(d_0) - \Phi(d_j) < 0$$

## Przykład - prosta kolejka

```
structure BatchedQueue: QUEUE =
```

```
  type a Queue = a list * a list
```

```
  val empty = ([], [])
```

```
  fun isEmpty (f,r) = null f
```

```
  fun checkf ([],r) = (rev r, [])  
    | checkf q = q
```

```
  fun snoc ((f,r), x) = checkf (f,x::r)
```

```
  fun head (x::f, r) = x
```

```
  fun tail (x::f, r) = checkf (f,r)
```

## Koszty

- head:
  - + stały koszt wydobywania elementu
- snoc:
  - + stały koszt dołożenia elementu
  - + jeden kredyt związany z dokładanym elementem
  - + (czasem) stały koszt rev
- tail:
  - + stały koszt wydobywania ogona
  - + (czasem)
    - $+|r|$  koszt rev
    - $-|r|$  kredytów leżących na przenoszonych elementach

## Niezmiennik rozkładu kredytów

Na każdym elemencie listy  $r$  leży jeden kredyt.

# Prosta kolejka - metoda potencjału

## Potencjał

$$\Phi(d) = |r|$$

## Koszty

- head:
  - + stały koszt wydobywania elementu
  - nie zmienia potencjału
- snoc:
  - + stały koszt dołożenia elementu
  - + 1 zmiana potencjału
  - + (czasem) rev stały koszt, zmiana potencjału -1
- tail:
  - + stały koszt wydobywania ogona
  - + (czasem)
    - +  $|r|$  koszt rev
    - -  $|r|$  zmiana potencjału

# Przykład - kopce dwumianowe

signature HEAP =

```
sig
  structure Elem:ORDERED
  type Heap

  val empty    : Heap
  val isEmpty (f,r): Heap -> bool

  val insert      :Elem.T * Heap -> Heap
  val merge       :Heap*Heap -> Heap

  val findMin     :Heap -> Elem.T
  val deleteMin   :Heap -> Heap
```

## Potencjał

$\Phi(d)$  = liczba drzew w kopcu

## Koszty (amortyzowane)

insert =  $O(1)$ ,  
merge, deleteMin =  $O(\log n)$

Powyższe przykłady przestają się amortyzować, jeśli struktury są używane *persistently*.

# Notacja \$

susp

```
datatype a susp = $ of a
```

Konstruowanie

```
(x:int susp) = $(1+2)
```

Odzyskiwanie wartości/Wymuszanie ewaluacji

Pattern matching:

```
val $y = x
```

Pomocnicza funkcja:

```
fun force ($y) = y
```

# Notacja \$ - przykłady

## Przykład 1

```
val s= $primes 1000000 (*fast*)  
val $x = s (*slow*)  
val $y = s (*fast*)
```

## Przykład 2

```
let  
    val s= $primes 1000000  
in 15 end
```



# Notacja \$ - uleniwanie pattern matchingu

```
fun plus ($m,$n)= $m+n
```

```
fun plus (x,y)= $ case (x,y)  
  of ($m,$n) => m+n
```

## Notacja

```
fun lazy plus ($m,$n) = $m+n  
fun plus (x,y)= $ case (x,y)  
  of ($m,$n) =>force ($m+n)
```

# Strumienie (leniwe listy)

## Stream

```
datatype a StreamCell = NIL | CONS of a * a Stream  
withtype a Stream = a StreamCell susp
```

## Stream- przykład

```
$CONS (1, $CONS (2, $CONS (3, $NIL)))
```

# Incremental vs monolithic

monolithic - int list susp

```
force ( $ [1,2,3] ) = [1,2,3]
```

++: a list susp -> a list susp -> a list susp

```
fun  s ++ t = $(force s @ force t)
```

incremental - int Stream

```
force ( $CONS (1, $CONS (2, $CONS (3, $NIL))) ) =  
      CONS (1, $CONS (2, $CONS (3, $NIL)))
```

++: a Stream -> a Stream -> a Stream

```
fun lazy ($Nil) ++ t = t  
  |($CONS (x,s)) ++ t = $CONS (x, s++t)
```

# Incremental vs monolithic

take: (int, a Stream) -> a Stream

```
fun lazy take (0,s) = $NIL
  | take (n,$NIL) = $NIL
  | take (n, $CONS (x,s)) = $CONS (x, take (n-1, s))
```

drop: (int, a Stream) -> a Stream

```
fun lazy drop (0,s) = s
  | drop (n,$NIL) = $NIL
  | drop (n, $CONS (x,s)) = drop (n-1,s)
```

reverse: a Stream -> a Stream

```
fun lazy reverse s = let
  fun reverse' ($NIL, r) = r
    | reverse' ($CONS (x,s),r) = reverse' (s, $CONS (x,r))
  in reverse' (s,$NIL)
```

## Execution trace (ślad wykonania?)

Digraf w którym wierzchołkami są operacje na danej strukturze (zbiorze struktur).

Krawędź  $(v, v')$  oznacza, że operacja  $v'$  używa któregoś z rezultatów operacji  $v$ .

$\hat{v}$  - zbiór wierzchołków, z których  $v$  jest osiągalny ( $v \in \hat{v}$ )

Execution trace dla struktury używanej *ephemeral* nie to ścieżka.

# Leniwa ewaluacja

sposób na dzielenie się kosztem z innymi kopiami struktury

## Koszt operacji

- *unshared cost* - czas wykonania operacji przy założeniu że wszystkie zawieszone dotąd obliczenia są już wykonane w chwili gdy zaczynamy wykonywać bieżącą operację
- *shared cost* - czas potrzebny na wykonanie wszystkich zawieszonych obliczeń, które są tworzone w wyniku operacji, przy założeniach j.w.

koszt całościowy operacji = shared + unshared  
(taki jak koszt w przypadku ewaluacji strict)

## Koszt 'ciągu' operacji

Suma kosztów unshared operacji

+

Suma kosztów shared operacji które zostały zrealizowane

## accumulated debt

- początkowo dług wynosi 0
- za każdym razem, gdy tworzymy zawieszone obliczenie dług jest zwiększany o koszt shared tego obliczenia
- każda operacja może spłacić część długu
- zawieszonego obliczenia nie można wznowić dopóki nie zostanie spłacony związany z nim dług

## dzielenie długu

W ogólności dług zawieszenia może być spłacany wspólnie przez różne 'wątki' które dzielą część struktury.

Zakładamy że cały dług obliczenia musi być spłacony przez operacje z  $\hat{v}$  aby obliczenie to mogło być wykonane w momencie wykonywania operacji  $v$ .

## Metoda bankiera (księgowania)

- dług jest reprezentowany przez debety (debits)
- każdy debet reprezentuje jednostkę ilości zawieszonych obliczeń
- każda operacja może spłacić pewną liczbę debetów
- każda operacja może wygenerować zawieszone obliczenie zwiększając dług struktury (generując debety)
- debety mogą być związane ze strukturą
- operacja może wykonać zawieszone obliczenie, jeśli dług tego obliczenia został spłacony
- $a_i$  (amortyzowany koszt) =  $t_i$  (kosz unshared)  
+  $c_i$  (liczba spłaconych debetów)



## Metoda bankiera (księgowania) - abstrakcyjne ujęcie

Etykietujemy wierzchołki śladu wykonania multizbiorami  $s(v)$ ,  $a(v)$ ,  $r(v)$  t. że:

- $v \neq w \Rightarrow s(v) \cap s(w) = \emptyset$
- $a(v) \subset \bigcup_{w \in \hat{v}} s(w)$
- $r(v) \subset \bigcup_{w \in \hat{v}} a(w)$

$s(v)$  - debety zaciągnięte przez operację  $v$  (zbiór)

$a(v)$  - debety spłacone przez operację  $v$

$r(v)$  - zawieszone obliczenia zrealizowane przez operację  $v$

## Koszty

- całkowity koszt shared:  $\sum_v |s(v)|$
- całkowity liczba spłaconych deбетów:  $\sum_v |a(v)|$
- zrealizowany koszt shared:  $|\bigcup_v r(v)|$

Stąd:

$$|\bigcup_v r(v)| \leq \sum_v |a(v)|$$

# Przykład - leniwa kolejka

```
structure BankersQueue: QUEUE =
```

```
  type a Queue = int * a Stream * int * a Stream
```

```
  val empty = (0,$NIL,0,$NIL)
```

```
  fun isEmpty (lenf,_,_,_) = (lenf=0)
```

```
  fun check (q as (lenf,f,lenr,r)) =
```

```
    if (lenr <= lenf)
```

```
      then q
```

```
      else (lenf+lenr,f++(reverse r),0,$NIL)
```

```
  fun snoc ((lenf,f,lenr,r), x) = check (lenf,f,lenr+1,$CONS (x,r) )
```

```
  fun head (lenf,$CONS (x,f'),lenr,r) = x
```

```
  fun tail (lenf,$CONS (x,f'),lenr,r) = check (lenf-1,f',lenr,r)
```

# Przykład - leniwa kolejka

## Debety

$d(i)$  - liczba debetów na  $i$  – *tym* elemencie zewalutowanego strumienia  $f$   
 $D(i) = \sum_{j=0}^i d(j)$

## Niezmiennik

$$D(i) \leq \min(2i, |f| - |r|)$$

(niezmiennik gwarantuje że można wykonać head ( $d(0) = 0$ ))

## Obserwacja

snoc oraz tail utrzymują niezmiennik spłacając odpowiednio 1 i 2 debety

## Metoda fizyka (potencjału)

- dług jest wspólny dla całej struktury, reprezentowany przez funkcję potencjału  $\Psi$
- dług reprezentuje koszt zawieszonych obliczeń
- każda operacja może spłacić pewną część długu
- każda operacja może wygenerować zawieszone obliczenie zwiększając dług struktury (generując debety)
- operacja może wykonać zawieszone obliczenie, jeśli dług całej struktury wynosi 0
- $a_i$  (amortyzowany koszt) =  $t_i$  (kosz unshared) + spłacona część długu

# Metoda potencjału

Przykład: leniwe kopce dwumianowe

```
structure LazyHeap: HEAP =  
  type Heap = Tree list susp  
  
  fun lazy insert (x, $ts) = $insTree (NODE (0,x,[]), ts)  
  ...
```

## Potencjał

- $\Psi(d)$  - liczba zer w binarnej reprezentacji  $|d|$
- `insert` - zamienia sufiks jedynek na sufiks zer i jedno zero na jedyne, sumaryczna zmiana rzędu ilości wywołań `link`.
- długi struktury rzędu  $\log(n)$

# Metoda potencjału: kolejka

```
structure PhysicistQueue: QUEUE =
```

```
  type a Queue = a list * int * a list susp * int * a list
```

```
  val empty = ([],0,$[],0,[])
```

```
  fun isEmpty (_,lenf,_,_,_) = (lenf = 0)
```

```
  fun checkw ([],lenf,f,lenr,r) = (force f,lenf,f,lenr,r)
    | checkw q = q
```

```
  fun check (q as (w,lenf,f,lenr,r)) =
    if (lenr<=lenf) then checkw q
    else let val f' = force f
         in checkw (f',lenf+lenr,$(f' @ rev r),0,[]) end
```

```
  fun snoc ((w,lenf,f,lenr,r), x) = check (w,lenf,f,lenr+1,x::r)
```

```
  fun head (x::w,lenf,f,lenr,r) = x
```

```
  fun tail (x::w,lenf,f,lenr,r) =
    check (w,lenf-1,$tl (force f),lenr,r)
```

$$\Psi(d) = \min(2|w|, |f| - |r|)$$

## Highlights

- Co by było gdybyśmy ewaluowali  $f$  dopiero jak  $w$  jest pusta?
- Co by było gdyby nie tworzyć  $\$t1$  (`force f`), tylko zapisać gdzieś że listę trzeba zmniejszyć?

# Metoda potencjału: Sortable collection

```
functor BootomUpMergeSort(Element:ORDERED): SORTABLE =
```

```
  structure Elem = Element
  type Sortable= int * Elem.T list list susp
  fun mrg ([],ys)= ys
    |mrg (xs as x::xs', yx as y::ys')=
      if Elem.T.leq(x,y) then x::mrg(xs',ys)
      else y::mr(xs,ys')

  val empty= (0,$[])
  fun add = (x,(size, segs)) =
    let fun addSeg (seg, segs, size) =
        if size mod 2 = 0 then seg::segs
        else addSeg(mrg(seg, hd segs),tl segs, size div 2)
    in (size+1, $addSeg([x],force segs, size)) end
  fun sort (size, segs) = foldl (mrg,[],force segs)
```

$$\Psi(n) = 2n - \sum_{i=0}^{\infty} b_i(n \bmod 2^i + 1)$$



# Eliminacja amortyzacji

- amortized data structure  $\Rightarrow$  worst-case data structure
- redukcja kosztu jednostkowych zawieszonych obliczeń (eliminacja funkcji monolitycznych)
- forsowanie explicite zawieszonych obliczeń (scheduling)

# Eliminacja amortyzacji: Real-time queue

$\text{rotate } (xs, ys, a) \equiv xs++\text{reverse } ys++a$

```
fun rotate ($NIL,$CONS (y,_),a) = $CONS (y,a)
  | rotate ($CONS (x,xs), $CONS (y,ys), a) =
    $CONS (x, rotate (xs,ys,$CONS (y,a)))
```

# Eliminacja amortyzacji: Real-time queue

```
structure RealTimeQueue: QUEUE =
```

```
  type a Queue = a Stream * a list * a Stream
```

```
  val empty = ($NIL, [], $NIL)
```

```
  fun isEmpty ($NIL, _, _) = true
```

```
    | isEmpty _ = false
```

```
  fun exec (f, r, $CONS (x, s)) = (f, r, s)
```

```
    | exec (f, r, $NIL) = let
```

```
      val f' = rotate(f, r, $NIL) i
```

```
    in (f', [], f') end
```

```
  fun snoc ((f, r, s), x) = exec (f, x::r, s)
```

```
  fun head ($CONS (x, f), ri, s) = x
```

```
  fun tail ($CONS (x, f), r, s) = exec (f, r, s)
```

$$|s| = |f| - |r|$$

# Eliminacja amortyzacji: Kopce dwumianowe

```
fun lazy insTree (t,$NIL)= $CONS (t,$NIL)
  |insTree (t, ts as $CONS (t', ts')) =
    if rank t < rank t' then $CONS(t,ts)
    else insTree(link (t,t'),ts')
```

(ciągłe monolityczna)

# Eliminacja amortyzacji: Kopce dwumianowe

```
datatype Tree= NODE of Elem.T * Tree list
datatype Digit= ZERO | ONE of Tree
type Heap= Digit Stream
```

```
fun lazy insTree (t,$NIL)= $CONS (ONE t,$NIL)
    |insTree (t,$CONS (ZERO, ds)) = $CONS (ONE t, ds)
    |insTree (t,$CONS (ONE t',ds))=
        $CONS (ZERO, insTree(link(t,t'),ds))
```

# Eliminacja amortyzacji: Kopce dwumianowe

```
datatype Tree= NODE of Elem.T * Tree list
datatype Digit= ZERO | ONE of Tree
type Schedule = Digit Stream list
type Heap= Digit Stream * Schedule
```

```
fun exec [] = []
  | exec (($CONS (ONE t,_))::sched) = sched
  | exec (($CONS (ZERO, job))::sched) = job::sched

fun insert (x,(ds,sched))= let
  val ds'=insTree(NODE (x,[]),ds)
in (ds', exec(exec(ds':::sched))) end
```

## Obserwacja

Pomiędzy każdymi dwoma zadaniami istnieje zewaluowane ZERO. Przed pierwszym zadaniem istnieją dwa zewaluowane ZERA.

# Eliminacja amortyzacji: Bottom-Up Mergesort

## incremental merge

```
fun lazy mrg ($NIL,ys) = ys
  |mrg (xs,$NIL) = xs
  |mrg (xs as $CONS (x,xs'), ys as $CONS (y,ys'))=
    if (x<=y) then $CONS (x, mrg (xs',ys))
    else $CONS (y, mrg (xs,ys'))
```

```
type Schedule = Elem.T Stream list
type Sortable = int * (Elem.T Stream * Schedule) list
```

# Eliminacja amortyzacji: Bottom-Up Mergesort

```
type Schedule = Elem.T Stream list
type Sortable = int * (Elem.T Stream * Schedule) list
```

```
fun exec1 [] = []
  | exec1 (($NIL)::sched) = exec1 sched
  | exec1 (($CONS (x,xs))::sched) = xs::sched

fun exec2 (xs,sched) = (xs, exec1(exec1 sched))
```

```
fun addSeg (xs,segs,size,rsched)=
  if size mod 2 = 0 then (xs, rev rsched)::segs
  else let val ((xa',[])::segs')=segs
        val xs''=mrg(xs,xs')
        in addSeg(xs'',segs',size div 2,xs''::rsched)

fun add (x,(size,segs))=let
  val segs'=addSeg ($CONS (x,$NIL),segs,size,[])
  in (size+1,map exec2 segs') end
```



## Obserwacja

Scheduler segmentu rozmiaru  $m = 2^k$  zawiera co najwyżej

$$2m - 2(n \bmod m + 1)$$

elementów (jednostkowych zawiesznień).

# Dequeues

```
signature DEQUE =
```

```
sig
```

```
type 'a Queue
```

```
val empty    : 'a Queue
```

```
val isEmpty  : 'a Queue -> bool
```

```
val cons     : 'a * 'a Queue -> 'a Queue
```

```
val head     : 'a Queue -> 'a
```

```
val tail     : 'a Queue -> 'a Queue
```

```
val snoc     : 'a * 'a Queue -> 'a Queue
```

```
val last     : 'a Queue -> 'a
```

```
val init     : 'a Queue -> 'a Queue
```

```
type 'a Queue = int * 'a Stream * int * 'a Stream
```

Perfect balance:  $|f| = |r| (+1)$ .

Balance invariant:  $|f| \leq c|r| + 1 \quad |r| \leq c|f| + 1$ .

```
fun check (lenf,f,lenr,r)=
  if (lenf> c*lenr+1) then let
    val i = (lenf+lenr) div 2
    val j = lenf+lenr-i
    val f' = take (i,f)
    val r' = r++ reverse (drop (i,f))
    in (i,f',j,r') end
  else if (lenr> c*lenf+1) then
    ...
  else q

fun cons (x,(lenf,f,lenr,r)) = check (lenf+1,$CONS (x,f),lenr,r )
fun head (lenf,$CONS (x,f'),lenr,r) = x
fun tail (lenf,$CONS (x,f'),lenr,r) = check (lenf-1,f,lenr,r)
...
```

Amortyzowane koszty  $\leq c$ .

# Real-Time Deques ( $c \in \{2, 3\}$ )

```
fun rotateDrop (f,j,r) = if j<c then rotateRev(f,drop(j,r),$NIL)
  else let val ($CONS(x,f')) = f
      in $(CONS (x, rotateDrop(f',j-c,drop(c,r)))) end

fun rotateRev ($NIL,r,a) = reverse r ++ a
  | rotateRev ($CONS (c,f),r,a) =
    $CONS (x, rotateRev (f, drop (c,r), reverse (take(c,r)) ++ a))
```

# Real-Time Deques ( $c \in \{2, 3\}$ )

```
type 'a Queue = int* 'a Stream * 'a Stream *int* 'a Stream *
```

```
fun check (lenf,f,lenr,r)=
  if (lenf> c*lenr+1) then let
    val i = (lenf+lenr) div 2      val j = lenf+lenr-i
    val f' = take (i,f)           val r' = rotateDrop (r,i,f)
    in (i,f',f',j,r',r') end
  else if (lenr> c*lenf+1) then
    ...
  else q

fun exec1 ($CONS (x,s))= s
  exec1 s = s
fun exec2 x= exec1 (exec1 s)

fun cons (x,(lenf,f,sf,lenr,r,sr)) =
  check (lenf+1,$CONS (x,f),exec1 sf,lenr,r, exec1 sr )
fun tail (lenf,$CONS (x,f'),sf,lenr,r,sr) =
  check (lenf-1,f',exec2 sf,lenr,r, exec2 sr )
...
```

# Numerical representations

## Unarna reprezentacja

```
datatype 'a List =  
  NIL  
  |CONS of 'a * 'a List  
  
fun tail (CONS (x,xs))= xs  
fun append (NIL,ys) = ys  
  |append (CONS (x,xs),ys)=  
    CONS (x,append (xs,ys))
```

```
datatype 'a List =  
  ZERO  
  |SUCC of Nat  
  
fun pred (SUCC n) = n  
fun plus (ZERO,n) = n  
  |plus (SUCC m,n) =  
    SUCC (plus (m,n))
```

# Random Access List

```
signature RANDOMACCESSLIST =
```

```
sig
```

```
type 'a RList
```

```
val empty    : 'a RList
```

```
val isEmpty  : 'a RList -> bool
```

```
val cons      : 'a * 'a RList -> 'a RList
```

```
val head      : 'a RList -> 'a
```

```
val tail      : 'a RList -> 'a RList
```

```
val lookup    : int * 'a RList -> 'a
```

```
val update    : int * 'a * 'a RList -> 'a RList
```

## Unarnie - zwykłe listy

- cons, head, tail -  $O(1)$
- lookup, update -  $O(n)$

## Binarnie (np. kopce dwumianowe)

- cons, head, tail -  $O(\log n)$
- lookup, update -  $O(\log n)$

# Random Access List

```
structure BRAL: RANDOMACCESSLIST =  
struct  
  datatype 'a Tree = LEAF of 'a | NODE of int * 'a Tree * 'a Tree  
  datatype 'a Digit = ZERO | ONE of 'a Tree  
  type 'a RList = 'a Digit list  
  ...  
  fun link ((s1,t1),(s2,t2)) = NODE (s1+s2,t1,t2)  
  fun consTree (t,[]) = [ONE t]  
    | consTree (t,ZERO::ts) = ONE t :: ts  
    | consTree (t1,ONE t2 ::ts) = ZERO :: consTree (link (t1,t2),ts)  
  
  fun cons (x,ts)= consTree (LEAF x, ts)
```



# Zeroless representation

```
structure ZLBAL: RANDOMACCESSLIST =
```

```
struct
```

```
datatype 'a Tree = LEAF of 'a | NODE of int * 'a Tree * 'a Tree
```

```
datatype 'a Digit = ONE of 'a Tree | TWO of 'a Tree * 'a Tree
```

```
type 'a RList = 'a Digit list
```

```
...
```

```
fun link ((s1,t1),(s2,t2)) = NODE (s1+s2,t1,t2)
```

```
fun constTree (t,[]) = [ONE t]
```

```
  | constTree (t1,ONE t2 ::ts) = TWO (t1,t2) :: ts
```

```
  | constTree (t1,TWO (t2,t3) ::ts) = ONE t1 :: constTree (link (t2,t3),t3 :: ts)
```

```
fun cons (x,ts) = constTree (LEAF x, ts)
```

## zeroless dec

```
fun dec [ONE] = []
```

```
  | dec (TWO::ds) = ONE::ds
```

```
  | dec (ONE::ds) = TWO::(dec ds)
```

## koszty operacji

- head -  $O(1)$
- cons,tail -  $O(\log n)$
- lookup,update -  $O(\log i)$

# Amortyzacja

```
datatype Digit = ZERO | ONE  
type Number = Digit Stream
```

## inc

- worst-case  $O(\log N)$
- amortyzowany  $O(1)$   
(inc, print)
- długość struktury – liczba jedynek

## dec

- worst-case  $O(\log N)$
- amortyzowany  $O(1)$   
(dec, print)
- długość struktury – liczba zer

Amortyzowany koszt inc dla operacji inc,dec,print to

$O(\log N)$

# Amortyzacja

```
datatype Digit = ZERO | ONE  
type Number = Digit Stream
```

## inc - amortyzacja

Połowa operacji które dotyczą  $k$ -tej cyfry dotyczy cyfry  $k + 1$ -szej. Stąd średnia liczba cyfr zmienianych przez operację to:

$$1 + \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \frac{1}{16} + \dots < 2$$

## bezpieczne cyfry

Cyfra jest bezpieczna dla operacji jeśli operacja zakończy działanie po natrafieniu na tą cyfrę.

- 0 bezpieczne dla inc
- 1 bezpieczne dla dec

# Niejednoznaczne reprezentacje

```
datatype Digit = ZERO | ONE | TWO
type Number = Digit Stream

fun lazy inc $NIL = $CONS (ONE, $NIL)
    |inc ($CONS (ZERO,ds)) = $CONS (ONE, ds)
    |inc ($CONS (ONE,ds)) = $CONS (TWO, ds)
    |inc ($CONS (TWO,ds)) = $CONS (ONE, inc ds)
fun lazy dec $CONS (ONE, $NIL) = $NIL
    |dec ($CONS (ONE,ds)) = $CONS (ZERO, ds)
    |inc ($CONS (TWO,ds)) = $CONS (ONE, ds)
    |inc ($CONS (ZERO,ds)) = $CONS (ONE, dec ds)
```

$(22222)+1 = (111111)$        $(111111)-1 = (011111)$

## inc,dec,print - amortyzacja

Jedynka jest bezpieczna dla obu operacji.

Debet struktury – liczba cyfr 1.

# Zeroless lazy representation

```
structure ZLLBRAL: RANDOMACCESSLIST =  
  ...  
  datatype 'a Digit = ONE    of 'a Tree  
                    | TWO    of 'a Tree * 'a Tree  
                    | THREE  of 'a Tree * 'a Tree * 'a Tree  
  type 'a RList = 'a Digit list  
  ...
```

## koszty operacji

- head -  $O(1)$
- cons, tail - amortyzowany  $O(1)$
- lookup, update -  $O(\log i)$

# Rzadka reprezentacja (sparse)

```
datatype DigitBlock = ZEROS of int | ONES of int  
type Number = DigitBlock list
```

```
fun zeros (i,[]) = []  
  | zeros (0,blks) = blks  
  | zeros (i, ZEROS j::blks) = ZEROS (i+j)::blks  
  | zeros (i, blks) = ZEROS i::blks
```

```
fun inc NIL = [ONES 1]  
  | inc (ZEROS n ::blks) = ones(1,zeros (i-1,blks))  
  | inc (ONES n ::blks) = ZEROS n::inc blks
```

$\text{inc, dec} - O(1)$

# Rzadka reprezentacja (sparse)

```
datatype Digits = ZERO | ONES of int | TWO
type Number = Digits list
...
```

## Niezmiennik

Ostatnia nie ONE przed TWO to ZERO.

```
fun simpleInc [] = [ONES 1]
  | simpleInc (ZERO::ds) = ones (1,ds)
  | simpleInc (ONES i::ds) = TWO:: one (i-1,ds)

fun fixup (TWO::ds) = ZERO::simpleInc ds (* następna cyfra to nie TWO *)
  | fixup (ONES i::TWO::ds) = ONES i:: ZERO :: simpleInc ds
  | fixup ds = ds
```

```
datatype Digit = ZERO | ONES of Tree list | TWO of Tree * Tree
type Heap = Digit list
...
```

## Skośna reprezentacja binarna

Waga  $i$ -tej cyfry to  $2^{i+1} - 1$ .

Cyfry:  $\{0,1,2\}$

## Reprezentacja kanoniczna

Dwójką może być co najwyżej najmniejsza niezerowa cyfra.

(Każda liczba naturalna ma unikalną skośną reprezentację kanoniczną.)

## Inkrementacja

$$1 + 2 \cdot (2^{i+1} - 1) = 2^{i+2} - 1$$

Jeśli najmniejsza niezerowa cyfra to 2 to zamień na 0 i zwiększ następną.

Wpp. zwiększ najmniejszą cyfrę.



# Skośne liczby - rzadka reprezentacja

```
type Nat = int list (* wagi cyfr *)

fun inc (ws as w1::w2::rest) = if (w1=w2) then (1+w1+w2)::rest
    else 1::ws
  | inc ws = 1::ws

fun dec (1::ws) = ws
  | dec (w::ws) = (w div 2)::(w div 2)::ws
```

## Skośne listy

...

# Skośne kopce dwumianowe

```
datatype Tree = NODE of int * Elem.T * ElemT.list * Tree list
type Heap = Tree list
```

- `int` oznacza rangę  $r$  drzewa (rozmiar  $2^r$ )
- lista elementów jest długości co najwyżej  $r$
- drzewa na liście mają rosnące rangi
- poza pierwszymi dwoma elementami rangi są silnie rosnące

```
fun link t1 t2) = ...
fun skewLink (x,t1,t2) =
  let val NODE (r,y,ys,c) = link (t1,t2)
  in
    if (x<y) then NODE (r,x,y::ys,c)
    else NODE (r,y,x::ys,c)
  end
```

# Skośne kopce dwumianowe

## insert

```
fun insert (x, ts as t1::t2::rest) =  
    if (rank t1 = rank t2) then skewLink (x,t1,t2)  
    else (NODE (0,x,[],[]))::ts  
|insert (x,ts) = NODE (0,x,[],[])::ts  
  
fun normalize (t::ts) = insTree (t,ts)  
fun merge (t1,t2) = mergeTrees (normalize t1, normalize t2)  
  
fun deleteMin ts = let  
    val (NODE (_,x,xs,ts1),ts2) = removeMinTree ts  
    fun insertAll ([],ts) = ts  
      |insertAll (x::xs) = insertAll (xs, insert(x,ts))  
    in insertAll (xs, merge (rev ts1, ts2)) end
```