Функционално програмиране
теория

бележке

I

Функционално програмирање је парадигма чије су карактеристике: имутабилност, функције вишег реда, чиста и прелиминарност

функција вишег реда је функција чији је један од аргумената ф-ја или је повратна вредност ф-ја

имутабилност је особина објекта да не може да мења своје стање након што је креиран

чиста функција (PURE) је ф-ја која:
а) за исти улаз увек даје исти излаз;
б) нема споредних ефеката

ленивост: израчунавање је одложено док не постане неопходно

строга типизираност: променљиве имају свој тип којим је ограничене. Пример декларације променљиве, додељивање јој тип.

*У Хаскелу додељујемо имена изразима
Када додељујемо име неком изразу A, не
можемо опет додељивати исто име изразу B
(у истом опсегу)

II

FOLD:

Репна рекурзија је рекурзија где се рекурзивни повик извршава на крају. Репну рекурзију можемо претворити у петљу, што је лакше оптимизовати.

Пример: Сума елемената листе (преко репне рек.)

```
• SUM xs =
    Let HELPER ACC VALUES =
                CASE VALUES OF
                  [] -> ACC
                  (Y:YS) -> HELPER (ACC+Y) YS
    IN HELPER 0 xs
```

• SUM [1,2,3] = HELPER 0 [1,2,3] = HELPER (0+1) [2,3]

= HELPER (1+2) [3] = HELPER (3+3) []

= (6+0) = 6

***ламбда функција:** анонимна ф-ја

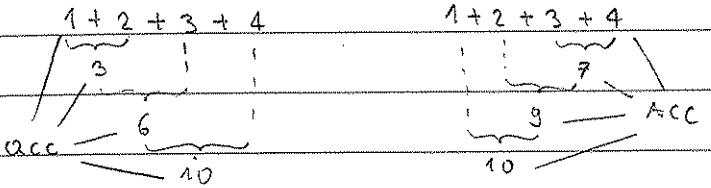
(\X Y -> X+Y) (синтакса)

арг. рез.

Пример: производ елемената листе преко репне р

```
• MUL xs =
    Let HELPER ACC VALUES =
                CASE VALUES OF
                  [] -> ACC
                  (Y:YS) -> HELPER (ACC*Y) YS
    IN HELPER 1 xs
```

*Пример 1 и 2 се разликују само на два места ūj. мењамо иницијалну вредност и оператор. Ово се може извршити!



'лево асоцијативна 'десно асоцијативна

• Шта се још може извршити?
-колекција, не мора да буде листа (FOLDABLE)
• да ли акумулатор (вредност која се акумулира од иницијалне вр. ~ тренутни резултат) мора да буде исти тип као елемент колекције? Пример: дата је листа карактера и треба одредити фреквенцију слова 'a'. Акумулатор је број, а елемент је карактер.

FOLD: "сабира" колекцију у резултат.

• FOLDR :: Foldable t => (a -> b -> b) -> b -> t a -> b
• FOLDL :: Foldable t => (b -> a -> b) -> b -> t a -> b
(5) (1) (2) (3) (4)

FOLDL и FOLDR се разликују по смеру "сабирања" ūj. по асоцијативности.

FOLDL = LEFT FOLD, FOLDR = RIGHT FOLD

елементи FOLDL-а:

1) оператор који узима старо акумулатор типа b и елемент типа a као аргументе, и враћа резултат типа b:

Пример: (\ACC x -> ACC + x)

или (+) који left постоји у Хаскелу

*За FOLDR је обрнут редослед и то је једина разлика између FOLDL и FOLDR у синтакси:

(\x ACC -> x + ACC)

2) иницијална вредност акумулатора

3) колекција која наслеђује FOLDABLE тј. може се применити FOLD на њу. (5)

4) резултат

FOLDL1, FOLDR1: узимају први елемент листе као иницијалну вредност акумулатора.

***оператор $:** f $ x = f x

Узимање оператора у Хаскелу је лево асоцијативно

f g x <=> (f g) x

f $ g x <=> f (g x)

***оператор .:** композиција ф-ја

f (g x) <=> f . g x

Пример 4:

а) FOLDR (+) 0 xs

б) FOLDR (*) 1 xs

в) FOLDR (++) [] xs , конкатенација низа листи

Пример 5: Која врста FOLD-а добија од низа низ па низ?

FOLDR (:) [] xs

-мора FOLDR, јер је (:) десно асоцијативан

*оператор : додаје елемент на почетак листе

(+) 2 3 <=> 2+3

обрнуто: FOLDR (:) [] [1,2,3]

=> 1 : 2 : 3 : []

[3]

[2,3]

[1,2,3]

Пример 6: филтрирати парне бројеве из листе:

onlyEven :: Int -> [Int] -> [Int] (декорација)
ARGS RETURN

onlyEven x acc = (дефиниција)

case (mod x 2) of

0 -> x : acc

1 -> acc

filterEven :: [Int] -> [Int]

filterEven xs = FOLDR onlyEven [] xs

*сложеност: O(n)

-Ако користимо FOLDL, онда можемо користити (++) ф-ју. конкатенирамо за додавање на крај Додавање на крај у Хаскелу је $O(n)$, да би комплексност целе ф-је била $O(n^2)$

?:t у ghci приказује информације о ф-ји: аргументи и њихови типови, повратна вр. и њен тип.

-На основу претходног примера видимо да на основу произвољног предиката можемо имплементирати филтрирање листе (колекције)

• да ли преко FOLD-a можемо да проверимо да ли у листи постоји елемент који задовољава неки услов? - да, али не можемо стићи код тачно тог елемента, јер FOLD пролази кроз целу листу.

• ACCUMULATE у C++ ⇝ FOLD у Хаскелу

custom list:

Рекурзивна дефиниција листе:

- листа је или празна или се састоји из главе (елемент) и репа (остали елементи)

Пример1: Листа целих бројева.

* data: кључна реч за дефинисање корисничких типова. Типови се пишу великим словом.

```
data IList = Empty                (празна листа)
           | Cons Int IList      (глава и реп)
           deriving Show          (*)
```

*=> Наслеђује Show за да бисмо могли представити објекат на излазу као ниску.

Конструктор типа: (IList) узима нула или више аргумената (нула у овом случају).

Конструктор вредности: (Empty, Cons) узима нула или више вредности (Empty - нула, Cons - два тј. главу и реп који представља листу.

Пример2: head за IList

```
head' :: IList -> Int
head' Empty = 666           - је било шта
head' (Cons x _) = x
```

(Cons x _) - мора у заграде

Пример 3: Универзална листа:

```
data UList a = Empty
             | Cons a (UList a)
             deriving Show
```

- конструктор типа сада има тип као аргумент
- остало је исто.

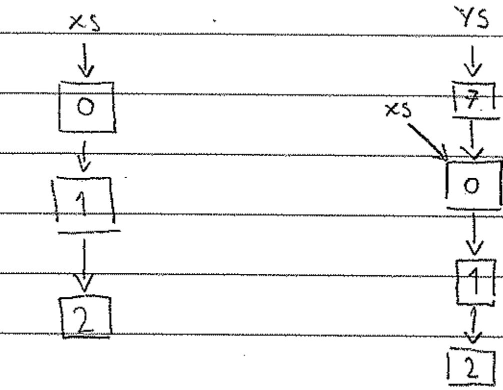
Пример 4: дубина (димензија) листе:

```
depth :: UList a -> Int
depth Empty = 0                        (аналогно: [])
depth (Cons a b) = 1 + depth b         (аналогно: (x:xs))
```

имутабилност листе:

Пример 5: додавање елемента на почетак листе:

xs = [0, 1, 2]   ys = 7:xs



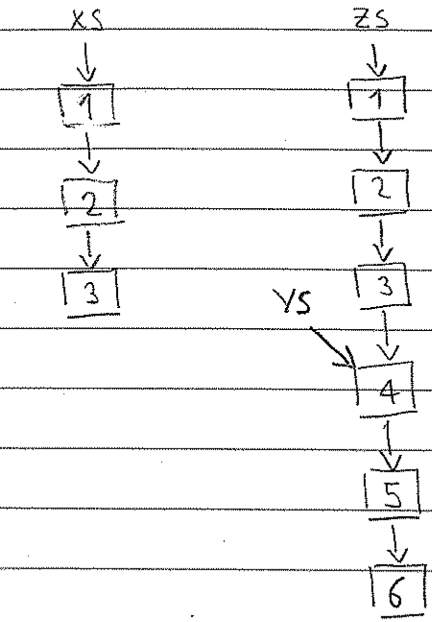
- Не врши се копирање!
- временска сложеност: O(1)
- меморијска сложеност: O(1)

*НАПОМЕНА: xs++[y] је временске и мем. с. O(n) због имутабилности.

Пример 6: конкатенација

xs = [1,2,3], ys = [4,5,6], len(xs) = n, len(ys) = m

zs = xs ++ ys, резултат



- копира се само xs листа а рет (ys) остаје заједнички део за ys и xs
- временска сложеност: O(n)
- меморијска сложеност: O(n

* операције код листе (временска сложеност):

а) додавање на почетак (:): O(1)

б) додавање на крај xs++[y]: O(n)

в) конкатенација (++): O(n)

г) индексирање: O(n)

* xs !! i , [1,2,3] !! 2 = 2

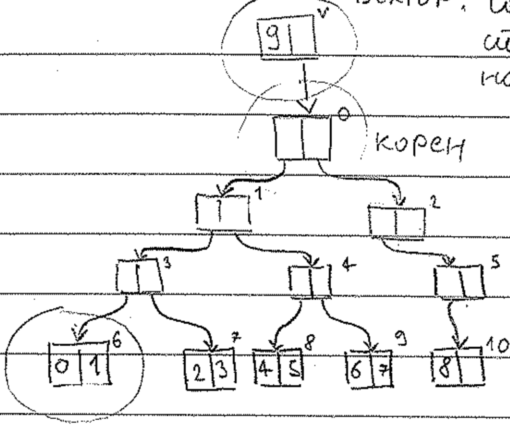
(!! — оператор индексирања)

д) брисање са почетка: O(1)

* додатна мана листе: неискоришћен потенцијал кеша због распореда у меморији (није блоковски)

PERSISTENT VECTOR:

Вектор: садржи димензију структуре и показивач на корен структуре



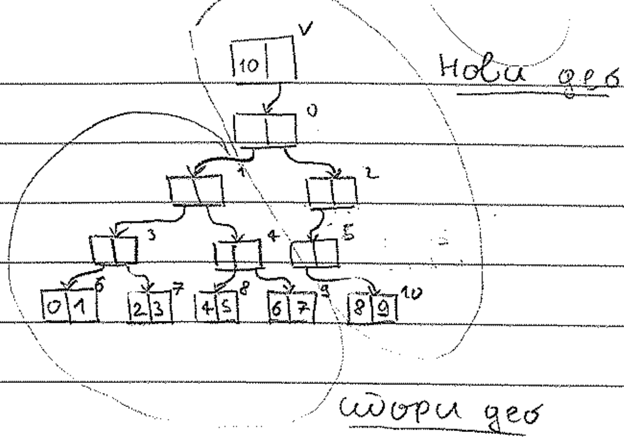
лист: садржи 1 или 2 елемента

=> вектор: [0,1,2,3,4,5,6,7,8}

а) додавање елемента у лист:

- желимо исту структуру, али са промењеним последњим листом на [8|9]. Проблем: не можемо да променимо лист (имутабилност)!

Решење је да креирамо нов вектор, али да сачувамо што више старих чворова. Довољно је заменити све претке од корена вектора до тог листа (пут) и тај лист. Са слике мењамо чворове v, 0, 2, 5 и 10



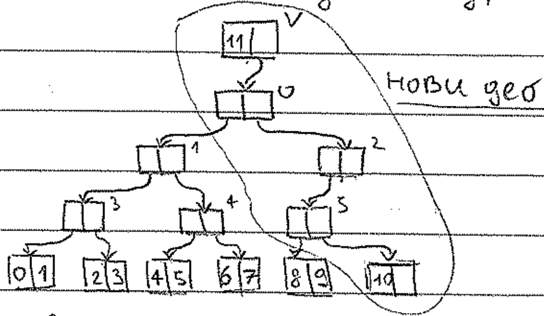
б) измена елемента је аналогно додавању: мењамо чворове од почетка до одговарајућег листа.

в) додавање елемента иде кроз случајеве

1) има места у последњем листу:

- своди се на (а);

2) Нема места у листу, али има у вектору:



3) вектор је попуњен:

- повећава се висина вектора. Тиме повећавамо његову димензију.

г) брисање са краја. Случајеви
1) последњи лист има више од једног елемента;
2) последњи лист има тачно један елемент
3) корен има само један елемент (леви) након брисања

- ове операције су инверзне са (в), па се врше аналогно томе

сложеност: (сложеност операција? $O(\log_2 n)$

Не морамо имати чворове димензије 2, већ могу бити произвољне димензије k. За k=32 имамо сложеност $O(\log_{32} n)$. Шта ако ограничимо висину стабла на нпр. 7? Тада је сложеност у теорији $O(c)$ тј. $O(1)$. Ова структура је спорија од вектора за константан фактор.

Сакупљачи отпадака

мане баратања меморијом:

1) низак ниво
2) ручно баратање меморијом води на грешке
3) алокација меморије на хипу.

• Ствари које нам више нису потребне морају да се обришу из меморије. У Хаскелу је ослобађање меморије аутоматизовано

garbage collector

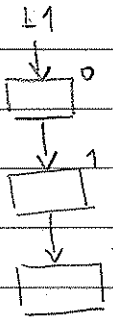
а) бројање референци:

• Бројимо колико ствари показује на неки објекат помоћу бројача (бројач референци)

• Ако на објекат ништа не показује, да ли можемо да му приступимо? Не, и зато желимо да га обришемо. Ако је бројач једнак нули, онда бришемо објекат

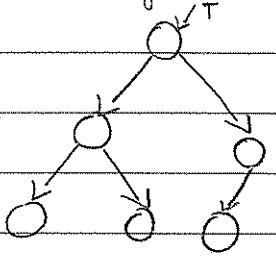
• Свако копирање објекта је само повећавање референци на тај објекат.

Пример 1: листа:



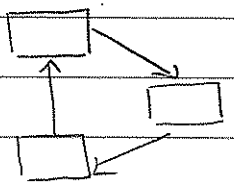
- ако текущо L1 референца, онда можемо да обришем чвор 0. Ако даден чвор листо има само референца од предходника онда се цела листа обрише

Пример 2: дрво



- слично као за листу

Пример 3: Кружна листа



- Нема ограничења на структуру, али број референци ових објеката није 0, да се не бришу

Пример 4: двоструко повезана листа

- исти проблем

предности:

П1) детерминизам;

П2) ефикасност;

П3) меморијска локалност;

мане:

М1) цикличне структуре;

М2) мутабилне структуре;

М3) све операције су мутабилне;

М4) фрагментација; М5) Немамо контролу;

Објашњење:

П1) У тренутку када нико не користи објекат, он бива уништен;

П2) ефикасне операције (али и даље се ради нешто што је скупље него ништа);

П3) бројачи се памте заједно са објектом

М1) Погледати примере 3 и 4;

М2/М3) бројачи се константно ажурирају што нарушава концепт имутабилности.

• Рецимо да имамо два имутабилна објекта A и B тако да A има референцу на B. У ком редоследу их креирамо?

* Први случај: Прво креирамо А. Овде је проблем што не можемо поставити референцу на В, јер не постоји тренутно.
* Други случај: Прво креирамо В. Овде је проблем што не можемо поставити референцу са А, јер није креиран још.

• У савршено чистом језику нема циклуса

• Не постоји савршено чист функционални језик, јер мора постојати циклус на неки начин.

М5) Објекат се уништава у тренутку када ништа не показује на њега. То може да доведе до ланчаног уништавања објеката, што може значајно да успори неки критичан део програма.

Критика: Пошто се објекат одмах обрише чим ништа не показује на њега, онда не можемо рећи да постоји ђубре у неком тренутку, па бројање референци не можемо назвати garbage collector-ом.

б) оптимизован бројач референци

1) одложено ажурирање бројача;
2) стапање узастопних промена бројача;
3) баферовано ажурирање података;

⇒ Ако у неком бројач референци за објекат постане 0, не уништавамо га и тај објекат постаје ђубре. Тиме бројач референци постаје garbage collector

в) MARK-AND-SWEEP:

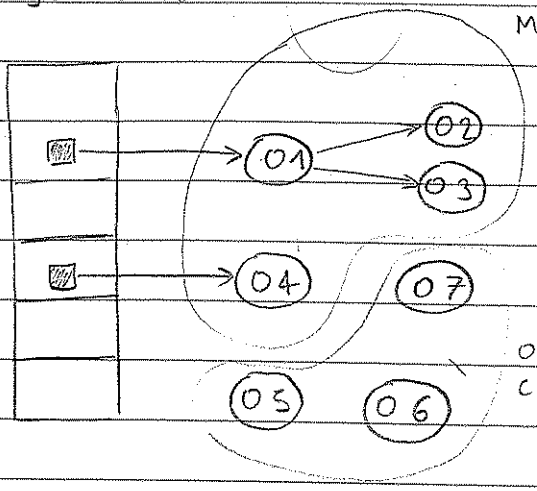
• Алгоритам који одређује и брише ђубре кроз два корака:

1) Покупи се показивачи са стека који формирају неки граф и обиђе тај граф (нпр. DFS) где у сваком чвору обележавамо објекте на хипу (иницијално су необележени). Сви елементи који остану необележени су по дефиницији ђубре. Овај корак се назива „MARK"

2) Пролазимо кроз објекте на хипу и бришемо их ако нису маркирани. Овај корак се назива „SWEEP"

Сви објекти постају необележени након итерације.

Објашњење:



- Не можемо знати да ли је нешто бубре на стеку, али можемо знати да ли је бубре на хипу. Стек сам чисти своје бубре
- Неправи објекти постоје хипу, морамо кренути од стека.

Предности:

1) Ради са циклусима;
2) Не оптерећа програм приликом копирања (дефинише копирање);
3) Не оптерећа програм када објекат постане бубре;

- У неком тренутку мора доћи до брисања бубрећа. Шта би се десило

ако бисмо дозволили да програм настави са радом током брисања?

- Може доћи до неког проблема тј. да се деси нешто што не желимо приликом промена показивача. Зато се користи STOP THE WORLD (заустави се свет) да се изврши брисање ђубрета. Брисање се најчешће примењује када понестане слободне меморије (ово је једна од мана).

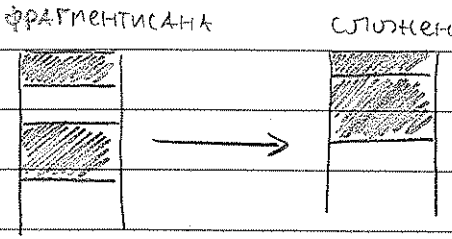
мане:

1) заустављање света;
2) локалност меморије;
3) фрагментација;

- Изгубили смо детерминизам, а добили ефикасност
- Морамо прећи цео стек (показиваче) и цео хип.
- У сваком кораку имамо промашај у кешу (сваки пут проверавамо објекат на другој меморијској позицији)

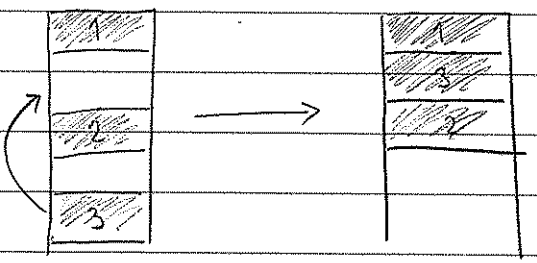
c) MARK-and-compact

- Решава проблем меморије: фрагментација
- Алгоритам MARK-And-compact представља модификацију претходног алгоритма, где врши и сабијање. Из „sweep" врши и „compact"



Проблем: Ово није ефикасна операција (са друге стране, није ни MARK-AND-sweep ефикасна операција)

Сабијање: Врши „SLIDING" објеката и тиме чува поредак. Овај поредак није битан. Ефикасније је ако можемо, у бајтима објекат између нека два објекта:



[*Приликом померања објеката, остаје FORWARD показивач на објекат до следећег „MARK"-а који разрешава локацију

Проблем: Скуп алгоритам, а фрагментација остаје.

Како у потпуности елиминисати фрагментацију? - Ако су сви објекти исте величине, онда можемо извршити ефикасно пребацивање и сабијање. Решење: партиционисање.

TWO FINGER COMPACTION:

а) погодан за објекте исте величине;
б) партиционисање са 2 показивача (почетак и крај);
в) притода произвољним алгоритмима сабијања.

*Подразумевано додељујемо 4KB програму

Предности:

1) мала фрагментација
2) локалност

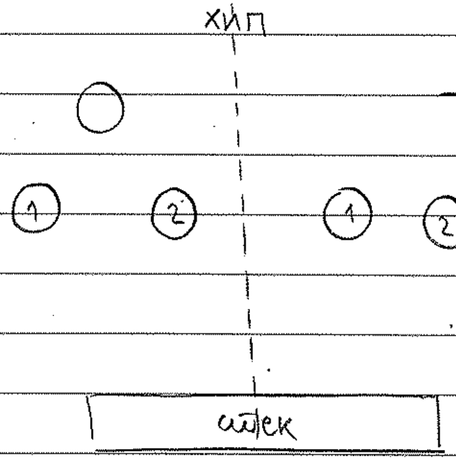
мане: споро скупљање отпадака

• Често се користи као комбинација са MARK-AND-SWEEP. Тек када

фрагментација постане велика се користи сабијање, не вршимо га када није потребно.

д) Копирајући GC

- Сваки пут копирају све
- Раздвајамо меморију на два дела тј. fromspace и tospace
- Копира се са fromspace у tospace
- На крају обрћемо fromspace и tospace.



- Обилазимо све објекте као у „MARK" и уместо обележавања, пребацујемо их у други део
- Не морамо да радимо било шта, довољно је да игноришемо стари део.

- Избацује се „SWEEP" део, јер не морамо да обилазимо све објекте на хипу. Доста ефикасна техника за програме који имају много ђубрета, а мало података

Предности:

1) локалност;
2) Једноставно скупљање објеката
3) брза алокација
4) мала фрагментација (приликом копирања вршимо сабијање)

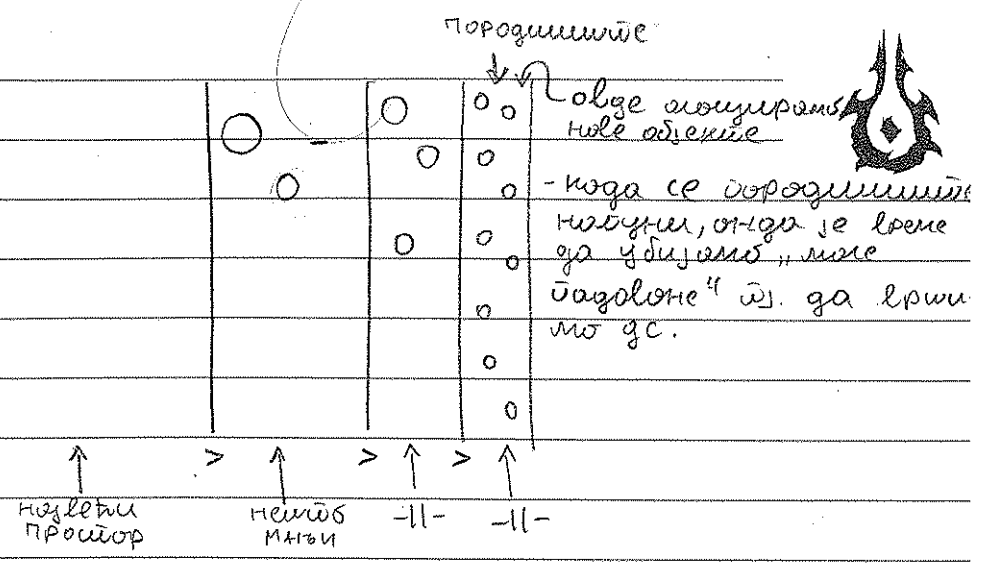
Мане:

1) дупло мања меморија
2) копирање података

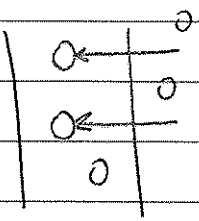
Не можемо користити на системима са малом меморијом. 3) копирање великих дуготрајних објеката.

Ђ) Генерацијски GC

- Заснован на претпоставкама:
  1) најмлађи објекти (скоро алоцирани) имају већу шансу да буду уништени.
  2) мали објекти се ћешће уништавају (ако имамо програм који константно алоцира и брише велике објекте, онда нешто није у реду)
- Делимо меморију на више делова различитих димензија.



ГС: ирши MARK-AND-SWEEP и чуваш бројач за дати објекат који означава колико је генерација иза преживео. Када бројач достигне одређену границу, ирши пребацивање. Овај чувају привремени FORWARD показивач до недостајућег „MARK"-а, када се показивачи ажурирају.



Добијамо сегментацију: најстарији, мањи, још мањи, најмлађи.

• Најстарији су они који су преживели највише генерација
• Овај GC има особину да ређа сличне објекте са већом вер. један до другог, што је корисно за кеш (хеуристика)

• Проблем са GC је што доста троше процесорско време. Сви ови алгоритми заустављају свет да би могли да почисте ђубре. Увек је јефтиније не радити ништа ако није потребно да се ради. Боље решење од GC је да имамо систем који не захтева чишћење преко GC.

е) линеарни типови: (Philip Wadler)
Linear types can change the world
• вредност сме <u>само једном</u> да се искористи;
• нема потребе за бројањем референци или GC
• није дозвољено копирање
• детерми-нистичко уништавање типова

• прелазимо из света у свет;
• све старо одбацујемо (заборављамо);

f :: Int → Int

g :: Int ⊸ Int lollipop - стрелочка означає лінійний тип

- Ніде дітись без Хаскела

V

Парцијална апликација и каријевање

Од n-арне ф-је f правимо нову ф-ју g тако што ф-ји f вежемо неке од аргумената за конкретне вредности

```
sum = \init xs -> foldr (+) init xs
g := sum (sum a b)
f := foldr (foldr a b c)
```

- фиксирали смо $a := (+)$ у foldr, где је (+) оператор сабирање тј. конкретна вр.

Ово се назива парцијална апликација

Каријевање: Све ф-је су унарне ф-је (сем const)

ф-ја f која је n-арна узима један елемент и враћа (n-1)-арну ф-ју

$f\ x\ y\ z = x + y + z$,

$f' = \backslash x \to \backslash y \to \backslash z \to x + y + z$

$$
\begin{aligned}
f'\ 1\ 2\ 3 &= f_1\ 2\ 3, & f_1\ y\ z &= 1 + y + z \\
&= f_2\ 3, & f_2\ z &= 1 + 2 + z \\
&= f_3, & f_3 &= 1 + 2 + 3
\end{aligned}
$$

Алгебарски типови података:

Проблем: Направити програм који за дату интернет адресу враћа колико речи на тој адреси има.

Шта нам је потребно?

- url : string;
- бројач : unsigned;
- web-page : socket;

Такође су нам потребне заставице за стање:

- started : bool;
- finished : bool;

У C-у би ово енкапсулирали у структуру

```
struct state_t{
    bool started = false;
    bool finished = false;
    unsigned count = 0;
    string url;
    socket web_page;
};
```

Шта је овде проблем?

- Вуче на много багова, могућности пропуста и дозвољава немогућа стања.

Производ типова A и B је тип чија вредност садржи тачно једну вредност типа A и тачно једну вредност типа B.

Пример1: (Int, String)

data Context c = MkContext String c

Пример2: Споменик мишеви хероји (посветили своје животе науци). DNK је производ типова који гради ланац. Могуће вредности су A, T, C, G. Број „могућих" ланаца DNK дужине n је 4^n. Велики број ових ланаца је неважан. DNK представља производ типова. У најбољем случају ако нешто промашимо то биће неће бити рођено. У горем случају можемо добити катастрофе (багове). Овај пример је метафора за програм и багове у зависности од могућих стања. Ако у програму направимо тип који има много већи број неважних стања од важних стања, онда можемо само да очекујемо проблеме. Овај пример илуструје проблем дефинисане структуре за почетни проблем:

$$P = S \times B \times B \times Z$$

S – Url, B – Started, B – Finished, Z – Бројаc

- Које стање за тајмер (Started, Finished) није могуће? (FALSE, TRUE)

=> додала сам нам три стања:

```
enum mode{
    INITIAL,        иницијално стање
    running,        стање бројања
    Finished        крајње стање
};
```

- Ако смо у иницијалном стању, није нам потребан бројач ни сокет, већ само url за повезивање. Ако смо у стању бројања онда нам је само битан бројач и сокет, а у крајњем стању је само битан бројач (резултат).

стање	шта нам је потребно
INITIAL	url
running	~~counter~~ socket
Finished	counter

• Треба водити рачуна о цурењу меморије приликом преласка из стања у стање.

• Овиме и даље имамо невалидна стања, али смо их смањили (нпр. невалидан пид)

• Уместо да правимо производ типова (овај и овај и ...), можемо да имамо суму (унију) типова (овај или овај или ...). Програм је у нашем случају или initial или running или finished

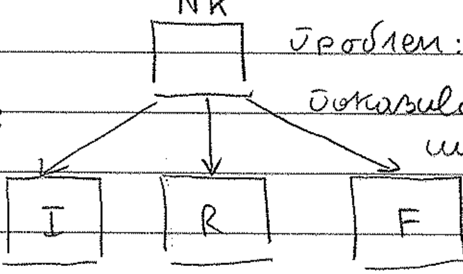
СУМА ТИПОВА A и B је тип чија је вредност садржи или једну вредност типа A или једну вредност типа B

• У C-у би ово имплементирали као унија са заставицама о стању.

• У C++ је то std::variant

• У OOP језицима можемо да имамо показивач надкласе који показује на једно стање:

НАШ ПРИМЕР:



Проблем: шта ако је показивач nullptr или невалидну локацију у меморији

Пример3: Maybe

data Maybe a = Nothing | Just a

- Nothing е конструктор който няма параметризован тип. Нема аргументи. Он има точно една предност
- Just a е конструктор който е параметризован типом a
- резултат е една предност която има един скрит а.

Пример4: data En = Val1 | Val2 | Val3

- няма при различие предности.
- Представва обикновен енумератор

Пример5: data Either a b = Left a | Right b

- няма скрит а и скрит b

Пример6: data Bool = True | False

Пример7: Що е листа?

*Листа е няма прозе листе и листе (рекурзивна деф.)

$La = \{[]\} \cup (a \times La)$

*$La = a \cup (a \times La)$ листа са борем једним елементом.

*$La = (a \times a) \cup (a \times La)$ листа са бором два елемента.

Напишок проблема: $P = I \cup R \cup F$

INITIAL Running Finished

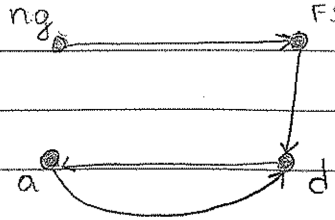
•R је произвољ скупа сокета и скупа целих бројева

```
data Program = INITIAL {url : String}
             | Running {s : socket
                        broj : Int}
             | Finished {broj : Int}
```

Проблем: Тенис кола

*Направити програм који води евиденцију о тенису. Поени: 0 15 30 40 A D

```
UNION tennis {
    normal game
    FORTY-something
    deuce
    advantage}
```



добијамо граф стања. У тренутку када пређемо са неких од стрелица, знамо да програм ради нешто погрешно. Можемо пре компилације да пустимо анализатор да провери да ли може доћи до грешке. Можемо додати и завршно стање тј. који је играч победио.

· Главна поента овог проблема је да минимизујемо неважна стања, а не чистоћа.

Теорија категорија: VI

*Апстрактна теорија ф-ја

*Апстрактна теорија бесмислица

-овако је зову људи који је не воле

*PRIMORDIAL SOUP - Биолошка супа

-овако је зову људи који је воле.

• Функције су уско везане за скупове. Ако избацимо скупове да дефинишемо нешто што личи на ф-је, али апстрактније.

Метаграф: састоји се из:

1) објеката: $a, b, c, \dots$
2) стрелица: $f, g, h, \dots$

(стрелице су исто објекти)

[*интуитивно ово личи на ф-је и скупове, али је заправо апстрактно]

• Свака стрелица има **домен** и **кодомен**

домен: $a = \mathrm{dom}\, f$ (додељује објекат a свакој стрелици)

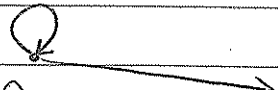
кодомен: $b = \mathrm{cod}\, f$ (додељује објекат b свакој стрелици)

$$f : a \to b$$

(f – стрелица, a – домен, b – кодомен)

a, b и f могу бити било шта.

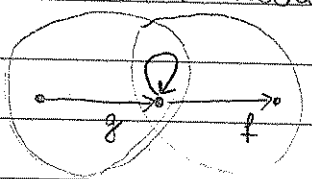
(усмерен граф)



Категорија: мета граф са следећим особинама:

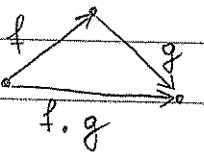
а) **идентитет:** додељује стрелицу $id_b = 1_b : b \to b$ сваком објекту b, где 1_b задовољава $1_b . f = f$ и $g . 1_b = g$ за сваке $f: a \to b$ и $g: b \to c$ ($.$ је композиција)

б) **композиција:** додељује стрелицу $g.f$ сваком пару стрелица f и g са $dom\, g = cod\, f$. Стрелица $g.f$ се назива композицијом стрелица g и f са $g.f: dom\, f \to cod\, g$. Композиција мора бити асоцијативна.



$id.f = f$

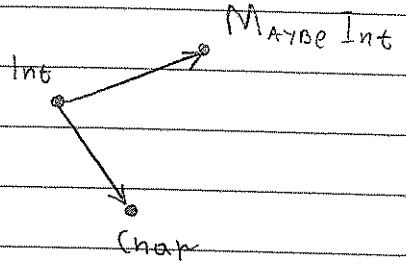
$g.id = g$



* Свака ф-ја је стрелица. Свака стрелица није ф-ја.
* Примери категорија:
  - скупови и ф-је
  - групе и хомоморфизми

*Пракшична категорија: шипови у Хаскелу

- објекши су шипови;
- сшрелице су ф-је;
- иденшишеш је ф-ја id;
- композиција је оџерашор (.).



Моноид (M,u,n): У кашеџорији $\langle C, ., 1 \rangle$ је објекаш M уз две сшрелице:

$u: M.M \to M$, мноџење

$n: 1 \to M$, јединица

*моноид у алџебри је сшрукшура која се сасшоји из нейрозној скуџа и оџерације која је асоцијашивна. У моноиду џосшоји јединшвени неуџрал (mono id).

У Хаскелу је u ~ mappend, а n mempty (u конкашенише ћредносши, а n је неуџрал)

*Пример моноида: (N, +, 0)

•Правила (Хаскел):

(<>) = mappend

x <> mempty = x

mempty <> x = x

(x <> y) <> z = x <> (y <> z)

Пример1: Низе (String):

<String, ++, "">. Може и уникалните ѝ:

за произволна листа:

```
instance Monoid [a] where
    mappend = (++)
    mempty = []
```

Пример2: Maybe

```
instance Monoid (Maybe a) where
    mappend = ???
    mempty = Nothing
```

~Как дефинираш mappend?

Jx ~ Just x , N ~ Nothing

•мора да бъди:

N · N = N

N · Jx = Jx

Jx · N = Jx

$Jx \cdot Jy = ?$

Постоји више могућности:

1) $Jx \cdot Jy = J(x \cdot y)$ (ово подсећа на списак)

2) $Jx \cdot Jy = Jx$

3) $Jx \cdot Jy = Jy$

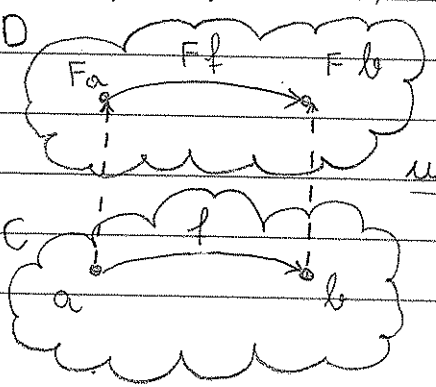
```
Instance Monoid (Maybe a) where
  mempty = Nothing
  mappend Nothing x = x
  mappend x Nothing = x
  mappend (Just x1) (Just x2) = Just (mappend x1 x2)
```

Функтор између категорија C и D је мапирање објеката на објекте и стрелица на стрелице које задовољава следећа правила:

a) $F(f) : F(dom\ f) \to F(cod\ f)$ за сваку стрелицу f у категорији C.

б) $F(g \cdot f) = F(g) \cdot F(f)$

в) $F(1_a) = 1_{F(a)}$



Шта је лифтинг (LIFTING)?

- Замислимо да имамо кутију у којој се налази број n. Желимо да пресликамо број n неком унарном ф-јом f у број m ($f(n)=m$) УНУТАР КУТИЈЕ. Да бисмо то урадили потребно је „отворити" кутију и узети број n, пресликати број n ф-јом f у број m и на крају „затворити" број m у кутији. Овај UNWRAP-AND-WRAP процес се назива „LIFTING"

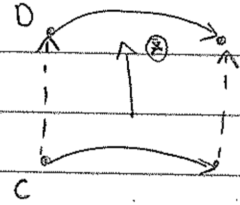
```
LIFTBox :: (a -> b) -> Box a -> Box b
            n    m         n        m
```

$f: a \to b$

$F(f): F(a) \to F(b)$

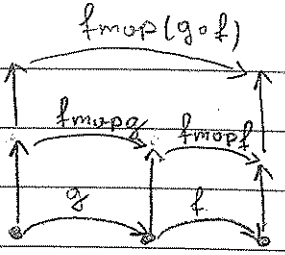
$F(id_a) = id_{Fa}$

$F(f \circ g) = F(f) \circ F(g)$



- У Хаскелу је Functor све што има fmap.
- Када кажемо fmap мислимо на стрелицу ⊛

```haskell
class Functor f where
  fmap :: (a -> b) -> f a -> f b
```



Ендофунктор е функтор категорије C на C

Примери: листе, Maybe, Either a

* Either има два објекта која се пресликавају на трећи, па мора да буде пресликавања као Either са фиксираним једним

* (->) стрелица.

Сама стрелица је параметризована на два типа. Онда можемо да очекујемо да је '(->)r' функтор. Ово неформално можемо да видимо као 'r ->'.

```
fmap :: Functor f => (a -> b) -> f a -> f b
```

(општи облик за fmap)

Заменимо f са (->)r и добијамо:

(a -> b) -> (->) r a -> (->) r b , тј.

(a -> b) -> (r -> a) -> (r -> b)

=> добили смо композицију fmap = (.)

```
Instance Functor ((->) r) where
    fmap f g = f . g
```

* Пошто fmap можемо да предефинишемо преко оператора (<$>), онда можемо да комбинујемо матрјошке на фин начин

(+2) <$> (*3) <$> Just 2 = 8

Монаде:

- Сваки код функтор може да буде монада ако дефинишемо return и join
- Монада M = <M, return, join> над категоријом C се састоји од ендофунктора M: C -> C и две трансформације return C -> M(C) и join: M(M(C)) -> M(C) које задовољавају следеће:

а) леви идентитет:

return a >>= f = f a

б) десни идентитет:

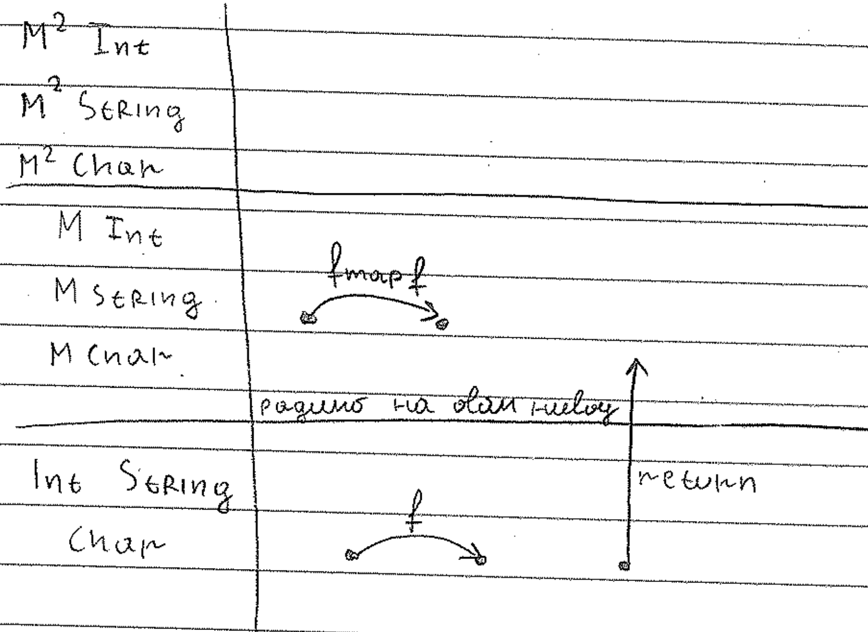
m >>= return = m

в) асоцијативност:

(m >>= f) >>= g = m >>= (\x -> f x >>= g)

оператор (>>=): bind оператор. Предефинисан композицијом join и fmap

```
class Monad m where
  return :: a -> m a
  (>>=) :: m a -> (a -> m b) -> m b
  (>>) :: m a -> m b -> m b
  x >> y = x >>= \_ -> y   (подразумевана деф
  fail :: String -> m a   (ово је за компајлер)
  fail msg = error msg   (подразумевана деф.
```



• M је било који функтор (пример Maybe)

• return нам враћа вредност ниво изнад:

```
return 2 >>= (\x -> Just (2*x))
```

(>>=) захтева m a за леви параметар

return узима 2 (a) и враћа

Just 2 (m a). резултат: Just 4

• Уколико ф-ју f лифтујемо, тада на нивоу више добијамо fmap f
• join не можемо да зовемо над доњим нивоом, јер мора бар два M да има
• join враћа ниво испод, тј. уз више позива join увек можемо да се вратимо на ниво Ma
• Ове две ф-је (return и join) нам дозвољавају да континуално живимо у средњем слоју (Ma). Овако деф. никад не можемо се вратити у почетни простор. За неке конкретне случаје је то могуће:
- за листу је могуће
- за Maybe је могуће (fromMaybe)
- У општем случају за IO није могуће

* bind -> лифтовано fmap нас пребацује у ниво изнад, а преко join-а се враћамо у тренутни

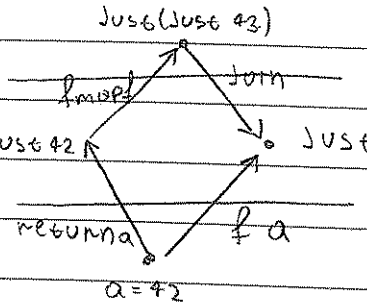
* Објашњење на примеру:

```
Just 2 >>= (\x -> Just (x+2)) = Just 4
join (fmap (\x -> Just(x+2)) (Just 2)) = Just 4
=> m >>= f <=> join (fmap f m)
```

* леви идентитет:

return a >>= f = f a

fmap . join



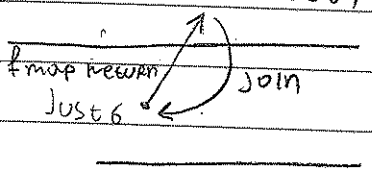
пример

a = 42

$$f(x) = \begin{cases} \text{Just } x+1, & x \geq 0 \\ \text{Nothing}, & x \leq 0 \end{cases}$$

* десни идентитет:

m >>= return = m (m је кутија са заноштом)



До нотација:

```
foo :: Maybe String
foo = Just 3 >>= (\x ->
      Just "!" >>= (\y ->
      Just (show x ++ y)))      = "3!"
```

Објашњење: Из кутије 'Just 3' извучемо 3 и дозвемо као улаз ф-је са десне стране (лямбда). Слично за 'Just "!"'

Имамо: x=3 и y="!", show x = "3"

show x ++ y = "3!"

Овај код је еквивалентан са

```
foo :: Maybe String
foo :: do
  x ← Just 3      (← извучено из кутије)
  y ← Just "!"
  Just (show x ++ y)
```

Напомена: Посматрати ствари као кутије је неформално :(

* Леви идентитет преко do нотације:

```
do { x' ← return x
   ; f x'
   }
```

Објашњење: Запакујемо вредност x у кутију и онда из кутије извучемо вредност преко (<-) и убацимо у ф-ју f. То је еквивалентно са f x.

* Десни идентитет преко do нотације:

```
do { x ← m
   ; return x
   }
```

Објашњење: Из кутије извучемо вредност x и опет је запакујемо. То је m

*Асоцијативност преко до нотације:

$(m >>= f) >>= g = m >>= (\lambda x \; f\, x >>= g)$

```
do { x ← m
   ; y ← f x
   ; g y
   }
```

-изачинимо x из куйице и убацимо у ф-ју f. Тада из резултата ф-је изачинимо y и убацимо у ф-ју g. Ово је екв. са:

$(m >>= f) >>= g$

```
do { x ← m
   ; do { y ← f x
        ; g y
        }
   }
```

-извачимо x из куйице m и убацимо у do блок. Добијамо исто као мономе.

```
do { y <- do { x <- m
             ; f x
             }
   ; g y
   }
```

- извлачимо x из кутије m и позивамо f x што враћа неку кутију из које извлачимо вредност y у На крају y убацујемо у g функцију.

Кисин оператор (Fish operator) >=> :

- Прави композицију монадичних вредности (кутија)
- Примењујемо ф-ју и опет добијамо монадичну вредност:

*леви и десни идентитет:

$f >=> return = return >=> f = f$

*асоцијативност:

$(f >=> g) >=> h = f >=> (g >=> h)$

$(>=>) :: (a \to m\ b) \to (b \to m\ c) \to a \to m\ c$

*Свако монада јесте функтор, али није свако функтор монада (монаде су подскуп функтора)

Проблем функтора: Можемо преко fmap да примењујемо ун. ф-је на „ствари које се налазе у кутијама":

fmap (*3) (Just 2) = Just 6

Шта добијамо са fmap (*) (Just 2)?

Добијамо Just ((*) 3) = Just (3*) тј. ф-ју унутар кутије. Можемо да урадимо следеће:

Let a = fmap (*) [1,2,3,4]

fmap (\f -> f 9) a = [9, 18, 27, 36]

□

□: узима ф-ју као аргумент и примењује на 9

Пример: (\f -> f 9) (3*) = (3*) 9 = 27

Проблем настаје када желимо да искори-стимо ф-ју унутар кутије за мапирање.

Пример: Имамо Just (3*) и Just 5 и желимо да извучемо (3*) из Just (3*) и мапира-мо Just 5. Ово не можемо са функтори-ма. Не можемо да мапирамо преко ф-је која се налази у функтору (кутији) неки функтор (кутију)

Тај проблем решавају ојачани функтори:

```
class (Function f) => Applicative f where
  pure :: a -> f a
  (<*>) :: f (a -> b) -> f a -> f b
```

↓

једна слобода која се разликује од fmap ~ <$>

Сада можемо да урадимо:

(Just (*3)) <$> (Just 5) = Just 15,

што решава претходни проблем pure узима вредност и ставља је у ојачани функтор (кутију)

*Ојачани функтори представљају нешто на шта можемо наставити идеју о монадама иако монаде не користе АФ у свој дефиницији.

Пример: Maybe. Треба дефинисати return и оператор (>>=):

```
return :: a -> m a          (опште)
(>>=) :: m a -> (a -> m b) -> m b
```

return задължително дефинира:

return x = Just x

За оператор (>>=):

Nothing >>= f = Nothing (ако приложим f на нищо, добиваме нищо)

Just x >>= f = f x

*FAIL _ = Nothing

Пример 2: Пор(,) : Ако е а моноид, онда е (,) а монада:

return b = (mempty, b)

join (a1, (c, b)) = (mappend a1 a2, b)

(a, b) >>= f

Пример 3: функцията:

join :: Monad m => m (m a) -> m a

• Заменим m са (->) r :

join :: (((->) r) (((->) r) a)) -> ((->) r) a

=> join :: (r -> (r -> a)) -> (r -> a)

=> join :: (r -> r -> a) -> (r -> a)

=> join f = \x -> f x x

Пример 4: Изразяване fmap чрез >>=, >> и return (успешен ДАН1 2019):

```
fakemap f z = z >>= (\x -> return $ f x)
```

Закључивање типова: VIII

* Обични дефинисани типови код дефинисања променљиве:

1) x :: [Char]
   i :: Int
   } => x !! i :: Char

2) Short i
   std::ARRAY<int, 4> a } a[i] = 6 је Int

Када ми закључимо тип (1) да је тип Char
(!!) :: [a] -> Int -> a => a = Char
x :: [Char]

Параметризовани типови: умемо да дамо конкретан тип, имамо типовске променљиве
f :: a -> Bool, а може да буде било шта
g :: a -> a -> a

```
template <typename T>
T id(T VAL, std::vector<T> xs) { ... }
```

(Прихвата један тип као параметар типа, док прихвата два аргумента)

- Ако знамо T, онда знамо тип оба аргумента ф-је.
- Слично као за ф-ју g у прећ. примеру

- Ова се нивоа генерички програмирање, полиморфизам преку типова, compile time полиморфизам.

- Код језика који се компајлирају, морате дедуковати типове током компилације

Имплицитни генерички типови:

Пример 1

Let f = \x -> x

=> f :: a -> a, јер две исте врсте => два иста типа

Let g = f

=> g :: a -> a, јер је f :: a -> a

Let h = \x -> f x , a = ?, b = ?, c = ?

(a; c; b)

- С је ф-ја облика a -> b ~> f :: a -> b
- из претходног знамо да је f = \x -> x

=> a = b => f :: a -> a , c = a -> a

=> h = a -> a

Пример 2: претпоставимо да нема класа типова.

let max $\underbrace{x}_{a}$ $\underbrace{y}_{b}$ = IF $\underbrace{x > y}_{c}$ then $\underbrace{x}_{a}$ else $\underbrace{y}_{b}$ (a brace under "x > y then x else y" is labelled d)

• Нема overloading => оператор > може да ради само за један тип x и један тип y. Пример (>) упоређује Int и String (што да не :o)

=> x :: Int, y :: String

=> a = Int, b = String

• Нека IF захтева Bool (стандардно)

• Што се дешава ако је (>) :: Int -> String -> Char?

- доводи до компилационе грешке, јер IF не добија Bool који захтева.

• Ако је (>) :: Int -> String -> Bool, онда се услов испуњава и не доводи до грешке.

• У Хаскелу, C-у, SQL-у IF не може да врати различите вредности у then и else

=> a = b => Int = String у грешка

=> код није добар, не пролази компилацију.

• Ако дозволимо overloading онда желимо да одредимо најопштији тип који испуњава два правила.

Тада би морало да буде c = Bool и a = b

max :: a -> a -> a ∧ x>y :: a -> a -> Bool

Пример 3:

let max cmp x y = if cmp x y then x else y

(max : d, cmp : a, x : b, y : c; у if: cmp : a, x : b, y : c, cmp x : e, cmp x y : f; then x : b, else y : c)

- then и else су истог типа => b = c
- d = Bool, јер IF тако захтева

=> cmp :: c -> g, g = ?

=> g y :: Bool (g пресликава нешто на Bool)

=> g = c -> Bool

=> cmp :: c -> c -> Bool , a = c -> c -> Bool

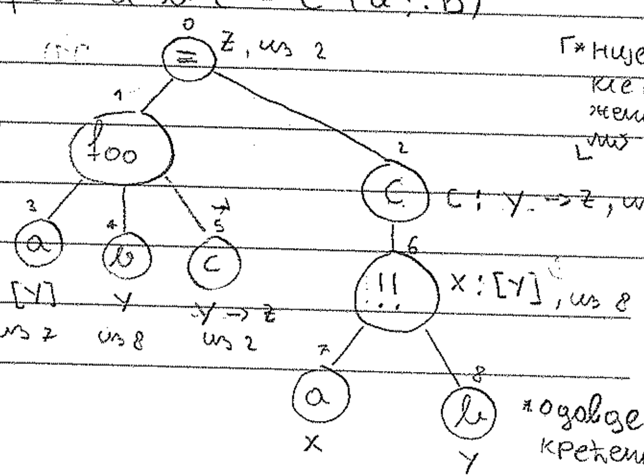
Шта је max?

max :: (c -> c -> Bool) -> c -> c -> c

(cmp : (c -> c -> Bool), x : c, y : c, рез. : c)

Пример 4:

let foo a b c = c (a!!b)



[* није битно одакле кренемо. можемо да користимо хеуристику]

=> foo :: [Y] -> Y -> (Y -> Z) -> Z

Пошто десни аргумент од !! мора да буде Int, онда важи: Y = Int

foo :: [Int] -> Int -> (Int -> Z) -> Z

Пример:

Let foo a b c = c (a !! b) + a

· C треба да враћа нешто што је истог типа као и a. Из претходног: a = [Y] ∧ c = Y -> Z

=> Z = [Y], сабирање низ листа није деф.

Hindley-Milner провера типова:

а) Закључује типове који нису експлицитно навели

б) језик, конструкти;

в) унификација типова;

- за произвољан бин. оп. нам је потребно три типа (a -> b -> c)
- ако је тај оператор релација, онда нам је потребно два типа, јер је резултат Bool (a -> b -> Bool)
- оператор < одреди два аргумента који су истог типа и враћа Bool (a -> a -> Bool

- оператор + који са леве стране а очекује а са десне стране и враћа а. Такође можемо да очекујемо да је а неки нумерички тип (у Хаскелу) (Num -> Num -> Num)

- оператор !! очекује са леве стране листу, са десне стране Int и као резултат враћа исто оно што је тип листе. - очекујемо да знамо тип константи: "Hello"::String, 42::Int

- ф-ја func a b c .... z. Треба одредити тип сваког од аргумената

*Ограничења:

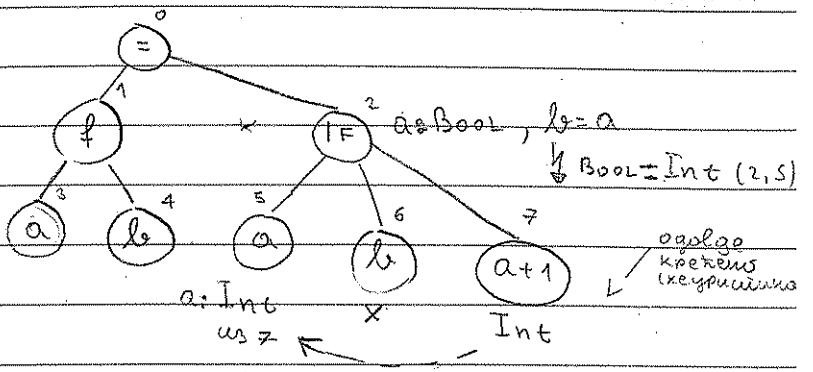
пример: IF-then-else, деф. ф-је

*Унификација:

- Крећемо од корена дрвета;
- Улазе налазимо на конструкт са слободним типовима, изједначимо ове типске параметре.
- Ако конструкт има дозвољена два типа T1 и T2, заменимо T1 са T2 у целом дрвету

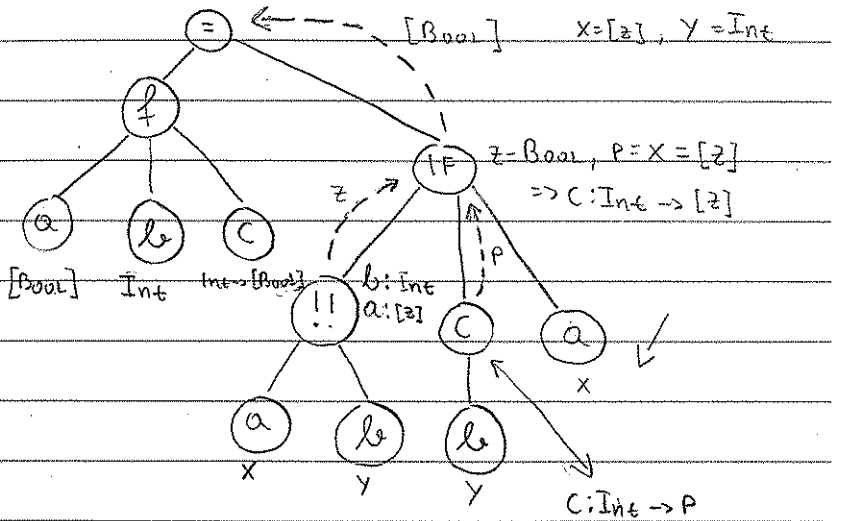
Пример 6:

Let f a b = IF a then b
else a+1



Пример 7:

Let f a b c = IF (a !! b) then c b else a



=> f :: [Bool] -> Int -> (Int -> [Bool]) -> [Bool]

Закључивање типова у C++-у: IX

* У C++ кода дефинишемо шаблон, он је дефинисан тј. да „ради" за шта год да нама падне на памет (може доћи до компајлерске грешке). Када ф-ју позовемо за неки конкретан тип, тада се врши копирање те ф-је са заменом параметарским типовима у конкретан тип. Компајлирањем нове ф-је се проверава да ли су испитована правилно. Када дефинишемо шаблон, ми смо само написали упутства како се прави нека конкретна ф-ја заменом типова у конкретан тип.

Пример1:

```
template<typename T>
void f(T value) {}
int main() {return 0;}
```

Колико ф-ја се поред main-а налази у извршном коду?

- Ниједна, имамо само шаблон, где нисмо инстанцирали ниједну ф-ју. Шаблон никад не завршава у извршном коду док се не инстанцира.

*Auto универсален заради силната дедукция

Auto value = 42; (int)

Пример 2:

template<typename T>

void f(T x);

string s="abc";

string& ss = s;

дедукцията

f(s) ~ T = string

f(ss) ~ T = string

-универсалният шаблон узнава какъв тип е Т и какъв референс, const...

Пример 3:

template<typename T>

void f(T*);

int* p = nullptr

int i = 0

дедукцията

f(p) ~ T = int

f(i) ~ error

*Пример 4:

```
template<typename T>
void f(T&& value){          (омотач)
    g(std::forward<T>(value));
}
```

-ово може да се позива са обичном вредношћу, са реф. или са привременом (&&) вредношћу.

-потребно је још да у ф-ју g проследимо како то што смо добили. Зато користимо **forwarding reference** што ради баш то што желимо.

* **decltype** дедукује тип израза.

Пример 5: perfect forwarding (return)

```
template <typename T>
decltype(auto) f(T&& value) {
    return g(std::forward<T>(value));
}
```

- исто као претх. пример, само за повратну вр. Оно што добијемо, то и вратимо

* Шаблони су Тјурин комплетан језик који се извршава током компилације

* vector<int> : vector је ф-ја која узима тип int и враћа тип vector<int>.
Да ли постоји ф-ја која узима vector<int> и враћа int? (мора да буде током компилације)

```
using inner_type_t =          (using ~ typedef)
    typename T::value_type;
```

inner_type_t<std::vector<int>> , где value_type враћа тип елемента колекције.

* За дефинисање компилационих ствари користимо struct/class:

```
template<typename T>
struct inner_type{
    using type = typename T::value_type;
};
inner_type<std::vector<int>>::type
```

Пример 6:

```
template<typename T, typename Expected = void>
struct expect_inner{
    using type = typename T::value_type;
    constexpr bool value =
        std::is_same_v<type, Expected>;
```

expect-inner<std::vector<int>>::value

constexpr: користимо над нешто да се нешто израчуна токам компилације

is_same_v: упоређујемо типове.

Зашто је ово корисно? Пример алгоритам за обрађивање које има ефикаснију верзију за низове него за листе. Још увек нешто да сакријемо по у једну ф-ју.

Концепти - c++ 20. Дефинишемо ресурикције преко предиката: HasValueType, Hashable