## Робота №6

Функції і Процедури

Розглядається проста процедурна мова, що містить чисті функції та «нечисті» імперативні процедури, котрі можуть змінювати глобальні змінні і повертати результат.

Функціональна компонента мови концентрується навколо *виразів*. Вираз може бути:

* *Константи*. Є значення двох типів: цілі або одновимірні масиви цілих (див. нижче). Логічних значень немає - вони кодуються цілими: true кодує ціле 1, а false - 0. В програмі можна вживати константи – цілі числа.
* З*мінні.* Ідентифікатори змінних визначаються користувачем і починаються з букви , наприклад і x, yy, VAR, cameCase т.д. Є змінні глобальні і локальні, котрі об’являються в процедурах.
* *Бінарні операції*. Є бінарні операції додавання, віднімання, множення, менше, дорівнює і індексування. Перші п’ять, що відповідають +, -, \*, < і == в Haskell, беруть лише цілі параметри (два масиви не порявнюються) і виробляють цілий результат. Для операцій менше і дорівнює результат буде або 0 або 1, представляючи false і true відповідно. Операція індексування реалізує доступ до елементів масиву: перший операнд - масив і другий - індекс (цілий).
* *Умовний* вираз має вигляд *тернарного умовного оператору* в Java - *P ? Q : R*.
* *Застосування функції визначеної користувачем* . В мові використовуються лише функції першого порядку (немає часткового застосування), застосування функціії - f (e1, e2, …, en), де e1, e2, …, en – вирази, що задають аргументи функції. Визначення функції включає ім’я функції f, список об’яв параметрів p1, p2, …, pn і вираз e з правого боку - на зразок *func* *f (p1, p2, …, pn) = e*. Об’ява параметру *x* (ціла змінна) або  *x[]* (цілий масив), де x – ім’я параметру.

На абстрактному рівні вирази і визначення функцій в Haskell описуються типами:

***type*** Id = String

***data*** Value = I Int | A [(Int, Int)]

***deriving*** (Eq, Show)

***data*** Op = Add | Minus | Mul | Less | Equal | Index

***deriving*** (Eq, Show)

***data*** Exp = Const Value |

Var Id |

OpApp Op Exp Exp |

Cond Exp Exp Exp |

FunApp Id [Exp]

***deriving*** (Eq, Show)

***data*** VarDef = Arr Id | Int Id ***deriving*** (Eq, Show)

***type*** FunDef = (Id, ([VarDef], Exp))

Значення цілих і масивів розрізняються конструкторами I і A в типі даних Value. Елементи масиву задаються списком пар, що відображають індекс масиву в значення.

Результат обрахування виразу – об’єкт типу Value, котрий може бути або ціле вигляду *I n* для деякого цілого n, або масив виду *A elems*, де elems - список відображення індексів у значення.

Наприклад:

* Масив представлений A [(3,1), (0,3), (5,12)] має значення, що визначені в позиціях 0, 3 і 5
* Операція індексування масиву з такими індексами поверне значення 3, 1 і 12 відповідно
* Порядок пар не важливий. Індекси не утворюють неперервний сегмент.
* Не повинно бути дублікатів входів для окремих індексів: масив зі значеннями, що визначені в n (унікальних) індексах повинен бути представлений списком пар точно з n елементів.
* Значення масиву в індексах, що не вказані, припускається нуль, тому при спробі індексувати попередній масив A в позиціях 1, 2, 4, 6, 7, … буде повернуто 0.
  + В цьому відношенні масиви в мові – *необмежені* навіть у випадку їх скінченного представлення.

Приклади означень функцій:

* Функцію biggest, що повертає більше з двох чисел m і n

*func biggest( m, n) = (m < n ? n : m)*

* Функція представляється наступним FunDef:

biggest :: FunDef

biggest =

("biggest",

([Int "m", Int "n"],

Cond (OpApp Less (Var "m") (Var "n")) (Var "n") (Var "m")

)

)

* Функція fib, що обчислює число Фібоначчі

*func fib(n) =*

*(n < 3 ? 1 : fib(n-1) + fib(n-2))*

* Її можна представити

fib :: FunDef

fib =

("fib",

([Int "n"], Cond (OpApp Less (Var "n") (Const (I 3)) )

(Const (I 1))

(OpApp Add

(FunApp "fib" [OpApp Minus (Var "n") (Const (I 1))])

(FunApp "fib" [OpApp Minus (Var "n") (Const (I 2))]))

)

)

* Наступна функція демонструє індексування масиву - sumA додає елементи 0 …n масиву A

*func sumA(a[], n) =*

*(n < 0 ? 0 : a[n] + sumA (a, n-1))*

* Її можна представити

sumA :: FunDef

sumA =

("sumA",

([Arr "a", Int "n"],

Cond (OpApp Less (Var "n") (Const (I 0)))

(Const (I 0))

(OpApp Add (OpApp Index (Var "a") (Var "n"))

(FunApp "sumA"

[Var "a", OpApp Minus (Var "n")(Const (I 1))]))

)

)

Інтерпретатор виразів (функція, що обчислює вирази) повинен працювати з оточенням – список означень функції (FunDef), введених користувачем, для того щоб реалізувати застосування функції, i *станом*, що надає значення змінних, котрі є в області дії, в першу чергу значення аргументів функції. Інтерпретатор буде трактувати останні, як *локальні* змінні.

В компіляції всі локальні змінні з їх поточними значеннями називають *записом активації*. Оскільки одна функція в процесі обчислення може викликати іншу (або саму себе, якщо функція рекурсивна), то стан одночасно може містити декілька записів активацій. Всі записи локальних активацій утворюють *стек*. Коли *функція викликається*, то в стек заноситься новий запис активації, що містить поточні значення всіх локальних даних функції. По *закінченню обчислення функції*, її запис активації вилучається зі стеку. Оскільки інтерпретатору доступні лише локальні дані викликаної функції, то він завжди працює лише з верхнім записом активації в стеку.

Пізніше необхідно буде розглянути також *глобальні* змінні, котрі утворюють глобальний запис активації. При інтерпретації доступні одночасно як локальні дані, так і глобальні, тому, стан містить стек локальних записів активацій і глобальний запис активації:

***import qualified*** Data.Map ***as*** M

***type*** Binding = M.Map Id Value

***type*** StateP = ([Binding], Binding)

Стан програми (st елемент StateP) *st = (lcx, gl)* містить *lcx* - стек локальних записів активації і *gl* - глобальний запис активації. При виконанні (інтерпретації) функції або процедури останній локальний запис активації і глобальний запис активації утворюють *область дії* всіх доступних змінних.

Для забезпечення ефективної роботи з записами активації (Binding), використовуються структури даних (Map) - таблиці (словники, відображення), котрі зв`язують ключі (Id) зі значеннями (Value). Для роботи з ними потрібно імпортувати модуль Data.Map. Імена багато функцій в цьому модулі співпадає з іменами функцій в модулі Prelude, тому для них потрібно використовувати кваліфіковані імена.

Залишок мови процедурний і його можна уявити як дуже спрощена версія Java, без об’єктів. Об’ява процедура (на зразок - *procedure* *np(p1, p2, …, pn) block*) включає ім’я процедури *np* та список об’яв параметрів *p1, p2, …, pn*, як для функції. Різниця в тому, що права частина процедури є блок *block -* список *операторів*. Оператор може бути:

* *Присвоювання змінній x*, *x = e*, де e – вираз.
  + Якщо змінна вже є в області дії імен, то її значення замінюється значенням виразу, що справа. В іншому випадку, присвоювання ініціює введення *x* як нової локальної змінної.
* *Присвоювання елементу масиву a*, *a[i] = e*, де *a* - масив, *i* - цілий індекс і *e* - вираз.
  + Масив *a* вже повинен бути в області дії імен, тобто присвоювання елементу масиву виду *a[i] = e* не можна використати, щоб об’явити новий масив *a*.
* *Оператор ‘if’*, еквівалентний оператору Java виду *if (P) {S1} else {S2}*, де P вираз з бульовим значенням (предикат) і S1 та S2 є список операторів. Предикат можна уявити, як вираз що обчислює або false (0) або true (1).
  + ‘if’ *оператори* не потрібно змішувати з умовними *виразами*.
* *Цикл while ,* *while (P) {S}*, де P - предикат і S - список операторів. Предикат P можна уявити, як вираз що обчислює або false (0) або true (1). Список операторів виконується повторно поки предикат є true.
* Виклик *процедури визначеної користувачем* з іменем *p* і виразами аргументами *e1, …, en .* 
  + *x := p(e1, …, en)* - якщо процедура повертає значення то це значення може бути зв’язаним зі змінною *x.* Замітимо, що виклик процедури – це не вираз в цій мові, тому зв’язування результату виклику процедури не теж саме, що присвоювання значення виразу змінній.
  + *call p(e1, …, en)* - якщо процедура не повертає значення
* Оператор *return*, *return e*, де *e* - вираз. Завершує обчислення процедури і повертає з неї значення виразу *e* .
  + Можна уявити, що оператор повернення буде завжди останнім оператором, що виконується в середині процедури.
* Обчислення процедури завершується або при виконанні оператору *return e* або після виконання останнього оператора блоку процедури

Процедури представляються наступними типами в Haskell.

***data*** Statement = Assign Id Exp |

AssignA Id Exp Exp |

If Exp Block Block |

While Exp Block |

Call Id Id [Exp] |

Return Exp

***deriving*** (Eq, Show)

***type*** Block = [Statement]

***type*** ProcDef = (Id, ([VarDef], Block))

Виклики процедури, коли вона повертає значення і коли НЕ повертає, представляються

* *x := p(e1, …, en)* - Call “x” “p” [e1, …, en].
* *call p(e1, …, en)* - Call “” “p” [e1, …, en].

Оскільки процедура може викликати, як функції визначені користувачем (протягом обчислення Expr), так і процедури визначені користувачем, то інтерпретатор операторів повинен мати доступ до

* списку функцій - FunDefs,
* списку процедур - ProcDef.
* значення змінних, що є в області дії – останній запис локальної активації та запис глобальної активації, - StateP.

При виклику процедури, як і при виклику функції, створюється новий запис локальної активації, що містить зв’язування для значень аргументів процедури, і заноситься в стек локальних записів активації. Таким чином стан, що переданий процедурі, має на вершині стеку запис локальної активації і запис глобальної активації, що містить значення глобальних змінних в точці виклику.

* Локальні змінні що є в області дії точки виклику зв»язані в записі активації, що знаходиться в стеку нижче останнього, тому що не видимі з процедури, що викликана.
* Якщо аргумент має таке ж саме ім’я, як і одна із глобальних змінних, то аргумент має переваги. Така глобальна змінна «невидима» протягом виконання тіла процедури.

При поверненні з процедури останній запис локальної активації вилучається зі стеку. Довільні зміни виконані з глобальними змінними в процедурі, яку викликали, залишаються в стані, що повернеться. Стан після виклику буде також містити на вершині стеку попередній локальний запис активації, тобто локальні змінні, котрі існували до виклику. Додаткові локальні змінні, котрі використовувалися процедурою, будуть вилучені разом з останнім локальним записом активації, тому що вони виходять з області дії імен при виході з процедури.

* Єдиний виняток - коли значення повертається з процедури, що викликана: оператор return e потрібно реалізувати присвоєнням спеціальній *локальній* змінній *$res* значення e і вона буде передана назад до місця виклику, як частина стану, що повертається.
* Якщо перший аргумент конструктора Call є непорожній рядок *“x”* , то значення змінної “x” в місті виклику процедури потрібно змінити на значення спеціальної змінної *$res* при поверненні з процедури. Якщо перший аргумент – порожній рядок “”, то зміна не потрібна.
  + Для вірної (well formed) процедури: кожний виклик виду *Call x p es*, де x *непорожнє,*  завжди закінчується оператором *Return e!*

Приклади означень процедур:

* Процедура gAdd додає два числа і присвоює результат змінній gSum

*proc gAdd(x, y) {*

*gSum = x + y;*

*}*

* + Якщо є глобальна змінна gSum в стані, з яким виконується gAdd, тоді –буде змінено її значення.
  + Якщо немає такої глобальної змінної - то присвоювання побудує нову локальну змінну з іменем gSum.
* Процедура представляється наступним ProcDef:

gAdd :: ProcDef

gAdd =

("gAdd",

([Int "x", Int "y"],

([], [Assign "gSum" (OpApp Add (Var "x") (Var "y"))])

)

)

* Далі процедурна версія функції sumA, що використовує цикл while для отримання суми масиву.

*proc sumA1 (a[], n) {*

*s = 0; i = 0; limit = n + 1;*

*while (i < limit) {*

*s = s + a[i]; i = i + 1*

*};*

*return s*

*}*

* Її можна представити

sumA1 :: ProcDef

sumA1 =

("sumA1",

([Arr "a", Int "n"],

[Assign "s" (Const (I 0)),

Assign "i" (Const (I 0)),

Assign "limit" (OpApp Add (Var "n") (Const (I 1))),

While (OpApp Less (Var "i") (Var "limit"))

[Assign "s" (OpApp Add (Var "s")

(OpApp Index (Var "a") (Var "i"))),

Assign "i" (OpApp Add (Var "i") (Const (I 1))) ],

Return (Var "s")]

)

)

В загальному випадку програма складається з трьох списків: глобальних змінних, функцій визначених користувачем і процедур визначених користувачем. Серед процедур завжди є одна процедура без параметрів з іменем main. Виконання програми: встановлюються значення всіх глобальних змінних в 0 і викликається процедура main.

***type*** Program = ([VarDef], [FuncDef], [ProcDef])

Приклад програми

*gSum;*

*proc gAdd(x, y) { gSum = x + y }*

*proc main() {call gAdd(5,10)}*

* Після її виконання стан буде містити одну глобальну змінну gSum зі значенням 15.

Надалі припускається, що всі програми і їх елементи являються вірними (well formed) в наступному значенні:

* Оператори, функції і процедури застосовуються до вірної кількості аргументів, кожний з яких має відповідний тип.
* Вирази, які повинні обчислювати логічні значення, будуть завжди обчислюватися до 0 (false) або 1 (true).
* В присвоєнні масиву a[i] = e масив завжди визначений (в області дії).
* Процедура, що викликається як *x := p(e1, …, en),* завжди буде повертати значення (закінчує своє обчислення оператором *return e*)

В допоміжному файлі, котрий включає визначення типів, допоміжних функцій і даних для тестування, надати визначення наступних функцій.

1. Функція *getValue id st*, котра знаходить значення змінної *id* в стані *st*. Передумова: значення змінної *id* є в стані *st*, але вона може знаходитися або в локальному запису активації або в глобальному. Наприклад:

getValue “x” sampleState = (I 5)

1. Функції *getLocals st* і *getGlobals st,* котрі відповідно повертають останній локальний запис активації або глобальний запис активації в стані *st*. Наприклад:

getLocals sampleState = fromList [(“x”, I 5]

getGlobals sampleState =

fromList [ (“a”, A [(0,4), (1,2), (2,7)]), (“y”, I 2)]

1. Функція *assignArray a i v*, що повертає масив такий самий як *a* за винятком його *i* елементу, значення якого потрібно зв’язати з *v*. Наприклад:

assignArray (getValue “a” sampleState) (I 2) (I 1) = A [ (0,4), (1,2), (2,1)]

1. Функція *updateVar (id,v) st*, котра змінює значення змінної *id* на *v* в стані *st,* якщо вона *зберігається в цій області дії*. Якщо змінна *id* не зв’язана в стані *st*, то необхідно додати нову **локальну** змінну *id.* Наприклад

updateVar ("x", I 6) sampleState =

([fromList [("x",I 6)]], fromList [ ("a", A [(0,4),(1,2),(2,7)]), ("y", I 2),])

updateVar ("z", I 3) ([M.empty], M.fromList [("g",I 8)]) =

([fromList [(“z”, I 3)] ], fromList [("g",I 8)])

1. Функція *applyOp op v1 v2*, котра застосовує оператор *op* до аргументів *v1* і  *v2.* Для *op =* *Index* (індексування масиву) результат повинен бути нуль (представляється I 0), якщо немає зв’язування для індексу *v2* в масиві *v1*. Наприклад.

applyOp Add (I 6) (I (-2)) = I 4

applyOp Mul (I 3) (I 4) = I 12

applyOp Less (I 7) (I 0) = I 0

applyOp Equal (I 2) (I 2) = I 1

applyOp Index (A [(1,1),(0,3)]) (I 0) = I 3

applyOp Index (A [(1,1),(0,3)]) (I 2) = I 0

1. Функція *bindArgs* *idx vx*, котра бере список ідентифікаторів змінних *idx* і список відповідних значень *vx* і генерувати запис активації (Binding), в якому кожна змінна зв’язується з відповідним значенням. Наприклад:

bindArgs ["x", "a"] [I 6, A [(1,1),(0,3)]] =

fromList [("a", A [(1,1),(0,3)]), ("x", I 6)]

Передумова: обидва списки мають однакову довжину.

1. Взаємно рекурсивні функції *eval* і *evalArgst* для обчислення виразів. Функція *eval e dfx st* обчислює значення вираз *e* для списку функцій визначених користувачем *dfx* і стану *st.*  *evalArgs ex dfx st* застосовує функцію *eval* до кожного елементу списку виразів *ex*, повертаючи список значень. Функція *eval e dfx st* вираховує значення за правилами:
   * Значення константи (*Const c*) є просто c.
   * Значення змінної отримується з стану *st* (використовуючи *getValue*).
   * Значення умовного виразу - значення одного з підвиразів, в залежності від значення предикату, значення якого є одне з цілих 0 (false) або 1 (true).
   * Значення бінарної операції – результат *applyOp* на обчислених аргументах.
   * Щоб обчислити застосування функції *f* до списку виразів *es* потрібно
     + Знайти означення *f* в списку *dfx*
     + Виділити з означення список імен аргументів *as* і вираз *ef* - тіло функції.
     + Використовуючи *evalArgs*, обчислити кожний вираз в *es*, отримуючи список значень *vs*.
     + Створити локальний запис активації, Зв’язавши імена з *as* зі значеннями з *vs,* використовуючи *binArgs* .
     + Обрахувати *ef* в стані *st*, в котрий додається новий локальний запис активації, що створений *binArgs*

Наприклад:

eval (Const (I 1)) [] sampleState = I 1

eval (Var "y") [] sampleState = I 2

eval (OpApp Add (Var "x") (Const (I 2))) [] sampleState = I 7

eval (Cond (Const (I 1)) (Var "x") (Const (I 9))) [] sampleState = I 5

eval (FunApp "fib" [Const (I 6)]) [fib] sampleState = I 8

FunApp f es dfx st

1. Взаємно рекурсивні функції *executeStatment s dfx dpx st* і *executeBlock bl dfx dpx st,* котрі виконують один оператор *s* і блок (список операторів) *bl*. Результат виконання – це перетворення вхідного стану у вихідний стан. Параметри функцій – список функцій визначених користувачем *dfx,* список процедур визначених користувачем *dpx* і вхідний стан *st.* Особливості виконання:
   * Якщо виконується оператор присвоювання (Assign) і немає в стані змінної, котрій присвоюється значення, то необхідно додати нову локальну змінну в стан (Використання *updateVar* робить це автоматично).
   * Присвоєння окремому елементу масиву (AssignA) припускає, що масив, котрий змінюється, є в області дії (Можна використати *updateVar* і *assignArray*).
   * При реалізації виклику процедури потрібно передати правильний стан в процедуру, що викликається, і повернути вірний стан вкінці.

Наприклад, використовуючи приклади і тестові функції з допоміжного файлу:

executeStatement (Call id1 id2 arrExp) dfx dpx st

executeBlock [Call "s" "sumA1" [sampleArray, intToExp 3]] [] [sumA1] ([], M.empty) = ([fromList [(“s”, I 22)]], fromlist [])

executeBlock [Call "s" "sumA1" [sampleArray, intToExp 3]] [] [sumA1] ([], M.fromList [(“s”, I 0)]) = ([], fromList [("s",I 22)])

program pr1 = ([], fromList [("gSum", I 15)])

getValue :: Id -> StateP -> Value

getLocals, getGlobal s :: StateP -> Binding

assignArray :: Value -> Value -> Value -> Value

updateVar :: (Id,Value) -> StateP -> StateP

applyOp :: Op -> Value -> Value -> Value

bindArgs :: [Id] -> [Value] -> Binding

eval :: Exp -> [FunDef] -> StateP -> Value

evalArgs :: [Expr] -> [FunDef] ->StateP -> [Value]

executeStatment :: Statement -> [FunDef] -> [ProcDef] -> StateP -> StateP

executeBlock :: Block -> [FunDef] -> [ProcDef] -> StateP -> StateP

Зауваження:

Назва файлу Family06.hs (Family – прізвище студента). Файл включає модуль Family06 і створюється на основі файла-заготовки HWP06.hs