Implementacja pracy *Linear types can change the world!*

Jarosław Kwiecień

1 Wprowadzenie

W ramach projektu wykonałem interpreter języka zaproponowanego w pracy *Linear types can change the world!*, rozszerzonego o podstawowy mechanizm polimorfizmu. Głównym elementem pracy są liniowe typy, które służą do reprezentacji mutowalnych obiektów oraz wejścia/wyjścia. Ich najważniejszą cechą jest to, że muszą zostać użyte dokładnie raz, czyli nie można ich zduplikować (stworzyć dodatkowych referencji), ani zapomnieć, co w innych językach często skutkuje trudnymi do wykrycia błędami oraz wyciekami pamięci. Dodatkowo wprowadzona została specjalna konstrukcja let pozwalająca przy odpowiednich założeniach użyć zmiennych liniowych wielokrotnie w trybie *tylko do odczytu*.

2 Przykład

Przykład definicji funkcji map działającej dla tablic:

```
let map = fun <?a> (f : !a -> !a) ->
  fix (
    fun (rec : int -> ![|!a|] -> ![|!a|]) ->
      fun (i : int) ->
      fun (arr : ![|!a|]) ->
        let {arr} n = len arr in
        if i == n then arr else
          update f i arr |> rec (i + 1)
  ) 0
in map (fun (x : int) -> x * x) [| 1, 2, 3, 4 |]
```

3 Składnia (nieformalny opis)

1. Typy
   (a) typ prosty (zwykły lub liniowy): `x` lub `!x`

(b) krotka: `(type_1, type_2, ..., type_n)`

(c) lista: `[type]`

(d) tablica (pierwszy wariant tylko do odczytu, drugi liniowy): `[|type|]` lub `![|type|]`

(e) funkcja (funkcja liniowa): `type_1 -> type_2 | type_1 -o type_2`

2. Definicja typu

(a) nieliniowy: `type x = C1 type_1 | C2 type_2 | C3 | C4 | ... | Cn type_n` , gdzie `type_1, ..., type_n` są nieliniowe

(b) liniowy: `type !x = ...`

3. Wzorzec

- _, `(pattern_1, ..., pattern_n)`, `pattern_1 :: pattern_2`, `Constructor pattern` lub `!Constructor pattern`

4. Wyrażenie

(a) liczby naturalne: `12345`

(b) znaki: `'x'`

(c) stringi: `"text"` - zamieniane na listę `['t', 'e', 'x', 't']`

(d) zmienna: `x`

(e) lista: `[e1, e2, ..., en]`

(f) tablica: `[|e1, e2, ..., en|]`

(g) krotka: `(e1, e2, ..., en)`

(h) aplikacja: `e1 e2`

(i) wyrażenia z operatorami: `e1` *op* `e2`

- dostępne operatory: `+, -, *, /, >, <, >=, <=, ==, !=, &&, ||, ::, ++, ;, |>`

(j) `if cond then e1 else e2`

(k) `case e of pattern_1 -> e1 | pattern_2 -> e2 | ... | pattern_n -> en`

(l) `let pattern = e in e1`

(m) Let z trybem tylko do odczytu: `let {v1, v2, ..., vk} pattern = e in e1`

(n) Funkcja (nieliniowa lub liniowa) `fun (pattern : type) -> e` lub `fun (pattern : type) -o e`

(o) Funkcja polimorficzna (o zmiennych typowych nieliniowych, liniowych lub uniwersalnych): `fun <a, !b, ?c> (pattern : type) -> e`. Liniowa wersja: `-o` zamiast `->`

5. Program: `[type_defs] [expr]` lub `[type_defs] use s1, ..., sn in [expr]`

4 Wbudowane typy i zmienne

1. Typy proste: `char, int, void, !stdin, !stdout`
2. `type bool = True | False`
3. `type line_opt = Line [char] | EOF`
4. `fix : <?a> (!a -> !a) -> !a`
5. `len : <?a> [|!a|] -> int`
6. `arr_from_elem : <a> int -> a -> ![|a|]`
7. `arr_from_list : <?a> [!a] -> ![|!a|]`
8. `lookup : <a> int -> [|a|] -> a`
9. `update : <?a> (!a -> !a) -> int -> ![|!a|] -> ![|!a|]`
10. `drop : <?a> ![|!a|] -> ()`
11. `print : [char] -> !stdout -> !stdout`
12. `read_line : !stdin -> (line_opt, !stdin)`
13. `int_of_string : [char] -> int`
14. `string_of_int : int -> [char]`

5 Polimorfizm

W języku zaimplementowany został podstawowy polimorfizm - przede wszystkim funkcje polimorficzne, w których zmienne typowe wprowadzane są bezpośrednio przed określeniem argumentu funkcji. Są trzy rodzaje zmiennych typowych:

- nieliniowe - unifikują się tylko z nieliniowymi typami
- liniowe - unifikują się z typami liniowymi lub nieliniowymi w zależności od definicji typu argumentu funkcji (na przykład w `fun <!a> (x : (!a, a)) -> ...` pierwszy element argumentu musi być typem liniowym, a drugi nieliniową wersją tego samego typu)

- uniwersalne - unifikują się z typami liniowymi lub nieliniowymi (ale konsekwentnie - nie z obiema wersjami jednocześnie). Wewnątrz funkcji traktowane są jak typy liniowe (dozwolone jest `fun <?a> (x : !a) -> ...` ale już nie `fun <?a> (x : a) -> ...`).

Zmiennym typowym przy wprowadzeniu przyznawane są świeże identyfikatory, a następnie wystąpienia tych zmiennych w funkcji są zamieniane na te identyfikatory (traktowane tak jak pozostałe typy).

Podczas aplikacji oraz wyrażeń operatorowych wyliczane jest najbardziej ogólne podstawienie unifikujące odpowiadające sobie typy, a następnie na jego podstawie ustalany jest typ wyniku.

Warto zauważyć, że nie tylko funkcje mogą mieć typy polimorficzne, na przykład lista pusta ma typ: `<?a> [!a]`

6 Wejście/wyjście

Język dostarcza dwie globalne zmienne nieliniowe: `stdin : !stdin` i `stdout : !stdout`. Aby ich użyć, trzeba to najpierw zadeklarować w konstrukcji `use ... in` przed głównym wyrażeniem programu. Wymienione zmienne wprowadzane są wtedy do środowiska jako zmienne liniowe. Warto zauważyć, że nie ma możliwości "eliminacji" tych zmiennych - muszą one zostać zwrócone w wyniku programu.

7 Korzyści z użycia liniowych typów

Dzięki liniowym typom możemy w bezpieczny sposób reprezentować mutowalne obiekty (tablice, wejście/wyjście) jako zmienne: ponieważ niedozwolone jest dwukrotne użycie tego samego obiektu, wszystkie operacje są "czyste" - nie powodują efektów ubocznych, co miałoby miejsce na przykład w takim programie:

```
let arr1 = [| 1, 2, 3 |] in
let arr2 = arr1 in
(update (fun (x : int) -> x + 1) 0 arr1, lookup 0 arr2)
```

Gdyby ten program się typował, obliczenie pierwszego elementu pary wpływałoby na wartość drugiego elementu.

Liniowe typy gwarantują też sekwencjonowanie tych operacji, które muszą być wykonane w odpowiedniej kolejności - dzięki temu można zastosować leniwą ewaluację (choć są z nią pewne problemy, o czym poniżej).

8 Problemy z ewaluacją leniwą

Leniwa ewaluacja przyjęta w tym języku prowadzi czasem do nieintuicyjnych zachowań programu, co ilustruje następujący przykład:

```
use stdin, stdout in
let loop = fix (fun (rec : (!stdin, !stdout) -> (!stdin, !stdout)) ->
    fun ((stdin, stdout) : (!stdin, !stdout)) ->
      let (line_opt, stdin) = read_line stdin in
      case line_opt of
        | Line line ->
            let stdout = stdout |> print line |> print "\n" in
            rec (stdin, stdout)
        | EOF -> (stdin, stdout))
  in
loop (stdin, stdout)
```

Choć sugeruje on, że program będzie naprzemiennie wykonywał sekwencję *wczytaj-wypisz*, wypisywanie jest leniwie odkładane na koniec działania programu. Aby zsynchronizować wejście i wyjście, wykorzystamy konstrukcję `let {} ...`, która wymusza gorliwą ewaluację:

```
type !io = IO (!stdin, !stdout)

let io_print =
  fun (x : [char]) ->
  fun (!IO (stdin, stdout) : !io) ->
    let {} stdout = print x stdout in
    !IO (stdin, stdout)
  in
let io_read_line =
  fun (!IO (stdin, stdout) : !io) ->
    let {} (line_opt, stdin) = read_line stdin in
    (line_opt, !IO (stdin, stdout))
  in
```

...

Tak zdefiniowane funkcje zapewniają synchronizację wejścia/wyjścia, a jednocześnie nie "psują" leniwości języka - same wywołania funkcji `io_print, io_read_line` wciąż wykonywane są leniwie!

9 Testy

W katalogu `tests` znajdują się następujące testy:

- `example.ll` - kod przykładu z tego raportu
- `lists.ll` - dwie proste operacje na listach - zdublowanie elementów i funkcja map
- `array.ll` - kilka ciekawych operacji na listach i tablicach, m. in. fold, range (zwraca listę liczb z danego zakresu) i suma dwóch tablic po współrzędnych
- `lazy.ll` - sprawdza kilka przypadków, gdzie leniwa ewaluacja pozwala uniknąć błędu wykonania
- `eager.ll` - pokazuje, że wyrażenie w konstrukcji `let {}` jest ewaluowane gorliwie
- `echo1.ll, echo2.ll` - wczytują linia po linii i wypisują je na wyjście - w wersji leniwej i zsynchronizowanej
- `linearity1.ll` - błąd typecheckera z powodu przypisania liniowej zmiennej do _ (*wildcard*)
- `linearity2.ll` - błąd typecheckera z powodu niekonsekwentnego użycia liniowej zmiennej
- `linearity3.ll` - błąd typecheckera z powodu wykorzystania liniowej zmiennej w nieliniowej funkcji
- `linearity4.ll` - poprawiona wersja poprzedniego testu, bez błędu
- `linearity5.ll` - błąd typecheckera z powodu dwukrotnego wywołania liniowej funkcji

10 Uruchomienie programu

Aby skompilować program potrzebny jest OCaml w wersji przynajmniej `4.07` oraz dodatkowe zależności `ocamlbuild, ocamlfind, menhir`. Program można zbudować poleceniem `make`. Aby uruchomić program należy wpisać `./main.native ścieżka_do_pliku_źródłowego`, np. `./main.native tests/array.ll`. Można też nie podać argumentu i uruchomić program w trybie interaktywnym - wczytuje on program aż do wystąpienia pustej linii a następnie uruchamia go i wypisuje typ oraz wynik.