Lista zadań nr 14

Zadanie 1. (2 pkt)

Dodaj do języka WHILE operatory preinkrementacji (`++x`), postinkrementacji (`x++`), predekrementacji (`--x`) i postdekrementacji (`x--`) z następującą semantyką:

- `x++` zwiększa wartość zmiennej `x` o 1 i zwraca oryginalną wartość,
- `++x` zwiększa wartość zmiennej `x` o 1 i zwraca nową wartość,
- `x--` zmniejsza wartość zmiennej `x` o 1 i zwraca oryginalną wartość,
- `--x` zmniejsza wartość zmiennej `x` o 1 i zwraca nową wartość.

Jaka jest semantyka wyrażenia

```
++x++ + ++x++
```

w Twoim interpreterze? Dlaczego?

Zadanie 2. (3 pkt)

Dodaj do języka WHILE konstrukcję

```
local x = e;
```

gdzie `x` to nazwa zmiennej, a `y` to wyrażenie. Konstrukcja ta wprowadza *zmienną lokalną*, której zasięgiem jest blok, w którym została ona zdefiniowana (od miejsca definicji do końca bloku). Np. program

```
a = 1;
if (true) {
  print(a);
  local a = 20;
  print(a);
}
print(a);
```

powinien wypisać na ekran

```
1
20
1
```

Zmienne lokalne przykrywają zmienne globalne oraz przykrywają się wzajemnie.

Wskazówka: Środowiska.

Zadanie 3. (2 pkt)

W pliku `spigot.while` znajdującym się w katalogu `examples` interpretera języka WHILE znajdą Państwo następujący program:

```
q := 1;
r := 0;
t := 1;
k := 1;
n := 3;
l := 3;

while (true) {
  if (4 * q + r - t < n * t) {
    print(n);
    qq := 10 * q;
    rr := 10 * (r - n * t);
    tt := t;
    lk := k;
    nn := ((10 * (3 * q + r)) / t) - (10 * n);
    ll := l;
  } else {
    qq := q * k;
    rr := (2 * q + r) * l;
    tt := t * l;
    kk := k + 1;
    nn := (q * (7 * k + 2) + r * l) / (t * l);
    ll := l + 2;
  }
  q := qq;
  r := rr;
  t := tt;
  k := kk;
  n := nn;
  l := ll;
}
```

Program ten wypisuje nieskończony ciąg bezsensownych liczb. Są jednak one bezsensowne tylko dlatego, że zmienne występujące w programie szybko za-

czynają się przepełniać – maszynowe inty (a tym bardziej ocamlowe inty[1]) są dla nich po prostu za małe. Użyjmy więc większych.

Zmodyfikuj interpreter jezyka WHILE tak, by zamiast ze stadardowych intów, korzystał on z liczb typu `Bigint.t` udostępnianych przez bibliotekę `bignum`[2]. Czy teraz rozpoznajesz ciąg liczb wypisywany na ekran przez powyższy program?

Uwaga: Bibliotekę `bignum` można zainstalwoać poleceniem

```
opam install bignum
```

By nasz progrm widział bibliotekę, należy dołączyć ją do listy zależności projektu, modyfikując plik `src/dune` tak, by wyglądał następująco:

```
(library
  (name while)
  (libraries bignum))

(menhir (modules parser))

(ocamllex lexer)
```

Wskazówka: Można zmniejszyć liczbę modyfikacji w kodzie używając składni w OCamlu

```
Moj_modul.(wyrazenie)
```

która sprawia, że lokalnie otwieramy moduł `Moj_modul` w wyrażeniu `wyrazenie`.

Zadanie 4. (2 pkt)

Udowodnij poprawność algorytmu szybkiego potęgowania używając logiki Hoare'a:

$\{\mathtt{x} = x \land \mathtt{n} = n \land n \geq 0\}$

```
z := 1;
while (n > 0) {
  if (n % 2 == 1)
    z := z * x;
  n := n / 2;
  x := x * x;
}
```

$\{\mathtt{z} = x^n\}$

[1] https://blog.janestreet.com/what-is-gained-and-lost-with-63-bit-integers/
[2] https://ocaml.org/p/bignum/latest/doc/Bigint/index.html

Zadanie 5. (3 pkt)

Pokaż poprawność tzw. dużej rutyny Turinga używając logiki Hoare'a:

$\{\mathtt{n} = n \land n \geq 0\}$

```
r := 0;
u := 1
while (r < n) {
  s := 1;
  v := u;
  while (s <= r) {
    u := u + v;
    s := s + 1;
  }
  r := r + 1;
}
```

$\{\mathtt{u} = n!\}$