MP20 @ II UWr 12 maja 2020 r.

Lista zagadnień nr 10

Ćwiczenie 1.

W notatkach do wykładu nr 10 napisane jest, że chcemy prześcignąć Racketa, ale nie pod względem prędkości ewaluacji programów. Zbadaj na kilku przykładach, jak dużo wolniejsze jest wykonanie programów w naszym języku w naszym interpreterze w porównaniu z odpowiadającymi programami w Rackecie. Ponieważ uruchamianie programów w DrRackecie nie jest miarodajne (nie są optymalizowane, a dodatkowo spowalniane są informacjami potrzebymi debuggerowi), lepiej programy racketowe i nasz interpreter najpierw skompilować przy użyciu narzędzia `raco`, np.

```
raco exe prog.rkt
```

skompiluje program w pliku `prog.rkt` do pliku wykonywalnego.

Możesz też porównać programy napisane w C do równoważnych programów w języku WHILE, ale też nie spodziewajmy się, że nasz interpreter będzie dużo szybszy.

Ćwiczenie 2.

Na wykładzie definiowaliśmy sobie różne rodzaje let-wyrażeń (nazwane dość ogólnikowo `bind`) mające formę procedur typu

```
(define (my-let c k) (k c))
```

Powyższa procedura definiuje zwykłe let-wyrażenie, ale używa się go tak:

```
(my-let e1 (lambda (x) e2))
```

Można się do takiej formy próbować przyzwyczaić, ale można też użyć faktu, że Racket pozwala definiować a potem używać własne fragmenty składni. Zadanie polega na doczytaniu, jak to zrobić, żeby używać naszych bind-ów w składni bardziej let-podobnej.

Można doczytać w Racket Reference, rozdział 1.2 (`https://docs.racket-lang.org/reference/syntax-model.html`). Jest to dość długi i skomplikowany dokument, ale wystarczy rzucić okiem na przykłady w sekcji 1.2.3.5 („Transformer

Bindings"), żeby umieć rozwiązać to zadanie w najprostszej wersji, w której chceilibyśmy móc napisać np.

```
(let-error [x e1] e2)
```

co miałoby oznaczać to, co poprzednio zapisywaliśmy jako

```
(let-error e1 (lambda (x) e2))
```

W wersji bardziej zaawansowanej, chcielibyśmy mieć składnię bardziej w stylu `let*`, np. móc napisać fragment interpretera w poniższy sposób:

```
(bind ([v1 (eval-env e1 env)]
       [v2 (eval-env e2 env)])
  ((op->proc op) v1 v2))
```

Ćwiczenie 3.

Rozbuduj język z pliku `error-composition.rkt` o dwuargumentową formę specjalną `handle` taką, że semantyka wyrażenia

```
(handle e1 e2)
```

jest następująca:

1. Oblicz wartość wyrażenia `e1`
2. Jeśli `e1` ma wartość, jest to wartość całego wyrażenia `(handle ...)`. Jeśli ewaluacja `e1` skoczyła się błędem, wartość całego wyrażenia `(handle ...)` to wartość wyrażenia `e2`.

Ćwiczenie 4.

Rozbuduj interpreter z niedeterminizmem (w wersji `composition` albo `monadic`) o procedurę wbudowaną `fail`. Wynikiem ewaluacji tej procedury jest brak wyników, czyli lista pusta.

Uwaga: Procedura `fail` nie potrzebuje żadnego argumentu, ale nie mamy bezargumentowych procedur. Zrób więc ją procedurą jednoargumentową, która ignoruje wartość swojego argumentu. Podobne rozwiązanie zastosowaliśmy przy okazji procedury `get` w notatkach do wykładu.

Ćwiczenie 5.

Zmodyfikuj procedurę wbudowaną `choose` w języku z niedeterminizmem, by brała jeden argument będący listą możliwych wartości. Wówczas `fail` można zdefiniować jako `(choose null)`.

Ćwiczenie 6.

Napisz w naszym języku z `choose` i `fail` program rozwiązujący problem 8 hetmanów (a najlepiej n hetmanów dla dowolnego n).

Ćwiczenie 7.

Zmodyfikuj interpreter z pliku `state-monadic.rkt` tak, by można było używać dowolnej liczby komórek pamięci, do których dostęp mamy przez ich identyfikator (będący np. symbolem, o ile ktoś rozwiązał ćwiczenie 4. z listy 9.). Procedury wbudowane `put` i `get` powinny brać dodatkowy argument, który mówi do jakiej komórki się odnoszą.

Np. chcielibyśmy móc napisać program

```
(begin
  (put 'x 10)
  (put 'y 20)
  (+ (begin (put 'y 30) (get 'x))
     (get 'y)))
```

Wartością tego programu powinno być `40` (w tym przykładzie użyliśmy lukru syntaktycznego `begin` z ćwiczenia 7. z listy 9.).

Semantykę dostępu do komórki, której wartość nie była ustawiona ani w stanie początkowym ani w trakcie działania programu, pozostawiamy w gestii Rozwiązującego.