Języki i paradygmaty programowania (Info, III rok) 16/17

NAWIGACJA

Kokpit
- Strona główna

Strony

Moje kursy
- POWI.INFO.I.16/17
- ZPP.INFO.III.16/17
- JiPP.INFO.III.16/17
  - Uczestnicy
  - Odznaki
  - Kompetencje
  - Oceny
  - Główne składowe
  - T1 27.2-5.3 Haskell 1
  - T2 6-12.3 Haskell 2
  - T3 13-19.3 Monady 1
  - T4 20-26.3 Monady 2
  - T5 27.3-2.4 Składnia 1
  - T6 3.4-9.4 Składnia 2
  - T7 10.4-23.4 Typy + semantyka
  - T8 24-30.4
  - T9 1.5-14.5
  - T10 15-21.5
  - T11 22.5-28.5
  - 29.5-4.6
  - 5.6-11.6
  - Ostateczna wersja interpretera
  - Laboratoriu

Zadanie 4 - Smalltalk

Protokół GoDo

Powiemy, że obiekty `subject` i `object` realizują *protokół GoDo*, jeżeli umożliwiają wykonanie akcji `action` dla dopasowań `subject` do `object`.

W Smalltalku z implementacji GoDo korzystamy instrukcją

```
subject go: object do: action
```

gdzie `action` to bezparametrowy blok.

Termy

Rozważamy termy zbudowane ze *stałych* i *zmiennych* za pomocą anonimowego dwuargumentowego symbolu funkcyjnego tworzącego *parę*.

Parę termów `s` i `t` budujemy komunikatem `#,`

```
s, t
```

Pierwszy element pary `p` odczytujemy komunikatem `#car`

```
p car
```

a *drugi element* komunikatem `#cdr`

```
p cdr
```

Fabryka stałych `C` produkuje stałą o wartości `wartość` w odpowiedzi na komunikat `#%`

```
C % wartość
```

Fabryka zmiennych `V` produkuje zmienną o nazwie `nazwa` w odpowiedzi na komunikat `#@`

```
V @ nazwa
```

Produkowane przez `V` zmienne są jednoznacznie identyfikowane nazwą. Jeżeli `nazwa1 = nazwa2` to wynikiem obliczenia wyrażeń `V @ nazwa1` i `V @ nazwa2` jest ta sama zmienna.

Dla ułatwienia budowy termów, każdy z nich rozumie komunikaty `#%` i `#@`. Jeżeli `t` jest termem, to obliczenie

m: Smalltalk 2
Zadanie 4 - Smalltalk
12-14.6
Bonus

ADMINISTRACJA
Administracja kursem

```
t % wartość
```

jest równoważne

```
t, (C % wartość)
```

a obliczenie

```
t @ nazwa
```

jest równoważne

```
t, (V @ nazwa)
```

Dodatkowo, istnieje globalna zmienna `L` o wartości

```
C % nil
```

stworzona instrukcją

```
Smalltalk at: #L put: (C % nil)
```

Interpretacja termu

Interpretacje termu `t` wyznaczamy komunikatem `#value`

```
t value
```

Interpretacją stałej `C % wartość` jest `wartość`.

Interpretacją pary `s, t` jest para zbudowana z interpretacji termu `s` oraz interpretacji termu `t`.

Bezpośrednio po utworzeniu zmienna jest nieustalona i nie ma interpretacji. Wysłanie do niej komunikatu `#value` powoduje zgłoszenie błędu. To samo dotyczy też komunikatów `#car` i `#cdr`.

Zmienna ustalona, której wartością jest term `t`, na komunikaty `#value`, `#car` i `#cdr` reaguje, dając wynik przesłania tych komunikatów do `t`. Interpretacją zmiennej ustalonej jest więc interpretacja jej wartości, o ile taka istnieje.

Uzgadnianie

Próba *uzgodnienia* termów kończy się *sukcesem* lub *porażką*. W przypadku sukcesu efektem uzgodnienia jest nadanie wartości nieustalonym zmiennym przez zastosowanie do termów *najbardziej ogólnego podstawienia uzgadniającego*.

Uzgadnialność jest relacją symetryczną na termach.

Stała o wartości `wartość1` jest uzgadnialna ze stałą o wartości `wartość2` jeśli `wartość1 = wartość2`.

Uzgodnienie pary `a1, b1` z parą `a2, b2` wymaga uzgodnienia `a1` z `a2` i `b1` z `b2`.

Uzgodnienie zmiennej ustalonej z termem `t` wymaga uzgodnienia z `t` wartości tej zmiennej.

Zmienna nieustalona jest uzgadnialna z termem `t`, jeśli jest mu równa lub w nim nie występuje. Efektem uzgodnienia zmiennej nieustalonej z `t` jest nadanie tej zmiennej wartości `t`.

Polecenie pierwsze (4 pkt)

Zrealizuj protokół GoDo dla termów.

Instrukcja

```
s go: t do: action
```

powinna wykonać bezparametrowy blok `action` jeśli termy `s` i `t` są uzgadnialne.

Przed wykonaniem `action` na nieustalone zmienne obu termów należy przypisać wartości określone przez podstawienie uzgadniające. Po wykonaniu `action` przypisania te należy cofnąć, przywracając zmiennym stan nieustalony.

Przykład pierwszy

Wykonanie w oknie *Workspace* poniższego fragmentu kodu powinno zakończyć się poprawnie, bez naruszenia asercji.

```
t := L % 1, (V @ 2 % 3).
self assert: [t car car value isNil].
self assert: [t car cdr value = 1].
self assert: [t car value car isNil].
self assert: [t car value cdr = 1].
self assert: [t cdr cdr value = 3].

a := V @ 1.
b := V @ 2.
c := V @ 1.
self assert: [a ~= b].
self assert: [a == c].

t := C % 1 @ #z.
self assert: [t car value = 1].
w := 0.
[t value] on: Error do: [:ex | w := w + 1].
self assert: [w = 1].

C % 1 go: C % 2 do: [self assert: [false]].

w := 0.
C % 1 go: C % 1 do: [w := w + 1].
self assert: [w = 1].

x := V @ #x.
y := V @ #y.
w := 0.
x % 1 go: C % 2, y do:
    [w := w + 1.
    self assert: [x value = 2].
    self assert: [y value = 1]].
self assert: [w = 1].

w := 0.
C % 1 % 2 go: x do:
    [w := w + 1.
    self assert: [x value car = 1].
    self assert: [x cdr value = 2]].
self assert: [w = 1].

w := 0.
x go: C % $a do:
    [w := w + 1.
    x go: C % $b do: [self assert: [false]]].
x go: C % $b do: [w := w + 1].
self assert: [w = 2].

w := 0.
x % 1 go: y, y do:
    [w := w + 1.
    self assert: [x value = 1].
    self assert: [y value = 1]].
self assert: [w = 1].

x go: L, x do: [self assert: [false]].
```

Interpreter Prologu

Obiekt stworzony wyrażeniem `Prolog new` jest interpreterem *języka programowania w logice*, dalej nazywanego Prologiem.

W klauzulach Horna obsługiwanych przez interpreter, będących częścią programu lub zapytaniem, wolno używać tylko jednego, anonimowego, jednoargumentowego predykatu. Atomy są reprezentowane przez term będący argumentem tego predykatu.

Interpreter `p` reaguje na komunikat `#fact:`

```
p fact: a
```

dodając do programu klauzulę unarną z atomem `a`.

Rezultatem wysłania do interpretera `p` komunikatu `#head:body:`

```
p head: a body: b
```

jest dodanie do programu klauzuli z nagłówkiem `a` i treścią `b`.

Treść klauzuli programu, jak również treść klauzuli negatywnej, czyli zapytania, jest zbudowana z jednego lub więcej atomów połączonych koniunkcją `#&`

```
a & b
```

Polecenie drugie (4 pkt)

Zrealizuj protokół GoDo dla interpretera Prologu oraz termu.

Instrukcja

```
p go: b do: action
```

powinna wykonać bezparametrowy blok `action` dla każdego sukcesu zapytania `b` w programie `p`.

Przed każdym wykonaniem `action` na nieustalone zmienne zapytania `b` należy przypisać wartości określone przez odpowiedź obliczoną. Po wykonaniu `action` przypisania te należy cofnąć, przywracając zmiennym stan nieustalony.

Podczas SLD-rezolucji stosujemy prologową regułę wyboru pierwszego atomu zapytania. SLD-drzewo przechodzimy w głąb, w porządku zgodnym z kolejnością dodawania klauzul do programu.

Przykład drugi

Wykonanie w oknie *Workspace* poniższego fragmentu kodu powinno zakończyć się poprawnie, bez naruszenia asercji.

```
x := V @ #x.
t := #(1 2 3) asOrderedCollection.
p := Prolog new.
t do: [:each | p fact: C % each].
w := OrderedCollection new.
p go: x do: [w add: x value].
self assert: [w = t].
w := 0.
p go: (C % 1) & (C % 2) & (C % 3) do: [w := w + 1].
self assert: [w = 1].
w := 0.
p go: (C % 1) & ((C % 2) & (C % 3)) do: [w := w + 1].
self assert: [w = 1].

y := V @ #y.
p := Prolog new
    fact: C % 1 % $a;
    fact: C % 2 % $b;
    head: x, y % $c body: (x % $a) & (y % $b);
    yourself.
w := 0.
p go: x, y % $c do:
    [w := w + 1.
    self assert: [x value = 1].
    self assert: [y value = 2]].
self assert: [w = 1].

z := V @ #z.
p := Prolog new
    fact: x, (y, x) % #member;
    head: x, (y, z) % #member body: x, y % #member;
    yourself.
w := OrderedCollection new.
p go: x, (L % 1 % 2 % 3) % #member do: [w add: x value].
self assert: [w = #(3 2 1) asOrderedCollection].

w := OrderedCollection new.
m := L % 1 % 2 % 3 % 4.
n := L % 0 % 2 % 4 % 6.
p go: (x, m % #member) & (x, n % #member) do: [w add: x value].
self assert: [w = #(4 2) asOrderedCollection].

a := V @ #a.
b := V @ #b.
c := V @ #c.
p fact: L, x, x % #append.
p head: (a, x), b, (c, x) % #append body: a, b, c % #append.
w := OrderedCollection new.
p go: x, y, (L % $c % $b % $a) % #append do:
    [[:q |
    s := OrderedCollection new.
    p go: a, q % #member do: [s add: a value].
    w add: (String withAll: s)]
        value: x;
        value: y].
self assert: [w = #('' 'abc' 'a' 'bc' 'ab' 'c' 'abc' '') asOrderedCollection]
```

Uwagi

- Nie trzeba sprawdzać poprawności argumentów komunikatu.
- Rozwiązanie powinno mieć formę pakietu, czyli pliku `.pac`, implementacji *Dolphin Smalltalk 7*.
- W pakiecie, oprócz zgodnych z powyższym opisem definicji `C`, `V`, `L` i `Prolog`, mają być wszystkie potrzebne niestandardowe klasy i metody.

Status przesłanego zadania

Status przesłanego zadania	Nie próbowano
Stan oceniania	Nie ocenione
Termin oddania	czwartek, 22 czerwiec 2017, 12:00
Pozostały czas	8 dni 1 godz.
Ostatnio modyfikowane	-
Komentarz do przesłanego zadania	▸ Komentarze (0)

Dodaj swoją pracę

Dodaj lub edytuj swoje zadanie