MP21 @ II UWr 8 czerwca 2021 r.

Lista zagadnień nr 13

Przed zajęciami

Tematem bieżącego tygodnia jest implementacja systemów typów. Należy zapoznać się z kodem (`typecheck.rkt`) oraz notatkami z wykładu.

Implementacja systemów typów

Ćwiczenie 1.

Rozbuduj kod z wykładu (`typecheck.rkt`) o symbole. W tym celu rozszerz:

- składnię konkretną (w języku Plait `''x` jest lukrem składniowym do `'(quote x)`),
- typ składni abstrakcyjnej,
- typ wartości,
- środowisko startowe o nową funkcję wbudowaną – operator porównywania symboli `eq-symbol?`,
- ewaluator,
- typ typów,
- procedurę `'typecheck-env`,
- środowisko startowe typów o typ operatora `eq-symbol?`,
- algorytm unifikacji.

Ćwiczenie 2.

W języku Plait nie występują modyfikowalne pary (`mcons`, `mcar`, `mcdr`), występuje natomiast typ modyfikowalnych „pudełek" `Boxof` oraz:

- procedura `box` typu `('a -> (Boxof 'a))`, tworząca nowe pudełko,
- procedura `unbox` typu `((Boxof 'a) -> 'a)`, zwracająca zawartość pudełka,
- procedura `set-box!` typu `((Boxof 'a) 'a -> Void)`, zmieniająca zawartość pudełka.

Wzorując się na kodzie z wykładu 9 (`letrec.rkt`), wprowadź do kodu z wykładu (`typecheck.rkt`) typ modyfikowalnych środowisk:

```
(define-type-alias (MEnv 'a) (Listof (Symbol * (Boxof 'a))))
```

Zdefiniuj dla tego typu stałą `menv-empty` oraz procedury `menv-lookup`, `menv-add`, `menv-update!`.

Zmodyfikuj ewaluator w kodzie z wykładu, aby używał modyfikowalnych środowisk zamiast zwykłych środowisk (nowym typem `eval-env` będzie `(Expr (MEnv Value)-> Value)`. Typechecker powinien nadal używać poprzedniej, niemodyfikowalnej implementacji środowisk.

Ćwiczenie 3.

Wykorzystując modyfikacje z poprzedniego zadania, rozbuduj kod z wykładu o formę **`letrec`** wzorując się na kodzie z wykładu 9 (`letrec.rkt`).

- Rozbudowując ewaluator, należy dodać konstruktor wartości `(blackhole)`.
- Rozbudowując typechecker, należy wykorzystać następującą regułę typowania dla **`letrec`**:

$$\frac{\Gamma, x : \tau_1 \vdash \texttt{e1} : \tau_1 \qquad \Gamma, x : \tau_1 \vdash \texttt{e2} : \tau_2}{\Gamma \vdash \texttt{(letrec (x e1) e2)} : \tau_2}$$

  Zwróć uwagę, czym ta reguła różni się od reguły dla **`let`**: założenie typowe na temat zmiennej x jest wprowadzane dla obu podwyrażeń – bo w obu podwyrażeniach ta zmienna jest widoczna.

  Wskazówka: regułę tę najłatwiej zaimplementować przez wygenerowanie świeżej zmiennej typowej dla τ_1 i wygenerowanie równania łączącego tę zmienną z typem wyrażenia `e1`.

- Typ typów oraz algorytm unifikacji pozostanie bez zmian.

Ćwiczenie 4.

Rozbuduj kod z wykładu o pary. W tym celu rozszerz:

- typ wartości,
- środowisko startowe o funkcje wbudowane `pair`, `fst` i `snd`,
- typ typów – o konstruktor `pair-type`,
- środowisko startowe typów o typy tych funkcji:
  - `pair` : `(function2-type (var-type 't1) (var-type 't2) (pair-type (var-type 't1) (var-type 't2)))`,
  - `fst` : `(function-type (pair-type (var-type 't1) (var-type 't2)) (var-type 't1))`,
  - `snd` : `(function-type (pair-type (var-type 't1) (var-type 't2)) (var-type 't2))`.

Ćwiczenie 5.

Zauważ, że typowanie par wprowadzone w poprzednim ćwiczeniu ma poważne ograniczenie – mianowicie w ramach jednego programu typ lewego elementu par musi być zawsze taki sam; ten sam problem dotyczy typu prawego elementu par. Przykładowo, poniższy program jest odrzucany przez typechecker:

```
(let (x (pair 1 2)) (pair true x))
```

Problem ten można rozwiązać, rozszerzając system typów o *schematy typów*. Zdefiniuj następujący typ schematów typów:

```
(define-type TypeSchema
    (type-schema [vs : (Listof Symbol)] [t : Type]))
```

Następnie zdefiniuj procedurę `instantiate` typu `(TypeSchema -> Type)`, która oblicza instancję schematu typów. Obliczenie instancji schematu typów polega na podstawieniu w typie `type-schema-t` za każdą zmienną z listy `type-schema-vs` świeżej zmiennej typowej. Można wykorzystać procedury `gen-var` oraz `substitute`.

Zmodyfikuj następnie procedurę `typecheck-env`, aby środowisko typów, zamiast typu `(Env Type)`, było typu `(Env TypeSchema)`. W tym celu należy:

- wykorzystać procedurę `instantiate` w przypadku dla `var-expr`,
- w przypadku dla `fun` i `let-expr` dodawać do środowiska, zamiast typu, schemat typów z pustą listą `type-schema-vs`,
- zmienić środowisko startowe, aby zawierało schematy typów z pustą listą `type-schema-vs`.

Upewnij się, że zmodyfikowana procedura `typecheck-env` działa prawidłowo. Następnie zmodyfikuj schematy typów dla `pair`, `fst` i `snd`, aby zawierały listę `type-schema-vs` postaci `(list 't1 't2)`. Sprawdź, że zmieniony typechecker może teraz otypować przykład podany na początku treści zadania.

Zadania domowe

Zadanie 13

Zaimplementuj w języku Plait algorytm unifikacji dla typu wyrażeń składniowych `S-Exp`. Dla przypomnienia – wyrażenie składniowe może być albo:

- stałą liczbową – spełniać predykat `s-exp-number?`,
- stałą logiczną – spełniać predykat `s-exp-boolean?`,
- symbolem – spełniać predykat `s-exp-symbol?`,
- listą wyrazeń składniowych – spełniać predykat `s-exp-list?`.

Uznajmy, że:

- symbole pełnią rolę zmiennych,
- dwie stałe liczbowe unifikują się wtedy i tylko wtedy, gdy ich wartości są równe,
- dwie stałe logiczne unifikują się wtedy i tylko wtedy, gdy ich wartości są równe,
- dwie listy wyrażeń składniowych unifikują się wtedy i tylko wtedy, gdy są tej samej długości oraz unifikują się parami ich pierwsze elementy, drugie elementy i tak dalej,
- nic innego się nie unifikuje.

Użyj definicji:

```
(define-type-alias Subst (Listof (Symbol * S-Exp)))
(define-type-alias Eq (S-Exp * S-Exp))
(define-type-alias Eqs (Listof Eq))
```

Rozwiązanie powinno zawierać procedurę `unify` typu `(Eqs -> (Optionof Subst))`.

Przykładowe wywołania tej procedury i ich wyniki:

```
> (unify (list (pair ‘x ‘1)))
(some (list (values 'x ‘1)))
> (unify (list (pair ‘x ‘1) (pair ‘y ‘#t) (pair ‘(z q) ‘(x y))))
(some (list (values 'q ‘#t) (values 'z ‘1) (values 'y ‘#t) (values 'x ‘
    1)))
> (unify (list (pair ‘x ‘1) (pair ‘y ‘#t) (pair ‘(z z) ‘(x y))))
(none)
```

Zadanie 13B

W ramach wyrażeń składniowych Plaita (tworzonych notacją ‘exp) mogą pojawić się dwie konstrukcje, które mogą być interpretowane w szczególny sposób: *cytowania* i *kwazicytowania*.

Cytowanie jest reprezentowane za pomocą dwuelementowej listy, której pierwszym elementem jest symbol **quote**:

```
> (list->s-exp (list (symbol->s-exp 'quote) ‘x))
‘'x
```

Zacytowane wyrażenie składniowe nie powinno podlegać interpretacji.

Kwazicytowanie jest reprezentowane za pomocą dwuelementowej listy, której pierwszym elementem jest symbol **quasiquote**:

```
> (list->s-exp (list (symbol->s-exp 'quasiquote) ‘x))
‘‘x
```

Podobnie jak dla zwykłego cytowania, symbole pojawiające się pod kwazicytowaniem nie powinny być interpretowane. Kwazicytowanie tym jednak różni się od cytowania, że wewnątrz niego może pojawić się *odcytowanie* – reprezentowane za pomocą dwuelementowej listy, której pierwszym elementem jest symbol **unquote**:

```
> (list->s-exp (list (symbol->s-exp 'quasiquote) (list->s-exp (list
    (symbol->s-exp 'unquote) ‘x))))
‘‘,x
```

Odcytowanie pozwala w pewnym sensie „uciec” spod cytowania – umieścić w zacytowanym wyrażeniu wyrażenie podlegające interpretacji. Ilustruje to poniższy przykład w Plaicie:

```
> ‘((number->s-exp (+ 2 2)) ,(number->s-exp (+ 2 2)))
‘((number->s-exp (+ 2 2)) 4)
```

Wewnątrz kwazicytowania mogą pojawić się kolejne kwazicytowania. Aby wyrażenie pod wieloma kwazicytowaniami podlegało interpretacji, konieczne jest odcytowanie wszystkich poziomów kwazicytowania:

```
> ``(,(number->s-exp (+ 2 2)) ,,(number->s-exp (+ 2 2)))
``(,(number->s-exp (+ 2 2)) ,4)
```

Zadanie polega na zaimplementowaniu algorytmu unifikacji dla wyrażeń składniowych zawierających cytowania i kwazicytowania. Symbole występujące pod cytowaniami nie powinny podlegać podstawieniom, ani być traktowanym jako zmienne w algorytmie unifikacji. W przypadku kwazicytowań, wyłącznie symbole całkowicie odcytowane powinny być traktowane jako zmienne.

Algorytm wymaga, aby dla unifikowanych wyrażeń pamiętać, pod jaką liczbą kwazicytowań się one znajdują. W tym celu można reprezentować równania za pomocą następującego typu:

```
(define-type EqLvl
  (equation [left : S-Exp] [right : S-Exp] [lvl : Number]))
```

Algorytm rozbudowujemy w następujący sposób:

- symbole pełnią rolę zmiennych, pod warunkiem, że nie znajdują się pod kwazicytowaniami,
- dwa zacytowane wyrażenia składniowe unifikują się wtedy i tylko wtedy, gdy te wyrażenia są równe (w sensie predykatu `equal?`),
- dwa kwazi-zacytowane wyrażenia składniowe unifikują się wtedy i tylko wtedy, gdy te wyrażenia unifikują się z poziomem kwazicytowań zwiększonym o 1,
- dwa odcytowane wyrażenia składniowe unifikują się wtedy i tylko wtedy, gdy te wyrażenia unifikują się z poziomem kwazicytowań zmniejszonym o 1,
- dwa symbole pod kwazicytowaniami unifikują się wtedy i tylko wtedy, gdy są równe,
- dwie stałe liczbowe unifikują się wtedy i tylko wtedy, gdy ich wartości są równe,
- dwie stałe logiczne unifikują się wtedy i tylko wtedy, gdy ich wartości są równe,
- dwie listy wyrażeń składniowych (nie definiujących cytowań, kwazicytowań ani odcytowań) unifikują się wtedy i tylko wtedy, gdy są tej samej długości oraz unifikują się parami ich pierwsze elementy, drugie elementy i tak dalej,

- nic innego się nie unifikuje.

Rozwiązanie powinno zawierać procedurę `unify` typu `(Eqs -> (Optionof Subst))`. Przykładowe wywołania tej procedury i ich wyniki:

```
> (unify (list (pair `x ``'x)))
(some (list (values 'x ``'x)))
> (unify (list (pair ``'x ``'x)))
(some '())
> (unify (list (pair ``'x ``'y)))
(none)
> (unify (list (pair ``'x ```x)))
(none)
> (unify (list (pair ```(f ,x) ```(f ,1))))
(some (list (values 'x `1)))
> (unify (list (pair ```(f ,x) ```(g ,1))))
(none)
> (unify (list (pair ```(f ,x ,x) ```(f ,1 ,2))))
(none)
> (unify (list (pair `x ```,y)))
(some (list (values 'x ```,y)))
> (unify (list (pair `x ```,x)))
(none)
> (unify (list (pair `x `1) (pair `y ```(`x x ,x))))
(some (list (values 'y ```(`x x ,1)) (values 'x `1)))
```