Lista zagadnień nr 7

Zadania na ćwiczenia

Ćwiczenie 1.

Rozważ poniższe przykłady składni konkretnej używanej w matematyce. Dla każdego przykładu wymyśl składnię abstrakcyjną i rozszerz o nią składnię wyrażeń arytmetycznych ze zmiennymi i let-wyrażeniami opisaną w notatkach, tzn. zdefiniuj odpowiednie struktury i rozszerz predykat `expr?`:

- Potęgowanie: a^b
- Suma wartości dla kolejnych liczb naturalnych: $\sum_{i=n}^{m} f(i)$.
- Całka oznaczona: $\int_a^b f(x)dx$
- Minimum zbioru: $\min\{f(i) \mid i \in \mathbb{N}\}$

Dobrze przemyśl, które przykłady składni wiążą zmienne, co jest stałą, a co jest wyrażeniem. Czy zaproponowana składnia abstrakcyjna umie reprezentować wyrażenie $\int_{1+1}^{\infty} \frac{1}{2^x} dx$?

Ćwiczenie 2.

Zaproponuj składnię konkretną dla przykładów z poprzedniego zadania poprzez odpowiednie rozszerzenie procedury `parse`.

Ćwiczenie 3.

Na liście zagadnień nr 5 zajmowaliśmy się rachunkiem zdań. W kolejnych dwóch zadaniach zajmiemy się rachunkiem QBF (*quntified boolean formulae*), w którym rozważamy spójniki takie jak w rachunku zdań, ale wprowadzamy też dwie nowe konstrukcje:

- Jeśli φ jest formułą, to $\exists x.\varphi$ też jest formułą
- Jeśli φ jest formułą, to $\forall x.\varphi$ też jest formułą

W powyższych konstrukcjach x to skwantyfikowana zmienna *zdaniowa*, która może przyjąć wartość prawda lub fałsz. Jeśli w formule nie ma zmiennych wolnych, to formuła jest albo prawdziwa albo fałszywa. Np.

- $\exists x.\neg x$ jest prawdziwa (świadek: x przyjmuje wartość „fałsz")
- $\forall x.x \vee \neg x$ jest prawdziwa
- $\exists x.x \wedge \neg x$ jest nieprawdziwa

Zaproponuj składnię konkretną (razem z procedurą `parse`) i abstrakcyjną dla rachunku QBF.

Ćwiczenie 4.

Zdefiniuj procedurę `eval`, która powie nam czy dana formuła QBF jest prawdziwa czy fałszywa. To zadanie można rozwiązać na dwa sposoby: używając modelu podstawieniowego i modelu ze środowiskiem.

Ćwiczenie 5.

Zdefiniuj procedurę, która przemianowuje zmienne związane w wyrażeniu arytmetycznym z let-wyrażeniami tak, by nigdy nie nastąpiło przysłanianie jednej zmiennej przez drugą, np.

```
> (rename (let-expr 'x (const 3)
            (binop
              '+
              (var-expr 'x)
              (let-expr 'x (const 5) (var-expr 'x)))))
(let-expr 'x1 (const 3)
  (binop
    '+1
    (var-expr 'x1)
    (let-expr 'x2 (const 5) (var-expr 'x2)))))
```

A także:

```
> (rename (binop '+
                 (let-expr 'x (const 1) (var-expr 'x))
                 (let-expr 'x (const 1) (var-expr 'x))))
(binop '+
       (let-expr 'x (const 1) (var-expr 'x))
       (let-expr 'x (const 1) (var-expr 'x)))
```

(oczywiście nazwy zmiennych po przemianowaniu mogą być bardzo różne od oryginalnych).

- W wersji łatwiejszej: użyj racketowej procedury `gensym` do generowania nowych nazw zmiennych
- W wersji trudniejszej: zdefiniuj rozwiązanie czysto funkcyjnie (mogą przydać się procedury `number->string`, `string-append` i `string->symbol`). Dla ułatwienia możesz założyć, że w oryginalnym wyrażeni nie ma zmiennych wolnych.

Wskazówka: Użyj środowiska, by pamiętać, jakie są nowe nazwy zmiennych.

Ćwiczenie 6.

Jak powyżej, ale przemianuj zmienne tak, by *wszystkie* zmienne związane w wyrażeniu miały różne nazwy, np.

```
> (rename (binop '+
                 (let-expr 'x (const 1) (var-expr 'x))
                 (let-expr 'x (const 1) (var-expr 'x))))
(binop '+
       (let-expr 'x1 (const 1) (var-expr 'x1))
       (let-expr 'x2 (const 1) (var-expr 'x2)))
```

Czy rozumiesz różnicę między tym zadaniem a poprzednim?

Ćwiczenie 7.

Zaimplementuj optymalizację, która działa na wyrażeniach arytmetycznych z `let`-wyrażeniami usuwając z wyrażenia nieużywane `let`-wyrażenia. Na przykład w programie

```
(let-expr 'x (binop '+ (const 2) (const 2))
  (let-expr 'y (binop '* (const 3) (var-expr 'x))
    (binop '+ (const 7) (var-expr 'x))))
```

definicja `'y` nie jest używana i całość może być uproszczona do

```
(let-expr 'x (binop '+ (const 2) (const 2))
  (binop '+ (const 7) (var-expr 'x)))
```

Zastanów się, czy i kiedy w prawdziwym języku programowania taka optymalizacja jest poprawna.

Zadania domowe

Zadanie 12.

Składnia abstrakcyjna używana wewnątrz kompilatora lub interpretera często udekorowana jest dodatkowymi informacjami: pozycją w pliku źródłowym, która odpowiada danemu węzłowi, czy informacjami o typach wyinferowanych we wcześniejszych fazach. W pliku `ex-free-and-bound.rkt` rozbudowujemy składnię abstrakcyjną wyrażeń arytmetycznych z `let`-ami o pozycję nazwy zmiennej związanej w konstrukcji `let`:

```
(struct pos (file col line)     #:transparent)
(struct let-expr (loc id e1 e2) #:transparent)

(define (expr? e)
  (match e
    ...
    [(let-expr loc x e1 e2)
     (and (pos? loc) (symbol? x) (expr? e1) (expr? e2))]
    ... ))
```

Dodatkowo, w tym samym pliku zdefiniowana jest procedura `parse` która umie automatycznie ozdobić składnię abstrakcyjną odpowiednią informacją o pozycji wiązania w kodzie źródłowym. Procedura przyjmuje jako argument trochę bardziej zaawansowaną formę cytowania (uzyskiwaną dodając # przed '):

```
Welcome to DrRacket, version 7.2 [3m].
Language: racket, with debugging, memory limit: 128 MB.
> (parse #'(let [x 5] (* y x)))
(let-expr
 (pos '|interactions from an unsaved editor| 3 17)
 'x
 (const 5)
 (binop '* (const 3) (var-expr 'x)))
```

W pliku znajduje się też kolejna definicja składni abstrakcyjnej wyrażeń, która zawiera dodatkowe informacje przy wystąpieniach zmiennych: czy jest wolna, czy jest związana (i gdzie):

```
(struct var-free  (id)     #:transparent)
(struct var-bound (pos id) #:transparent)

(define (expr-annot? e)
  (match e
    ...
    ; bez klauzuli dla var-expr!
    [(var-free x)      (symbol? x)]
    [(var-bound loc x) (and (pos? loc) (symbol? x))]
```

```
... ))
```

Zdefiniuj procedurę `annotate-expression`, która rozwiązuje zadanie z kartkówki z 10 marca: Jeśli `e` to wyrażenie w składni `expr?`, to `(annotate-expression e)` jest wyrażeniem w składni `expr-annot?`, w której każda zmienna z oryginalnego wyrażenia jest oznaczona albo jako wolna (czyli reprezentowana jest jako węzeł `var-free`) albo jako związana (węzeł `var-bound`) z dodatkową informacją, gdzie znajduje się wystąpienie wiążące (czyli wartość pola `loc` z wyrażenia `let` wiążącego tą zmienną), np.

```
> (annotate-expression (parse #'(let [x 5] (* y x))))
(let-expr
 (pos '|interactions from an unsaved editor| 3 38)
 'x
 (const 5)
 (binop
  '*
  (var-free 'y)
  (var-bound (pos '|interactions from an unsaved editor| 3 38) 'x)))
```

Rozwiązanie powinno rozszerzać szablon zawarty w pliku `ex-free-and-bound.rkt`.

Wskazówka: Użyj środowiska, by pamiętać, gdzie dana zmienna jest związana.

Zadanie 13.

Racket, tak jak zdecydowana większość języków funkcyjnych, korzysta z automatycznego zarządzania pamięcią. Znaczy to, że możemy stworzyć strukturę w pamięci używając np. procedury `cons` i ta struktura żyje w pamięci, dopóki jest potrzebna (w praktyce: można do niej „dotrzeć" przez dereferencję łańcuszka wskaźników począwszy od aktualnego środowiska i/lub stosu). Raz na jakiś czas, gdy system zarządzania pamięcią uzna, że już nadszedł odpowiedni moment, uruchamiany jest odśmiecacz (ang. *garbage collector, GC*), który usuwa z pamięci wszystkie niepotrzebne już struktury.

Przykładowo, zakładając że Racket oblicza argumenty procedur od lewej do prawej strony, rozważmy wyrażenie:

```
(+ (let ([x (bardzo-dluga-lista)]) (length x))
   (bardzo-dlugie-obliczenie))
```

Gdy obliczymy już długość bardzo długiej listy i przejdziemy do wykonywania bardzo długiego obliczenia, zmienna `x` przestaje być w zasięgu i GC może skasować listę z pamięci.[1]

[1] Chyba że tworząc bardzo długą listę zakulisowo zrobiliśmy jakieś efekty uboczne (typu: zapisaliśmy wskaźnik do jej fragmentu w jakieś mutowalnej zmiennej) i lista wciąż jest żywa – to już zadanie dla GC określić, które komórki pamięci można zwolnić.

A teraz rozważmy taki przykład:

```
(let ([x (bardzo-dluga-lista)])
  (+ (length x)
     (bardzo-dlugie-obliczenie)))
```

Jeśli zmienna x **nie** występuje w definicji bardzo długiego obliczenia, w sumie GC mógłby już usunąć bardzo długą listę z pamięci, ale nie może tego zrobić, bo wciąż jesteśmy w zasięgu zmiennej x, więc jest ona w środowisku, więc GC musi uznać listę za żywą. Dlatego kompilator często robi optymalizację i oznacza sobie w składni abstrakcyjnej, że od pewnego momentu można uznać zmienną za martwą i pozwolić GC usunąć listę.

W tym zadaniu zrobimy sobie taką zabawkową optymalizację programów. Rozszerzamy składnię wyrażeń arytmetycznych z let-wyrażeniami o strukturę, która anotuje ostatnie wystąpienie danej zmiennej, zakładając gorliwą ewaluację let-wyrażeń i obliczanie argumentów operatorów arytmetycznych od lewej do prawej.

Nowa składnia i szablon rozwiązania dostępne są w pliku `ex-dead-variables.rkt`. Rozbudowujemy naszą podstawową składnię o nową strukturę:

```
(struct var-dead (id) #:transparent)

(define (expr? e)
  (match e
    ...
    [(var-dead x) (symbol? x)]
    ... ))
```

Zadanie polega na zdefiniowaniu procedury `find-dead-vars`, która przekształca wyrażenia (bez wystąpień anotacji `var-dead`), oznaczając ostatnie w kolejności obliczeń wystąpienie każdej zmiennej jako `var-dead`. Na przykład:

```
(find-dead-vars (let-expr 'x (const 3)
                  (binop '+ (var-expr 'x) (var-expr 'x))))
```

powinno dać rezultat

```
(let-expr 'x (const 3)
   (binop '+ (var-expr 'x) (var-dead 'x)))
```

Natomiast

```
(find-dead-vars (let-expr 'x (const 3)
                  (binop '+ (var-expr 'x)
                            (let-expr 'x (const 5)
                               (bonop '+ (var-expr 'x)
                                         (var-expr 'x))))
```

powinno dać rezultat

```
(let-expr 'x (const 3)
   (binop '+ (var-dead 'x)
              (let-expr 'x (const 5)
                 (bonop '+ (var-expr 'x)
                           (var-dead 'x)))
```

Proszę założyć, że wejście jest programem, to znaczy nie zawiera zmiennych wolnych.

Potrzebna definicja składni i szablon rozwiązania znajdują się w pliku `ex-dead-variables.rkt`.