Lista zadań nr 11

Zadanie 1. (1 pkt)

Uzupełnij implementację sprawdzania typów z wykładu o raportowanie błędów innych niż niezdefiniowana zmienna.

Zadanie 2. (2 pkt)

Zaimplementowane na wykładzie sprawdzanie typów przerywa pracę przy pierwszym napotkanym błędzie. Zmodyfikuj algorytm tak, by był w stanie zgłosić więcej niż jeden błąd.

Wskazówka: Jeżeli błąd wystąpi w funkcji `check_type`, to możemy go zgłosić, ale nie trzeba przerywać pracy algorytmu.

Zadanie 3. (2 pkt)

Rozszerz język z wykładu o wieloargumentowe funkcje rekurencyjne. Rozszerzenie to powinno zostać zaimplementowane jako lukier składniowy, tzn. cała praca powinna zostać wykonana w parserze, jeszcze przed sprawdzaniem typów. Np przykład, wyrażenie

```
funrec fact (n : int) (r : int) : int ->
  if n = 0 then r
  else fact (n - 1) (r * n)
```

powinno zostać potraktowane jako

```
funrec fact (n : int) : (int -> int) -> fun (r : int) ->
  if n = 0 then r
  else fact (n - 1) (r * n).
```

Zadanie 4. (2 pkt)

Implementacja języka z wykładu dwa razy przechodzi przez program otrzymany z parsera: raz sprawdzając typy, i drugi raz tłumacząc do reprezentacji używanej w ewaluatorze. Połącz te dwie operację w jedną: tzn. napisz funkcję która jednocześnie sprawdza/wyprowadza typy i tłumaczy program do innej reprezentacji.

Zadanie 5. (1 pkt)

Dojrzałe kompilatory wielokrotnie transformują programy (w reprezentacji pośredniej) zanim wyprodukują kod maszynowy. Takie transformacje często wymagają generowania świeżych zmiennych. Jeżeli zmienne reprezentujemy jako identyfikatory wprowadzone przez programistę, to generowanie świeżych unikatowych nazw jest problematyczne. Dlatego w reprezentacjach pośrednich często stosuje się inną reprezentację zmiennych (np. kolejne liczby naturalne).

Zmodyfikuj składnię abstrakcyjną używaną przez interpreter (`Ast`) tak by zmienne reprezentować przy pomocy typu `int`. Zmodyfikuj upraszczanie wyrażeń tak, by zamieniać reprezentację zmiennych z typu `string` na typ `int`. Możesz użyć mutowalnego stanu do generowania świeżych liczb. To, i następne zadanie jest łatwiej rozwiązać, bazując na rozwiązaniu poprzedniego zadania.

Zadanie 6. (3 pkt)

Czasami zachowanie pewnych konstrukcji języka zależy od typów, wyliczonych w fazie wyprowadzania typów. Wówczas wyprowadzanie typów powinno zostawić jakąś informację dla dalszych faz kompilatora/ewaluatora, lub — jak w zadaniu 4 — być formą translacji. By to zademonstrować, rozszerz język o nowy rodzaj funkcji (które będziemy nazywać *metodami*), które można przeciążać. Dodaj do języka następujące dwie konstrukcje.

- `method (`x `:` τ`).`n `=` e_1 `in` e_2 — definiuje metodę o nazwie n dla typu τ (n i x są dowolnymi identyfikatorami). Ciało metody (e_1) może odwoływać się do zmiennej x, która reprezentuje obiekt, dla którego metoda jest wywoływana. Zdefiniowana metoda może być używana w wyrażeniu e_2. Metody nie są rekurencyjne.

- $e.n$ — wywołuje metodę n na obiekcie e. Wybierana jest (leksykalnie najbliższa) metoda o nazwie n zdefiniowana dla typu wyrażenia e. Konstrukcja ta wiąże silniej niż aplikacja funkcji.

Wspomniane konstrukcje powinny zostać dodane do języka źródłowego (`RawAst`), ale w języku docelowym (`Ast`) powinny zostać zamienione na zwykłe funkcje. Na przykład program

```
method (self : int).toInt = self in
method (self : bool).toInt = if self then 1 else 0 in
true.toInt + 1.toInt
```

powinien zostać przetłumaczony na

```
let toInt1 = fun (self : int) -> self in
let toInt2 = fun (self : bool) -> if self then 1 else 0 in
toInt2 true + toInt1 1.
```

W tym zadaniu są zasadniczo dwie trudności. Po pierwsze dla metod o tej samej nazwie, ale różnych typach trzeba wygenerować różne identyfikatory. W tym celu pomocne okaże się zadanie 5. Po drugie, trzeba sobie odpowiedzieć na pytanie, jak wygląda środowisko sprawdzania typów. Będzie to para dwóch map. Dziedziną pierwszej, tak jak w kodzie z wykładu, powinny być zmienne lokalne. Dziedziną drugiej powinny być pary złożone z nazwy metody i typu obiektu.

Zadanie 7. (1 pkt)

Rozważmy rozszerzenie języka o typ $\tau_1 + \tau_2$ (w OCamlu nazywa się on `Either`), który ma dwa konstruktory `Left` i `Right`, a eliminujemy go przez płytkie dopasowanie wzoraca, np.

```
let swap = fun (x : int + bool) ->
  match x with
  | Left n -> Right n
  | Right b -> Left b
in
...
```

Nie musisz implementować tego rozszerzenia, ale zaproponuj reguły (relacji) typowania dla nowych konstrukcji języka. Jakiemu spójnikowi logicznemu odpowiada typ $\tau_1 + \tau_2$?