Ćwiczenia z Agdy – część druga

Wojciech Jedynak Piotr Polesiuk

7 stycznia 2014

1 Uwaga

Niniejszy dokument stanowi drugą część zadań z Agdy. Realizując wszystkie polecenia można uzyskać zaliczenie jednej listy seminaryjnej.

Rozwiązania niniejszych zadań przyjmujemy do **31 stycznia 2014 roku**. Rozwiązania w postaci pliku .agda należy wysyłać e-mailem na adres `wjedynak@gmail.com` lub `piotr.polesiuk@gmail.com`. Zachęcamy także do zadawania pytań!

2 Zadanie

Rozważmy następujący język λ_1:

```
infixr 20 _⇒_
infixl 25 _⊗_
data Tp : Set where
  nat : Tp
  _⇒_ : Tp → Tp → Tp
  _⊗_ : Tp → Tp → Tp
data Expr : Set where
  N    : ℕ → Expr
  if0  : Expr → Expr → Expr → Expr
  V    : Expr
  lam  : Tp → Expr → Expr
  _•_  : Expr → Expr → Expr
  [_,_] : Expr → Expr → Expr
  fst  : Expr → Expr
  snd  : Expr → Expr
```

Jest to typowy rachunek lambda z parami, liczbami naturalnymi i testem na zero. Dla uproszczenia używamy tylko jednej zmiennej (V), a każda abstrakcja wiąże ją na nowo przysłaniając starą wartość.

Polecenie 1 *Zdefiniuj relację typowania dla rachunku λ_1. Zwróć uwagę, żeby uwzględnić jednoelementowe środowisko. Relacja typowania powinna być parametryzowana typem początkowym dla zmiennej.*

Na dobry początek podajemy pierwszą regułę :-)

```
infix 10 _⊢_::_
data _⊢_::_ : Tp → Expr → Tp → Set where
  T-Nat : ∀ { T n} → T ⊢ N n :: nat
```

Polecenie 2 *Napisz ewaluator dla λ_1. Gdybyśmy pisali interpreter w Haskellu, to jego sygnatura mogłaby wyglądać następująco*

```
data Value = Nat Integer | Fun (Value → Value) | Pair Value Value
eval :: Expr → Maybe Value
```

W Agdzie możemy zrobić to lepiej! Po pierwsze interesuje nas tylko wykonywanie dobrze typowanych programów. Interpreter jako dodatkowy argument powinien brać dowód tego, że term jest dobrze typowany – dzięki temu pozbywamy się Maybe. Po drugie mając dowód typowania wiemy jaki dokładnie powinien być typ wyniku.

Ponieważ nasz rachunek można osadzić w systemie typów Agdy, napiszemy funkcję tłumaczącą typy z λ_1 do typów Agdy.

```
⟦_⟧ : Tp → Set
⟦ nat ⟧ = ℕ
⟦ T ⇒ S ⟧ = ⟦ T ⟧ → ⟦ S ⟧
⟦ T ⊗ S ⟧ = ⟦ T ⟧ × ⟦ S ⟧
```

Mając powyższą funkcję łatwo napisać sygnaturę funkcji eval:

```
eval : { T S : Tp} → (e : Expr) → T ⊢ e :: S → ⟦ T ⟧ → ⟦ S ⟧
eval e prf v₀ = ?
```

Polecenie 3 *Zaimplementuj typechecker dla λ_1. Konkretniej napisz dwie wzajemnie rekurencyjne funkcje:*

```
  -- Pomocniczy sigma-typ do inferencji
data WellTyped (T : Tp) (e : Expr) : Set where
  hasType : (S : Tp) → T ⊢ e :: S → WellTyped T e

mutual
  infer-type : ∀ T e → Maybe (WellTyped T e)
  infer-type T e = ?

  check-type : ∀ T e S → Maybe (T ⊢ e :: S)
  check-type T e S = ?
```

Postaraj się, żeby kod był czytelny. Pomocne mogą okazać się operatory monadyczne.