Programowanie 2009 Programming 2009

Lista zadań nr 15 Problem set no. 15

Na zajęcia 16–17 czerwca 2009 Due June 16–17, 2009

Grupy zasadnicze Basic groups

Zadanie 1 (1 pkt). Podaj typy następujących funkcji w Haskellu:

Problem 1 (1 p). Reveal the types of the following Haskell functions:

```
(f . g) x = f (g x)
f $ x = f x
curry f x y = f (x,y)
uncurry g (x,y) = g x y
flip f x y = f y x
const x _ = x
```

Zadanie 2 (1 pkt). Oto funkcje zapisane w Haskellu w notacji bezpunktowej:

Problem 2 (1 p). Here are some functions written in Haskell in the point-free notation:

```
dot = ((.).(.))
owl = ((.)$(.))
swing = flip . (. flip id)
```

Podaj ich typy i przepisz je w notacji punktowej.

Reveal their types and rewrite them in the pointwise form.

Zadanie 3 (2 pkt). Niech

Problem 3 (2 p). Let

```
data Tree = Node Tree Int Tree | Leaf
```

Zaprogramuj funkcję

Define a function

```
relabel :: Tree -> Tree
```

tworzącą kopię podanego drzewa, w której etykiety mniejsze niż liczba wierzchołków drzewa są podwojone, pozostałe zaś skopiowane bez zmian. *Uwaga:* wolno tylko raz przejść rekurencyjnie przez drzewo. *Wskazówka:* wykorzystaj leniwość.

creating a copy of a given tree in which labels less than the number of nodes of the tree are doubled, while the others are copied unchanged. *Attention:* you are allowed to traverse a tree only once. *Hint:* use laziness.

Zadanie 4 (2 pkt). Zdefiniuj w Prologu predykat `relabel/2`, który tworzy kopię podanego drzewa, zbudowanego za pomocą funktora `node/3` i atomu `leaf/0`, w której wszystkie etykiety są zastąpione liczbą wierzchołków drzewa. *Uwaga:* wolno tylko raz przejść rekurencyjnie przez drzewo. *Wskazówka:* wykorzystaj struktury otwarte. *Pytanie:* czy potrafisz rozwiązać w Prologu poprzednie zadanie?

Problem 4 (2 p). Define in Prolog a predicate `relabel/2` that makes a copy a given tree built with a functor `node/3` and an atom `leaf/0` in which all labels are replaced with the number of nodes of the tree. *Attention:* you are allowed to traverse a tree only once. *Hint:* use open structures. *Question:* can you solve in Prolog the previous problem?

Zadanie 5 (3 pkt). Lambda wyrażenia reprezentujemy w Prologu w postaci termów zbudowanych za pomocą funktorów `app/2` i `abs/2`. Na przykład term

Problem 5 (3 p). We represent lambda terms in Prolog as terms built with functors `app/2` and `abs/2`. For example a term

```
abs(X,abs(Y,app(Y,X)))
```

reprezentuje wyrażenie $\lambda xy.yx$. Typy reprezentujemy za pomocą zmiennych i funktora `fun/2`. Zaprogramuj w Prologu algorytm rekonstrukcji typów w rachunku lambda z typami prostymi. *Wskazówka:* przyda Ci się predykat `unify_with_occurs_check`.

represents the term $\lambda xy.yx$. We represent types using variables and the functor `fun/2`. Define i Prolog the algorithm for type reconstruction in the simply typed lambda calculus.

Grupy rozszerzone

Extended groups

Zadanie 1 (1 pkt). Pokaż, że jeśli w regule `let` systemu z polimorfizmem parametrycznym opuścimy założenie, że wolno generalizować jedynie zmienne, które nie mają wolnych wystąpień w kontekście typowym, to wyrażenie $1 + \texttt{True}$ posiada typ.

Problem 1 (1 p). Show that if in the `let` rule of the type system with parametric polymorphism we omit the assumption that it is allowed to generalize only variables that have no free occurrences in the type context, then the expression $1 + \texttt{True}$ is typable.

W kolejnych zadaniach rozważamy system typów drugiego rzędu w wersji Curry'ego:

In the next problems we consider the Curry-style second order type system:

$$\begin{aligned} t &::= x \mid t_1 t_2 \mid \lambda x.t \\ \sigma &::= \alpha \mid \sigma_1 \to \sigma_2 \mid \forall \alpha.\sigma \end{aligned}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash t : \forall \alpha.\sigma}\ \alpha \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash t : \forall \alpha.\sigma}{\Gamma \vdash t : \sigma[\alpha/\tau]}$$

Zadanie 2 (1 pkt). Pokaż, że w tym systemie wyrażenie $\lambda x.xx$ posiada typ, podczas gdy w języku z polimorfizmem parametrycznym — nie. Wskaż przykład termu, który nie posiada typu w systemie drugiego rzędu. Pokaż, że każdy term, który posiada typ w systemie typów z polimorfizmem parametrycznym posiada go też w systemie drugiego rzędu. Pokaż, że jeżeli do składni wyrażeń dodać `let`, to każdy `let`-term typowalny w systemie z polimorfizmem parametrycznym ma też typ w systemie drugiego rzędu, w którym typ wyrażenia `let` jest wyprowadzony na podstawie tłumaczenia Landina (zatem `let` jest jedynie cukrem syntaktycznym w monomorficznym systemie typów i w systemie drugiego rzedu, lecz ma kluczowe znaczenie dla polimorfizmu parametrycznego).

Problem 2 (1 p). Show that the term $\lambda x.xx$ is typable in this system, while in the system with parametric polymorphism it is not. Give an example of a term not typable in the second order system. Show that any term typable in the system with parametric polymorphism is typable in the second order system. Show that if we add the `let` construct to the syntax of terms, then every `let`-term typable in the system with parametric polymorphism is also typable in the second order type system with type for the `let` expression derived from the Landin translation (hence the `let` construct is merely a syntactic sugar in the monomorphic type system and in the second order system, but plays a crucial role in the system with parametric polymorphism).

Zadanie 3 (1 pkt). Zauważ, że w systemie drugiego rzedu istnieją dokładnie dwa zamknięte termy w postaci normalnej typu $\forall\alpha.\alpha \to \alpha \to \alpha$. Typ ten może więc służyć do zakodowania typu `bool`. Zdefiniuj wartości `true` i `false` oraz komplet operacji logicznych. Zauważ, że wszystkie dają się wyrazić za pomocą operatora `if` i że operator ten jest kodowany przez wyrażenie $\lambda x.x$ (to nie przypadek). Pamiętając o liczebnikach Churcha zauważ dalej, że $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ może reperezentować typ `nat`. Pokaż, że dowolny zamknięty term w postaci normalnej jest typu $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$, wtedy i tylko wtedy, gdy jest liczebnikiem Churcha. Zdefiniuj podstawowe operacje arytmetyczne. Zauważ, że wszystkie dają się wyrazić przy pomocy iteratora `iter`, takiego, że `iter` $n\, f\, c = f^n(c)$. Zauważ, że `iter` jest kodowany przez wyrażenie $\lambda x.x$ (to nie przypadek). Zauważ dalej, że typ $\texttt{list}(\beta) \equiv \forall\alpha.(\beta \to \alpha \to \alpha) \to (\alpha \to \alpha)$ koduje listy typu β. Zdefiniuj konstruktory i operacje działające na listach. Zauważ, że dają się one wyrazić przy pomocy iteratora `foldr` i że `foldr` jest kodowany przez wyrażenie $\lambda x.x$ (to nie przypadek). Wywnioskuj z tych przykładów, że system drugiego rzedu może służyć do zadania semantyki takich typów danych, jak listy, krotki, liczby itp. Zakoduj kilka innych typów danych (`unit`, `void`, krotki, drzewa binarne itp.).

Problem 3 (1 p). Notice that in the second order system there exist exactly two closed terms in normal form of type $\forall\alpha.\alpha \to \alpha \to \alpha$. Thus this type can serve as an encoding of the `bool` type. Define the values `true` and `false`, and a set of logical operations. Notice, that all of them can be expressed using the `if` operator and that this operator is encoded by the term $\lambda x.x$ (this is not a coincidence). Remembering Church numerals notice further, that $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ can represent the type `nat`. Show that any closed term in normal form has type $\forall\alpha.(\alpha \to \alpha) \to (\alpha \to \alpha)$ if and only if it is a Church numeral. Define basic arithmetic operations. Notice that all of them can be expressed using the iterator `iter`, such that `iter` $n\, f\, c = f^n(c)$. Notice that `iter` is encoded by the term $\lambda x.x$ (this is not a coincidence). Notice further that the type $\texttt{list}(\beta) \equiv \forall\alpha.(\beta \to \alpha \to \alpha) \to (\alpha \to \alpha)$ encodes lists of type β. Define list constructors and operations acting on lists. Notice that they can be expressed using the `foldr` iterator and that `foldr` is encoded by the term $\lambda x.x$ (this is not a coincidence). Infer from the examples above that the second order type system can be used to give sematics for such data types as lists, tuples, numbers, *etc.* Encode some other data types (`unit`, `void`, tuples, binary trees, *etc.*).

Zadanie 4 (1 pkt). Rozważmy system typów drugiego rzędu w stylu Churcha:

Problem 4 (1 p). Consider the Church-style second order type system:

$$\begin{aligned} t &::= x \mid t_1 t_2 \mid \lambda x : \sigma.t \mid t\langle\sigma\rangle \mid \Lambda\alpha.t \\ \sigma &::= \alpha \mid \sigma_1 \to \sigma_2 \mid \forall\alpha.\sigma \end{aligned}$$

$$\frac{}{\Gamma, x : \sigma \vdash x : \sigma} \qquad \frac{\Gamma \vdash t_1 : \sigma \to \tau \quad \Gamma \vdash t_2 : \sigma}{\Gamma \vdash t_1 t_2 : \tau} \qquad \frac{\Gamma, x : \sigma \vdash t : \tau}{\Gamma \vdash \lambda x.t : \sigma \to \tau}$$

$$\frac{\Gamma \vdash t : \sigma}{\Gamma \vdash \Lambda\alpha.t : \forall\alpha.\sigma}\ \alpha \notin \mathrm{FV}(\Gamma) \qquad \frac{\Gamma \vdash t : \forall\alpha.\sigma}{\Gamma \vdash t\langle\sigma\rangle : \sigma[\alpha/\tau]}$$

$$\begin{aligned} (\lambda x.t_1)t_2 &\to_\beta t_1[x/t_2] \\ \lambda x.tx &\to_\eta t, \qquad x \notin \mathrm{FV}(t) \\ (\Lambda\alpha.t)\langle\sigma\rangle &\to_\Lambda t[\alpha/\sigma] \end{aligned}$$

Wymyśl algorytm, który dla podanego kontekstu Γ i takiego termu t, że $\mathrm{FV}(t) \subseteq \mathrm{Dom}(\Gamma)$ znajduje taki typ σ, że $\Gamma \vdash t : \sigma$ lub informuje, że term t w tym kontekście typu nie posiada.

Devise an algorithm that for a given context Γ and such a term t that $\mathrm{FV}(t) \subseteq \mathrm{Dom}(\Gamma)$ finds such a type σ, that $\Gamma \vdash t : \sigma$ or informs that a term t in ths context has no type.

Zadanie 5 (1 pkt). W poprzednich zadaniach widzieliśmy, że kwantyfikator ogólny odpowiada typom polimorficznym. Nasuwa się pytanie, jaka jest interpretacja kwantyfikatora egzystencjalnego. Widzieliśmy już, że wszystkie zmienne występujące w typach argumentów konstruktorów muszą być wymienione wśród parametrów definiowanego typu, inaczej system staje się sprzeczny. Dlatego odrzucamy deklarację

Problem 5 (1 p). In the previous problems we saw that the universal quantifier corresponds to polymorphic types. The question arises what is the interpretation of the existential quantifier. We saw that all variables occuring in types of arguments of constructors have to be mentioned among parameters of the defined type, otherwise the system becomes unsound. This is why we reject the declaration

```
data T = C a
```

czyli

that is

```
data T where C :: a -> T
```

Jeśli jednak skwantyfikujemy uniwersalnie zmienną `a` w typie konstruktora `C`, to system będzie wolny od sprzeczności:

If however we quantify universally the variable `a` in the type of the `C` constructor, then the system is free from contradiction:

```
data T where C :: forall a. a -> T
```

Zauważmy, że formuły $\forall a.a \to T$ oraz $(\exists a.a) \to T$ są logicznie równoważne. Na mocy izomorfizmu Curry'ego-Howarda równoważne (w pewnym sensie) powinny być także odpowiadające im typy. Zatem $C : (\exists a.a) \to T$. Typ konstruktora C możemy przeczytać następująco: jeżeli istnieje typ a taki, że $x : a$, to $C\,x : T$. Skoro jednak o typie a nic nie wiadomo, to z wartością $C\,x$ nie da się nic zrobić. Jeżeli jednak wyrazimy specyfikację typów w postaci pewnej klasy S i zażądamy, by $a \in S$:

Notice that formulas $\forall a.a \to T$ and $(\exists a.a) \to T$ are logically equivalent. By the Curry-Howard isomorphism the types that corresond to them should be (somehow) equivalen as well. Thus $C : (\exists a.a) \to T$. We can read the type of the `C` constructor as follows: if there exists a type a, such that $x : a$, then $C\,x : T$. Since we no nothing about the type a, that we can do nothing with the value $C\,x$. If we however express a type specification in the form of some class S and postulate that $a \in S$:

```
data T where C :: forall a. S a => a -> T
```

to także $T \in S$. Teraz typ $C : (\exists a \in S.a) \to T$ oznacza, że jeśli istnieje taki typ a spełniający specyfikację S, że $x : a$, to $C\,x$ jest typu T. O wartościach typu T wiemy, że spełniają specyfikację S, ale nie wiemy, jak są zaimplementowane. Kwantyfikator egzystencjalny odowiada więc abstrakcji danych. Rozważmy dla przykładu specyfikację stosów (używamy rozszerzeń `GADTs`, `KindSignatures` i `Rank2Types` standardu Haskella):

than $T \in S$ as well. Now the type $C : (\exists a \in S.a) \to T$ means that if there exists such a type a that satisfies the specification S and $x : a$, then $C\,x$ has type T. About the values of type T we know that they satisfy the sepcification S, but we do not know how they are implemented. Thus the existstential quantifier corresponds to data abstraction. Consider for example the specification of stacks (we use the extensions `GADTs`, `KindSignatures` and `Rank2Types` of the Haskell standard):

```
class StackSpc (s :: * -> *) where
   push :: (a, s a) -> s a
   pop :: s a -> (a, s a)
```

Definiujemy teraz abstrakcyjną implementację stosów:

We now define the abstract implementation of stacks:

```
data StackAbs :: * -> * where
   StackAbs :: forall s. StackSpc s => s a -> StackAbs a

instance StackSpc StackAbs where
   push (x, StackAbs xs) = StackAbs (push (x,xs))
   pop (StackAbs s) = (x, StackAbs xs) where (x,xs) = pop s
```

Wybierzmy konkretną reprezentację stosów za pomocą list:

Let us choose the concrete representation of stacks using lists:

```
instance StackSpc [] where
   push (x,xs) = (x:xs)
   pop  (x:xs) = (x,xs)

empty = StackAbs []
```

Wartość `empty :: StackAbst` jest listą, ale jest abstrakcyjna w tym sensie, że wolno na niej wykonywać jedynie operacje `push` i `pop`:

The value `empty :: StackAbst` is a list, but is abstract in the sense that we can only apply the operation `push` and `pop` to it:

```
*Main> fst . pop . snd . pop $ push (2, push (1, empty))
1 :: Integer
```

ale nie mamy dostępu do jej reprezentacji:

but we have no access to its representation:

```
*Main> case empty of StackAbs x -> x
    Inferred type is less polymorphic than expected
      Quantified type variable ‘s’ escapes
    When checking an existential match that binds
        x :: s a
    The pattern(s) have type(s): StackAbs a
    The body has type: s a
    In a case alternative: StackAbs x -> x
    In the expression: case empty of StackAbs x -> x
```

Na wzór powyższego programu napisz specyfikację i abstrakcyjną implementację kolejek FIFO.

Following the program above write a specification and abstract implementation of the FIFO queues.

Zadanie 6 (2 pkt). Listy heterogeniczne, to listy, które mogą mieć elementy różnych typów. System typów Haskella nie pozwala utworzyć listy `[True, 1]`. Heterogeniczność jest dualna do polimorfizmu (listy Haskellowe są polimorficzne) i można ją, podobnie jak abstrakcję w poprzednim zadaniu, osiągnąć za pomocą polimorfizmu — parametrycznego:

Problem 6 (2 p). Heterogenic lists are lists that can contain elements of different types. The Haskel type system does not allow to create the list `[True, 1]`. Heterogeneity is dual to polymorphism (Haskell lists are polymorphic), and it can be achieved by the polymorphism — parametric:

```
data HeteroList where
   consBool :: Bool -> HeteroList -> HeteroList
   consInteger :: Integer -> HeteroList -> HeteroList
   nil :: HeteroList
xs :: HeteroList
xs = True ‘consBool‘ 1 ‘consInteger‘ nil
```

bądź inkluzyjnego:

or the inclusion one:

```
class ListElem where
   ...
data HeteroList :: * where
   cons :: forall e. ListElem e => e -> HeteroList -> HeteroList
   nil :: HeteroList
xs :: HeteroList
xs = True ‘cons‘ 1 ‘cons‘ nil
```

W drugim przypadku użyliśmy typów egzystencjalnych rozważanych w poprzednim zadaniu. Definując powyższe listy ustalamy albo strukturę (w pierwszym przypadku) albo funkcjonalność (w drugim przypadku) elementów list. W wielu zastosowaniach jest to zbyt ograniczające. Klasę `ListElem` moglibyśmy pozostawić bez metod. Wówczas każdy typ dałby się w niej zaistalować, ale też niczego nie dałoby się zrobić z już skonstruowaną listą. Zauważmy, że listy w Haskellu są polimorficzne, tj. wspólny typ elementów jest odzwierciedlony w typie listy. Aby zdefiniować listy heterogeniczne możemy w typie listy ująć typy wszystkich jej elementów osobno:

In the second case we used existential types cosidered in the previous problem. Defining the lists above we fix either structure (in the first case) or functionality (in the second one) of list elements. In many applications it is too restrictive. The `ListElem` class can be left without methods. Then every type could be installed in it, but we could do nothing with already constructed list. Notice that lists in Haskell are polymorphic, *i.e.*, the common type of all elements of a list is expressed in its type. To define heterogenic lists we can capture the types of all elements of a list separtely in its type:

```
class TyList (hlist :: *)
data Nil
data TyList tail => head :. tail
instance TyList Nil
instance TyList tail => TyList (head :. tail)
data HList :: * -> * where
   Nil :: HList Nil
   (:.) :: TyList tail => head -> HList tail -> HList (head :. tail)
infixr 5 :.
```

(używamy tu roszerzeń `KindSignatures`, `GADTs`, `EmptyDataDecls` i `TypeOperators` standardu języka Haskell). Zainstaluj powyższe listy w klasie `Show` i zdefinuj dla nich operacje ujawniania długości listy, konkatenacji list i odwracania.

(we use the extensions `KindSignatures`, `GADTs`, `EmptyDataDecls` and `TypeOperators` of the Haskell standard). Install these lists in the `Show` class and define for them the length, concatenate and reverse operations.

Zadanie 7 (2 pkt). Typy elementów list z poprzedniego zadania mogą być całkowicie dowolne. W wielu zastosowaniach wszystkie elementy listy są różnymi instancjami jednego polimorficznego typu. Jeśli typ ten będzie parametrem listy heterogenicznej, to jej elementy będzie można przetwarzać dużo łatwiej. Niech więc

Problem 7 (2 p). The types of elements of lists from the previous problem can be completely arbitrary. In many applications all elements of a list are different instances of the same type. If this type is a parameter of a heterogenic list, then its elements can be processed much easier. So let

```
class ListElem (e : *) where
   ...
class TyList (hlist :: *)
data Nil
data TyList tl => hd :. tl
instance TyList Nil
instance TyList tl => TyList (hd :. tl)
data HList ty xs where
   Nil :: HList ty Nil
   (:.) :: (ListElem hd, TyList tl) =>
               ty hd -> HList ty tl -> HList ty (hd :. tl)
infixr 5 :.
```

Zainstaluj powyższe listy w klasie `Show` i zdefinuj dla nich operacje ujawniania długości listy, konkatenacji list i odwracania. Zdefinuj dla nich funkcję

Install these lists in the `Show` class and define for them the length, concatenate and reverse operations. Define for them the function

```
hmap :: (forall s. ListElem s => a s -> b s) -> List a xs -> List b xs
```

Zauważ, że powyższe listy nie są szczególnym przypadkiem, lecz uogólnieniem list z poprzedniego zadania — wystarczy wziąć `HList Id`.

Notice the lists above are not the special case but the generalization of lists from the previous problem — it suffices to take `HList Id`.

Grupa zaawansowana

W tym tygodniu grupa zaawansowana nie otrzymuje specjalnych zadań.

Advanced group

In this week the advanced group gets no special problems.