Programowanie 2009 Programming 2009

Lista zadań nr 14 Problem set no. 14

Na zajęcia 9–10 czerwca 2009 Due June 10, 2009

W poniższych zadaniach rozważamy system typów z polimorfizmem parametrycznym zadany następującymi regułami:

In the problems below we consider the type system with parametric polymorphism given by the following rules:

$$
\begin{array}{lcl}
t & ::= & x \mid t_1 t_2 \mid \lambda x.t \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid \texttt{letrec}\ x = t_1\ \texttt{in}\ t_2 \mid \texttt{letpolyrec}\ x = t_1\ \texttt{in}\ t_2 \mid \ldots \\
\tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \ldots \\
\sigma & ::= & \forall \vec{\alpha}.\tau
\end{array}
$$

$$
\frac{}{\Gamma, x : \forall \vec{\alpha}\tau \vdash x : \tau[\vec{\alpha}/\vec{\rho}]} \qquad \frac{\Gamma \vdash t_1 : \tau_2 \to \tau_1 \quad \Gamma \vdash t_2 : \tau_2}{\Gamma \vdash t_1 t_2 : \tau_1} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}
$$

$$
\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}\ \vec{\alpha} = \mathrm{FV}(\tau_1) \setminus \mathrm{FV}(\Gamma)
$$

$$
\frac{\Gamma, x : \tau_1 \vdash t_1 : \tau_1 \quad \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{letrec}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}\ \vec{\alpha} = \mathrm{FV}(\tau_1) \setminus \mathrm{FV}(\Gamma)
$$

$$
\frac{\Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash t_1 : \tau_1 \quad \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{letpolyrec}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}\ \vec{\alpha} = \mathrm{FV}(\tau_1) \setminus \mathrm{FV}(\Gamma)
$$

Grupy zasadnicze

Basic groups

Zadanie 1 (1 pkt). Wzorując się na wykładzie napisz kompletny algorytm znajdowania typu w systemie z polimorfizmem parametrycznym i definicjami rekurencyjnymi (oczywiście bez rekursji polimorficznej).

Problem 1 (1 p). Following the lecture write the complete algorithm for finding a type in the system with parametric polymorphism and recursive definitions (of course without polymorphic recursion).

Zadanie 2 (1 pkt). Dla konstrukcji `let` wprowadziliśmy na wykładzie (za Landinem) regułę `let`-redukcji:

Problem 2 (1 p). During the lecture we introduced (following Landin) the `let`-reduction rule:

$$\texttt{let}\ x = t_1\ \texttt{in}\ t_2 \quad \to \quad t_2[x/t_1].$$

Wynajdź podobną regułę dla konstrukcji `letrec` i `letpolyrec` (ich semantyka dynamiczna powinna być taka sama — one różnią się tylko semantyką statyczną, tj. typowalnością).

Invent a similar rule for the `letrec` and `letpolyrec` constructions (their dynamic semantics should be the same — they only differ with respect to the static semantics, *i.e.*, typability).

Zadanie 3 (1 pkt). Zamiast konstrukcji `letrec` wprowadzamy do języka „rekursję anonimową" (tak jak funkcje anonimowe):

Problem 3 (1 p). Instead of the `letrec` construction we introduce to the language the "anonymous recursion" (like anonymous functions):

$$\mu x.t.$$

Wyrażenie powyższe oznacza „najmniejszy punkt stały funkcji $\lambda x.t$". Zaproponuj odpowiednie reguły typowania (semantyka statyczna) i redukcji (semantyka dynamiczna) dla tej konstrukcji.

The expression above stands for the "smallest fixpoint of the function $\lambda x.t$". Suggest appropriate typing (static semantics) and reduction rules (dynamic semantics) for this construction.

Zadanie 4 (1 pkt). Zamiast konstrukcji `letrec` wprowadzamy do języka „operator punktu stałego" — specjalna stałą `Y`. Zaproponuj odpowiednie reguły typowania (semantyka statyczna) i redukcji (semantyka dynamiczna) dla niego.

Problem 4 (1 p). Instead of the `letrec` construction we introduce to the language the "fixpoint operator" — a special constant `Y`. Suggest appropriate typing (static semantics) and reduction rules (dynamic semantics) for this construction.

Zadanie 5 (1 pkt). Przypomnijmy, że w prostym (monomorficznym) systemie typów konstrukcja `let` była typowana następujaco:

Problem 5 (1 p). Recall that in the simple (monomorphic) type system the `let` construction was typed as fallows:

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \tau_1 \vdash t_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}$$

Podaj przykład termu, który i) ma typ w systemie z polimorfizmem parametrycznym, a nie posiada go w systemie monomorficznym, ii) posiada typy w obu systemach, ale jego typy główne w obu systemach są różne.

Give an example of a term which i) is typable in the system with parametric polymorphism but untypable in the monomorphic system, b) is typable in both system but has different principal types in those systems.

Zadanie 6 (1 pkt). Do języka dodajemy pary:

Problem 6 (1 p). We add pairs to the language:

$$\begin{aligned} t &::= \ldots \mid (t_1, t_2) \mid \texttt{fst}\ t \mid \texttt{snd}\ t \mid \ldots \\ \tau &::= \ldots \mid \tau_1 \times t_2 \mid \ldots \end{aligned}$$

unie:

unions:

$$\begin{aligned} t &::= \ldots \mid \texttt{left}\ t \mid \texttt{right}\ t \mid \texttt{case}\ t\ \texttt{of left}\ x \to t_1\ \texttt{right}\ x \to t_2 \mid \ldots \\ \tau &::= \ldots \mid \tau_1 + t_2 \mid \ldots \end{aligned}$$

typ jednoelementowy:

one-element type:

$$\begin{aligned} t &::= \ldots \mid \texttt{()} \mid \ldots \\ \tau &::= \ldots \mid \texttt{unit} \mid \ldots \end{aligned}$$

typ pusty:

the empty type:

$$\begin{aligned} t &::= \ldots \mid ? \mid \ldots \\ \tau &::= \ldots \mid \texttt{void} \mid \ldots \end{aligned}$$

wartości logiczne:

logical values:

$$\begin{aligned} t &::= \ldots \mid \texttt{true} \mid \texttt{false} \mid t_1 \vee t_2 \mid t_1 \wedge t_2 \mid \texttt{if}\ t_0\ \texttt{then}\ t_1\ \texttt{else}\ t_2 \mid \ldots \\ \tau &::= \ldots \mid \texttt{bool} \mid \ldots \end{aligned}$$

i liczby całkowite:

and the integer numbers:

$$\begin{aligned} t &::= \ldots \mid n \mid t_1 + t_2 \mid t_1 \cdot t_2 \mid t_1 \leq t_2 \mid \ldots \\ \tau &::= \ldots \mid \texttt{int} \mid \ldots \end{aligned}$$

Zaproponuj dla nich odpowiednie reguły typowania (semantyka statyczna) i redukcji (semantyka dynamiczna).

Suggest appropriate typing rules (static semantics) and reduction rules (dynamic semantics) for them.

Zadanie 7 (1 pkt). Liczby naturalne kodujemy w prostym systemie typów tak samo, jak w rachunku beztypowym — za pomocą liczebników Churcha:

Problem 7 (1 p). In the simply typed lambda calculus we code the natural numbers in the same way we did in the untyped calculus — using the Church numerals:

$$\underline{n} \quad = \quad \lambda xy.x^{(n)}y.$$

Wszystkie liczebniki mają wspólny typ $nat = (\alpha \to \alpha) \to (\alpha \to \alpha)$. Zdefiniuj wyrażenia

All the numerals have the common type $nat = (\alpha \to \alpha) \to (\alpha \to \alpha)$. Define the terms

$$\begin{aligned} \mathit{plus} &: \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\ \mathit{times} &: \mathit{nat} \to \mathit{nat} \to \mathit{nat} \\ \mathit{iszero} &: \mathit{nat} \to \mathit{nat} \end{aligned}$$

tak by

so that

$$\begin{aligned} \mathit{plus}\ \underline{n}\ \underline{m} &=_\beta \underline{n+m} \\ \mathit{times}\ \underline{n}\ \underline{m} &=_\beta \underline{n \cdot m} \\ \mathit{iszero}\ \underline{0} &=_\beta \underline{0} \\ \mathit{iszero}\ \underline{n+1} &=_\beta \underline{1} \end{aligned}$$

dla wszelkich liczb naturalnych $n, m \in \mathbb{N}$.

for any natural numbers $n, m \in \mathbb{N}$.

Zadanie 8 (1 pkt). Pokaż, że wyrażenie

Problem 8 (1 p). Show that the term

$$\texttt{letrec}\ f\ x = f\ (\lambda y.x)\ \texttt{in}\ f$$

nie posiada typu, ale wyrażenie

is not typable, but the term

$$\texttt{letpolyrec}\ f\ x = f\ (\lambda y.x)\ \texttt{in}\ f$$

posiada.

is.

Zadanie 9 (1 pkt). Drzewa binarne o etykietowanych liściach definiujemy w Haskellu następująco:

Problem 9 (1 p). We define binary trees with labelled leaves in Haskell as follows:

```
data Tree a = Node (Tree a, Tree a) | Leaf a
```

Jeśli chcemy zdefiniować typ, którego wartościami będą jedynie drzewa pełne, to wystarczy zamienić typ `Tree` i typ pary miejscami:

If we want to define a type whose values are only full binary trees, then it suffices to swap the `Tree` and pair types:

```
data Tree a = Node (Tree (a, a)) | Leaf a
```

Zaprogramuj w Haskellu następujące funkcje:

Define in Haskell the following functions:

```
height :: Tree a -> Int
flatten :: Tree a -> [a]
```

Wyjaśnij, gdzie pojawia się rekursja polimorficzna.

Explain where the polymorphic recursion occurs.

Grupy rozszerzone

Extended groups

Zadanie 1 (1 pkt). Rozważmy następujący wariant rachunku lambda z typami prostymi:

Problem 1 (1 p). Consider the following variant of the simply typed lambda calculus:

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

gdzie o jest stałą (o oznacza typ *bazowy*, taki jak `int`). Nie ma zmiennych typowych. Jest to zatem język, w którym typy są monomorficzne i występują w termach *explicite* (tak jak w Pascalu). Reguły typowania są następujące:

where o is a constant (o stands for a *base* type, like `int`). There are no type variables. Hence this is a language in which types are monomorphic and occur *explicite* in terms (like in Pascal). The typing rules are as follows:

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

Wyrażenie jest *zamknięte*, jeśli nie zawiera zmiennych wolnych. Wprowadzamy pojęcie *rzędu* typu:

An term is *closed* if does not contain free variables. We introduce the notion of the *order* of a type:

$$\begin{aligned} \mathrm{order}(o) &= 1, \\ \mathrm{order}(\sigma_1 \to \ldots \to \sigma_n \to o) &= 1 + \max(\mathrm{order}(\sigma_1), \ldots, \mathrm{order}(\sigma_n)). \end{aligned}$$

Moc typu, oznaczana $\mathrm{card}(\sigma)$, to liczba różnych zamkniętych wyrażeń tego typu w postaci normalnej (utożsamiamy wyrażenia różniące się jedynie nazwami zmiennych związanych). Napis $\sigma^k \to \tau$ oznacza $\sigma \to \ldots \to \sigma \to \tau$, gdzie σ występuje k razy. Formalnie: $\sigma^0 \to \tau = \tau$ oraz $\sigma^{k+1} \to \tau = \sigma \to \sigma^k \to \tau$. Udowodnij twierdzenie:

The cardinality of a type, denoted $\mathrm{card}(\sigma)$, is the number of different closed terms of that type in normal form (we identify terms that differ only in names of bound variables). The formula $\sigma^k \to \tau$ stands for $\sigma \to \ldots \to \sigma \to \tau$, where σ occurs k times. Formally, $\sigma^0 \to \tau = \tau$, and $\sigma^{k+1} \to \tau = \sigma \to \sigma^k \to \tau$. Prove the theorem:

1. Jeśli $\mathrm{order}(\sigma) = 1$, to $\sigma = o$ i $\mathrm{card}(\sigma) = 0$.
2. Jeśli $\mathrm{order}(\sigma) = 2$, to $\sigma = o^k \to o$ i $\mathrm{card}(o^k \to o) = k$.
3. Jeśli $\mathrm{order}(\sigma) > 2$, to $\mathrm{card}(\sigma) = 0$ lub $\mathrm{card}(\sigma) = \infty$. Ponadto $\mathrm{card}(\sigma \to \tau) = \infty$ wtedy i tylko wtedy, gdy $\mathrm{card}(\sigma) = 0$ lub $\mathrm{card}(\tau) = \infty$.

1. If $\mathrm{order}(\sigma) = 1$, then $\sigma = o$ and $\mathrm{card}(\sigma) = 0$.
2. If $\mathrm{order}(\sigma) = 2$, then $\sigma = o^k \to o$ and $\mathrm{card}(o^k \to o) = k$.
3. If $\mathrm{order}(\sigma) > 2$, then $\mathrm{card}(\sigma) = 0$ or $\mathrm{card}(\sigma) = \infty$. Moreover $\mathrm{card}(\sigma \to \tau) = \infty$ if and only if $\mathrm{card}(\sigma) = 0$ or $\mathrm{card}(\tau) = \infty$.

Wniosek: dla dowolnego typu σ mamy: $\mathrm{card}(\sigma) > 0$ wtedy i tylko wtedy, gdy σ przeczytane jako formuła klasycznego rachunku zdań, w której o oznacza fałsz, zaś $\to$ implikację jest prawdziwa. Typy możemy więc utożsamiać z formułami logicznymi a wyrażenia tych typów z ich dowodami. Np. wyrażenie $\lambda x : o.x$ jest dowodem formuły $o \to o$. Nie ma wyrażeń typu $(o \to o) \to o$ (bo ta formuła nie jest prawdziwa, więc nie ma dowodu). Jest to tzw. *izomorfizm Curry'ego-Howarda*.

Corollary: for any type σ one has $\mathrm{card}(\sigma) > 0$ if and only if σ considered as a formula of the classical propositional calculus in which o stands for falsity and $\to$ for implication is valid. Thus we can identify types with propositional formulas and terms with their proofs. For example the term $\lambda x : o.x$ is a proof of the formula $o \to o$. There are no terms of type $(o \to o) \to o$ (because this formula is not valid). This is so called *Curry-Howard isomorphism*.

Zadanie 2 (1 pkt). Rozważmy język:

Problem 2 (1 p). Consider the language:

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

z następującymi regułami typowania:

with the following typing rules:

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

Rozważmy opisaną na wykładzie dziedzicznie skończoną strukturę typów $\mathfrak{M}_2$:

Consider the hereditarily finite type structure $\mathfrak{M}_2$ described during the lecture:

$$\begin{aligned} [\![o]\!] &= \{0, 1\} \\ [\![\sigma \to \tau]\!] &= [\![\tau]\!]^{[\![\sigma]\!]} \\ [\![x^\sigma]\!]\eta &= \eta_\sigma(x) \\ [\![t_1 t_2]\!]\eta &= ([\![t_1]\!]\eta)([\![t_2]\!]\eta) \\ ([\![\lambda x^\sigma . t]\!]\eta)(d) &= [\![t]\!](\eta[x \to d]), \quad d \in [\![\sigma]\!] \end{aligned}$$

Wskaż elementy nieosiągalne w zbiorach $[\![o \to o \to o]\!]$ oraz $[\![(o \to o) \to (o \to o)]\!]$.

Zadanie 3 (1 pkt). Pokaż, że w systemie z polimorfizmem parametrycznym można napisać program P_n rozmiaru n, którego typ ma rozmiar $2^{2^{\Omega(n)}}$.

Pokaż, że typ wyrażenia rozmiaru n w systemie z polimorfizmem parametrycznym ma rozmiar $2^{2^{O(n)}}$.

Zadanie 4 (1 pkt). Na wykładzie rozważaliśmy wprowadzenie polimorfizmu parametrycznego do systemu typów za pomocą reguły

Indicate unreachable elements in sets $[\![o \to o \to o]\!]$ and $[\![(o \to o) \to (o \to o)]\!]$.

Problem 3 (1 p). Show that in the system with parametric polymorphism it is possible to write a program P_n of size n whose type has size $2^{2^{\Omega(n)}}$.

Show that a type of an expression of size n in the system with parametric polymorphism has size $2^{2^{O(n)}}$.

Problem 4 (1 p). During the lecture we considered an introduction of the parametric polymorphism to the type system using the rule

$$\frac{\Gamma \vdash t_2[x/t_1] : \tau}{\Gamma \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau}$$

Pokaż, że jeśli $\Gamma \vdash t : \tau$ daje się udowodnić w systemie ze schematami typów, to także daje się udowodnić za pomocą powyższej reguły. Zauważ, że twierdzenie odwrotne nie jest prawdziwe. Wskaż przykład. Popraw odpowiednio powyższą regułę i udowodnij równoważność obu systemów.

Show that if $\Gamma \vdash t : \tau$ is provable in the system with type schemes, then it can be proven using the rule above. Note that the converse theorem is not true. Show an example. Correct appropriately the rule above and prove the equivalence of both systems.

Zadanie 5 (1 pkt). Pokaż, że polimorfizm parametryczny „kłóci się" ze skutkami ubocznymi. Rozważmy język

Problem 5 (1 p). Show that the parametric polymorphism "disagrees" with side effects. Consider the language

$$\begin{array}{rcl} t & ::= & x \mid t_1 t_2 \mid \lambda x.t \mid \texttt{let}\ x = t_1\ \texttt{in}\ t_2 \mid n \mid t_1 + t_2 \mid \texttt{True} \mid \texttt{False} \mid \texttt{ref}\ t \mid t_1\ \texttt{:=}\ t_2 \mid \texttt{!}\ t \\ \tau & ::= & \alpha \mid \tau_1 \to \tau_2 \mid \texttt{Int} \mid \texttt{Bool} \mid \texttt{Ref}\ \tau \\ \sigma & ::= & \forall \vec{\alpha}.\tau \end{array}$$

$$\frac{}{\Gamma, x : \forall \vec{\alpha}.\tau \vdash x : \tau[\vec{\alpha}/\vec{\tau}']} \qquad \frac{\Gamma \vdash t : \tau_1 \to \tau_2 \quad \Gamma \vdash s : \tau_2}{\Gamma \vdash ts : \tau_2} \qquad \frac{\Gamma, x : \tau_1 \vdash t : \tau_2}{\Gamma \vdash \lambda x.t : \tau_1 \to \tau_2}$$

$$\frac{\Gamma \vdash t_1 : \tau_1 \quad \Gamma, x : \forall \vec{\alpha}.\tau_1 \vdash t_2 : \tau_2}{\texttt{let}\ x = t_1\ \texttt{in}\ t_2 : \tau_2}\ \vec{\alpha} = \mathrm{FV}(\tau_1) \setminus \mathrm{FV}(\Gamma)$$

$$\frac{}{\Gamma \vdash n : \texttt{Int}} \qquad \frac{\Gamma \vdash t_1 : \texttt{Int} \quad \Gamma \vdash t_2 : \texttt{Int}}{\Gamma \vdash t_1 + t_2 : \texttt{Int}} \qquad \frac{}{\Gamma \vdash \texttt{True} : \texttt{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{False} : \texttt{Bool}}$$

$$\frac{\Gamma \vdash t : \tau}{\Gamma \vdash \texttt{ref}\ t : \texttt{Ref}\ \tau} \qquad \frac{\Gamma \vdash t_1 : \texttt{Ref}\ \tau \quad \Gamma \vdash t_2 : \tau}{\Gamma \vdash t_1\ \texttt{:=}\ t_2 : \tau} \qquad \frac{\Gamma \vdash t : \texttt{Ref}\ \alpha}{\Gamma \vdash \texttt{!}\ t : \alpha}$$

Operator `ref` alokuje pamięć dla nowej zmiennej i inicjuje ją podaną wartością, `:=` jest operatorem przypisania, a `!` — dereferencji. Np. program

The `ref` operator allocates memory for a new variable, `:=` is the assignment, and `!` — the dereference operator. For example the program

$$\texttt{let}\ x = \texttt{ref}\ 1\ \texttt{in}\ x\ \texttt{:=}\ 2$$

alokuje pamięć dla komórki przechowującej dane typu `Int`, związuje adres tej komórki ze zmienną x typu `Ref Int`, inicjuje tę komórkę wartością 1, a następnie przypisuje jej wartość 2. Pokaż, że powyższy system jest sprzeczny i pozwala napisać program, który dodaje 1 do `True`. Wykorzystaj ten trik do zdefiniowania funkcji $\texttt{cast} : \alpha \to \beta$.

allocates memory for a cell that stores data of type `Int`, binds the address of that cell to the name x of type `Ref Int`, initializes that cell with value 1, then assigns the value 2 to it. Show that the system above is unsound and allows to write a program which adds 1 to `True`. Use this trick to define a function $\texttt{cast} : \alpha \to \beta$.

Komentarz: Istnieje wiele rozwiązań tego problemu, żadne jednak nie jest eleganckie. Konserwatywne rozszerzenie systemu Hindley'a-Milnera wymaga wprowadzenia dodatkowych zmiennych typowych (tzw. słabych typów) i mało intuicyjnych reguł typowania. Prostsze rozwiązanie (Andrew Wrighta) psuje system typów tak, że przestaje być dla niego słuszne twierdzenie o *subject reduction* dla reguły η. W Haskellu problem znika, bo nie ma skutków ubocznych (choć pojawia się dla funkcji `unsafePerformIO` która powoduje skutki uboczne i nie jest dla niej w żaden sposób rozwiązany — funkcja ta pozwala zdefiniować funkcję $\texttt{cast} : \alpha \to \beta$).

Note: There exist many solutions to this problem, however none of them is elegant. A conservative extension of the Hindley-Milner system needs additional type variables (so called weak types) and unintuitive typing rules to be introduced. A simpler solution (by Andrew Wright) spoils the type system so that the subject reduction theorem for η-reduction rule is not valid in this system. The problem disappears in Haskell because of lack of side effects (but reappears for the `unsafePerformIO` function, which does cause side effects, and is not solved in any way — this function allows to define a function $\texttt{cast} : \alpha \to \beta$).

Zadanie 6 (1 pkt). Pokaż, że w języku z poprzedniego zadania można przywrócić niesprzeczność jeśli wprowadzimy nowy rodzaj zmiennych typowych. Nazwijmy je *słabymi* i oznaczmy ich zbiór przez $\mathcal{W}$. Zmienne te nie mogą wystąpić w schemacie typu pod kwantyfikatorem, tj. nie wolno ich generalizować w regule `let` nawet, jeśli nie występują w Γ. Warunek w regule `let` ma więc postać $\vec{\alpha} = \mathrm{FV}(\tau_1) \setminus (\mathrm{FV}(\Gamma) \cup \mathcal{W})$. Przyjmij, że typy mają teraz postać $(\texttt{Weak}\ \alpha_1, \ldots, \texttt{Weak}\ \alpha_n) \Rightarrow \tau$ w której prefiks wskazuje, które zmienne są słabe.

Problem 6 (1 p). Show that we can restore soundness in the system from the previous problem if we introduce a new kind of type variables. Les us call them *weak* and let $\mathcal{W}$ stands for the set of such variables. Those variables cannot appear in a type scheme under the quantifier, *i.e.*, it is not allowed to generalize them in the `let` rule even if they do not occur in Γ. The condition in the `let` rule now has the form $\vec{\alpha} = \mathrm{FV}(\tau_1) \setminus (\mathrm{FV}(\Gamma) \cup \mathcal{W})$. Assume the types now have the form $(\texttt{Weak}\ \alpha_1, \ldots, \texttt{Weak}\ \alpha_n) \Rightarrow \tau$ in which the prefix indicates which variables are weak.

Zadanie 7 (1 pkt). Zauważ, że algebraiczne typy danych świetnie współgrają z polimorfizmem parametrycznym — deklaracja

Problem 7 (1 p). Note that algebraic data types blend well with parametric polymorphism — the declaration

$$\texttt{data}\ T\ \vec{\alpha} = C_1\ \tau_1^1\ \ldots\ \tau_1^{n_1} \mid \ldots \mid C_k\ \tau_k^1\ \ldots\ \tau_k^{n_k}$$

dodaje po prostu do składni termów stałe $C_1, \ldots, C_k$ i wyrażenie `case` pozwalające dopasowywać wyrażenia do wzorców, do typów wyrażenie typowe $T\ \tau_1\ \ldots\ \tau_k$, a do systemu typów — reguły:

simply adds constants $C_1, \ldots, C_k$ and expression `case` which allows to match expressions with patterns to the syntax of terms, type expression $T\ \tau_1\ \ldots\ \tau_k$ to types, and the following rules to the type system:

$$\frac{}{\Gamma \vdash C_i : \tau_i^1 \to \ldots \to \tau_i^{n_i} \to T\ \vec{\alpha}}$$

Zdefiniuj składnię abstrakcyjną takiego języka i napisz algorytm rekonstrukcji typów dla takiego języka.

Zauważ, że musimy przyjąć następujące założenie:

Define the abstract syntax of such language and write a type reconstruction algorithm for it.

Note we need to assume that:

$$\bigcup_{j=1}^{n_i} \mathrm{FV}(\tau_i^j) \subseteq \vec{\alpha}.$$

Inaczej system jest sprzeczny — pokaż, że jeśli dopuścilibyśmy deklarację

Otherwise the system becomes unsound — show that if we allowed the declaration

```
data T = C a
```

to łatwo moglibyśmy napisać funkcję `cast :: a -> b`. (Tu jesteśmy o krok od wynalezienia typów egzystencjalnych).

then we could easily write a function `cast :: a -> b`. (We are one step before inventing existential types here.)

Zadanie 8 (1 pkt). W zadaniu 9 dla grup podstawowych rozważamy nietrywialne zastosowanie rekursji polimorficznej. Wykorzystaj zdefiniowane tam pełne drzewa binarne, by poprawić poniższą implementację kopców dwumianowych (Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998):

Problem 8 (1 p). In the Problem 9 for basic groups we investigate a nontrivial application of the polymorphic recursion. Use the full binary trees defined there to improve the implementation of binomial heaps presented below (Chris Okasaki, *Purely Functional Data Structures*, Cambridge University Press, 1998):

```
data Tree a = Node Int a [Tree a]

rank :: Tree a -> Int
rank (Node r x c) = r

root :: Tree a -> a
root (Node r x c) = x

link :: Ord a => Tree a -> Tree a -> Tree a
link t1@(Node r x1 c1) t2@(Node _ x2 c2) =
   if x1 <= x2
      then Node (r+1) x1 (t2:c1)
      else Node (r+1) x2 (t1:c2)

type BinomialHeap a = [Tree a]

insTree :: Ord a => Tree a -> BinomialHeap a -> BinomialHeap a
insTree t [] = [t]
insTree t ts@(t':ts') =
   if rank t < rank t'
      then t:ts
      else insTree (link t t') ts'

empty :: BinomialHeap a
empty = []

insert :: Ord a => a -> BinomialHeap a -> BinomialHeap a
insert x = insTree (Node 0 x [])

merge :: Ord a => BinomialHeap a -> BinomialHeap a -> BinomialHeap a
merge ts1 [] = ts1
merge [] ts2 = ts2
merge ts1@(t1:ts1') ts2@(t2:ts2')
    | rank t1 < rank t2 = t1 : merge ts1' ts2
    | rank t2 < rank t1 = t2 : merge ts1 ts2'
    | otherwise = insTree (link t1 t2) (merge ts1' ts2')

findMin :: Ord a => BinomialHeap a -> a
deleteMin :: Ord a => BinomialHeap a -> (a, BinomialHeap a)
```

Kopiec dwumianowy to lista *pełnych* drzew binarnych. W powyższej implementacji jest wiele wartości typu `BinomialHeap`, które nie reprezentują kopców, ponieważ drzewa występujące na liście nie są pełne. Zaprogramuj funkcje `insert`, `merge`, `findMin` i `deleteMin` dla następującej implementacji kopców dwumianowych:

A binary heap is a list of *full* binary trees. There are many values of type `BinomialHeap` in the implementation above, which do not represent any heap, since trees put on a list are not full. Program functions `insert`, `merge`, `findMin` and `deleteMin` for the following implementation of binomial heaps:

```
data Tree a = Node (Tree (a,a)) | Leaf a
type BinomialHeap a = [Tree a]
```

Zadanie 9 (1 pkt). Kopiec dwumianowy, to lista pełnych drzew binarnych *o ściśle rosnących wysokościach*. Zauważ, że w implementacji z poprzedniego zadania nadal występują wartości typu `BinomialHeap`, które nie reprezentują żadnego kopca dwumianowego. Wydaje się, że do sparametryzowania typu wartościami potrzeba typów zależnych, ale jeśli parametr typu jest np. liczbą naturalną, to łatwo możemy symulować wartości za pomocą typów (Oleg Kiselyov, Number-parameterized types, *The Monad.Reader*, Issue Five, October 2005):

Problem 9 (1 p). A binary heap is a list of full binary trees *of strictly increasing height*. Note that in the implementation from the previous problem still there exist values which do not represent any binary heap. It seems that one needs dependent types to parametrize types with values, but if a type prarameter is, *e.g.*, a natural number, then we can easily simulate values by types (Oleg Kiselyov, Number-parameterized types, *The Monad.Reader*, Issue Five, October 2005):

```
class Nat (t :: *)
data Zero
data Nat n => Succ n
instance Nat Zero
instance Nat n => Nat (Succ n)
```

Teraz łatwo zdefiniować pełne binarne drzewa, których wysokość jest parametrem ich typu:

Now it is easy to define full binary trees whose height is a parameter of their type:

```
data Tree h a where
   Node :: (Nat h, Ord a) => Tree h a -> Tree h a -> Tree (Succ h) a
   Leaf :: Ord a => a -> Tree Zero a
```

(W powyższym kodzie użyliśmy następujących rozszerzeń standardu języka Haskell: `KindSignatures`, `GADTs` i `EmptyDataDecls`). Zaprogramuj typ `BinomialHeap` tak, by każda jego wartość była listą pełnych drzew o ściśle rosnących wysokościach, tj. by reprezentowała jakiś poprawnie zbudowany kopiec.

(In the code above we use the following extensions of the Haskell standard: `KindSignatures`, `GADTs` and `EmptyDataDecls`). Program a type `BinomialHeap` so that any value of that type is a list of full binary trees of strictly increasing height, *i.e.*, represents some properly built heap.

Grupa zaawansowana

Advanced group

Zadanie 1 (2 pkt). Udowodnij twierdzenie o istnieniu typu głównego dla języka z polimorfizmem parametrycznym.

Problem 1 (2 p). Prove the theorem on the existence of principal types for the language with parametric polymorphism.

Zadanie 2 (2 pkt). Rozważmy język:

Problem 2 (2 p). Consider the language:

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

z następującymi regułami typowania:

with the following typing rules:

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

Opisana na wykładzie struktura typów zależy od wyboru dziedziny interpretacji typu bazowego (a dokładniej od jego mocy). Niech $\mathfrak{M}_\kappa$ oznacza strukturę typów w której dziedzina interpretacji typu bazowego ma moc κ. Udowodnij, że

The type structure described during the lecture depends on the choice of the domain of interpretation of the base type (more specifically on its cardinality). Let $\mathfrak{M}_\kappa$ stands for the type structure in which the domain of interpretation of the base type has cardinality κ. Prove that

$$t_1 =_{\beta\eta} t_2 \quad \Longleftrightarrow \quad \mathfrak{M}_{\aleph_0} \models t_1 = t_2.$$

Pokaż, że nie jest to prawdą dla $\mathfrak{M}_\kappa$ gdzie $\kappa < \aleph_0$, ale że dla każdego termu t_1 istnieje takie $\kappa \in \mathbb{N}$, że dla każdego termu t_2 zachodzi

Show that it is not true for $\mathfrak{M}_\kappa$ where $\kappa < \aleph_0$, but that for any term t_1 there exists such $\kappa \in \mathbb{N}$, that for any term t_2 one has

$$t_1 =_{\beta\eta} t_2 \quad \Longleftrightarrow \quad \mathfrak{M}_\kappa \models t_1 = t_2.$$