

System typów F_ω

Systemy Typów 2010/11
Prowadzący: dr Dariusz Biernacki

Piotr Polesiuk	Małgorzata Jurkiewicz
bassists@o2.pl	gosia.jurkiewicz@gmail.com

Wrocław, dnia 13 lutego 2011 r.

1. Wstęp

No to na razie taki bałagan

2. System F_ω

W rozdziale tym chcielibyśmy się skupić na systemie F_ω okrojonym do niezbędnego minimum. Przedstawimy, jak wyglądają termy, typy i wartości tego języka, a także pokażemy, jak przebiega typowanie, znajdowanie rodzaju, ewaluacja czy sprawdzanie równości typów. Postaramy się pisać jasno i pokażemy parę przykładów, aby nieobyty w temacie Czytelnik nie zgubił się. W rozdziale trzecim do tak zdefiniowanego systemu będziemy wprowadzać rozszerzenia.

2.1. Termy i typy w F_ω

System F_ω to rachunek będący rozszerzeniem λ_ω oraz systemu F . Wszystkie trzy wywodzą się z rachunku lambda z typami prostymi. Termy oraz typy definiujemy w λ_\rightarrow następująco:

$t ::=$	<i>termy</i>
x	<i>zmienne</i>
$\lambda x : T. t$	<i>abstrakcja anotowana</i>
$\lambda x. t$	<i>abstrakcja nieanotowana</i>
$t t$	<i>aplikacja</i>
$T ::=$	<i>typy</i>
X	<i>zmienna typowa</i>
$T \rightarrow T$	<i>typ funkcji</i>

2.1.1. System λ_ω

Główną cechą systemu λ_ω jest to, że oprócz termów zależnych od termów mamy typy zależne od typów, czyli możemy mówić o aplikacji i abstrakcji typowej, a tak powstałe 'typy' będziemy nazywać konstruktorami. By nam się nie pomyliło z abstrakcją na termach, zmienne konstruktorowe będziemy zaczynać dużą literą. Przykładowo $Tb = \lambda X. X \rightarrow Bool$ i $\lambda X. X$ są abstrakcjami konstruktowymi, ale $\lambda x. x$ jest abstrakcją na termach. Do konstruktora Tb możemy zaaplikować $Bool$ i dostaniemy $(\lambda X. X \rightarrow Bool)Bool$ równoważne $Bool \rightarrow Bool$. Jak widać, użyliśmy słowa *równoważne*. W rachunku lambda z typami prostymi sposób konstrukcji typów gwarantował nam, że dwa typy T_1 i T_2 na pewno są różne (zakładając, że typy bazowe były sobie różne). W λ_ω jest inaczej – konstruktory tego systemu możemy podzielić na klasy równoważności. Do klasy $Bool \rightarrow Bool$ należą również $(Tb^n)Bool$ dla n naturalnego, a T^n oznacza aplikację n konstruktorów T . Zauważmy, że odpowiednikiem takiej relacji równoważności w λ_\rightarrow jest β -równoważność. W świecie typów nazwiemy taką relację \equiv^1 . Każdy konstruktor typu jest silnie normalizowalny i zachodzi własność Churcha-Rossera. Przez $nf(T)$ oznaczamy postać normalną konstruktora rodzaju T . Dodatkowo wprowadzimy następującą regułę: $\frac{\Gamma \vdash t : S \quad S \equiv T}{\Gamma \vdash t : T}$ mówiącą, że jeżeli S jest konstruktorem termu t , to dowolny konstruktor S równoważny z T również jest konstruktorem t .

¹formalnie zdefiniujemy tą relację w rozdziale 2.2.3

Niestety, w tak zdefiniowanym systemie powstaje jeden problem. Nie chcielibyśmy, aby `Bool Bool` było dozwolone, tak samo, jak w świecie termów nie chcielibyśmy, by `true true` było dozwolone. W świecie termów, by rozwiązać ten problem, wprowadziliśmy typy na termach, w świecie typów wprowadzimy *rodzaje* na konstruktorach. Piszemy, że $T :: K$, czyli konstruktor T jest rodzaju K . Wprowadzimy też jeden rodzaj bazowy $*$.

Wszystkie typy, jakie pojawiły się w λ_{\rightarrow} , są rodzaju $*$. Np. $Bool :: *$, $Nat \rightarrow Nat$, $(Bool \rightarrow Nat) \rightarrow Nat :: *$, itd. Rodzaj $* \Rightarrow *$ będzie odpowiadał funkcjom z konstruktorów w konstruktory, np. $\lambda X.X \rightarrow Bool :: * \Rightarrow *$. $* \Rightarrow * \Rightarrow *$ bierze konstruktor i zwraca funkcję konstruktorową, np. $\lambda X.\lambda Y.X \rightarrow Y :: * \Rightarrow * \Rightarrow *$, itd.

Teraz możemy λ_{\rightarrow} rozszerzyć o następujące konstrukcje:

- rodzaje

$K ::=$	$*$	<i>rodzaje</i>
	$K \Rightarrow K$	<i>rodzaj wszystkich typów</i>
		<i>rodzaj funkcji typowej</i>

- abstrakcję i aplikację typową na typach

$T ::=$	\dots	<i>typy</i>
	$\lambda X :: K.T$	<i>abstrakcja konstruktorowa anotowana</i>
	$\lambda X.T$	<i>abstrakcja konstruktorowa nieanotowana</i>
	$T T$	<i>aplikacji konstruktorowa</i>

Powstaje pytanie, czy wszystkie konstruktory są typami? Otóż nie, typy to konstruktory rodzaju $*$.

2.1.2. System F

System F jest systemem, w którym dodatkowo, oprócz termów zależnych od termów, mamy termy zależne od typów. Wprowadzimy trzeci już rodzaj abstrakcji i aplikacji, poprzedni był w świecie typów, ten będzie w świecie termów. Znana jest nam funkcja identycznościowa $\lambda x.x$, w λ_{\rightarrow} możemy ją napisać na wiele sposobów: $\lambda x : Bool.x$, $\lambda x : Nat.x$, $\lambda x : Bool \rightarrow Nat.x$. W systemie F możemy wszystkie te funkcje zapisać jako: $\lambda X.\lambda x : X.x$. Zauważmy, że ten *term* przyjmuje jako pierwszy argument typ, następnie term tego typu i zwraca term. Przykładem użycia takiego termu mogą być: $(\lambda X.\lambda x : X.x) [Bool] \text{ true}$, co daje `true`, albo $(\lambda X.\lambda x : X.x) [Nat] 1$, co daje `1`. W ten sposób powstała nam *uniwersalna* funkcja identycznościowa, której nadamy tzw. uniwersalny typ: $\lambda X.\lambda x : X.x : \forall X.X \rightarrow X$. Dodatkowo, jako że dodaliśmy już do systemu rodzaje, napiszemy $\lambda X :: *. \lambda x : X.x : \forall X :: *. X \rightarrow X :: *$.

Czy moglibyśmy napisać $\lambda X :: * \Rightarrow *. \lambda x : X.x : \forall X :: * \Rightarrow *. X \rightarrow X :: * \Rightarrow *$? Jak już mówiliśmy, tylko konstruktory rodzaju $*$ są typami, więc powyższy term nie jest dobry.

Po tym krótkim wstępie możemy już zdefiniować odziedziczone z systemu F własności takie, jak:

- abstrakcję i aplikację typową na termach

$t ::=$	\dots	<i>termy</i>
	$\lambda X :: K.t$	<i>abstrakcja typowa anotowana</i>
	$\lambda X :: K.t$	<i>abstrakcja typowa nieanotowana</i>
	$t[T]$	<i>aplikacja typowa</i>

- typ uniwersalny

$T ::=$	\dots	<i>typy</i>
	$\forall X :: K.T$	<i>typ uniwersalny anotowany</i>
	$\forall X.T$	<i>typ uniwersalny nieanotowany</i>

2.2. Typowanie

2.2.1. Kontekst

Kontekst typowania opisany jest następującą składnią abstrakcyjną:

$\Gamma ::=$		<i>kontekst</i>
	\emptyset	<i>pusty kontekst</i>
	$\Gamma, x : T$	<i>wiązanie typu</i>
	$\Gamma, X :: K$	<i>wiązanie rodzaju</i>

Konteksty typowania będziemy często traktować jako skończone zbiory wiązań i będziemy używać teoriomnogościowych symboli na nich. Np. przynależność do kontekstu formalnie definiujemy jako:

$$\frac{}{B \in \Gamma, B} \quad \frac{B \in \Gamma}{B \in \Gamma, B'}$$

Definicje pozostałych operacji teoriomnogościowych są na tyle naturalne, że zostawiamy je Czytelnikowi do uzupełnienia.

2.2.2. Podstawienia

Oprócz zwykłego podstawienia za zmienne, które pozostawiamy Czytelnikowi do uzupełnienia, powinniśmy zdefiniować podstawienie za zmienne konstruktorowe.

- $[Y \mapsto T]X = \begin{cases} T & Y = X \\ X & \text{w.p.p} \end{cases}$
- $[Y \mapsto T](X_1 X_2) = [Y \mapsto T]X_1 [Y \mapsto T]X_2$
- $[Y \mapsto T](S_1 \rightarrow S_2) = [Y \mapsto T]S_1 \rightarrow [Y \mapsto T]S_2$
- $[Y \mapsto T]\forall X.S = \begin{cases} \forall X.S & Y = X \text{ lub } Y \notin FV(S) \\ \forall X.[Y \mapsto T]S & X \notin FV(S) \text{ i } Y \in FV(S) \end{cases}$
- $[Y := T]\lambda X.S = \begin{cases} \lambda X.S & Y = X \text{ lub } Y \notin FV(S) \\ \lambda X.[Y \mapsto T]S & X \notin FV(S) \text{ i } Y \in FV(S) \end{cases}$

2.2.3. Relacja \equiv

Jak wspomnieliśmy w rozdziale 2.1.1, definiujemy na typach relację równoważności. W poniższych wzorach S, S_1, S_2, T, T_1, T_2 to typy, K to rodzaj. Następujące trzy reguły:

$$\frac{}{T \equiv T} \quad \frac{S \equiv T}{T \equiv S} \quad \frac{S \equiv U \quad U \equiv T}{S \equiv T}$$

gwarantują nam równoważność relacji \equiv . Pozostałe reguły jak następuje:

$$\frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 \rightarrow S_2 \equiv T_1 \rightarrow T_2} \quad \frac{S_1 \equiv T_1 \quad S_2 \equiv T_2}{S_1 S_2 \equiv T_1 T_2}$$

$$\frac{S \equiv T}{\lambda X :: K.S \equiv \lambda X :: K.T} \quad (\lambda X :: K.S)T \equiv [X \mapsto T]S$$

definiują równoważność funkcji typowych, aplikacji i abstrakcji konstruktorowych oraz typów uniwersalnych.

2.2.4. Reguły znajdowania rodzaju

W systemie F_ω każdemu poprawnie zbudowanemu typowi przyporządkowujemy rodzaj. Przyporządkowanie to określa relacja $(\vdash \cdot :: \cdot)$ zdefiniowana następująco.

Jeżeli zachodzi $\Gamma \vdash T :: K$, to powiemy, że *typ T jest rodzaju K w kontekście Γ* , gdzie relacja określenia rodzaju $(\vdash \cdot :: \cdot) \subseteq \Gamma \times T \times K$ jest najmniejszą relacją zamkniętą na reguły:

$$\frac{X :: K \in \Gamma}{\Gamma \vdash X :: K} \quad \frac{\Gamma \vdash T_1 :: K_1 \Rightarrow K_2 \quad \Gamma \vdash T_2 :: K_1}{\Gamma \vdash T_1 T_2 :: K_2}$$

$$\frac{\Gamma \vdash X :: K_1 \quad \Gamma \vdash T :: K_2}{\Gamma \vdash \lambda X :: K_1.T :: K_1 \Rightarrow K_2} \quad \frac{\Gamma \vdash X :: K \quad \Gamma \vdash T :: *}{\Gamma \vdash \forall X :: K.T :: *}$$

$$\frac{\Gamma \vdash T_1 :: * \quad \Gamma \vdash T_2 :: *}{\Gamma \vdash T_1 \rightarrow T_2 :: *}$$

2.2.5. Reguły typowania

Jesteśmy już gotowi przedstawić reguły typowania zdefiniowanego wyżej systemu F_ω . Każdemu poprawnie zbudowanemu termowi przyporządkowujemy typ. Przyporządkowanie to określa relacja $(\vdash \cdot :: \cdot)$ zdefiniowana następująco.

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \quad \frac{\Gamma \vdash T_1 :: * \quad \Gamma, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \lambda x : T_1.t_2 : T_1 \rightarrow T_2}$$

$$\frac{\Gamma \vdash t_1 : T_1 \rightarrow T_2 \quad \Gamma \vdash t_2 : T_1}{\Gamma \vdash t_1 t_2 : T_2} \quad \frac{\Gamma \vdash t : S \quad S \equiv T \quad \Gamma \vdash T :: *}{\Gamma \vdash t : T}$$

$$\frac{\Gamma, X :: K \vdash t : T}{\Gamma \vdash \lambda X :: K.t : \forall X :: K.T} \quad \frac{\Gamma \vdash t : \forall X :: K.T \quad \Gamma \vdash T' :: K}{\Gamma \vdash t[T'] : [X \mapsto T']T}$$

2.3. Ewaluacja

Wartości w F_ω zdefiniujemy dokładnie jak w λ_{\rightarrow} .

$v ::=$	<i>wartości</i>
$\lambda x : T.t$	<i>wartość abstrakcji</i>

Ewaluacja przebiega w sposób standardowy dla aplikacji i abstrakcji termów. Teraz, dla czytelności, przetoczmy te reguły ewaluacji (t_1, t'_1, t_2, t'_2, t to termy, v to wartość, $x:T$ to zmienna x typu T):

$$\frac{t_1 \longrightarrow t'_1}{t_1 t_2 \longrightarrow t'_1 t_2} \quad \frac{t_2 \longrightarrow t'_2}{v_1 t_2 \longrightarrow v_1 t'_2}$$

$$(\lambda x : T.t)v \longrightarrow [x \mapsto v]t$$

Do tego dochodzą reguły dla nowych w języku abstrakcji typowych i aplikacji typowych.

$$\frac{t \longrightarrow t'}{t[T] \longrightarrow t'[T]}$$

$$(\lambda X :: K.t)[T] \longrightarrow [X \mapsto T]t$$

3. Rozszerzenia F_ω

W rozdziale tym chcielibyśmy poruszyć, jak w systemie F_ω zdefiniować najprostsze konstrukcje, takie jak wyrażenia arytmetyczne i logiczne, warianty, sekwencje wyrażeń, typy egzystencjalne, rekordy i inne. Pokażemy również, jak przebiega typowanie, ewaluacja i gdzieś tam dodamy reguły tworzenia rodzaju.

Chcielibyśmy podkreślić, że następująca reguła typowania:

$$\frac{\Gamma \vdash t : T \quad S \equiv T \quad \Gamma \vdash S :: *}{\Gamma \vdash t : S}$$

bardzo ułatwia definiowanie reguł typowania w F_ω . W większości przypadków są one takie same lub lekko zmodyfikowane, dlatego nie powinny nastęrczać trudności.

3.1. wyrażenia arytmetyczne i logiczne

Wyrażenia arytmetyczne i logiczne to część, bez której żaden język się nie obędzie. Oczywiście można je sobie zakodować w systemie F_ω , ale normą są wbudowane w język wyrażenia. Termy, typy i wartości wyrażeń zdefiniujemy następująco:

$t ::=$...	<i>termy</i>
	true	<i>prawda</i>
	false	<i>fałsz</i>
	zero	<i>zero</i>
	succ t	<i>następnik</i>
	pred t	<i>poprzednik</i>
	iszero	<i>test na zero</i>
	if t then t else t	<i>warunek</i>
$T ::=$...	<i>typy</i>
	Nat	<i>typ liczbowy</i>
	Bool	<i>typ boolowski</i>
$v ::=$...	<i>typy</i>
	true	<i>wartość prawdy</i>
	false	<i>wartość fałszu</i>
	nv	<i>wartość liczbową</i>
$nv ::=$...	<i>wartość liczbową</i>
	zero	<i>wartość zera</i>
	succ nv	<i>wartość następnika</i>

Na pewno musimy dodać reguły tworzenia rodzaju dla Nat i Bool , którym nadamy rodzaj $*$:

$$\frac{}{\Gamma \vdash \text{Bool} :: *} \quad \frac{}{\Gamma \vdash \text{Nat} :: *}$$

Typowanie wygląda dokładnie tak samo jak w rachunku lambda z typami prostymi. Możemy sobie pozwolić na takie reguły dzięki regule XXX. Przykładowo, nie tylko termny typu Nat mogą się dobrze otypować, gdy zaaplikujemy je do succ . Dla $t : (\lambda X.X)\text{Nat}$ otrzymamy:

$$\frac{\frac{\Gamma \vdash t : (\lambda X.X)\text{Nat} \quad \Gamma \vdash (\lambda X.X)\text{Nat} \equiv \text{Nat} \quad \Gamma \vdash (\lambda X.X)\text{Nat} :: *}{\Gamma \vdash t : \text{Nat}}}{\Gamma \vdash \text{succ } t : \text{Nat}}$$

Nie musimy również pisać reguł typu:

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T \quad \Gamma \vdash T :: *}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

ponieważ posiadanie typu przez term t_2 gwarantuje nam, że ten typ będzie rodzaju $*$. Stąd, reguły typowania wyrażeń arytmetycznych i logicznych wyglądają następująco w F_ω :

$$\frac{}{\Gamma \vdash \text{true} : \text{Bool}} \quad \frac{}{\Gamma \vdash \text{false} : \text{Bool}} \quad \frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{iszero } t : \text{Bool}}$$

$$\frac{\Gamma \vdash t_1 : \text{Bool} \quad \Gamma \vdash t_2 : T \quad \Gamma \vdash t_3 : T}{\Gamma \vdash \text{if } t_1 \text{ then } t_2 \text{ else } t_3 : T}$$

$$\frac{}{\Gamma \vdash \text{zero} : \text{Nat}} \quad \frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{succ } t : \text{Nat}} \quad \frac{\Gamma \vdash t : \text{Nat}}{\Gamma \vdash \text{pred } t : \text{Nat}}$$

Zdefiniowanie reguł ewaluacji pozostawiamy Czytelnikowi.

3.2. Unit i sekwencje

W rachunku lambda z typami prostymi dodaliśmy do składnię języka rozszerzaliśmy o konstrukcje takie, jak:

$t ::=$...	<i>termy</i>
	unit	<i>term unit</i>
$T ::=$...	<i>typy</i>
	Unit	<i>typ unit</i>
$v ::=$...	<i>wartości</i>
	unit	<i>wartość unit</i>

natomiast typowanie przebiegało następująco:

$$\frac{}{\Gamma \vdash \text{unit} : \text{Unit}}$$

a sekwencje definiowaliśmy jako:

$$t_1; t_2 \stackrel{\text{def}}{=} (\lambda x : \text{Unit}. t_2) t_1 \quad \text{gdzie } x \notin \text{FV}(t_2)$$

Aby pozostać przy wbudowanym **unit** w język wystarczy dodać regułę znajdowania rodzaju dla typu **Unit**:

$$\frac{}{\Gamma \vdash \text{Unit} :: *}$$

W rachunku F_ω pojawia się możliwość zakodowania **unit** i **Unit**. Robimy to w taki sposób:
 $\text{unit} \stackrel{\text{def}}{=} \lambda X :: *. \lambda x : X. x$
 $\text{Unit} \stackrel{\text{def}}{=} \forall X :: *. X \rightarrow X$

3.3. Anotacje typowe

Anotacje typowe są przydatną konstrukcją używaną na przykład przy typach egzystencjalnych.

$t ::=$	\dots	<i>termy</i>
	$t \text{ as } T$	<i>anotacja typowa</i>

Ewaluacja i typowanie nie zmieniają się.

3.4. Definicje lokalne

$$\text{let } x = t_1 \text{ in } t_2 \stackrel{\text{def}}{=} (\lambda x : T. t_2) t_1 \quad \text{gdzie } t_1 : T$$

3.5. Rekordy

Składnię rekordów zdefiniujemy następująco:

$t ::=$	\dots	<i>termy</i>
	$\{l_i = t_i^{i \in 1..n}\}$	<i>rekord</i>
	$t.l$	<i>projekcja</i>
$T ::=$	\dots	<i>typy</i>
	$\{l_i : T_i^{i \in 1..n}\}$	<i>typ rekordu</i>
$v ::=$	\dots	<i>wartości</i>
	$\{l_i = v_i^{i \in 1..n}\}$	<i>wartość rekordu</i>

Do relacji tworzenia rodzaju dodamy regułę nadającą rodzaj typowi $\{l_i : T_i^{i \in 1..n}\}$:

$$\frac{\Gamma \vdash T_1 :: * \dots \Gamma \vdash T_n :: *}{\Gamma \vdash \{l_i : T_i^{i \in 1..n}\} :: *}$$

oraz wprowadzimy niewielkie zmiany w regułach typowania:

$$\frac{\Gamma \vdash t_1 : T_1 \dots \Gamma \vdash t_n : T_n \quad \Gamma \vdash \{l_i : T_i^{i \in 1..n}\} :: *}{\Gamma \vdash \{l_i = t_i^{i \in 1..n}\} : \{l_i : T_i^{i \in 1..n}\}}$$

$$\frac{\Gamma \vdash t : \{l_i : T_i^{i \in 1..n}\}}{\Gamma \vdash t.i : T_i}$$

a ewaluację pozostawimy bez zmian:

$$\{l_i = v_i^{i \in 1..n}\}.i \longrightarrow v_i \quad \frac{e \longrightarrow e'}{e.i \longrightarrow e'.i}$$

$$\frac{t_i \longrightarrow t'_i}{\{l_1 = v_1, \dots, l_{i-1} = v_{i-1}, l_i = t_i, \dots, l_n = t_n\} \longrightarrow \{l_1 = v_1, \dots, l_{i-1} = v_{i-1}, l_i = t'_i, \dots, l_n = t_n\}}$$

3.6. Warianty

Składnię wariantów zdefiniujemy następująco:

$t ::=$	\dots	<i>termy</i>
	$\langle l = t \rangle \text{ as } T$	<i>tagowanie</i>
	$\text{case } t \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \text{ }^{i \in 1..n}$	<i>case</i>
$T ::=$	\dots	<i>typy</i>
	$\langle l_i : T_i \text{ }^{i \in 1..n} \rangle$	<i>typ wariantu</i>

Podobnie jak przy rekordach, typ wariantu dostanie rodzaj $*$:

$$\frac{\Gamma \vdash T_1 :: * \dots \Gamma \vdash T_n :: *}{\Gamma \vdash \langle l_i : T_i \text{ }^{i \in 1..n} \rangle :: *}$$

a w regułach typowania wprowadzimy małe zmiany:

$$\frac{\Gamma \vdash t_0 : \langle l_i : T_i \text{ }^{i \in 1..n} \rangle \quad \Gamma, x_1 : T_1 \vdash t_1 : T \dots \Gamma, x_n : T_n \vdash t_n : T}{\Gamma \vdash \text{case } t_0 \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \text{ }^{i \in 1..n} : T}$$

$$\frac{\Gamma \vdash t_j : T_j \quad \Gamma \vdash \langle l_i : T_i \text{ }^{i \in 1..n} \rangle :: *}{\Gamma \vdash \langle l_j = t_j \rangle \text{ as } \langle l_i : T_i \text{ }^{i \in 1..n} \rangle : \langle l_i : T_i \text{ }^{i \in 1..n} \rangle}$$

natomiast ewaluacja pozostanie bez zmian:

$$\text{case } (\langle l_j = t_j \rangle \text{ as } T) \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \text{ }^{i \in 1..n} \longrightarrow [x_j \mapsto v_j]t_j$$

$$\frac{t \longrightarrow t'}{\text{case } t \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \text{ }^{i \in 1..n} \longrightarrow \text{case } t' \text{ of } \langle l_i = x_i \rangle \Rightarrow t_i \text{ }^{i \in 1..n}}$$

$$\frac{t_j \longrightarrow t'_j}{\langle l_j = t_j \rangle \text{ as } T \longrightarrow \langle l_j = t'_j \rangle \text{ as } T}$$

3.7. Punkt stały

$t ::=$	\dots	<i>termy</i>
	$\text{fix } t.v$	<i>punkt stały</i>

Typowanie

$$\frac{\Gamma, f : T \vdash v : S \quad \Gamma \vdash T :: * \quad \Gamma \vdash S :: * \quad S \equiv T}{\Gamma \vdash \text{fix } f.v : T}$$

Ewaluacja

$$\text{fix } f.v \longrightarrow [f \mapsto \text{fix } f.v]v$$

3.8. Listy

Jako przykład wbudowanych typów danych wybraliśmy listy. Podobne rekursywne struktury, jak na przykład drzewa, możemy dodać do języka w analogiczny sposób, jednak rekurencyjne typy danych odwiodą nas od tej konieczności.

$t ::=$	\dots	<i>termy</i>
	$\text{nil}[T]$	<i>lista pusta</i>
	$\text{cons}[T] \ t \ t$	<i>konstruktor listy</i>
	$\text{isnil}[T] \ t$	<i>test na pustość listy</i>
	$\text{head}[T] \ t \ t$	<i>głowa listy</i>
	$\text{tail}[T] \ t$	<i>ogon listy</i>
$T ::=$	$\text{List } T$	<i>typy</i> <i>typ listy</i>
$v ::=$	$\text{nil } [T]$	<i>typy</i> <i>wartość pustej listy</i>
	$\text{cons } [T] \ v \ v$	<i>wartość listy niepustej</i>

Tworzenie rodzaju:

$$\frac{\Gamma \vdash T :: *}{\Gamma \vdash \text{List } T :: *}$$

Typowanie:

$$\frac{\Gamma \vdash \text{List } T :: *}{\Gamma \vdash \text{nil}[T] : \text{List } T} \quad \frac{\Gamma \vdash t_1 : T \quad \Gamma \vdash t_2 : \text{List } T}{\Gamma \vdash \text{List}[T] \ t_1 \ t_2 : \text{List } T}$$

$$\frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{head}[T] \ t : T} \quad \frac{\Gamma \vdash t : \text{List } T}{\Gamma \vdash \text{tail}[T] \ t : \text{List } T}$$

Ewaluacja:

aaa

3.9. Typy egzystencjalne

System F_ω jest już w stanie zakodować typy egzystencjalne, choć wbudowane typy egzystencjalne niczemu nie szkodzą. Pokażemy oba podejścia do tego problemu, zaczynając od przedstawienia składni:

$t ::=$	\dots	<i>termy</i>
	$\{ *T :: K, t \} \text{ as } T$	<i>pakowanie</i>
	$\text{let } \{ X, x \} = t \text{ in } t$	<i>odpakowanie</i>
$T ::=$	\dots	<i>typy</i>
	$\{ \exists X :: K, T \}$	<i>typ egzystencjalny</i>
$v ::=$	\dots	<i>wartości</i>
	$\{ *T, v \} \text{ as } T$	<i>pakowanie</i>

W systemie F_ω powyższe elementy języka możemy zdefiniować następująco:

$$\{ \exists X :: K, T \} \stackrel{\text{def}}{=} \forall Y :: *. (\forall X :: K. T \rightarrow Y) \rightarrow Y$$

$$\{ *U :: K, t \} \text{ as } \{ \exists X :: K, T \} \stackrel{\text{def}}{=} \text{let } x = t \text{ in } \lambda Y :: *. (\lambda f : \forall X :: K. T \rightarrow Y). f \ [U] \ x$$

$$\text{let } \{ X :: K, x \} = t \text{ in } t' \stackrel{\text{def}}{=} t[t'](\lambda X :: K. \lambda x : T. t') \quad \text{gdzie } t' : T'$$

Zauważmy, że dopiero obecność rodzajów pozwoliła nam na tego rodzaju sztuczki. W systemie F nie umiemy tak zrobić.

Na pierwszy rzut oka termy te są niezrozumiałe. Ależ jak bardzo można się mylić – są miłe i przyjemne dla swych wielbicieli. Pokażemy, że zachodzą podstawowe własności pakowania i odpakowania. Rozważmy term $\{ *U :: K, t \} \text{ as } \{ \exists X :: K, T \}$.

$$\{ *U :: K, t \} \text{ as } \{ \exists X :: K, T \} \\ = \text{let } x = t \text{ in } \lambda Y :: *. \lambda f : (\forall X :: K. T \rightarrow Y). f [U] x =$$

$= (\lambda x : [X \mapsto U]T. \lambda Y :: *. \lambda f : (\forall X :: K. T \rightarrow Y). f[U]x) t =$
 $\stackrel{t : [X \mapsto U]T}{=} \lambda Y :: *. \lambda f : (\forall X :: K. T \rightarrow Y). f[U](t : [X \mapsto U]T)$
 co jest typu $\forall Y :: *. (\forall X :: K. T \rightarrow Y) \rightarrow Y$, czyli z definicji $\{\exists X :: K, T\}$.

Rozważmy bardziej życiowy przykład, aby Czytelnik mógł jeszcze raz przeanalizować pakowanie. Oto typowanie w systemie F przykładowego termu:

$$\frac{\Gamma \vdash \{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} : [X \mapsto \text{Nat}]\{a : X, f : X \rightarrow \text{Nat}\}}{\Gamma \vdash *X, \{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} \text{ as } \{\exists X, \{a : X, f : X \rightarrow \text{Nat}\}\}}$$

Następnie wyprowadzimy ten term w F_ω :

$\{\text{Nat} :: K, \{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\}\} \text{ as } \{\exists X :: K, \{a : X, f : X \rightarrow \text{Nat}\}\} =$
 $= \text{let } x = \{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} \text{ in } \lambda Y :: *. \lambda f : (\forall X :: K. \{a : X, f : X \rightarrow \text{Nat}\} \rightarrow Y). f[\text{Nat}]x =$
 $= (\lambda x : [X \mapsto \text{Nat}]\{a : X, f : X \rightarrow \text{Nat}\}. \lambda Y :: *. \lambda f : (\forall X :: K. \{a : X, f : X \rightarrow \text{Nat}\} \rightarrow Y). f[\text{Nat}]x)$
 $\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} \stackrel{\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} : [X \mapsto \text{Nat}]\{a : X, f : X \rightarrow \text{Nat}\}}{=} \lambda Y :: *. \lambda f : (\forall X :: K. \{a : X, f : X \rightarrow \text{Nat}\} \rightarrow Y). f[\text{Nat}]\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\}$

co jest typu $\forall Y :: *. (\forall X :: K. \{a : X, f : X \rightarrow \text{Nat}\} \rightarrow Y) \rightarrow Y$, czyli z definicji $\{\exists X :: K, \{a : X, f : X \rightarrow \text{Nat}\}\}$.

Uważne odpakowanie otrzymanego termu pozostawiamy Czytelnikowi jako ćwiczenie, my pozwolimy sobie przeprowadzać schemat wyvodu:

$\text{let } \{X, x\} = \lambda Y :: *. \lambda f : (\forall X :: K. \{a : X, f : X \rightarrow \text{Nat}\} \rightarrow Y). f[\text{Nat}]\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} \text{ in } (x.f \ x.a) =$
 $= (\lambda Y :: *. \lambda f : (\forall X :: K. \{a : X, f : X \rightarrow \text{Nat}\} \rightarrow Y). f[\text{Nat}]\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\})[\text{T}'](\lambda X :: K. \lambda x : \text{T}. (x.f \ x.a)) =$
 $= ((\lambda X :: K. \lambda x : \text{T}. (x.f \ x.a))[\text{Nat}](\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\} : [X \mapsto \text{Nat}]\{a : X, f : X \rightarrow \text{Nat}\})) =$
 $= (\{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\}.f \ \{a = \text{zero}, f : \lambda x : \text{Nat.succ } x\}.a) =$
 $= (\lambda x : \text{Nat.succ } x)\text{zero} = \text{succ zero}$

Przykłady powyższe obrazują działanie zakodowanych typów rekurencyjnych. Teraz zdefiniujemy wbudowane w język konstrukcje typów rekurencyjnych dla systemu F_ω . Do definicji termów, typów i wartości dodaliśmy już elementy w tabelce na początku rozdziału. Pokażemy, w jaki sposób przebiega typowanie i ewaluacja.

Tworzenie rodzaju:

$$\frac{\text{a tutaj co?}}{\Gamma \vdash \{\exists X :: K, T\} :: K}$$

Typowanie:

$$\frac{\Gamma \vdash t : [X \mapsto U]T \quad \Gamma \vdash U :: K \quad \Gamma \vdash \{\exists X :: K, T\} :: *}{\Gamma \vdash \{*U :: K, t\} \text{ as } \{\exists X :: K, T\} : \{\exists X :: K, T\}}$$

$$\frac{\Gamma \vdash t_1 : \{\exists X :: K, T_1\} \quad \Gamma, X :: K, x : T_1 \vdash t_2 : T_2}{\Gamma \vdash \text{let } \{X, x\} = t_1 \text{ in } t_2 : T_2}$$

Ewaluacja:

$$\text{let } \{X, x\} = (\{*U :: K, v\} \text{ as } T) \text{ in } t \longrightarrow [X \mapsto U][x \mapsto v]t$$

$$\frac{t \longrightarrow t'}{\{*U :: K, t\} \text{ as } T \longrightarrow \{*U :: K, t'\} \text{ as } T}$$

$$\frac{t_1 \longrightarrow t'_1}{\text{let } \{X, x\} = t_1 \text{ in } t_2 \longrightarrow \text{let } \{X, x\} = t'_1 \text{ in } t_2}$$

3.10. Typy rekurencyjne

Hmmm, no tu muszę się zastanowić, to na dole dla zwykłej wersji.

$t ::=$	\dots	<i>termy</i>
	$\text{fold}[T] \ t$	<i>folding</i>
	$\text{unfold}[T] \ t$	<i>unfolding</i>
$T ::=$	\dots	<i>typy</i>
	$\mu X.T$	<i>typ rekursywny</i>
$v ::=$	\dots	<i>wartości</i>
	$\text{fold}[T] \ v$	<i>folding</i>

Tworzenie rodzaju:

$$\frac{}{\Gamma \vdash \mu X.T :: *}$$

Typowanie:

$$\frac{U = \mu X.T \quad \Gamma \vdash t : [X \mapsto U]U}{\Gamma \vdash \text{fold}[U] \ t : U}$$

Ewaluacja:

3.11. dopasowanie wzorca

4. Rekonstrukcja typów

4.1. Kilka słów o składni.

Rozważmy język o składni z 2 rozdziału rozszerzony o następujące konstrukcje:

$t ::=$	\dots	
	$\text{let } x = t \text{ in } t$	<i>zmienna lokalna</i>
	$\text{tlet } X = T \text{ in } t$	<i>typ lokalny</i>
$T ::=$	\bar{X}	<i>zmienna typowa (kwantyfikowana schematem)</i>
$K ::=$	\hat{X}	<i>zmienna rodzajowa (kwantyfikowana schematem)</i>

Ważnym założeniem jest to, że $\bar{X} :: *$, co nam ułatwi sprawę przy unifikacji, a zarazem zbytnio nie ograniczy języka.

Jeżeli chcemy dodać ML-polimorfizm, będą nam potrzebne schematy typów i schematy rodzajów:

$\langle T \rangle ::=$	T	<i>schemat typu</i>
	$\Omega \bar{X}. \langle T \rangle$	<i>typ schematu</i>
	$\Omega \hat{X}. \langle T \rangle$	<i>kwantyfikacja zmiennej typowej</i>
	$\Omega \bar{X}. \langle T \rangle$	<i>kwantyfikacja zmiennej rodzajowej</i>
$\langle K \rangle ::=$	K	<i>schemat rodzaju</i>
	$\Omega \hat{X}. \langle K \rangle$	<i>rodzaj schematu</i>
	$\Omega \bar{X}. \langle K \rangle$	<i>kwantyfikacja zmiennej rodzajowej</i>

Będziemy używać następującego cukru syntaktycznego:

$$\Omega \bar{X}_1 \dots \bar{X}_n \hat{X}_1 \dots \hat{X}_m . T \equiv \Omega \bar{X}_1 \dots \Omega \bar{X}_n . \Omega \hat{X}_1 \dots \Omega \hat{X}_m . T$$

I analogicznie dla rodzajów.

Teraz sam kontekst typowania ma postać:

$\Gamma ::=$	\emptyset
	$\Gamma, x : \langle T \rangle$
	$\Gamma, X :: \langle K \rangle$

4.2. β -unifikacja

Podczas rekonstrukcji typów pojawiają się równania więzów, które należy rozwiązać. Samo rozwiązanie sprowadza się do unifikacji pewnych termów, ale tutaj, ze względu na możliwość występowania funkcji typowych, termy równoważne nie muszą być równe, więc sama unifikacja powinna sprowadzać termy do β -równych sobie.

Taka unifikacja niesie ze sobą wiele problemów. Po pierwsze, będziemy chcieli używać podstawienia, będącego unifikatorem, na innych termach. To zaś grozi uzewnętrznieniem zmiennych związanych. Drugi problem jest taki, że samo podstawienie nie musi zrównywać termów, które da się zrównać, ale różnią się nazwami zmiennych związanych.

Z pierwszym problemem poradzimy sobie, traktując podstawienia jako funkcję częściową, tzn. samo podstawienie oprócz przyporządkowań postaci $[\bar{X} := T]$, może również zawierać podstawienia postaci $[\bar{X} := fail]$. Dodając operację anulowania podstawienia dostajemy:

$$[\bar{X} := T] \setminus X = \begin{cases} [\bar{X} := fail] & X \in Var(T) \\ [\bar{X} := T] & X \notin Var(T) \end{cases}$$

Z drugim problemem poradzimy sobie traktując unifikator jako parę zawierającą podstawienie, które dobrze działa na zewnątrz, oraz term będący wynikiem unifikacji.

Definicja 1. β -unifikatorem dla konstruktorów typów T_1 i T_2 nazwiemy taką parę (σ, S) , że istnieją podstawienia θ_1 i θ_2 takie, że:

$$\theta_1 T_1 =_\beta S =_\beta \theta_2 T_2$$

oraz

$$\theta_1 \setminus BTV(T_1) = \sigma = \theta_2 \setminus BTV(T_2)$$

Definicja 2. Powiemy, że β -unifikator (σ, S) jest *ogólniejszy* od β -unifikatora (θ, T) , jeżeli istnieje takie podstawienie ρ i przemianowanie α , że

$$\rho\alpha S =_{\beta} T \quad \text{oraz} \quad \rho\alpha\sigma = \theta$$

Definicja 3. *Najogólniejszym β -unifikatorem dla konstruktorów typów T_1 i T_2 nazwiemy taki β -unifikator, który jest ogólniejszy od wszystkich innych β -unifikatorów tychże konstruktorów typów.*

Lemat 1. *Niech A i B będą podzbiarami zbioru zmiennych typowych kwantyfikowanych abstrakcją, niech θ będzie podstawieniem. Wówczas zachodzi*

$$(\theta \setminus A) \setminus B = \theta \setminus (A \cup B).$$

Lemat 2. *Niech A będzie podzbiorem zbioru zmiennych typowych kwantyfikowanych abstrakcją, niech θ i ρ będą podstawieniami, oraz niech zachodzi $\text{Var}(\rho) \cap A = \emptyset$. Wówczas zachodzi*

$$(\rho \circ \theta) \setminus A = \rho \circ (\theta \setminus A)$$

.

Lemat 3. *Niech A będzie podzbiorem zbioru zmiennych typowych kwantyfikowanych abstrakcją, niech θ będzie dowolnym podstawieniem, niech ρ będzie podstawieniem takim, że $\text{Var}(\rho) \cap A = \emptyset$ (w szczególności zawierającym tylko zmienne rodzajowe). Wówczas zachodzi*

$$(\theta \circ \rho) \setminus A = (\theta \setminus A) \circ \rho$$

Lemat 4. *Niech (σ, S) będzie β -unifikatorem dla konstruktorów typów T_1 i T_2 , oraz niech ρ będzie podstawieniem takim, że $\text{Var}(\rho) \cap \text{BTV}(T_1) = \emptyset$ oraz $\text{Var}(\rho) \cap \text{BTV}(T_2) = \emptyset$. Wówczas $(\rho \circ \sigma, \rho S)$ również jest β -unifikatorem dla konstruktorów typów T_1 i T_2 .*

Dowód. Wiemy, że istnieją takie θ_1 i θ_2 , że

$$\theta_1 T_1 =_{\beta} S =_{\beta} \theta_2 T_2 \quad \text{oraz} \quad \theta_1 \setminus \text{BTV}(T_1) = \sigma = \theta_2 \setminus \text{BTV}(T_2)$$

A wtedy zachodzi

$$(\rho \circ \theta_1) T_1 = \rho(\theta_1 T_1) =_{\beta} \rho S =_{\beta} \rho(\theta_2 T_2) = (\rho \circ \theta_2) T_2$$

oraz z lematu 2

$$(\rho \circ \theta_1) \setminus \text{BTV}(T_1) = \rho \circ (\theta_1 \setminus \text{BTV}(T_1)) = \rho \circ \sigma$$

$$(\rho \circ \theta_2) \setminus \text{BTV}(T_2) = \rho \circ (\theta_2 \setminus \text{BTV}(T_2)) = \rho \circ \sigma$$

□

Lemat 5. *Niech T_1 i T_2 będą konstruktorami typów, ρ niech będzie podstawieniem takim, że $\text{Var}(\rho) \cap \text{BTV}(T_1) = \emptyset$ oraz $\text{Var}(\rho) \cap \text{BTV}(T_2) = \emptyset$, a (σ, S) niech będzie β -unifikatorem konstruktorów $(\rho T_1, \rho T_2)$. Wówczas $(\sigma \circ \rho, S)$ jest β -unifikatorem konstruktorów (T_1, T_2) .*

Dowód. Niech $i \in \{1, 2\}$.

Z tego, że (σ, S) jest β -unifikatorem konstruktorów $(\rho T_1, \rho T_2)$ wiemy, że istnieje takie θ_i , że

$$\theta_i \rho T_i =_\beta S \quad \text{oraz} \quad \theta_i \setminus BTV(\rho T_i) = \sigma$$

A wtedy

$$(\theta_i \circ \rho) T_i =_\beta S$$

oraz z lematu 3 i z tego że $BTV(\rho) \cap BTV(\theta_i) = \emptyset$ mamy

$$(\theta_i \circ \rho) \setminus BTV(T_i) = (\theta_i \circ BTV(T_i)) \circ \rho = (\theta_i \circ BTV(\rho T_i)) \circ \rho = \sigma \circ \rho$$

co kończy dowód. □

4.2.1. Algorytm β -unifikacji

Przy unifikacji wszystkie nieanotowane kwantyfikatory traktujemy jak anotowane unikatową zmienną. Funkcja *unify* to klasyczna unifikacja (na rodzajach).

```

unify $_{\beta}(T_1, T_2)$  when  $T_1 =_{\beta} T_2$  =
  ( $\square$ ,  $T_1$ )
unify $_{\beta}(\bar{X}, T)$  =
  if  $\bar{X} \in \text{Var}(T)$  then fail
  else ( $[\bar{X} := T]$ ,  $T$ )
unify $_{\beta}(T, \bar{X})$  =
  if  $\bar{X} \in \text{Var}(T)$  then fail
  else ( $[\bar{X} := T]$ ,  $T$ )
unify $_{\beta}(\forall X_1 :: K_1.T_1, \forall X_2 :: K_2.T_2)$  =
  let  $\sigma_K = \text{unify}(K_1, K_2)$  in
  let  $(\sigma_T, T) = \text{unify}_{\beta}(\sigma_K T_1, \sigma_K \{X_2 := X_1\} T_2)$  in
   $((\sigma_T \setminus X_1) \circ \sigma_K, \forall X_1 :: \sigma_T \sigma_K K_1.T)$ 
unify $_{\beta}(\lambda X_1 :: K_1.T_1, \lambda X_2 :: K_2.T_2)$  =
  let  $\sigma_K = \text{unify}(K_1, K_2)$  in
  let  $(\sigma_T, T) = \text{unify}_{\beta}(\sigma_K T_1, \sigma_K \{X_2 := X_1\} T_2)$  in
   $((\sigma_T \setminus X_1) \circ \sigma_K, \lambda X_1 :: \sigma_T \sigma_K K_1.T)$ 
unify $_{\beta}(T_1 \rightarrow S_1, T_2 \rightarrow S_2)$  =
  let  $(\theta, T) = \text{unify}_{\beta}(T_1, T_2)$  in
  let  $(\sigma, S) = \text{unify}_{\beta}(\theta S_1, \theta S_2)$  in
   $(\sigma \circ \theta, \sigma T \rightarrow S)$ 
unify $_{\beta}(T_1, T_2)$  when  $T_1 \rightarrow_{\beta} S_1$  =
  unify $_{\beta}(S_1, T_2)$ 
unify $_{\beta}(T_1, T_2)$  when  $T_2 \rightarrow_{\beta} S_2$  =
  unify $_{\beta}(T_1, S_2)$ 
unify $_{\beta}(V_1 T_1, V_2 T_2)$  =
  let  $(\theta, V) = \text{unify}_{\beta}(V_1, V_2)$  in
  let  $(\sigma, T) = \text{unify}_{\beta}(\theta T_1, \theta T_2)$  in
   $(\sigma \circ \theta, \sigma V T)$ 

```

4.2.2. Własności

Fakt 1. *Algorytm unify_β dla konstruktorów typów, dla których daje się określić rodzaj, zatrzymuje się.*

Dowód. Trywialne. Wynika z tego, że algorytm jest sterowany składnią, oraz z silnej normalizowalności konstruktorów typów. \square

Twierdzenie 1. *Niech T_1 oraz T_2 będą konstruktorami typów, które daje się orodzajować. Wówczas jeśli $\text{unify}_\beta(T_1, T_2) = (\sigma, S)$, to (σ, S) jest β -unifikatorem konstruktorów typów T_1 i T_2*

Dowód. Indukcja po głębokości rekursji.

Możliwe są następujące przypadki:

- $T_1 =_\beta T_2$ lub $T_1 \equiv \bar{X}$ lub $T_2 \equiv \bar{X}$
dla tych przypadków teza zachodzi trywialnie.
- $T_1 \equiv \forall X_1 :: K_1.T'_1$ oraz $T_2 \equiv \forall X_2 :: K_2.T'_2$
Wtedy z założenia indukcyjnego (σ_T, T) jest β -unifikatorem konstruktorów typów $\sigma_K T'_1$ oraz $\sigma_K \{X_2 := X_1\} T'_2$, gdzie σ_K jest unifikatorem rodzajów K_1 i K_2 . Istnieją więc takie podstawienia θ_1 i θ_2 , że zachodzą równości

$$\theta_1 \sigma_K T'_1 =_\beta T =_\beta \theta_2 \sigma_K \{X_2 := X_1\} T'_2$$

$$\theta_1 \setminus \text{BTV}(\sigma_K T'_1) = \sigma_T = \theta_2 \setminus \text{BTV}(\sigma_K \{X_2 := X_1\} T'_2)$$

A wtedy

$$\begin{aligned} & (\theta_1 \circ \sigma_K) \forall X_1 :: K_1.T'_1 = \forall X_1 :: \theta_1 \sigma_K K_1. \theta_1 \sigma_K T'_1 =_\beta \\ & =_\beta \forall X_1 :: (\theta_1 \setminus \text{BTV}(\sigma_K T'_1)) \sigma_K K_1.T = \forall X_1 :: \sigma_T \sigma_K K_1.T \end{aligned}$$

oraz

$$\begin{aligned} & (\theta_1 \circ \sigma_K) \setminus \text{BTV}(\forall X_1 :: K_1.T'_1) = (\theta_1 \setminus (\{X_1\} \cup \text{BTV}(\sigma_K T'_1))) \circ \sigma_K = \\ & = (\sigma_T \setminus X_1) \circ \sigma_K \end{aligned}$$

Analogicznie pokażemy pozostałe dwie równości dla konstruktora T_2 . Zatem $((\sigma_T \setminus X_1) \circ \sigma_K, \forall X_1 :: \sigma_T \sigma_K K_1.T)$ jest β -unifikatorem konstruktorów T_1 i T_2 .

- $T_1 \equiv \lambda X_1 :: K_1.T'_1$ oraz $T_2 \equiv \lambda X_2 :: K_2.T'_2$
Dla tego przypadku dowód przeprowadzamy analogicznie do przypadku poprzedniego.
- $T_1 \equiv T'_1 \rightarrow S_1$ oraz $T_2 \equiv T'_2 \rightarrow S_2$
Teza wynika wprost z założenia indukcyjnego oraz lematów 4 i 5.
- $T_1 \longrightarrow_\beta S_1$ lub $T_2 \longrightarrow_\beta S_2$
Wynika z założenia indukcyjnego i z faktu, że β -redukcja zachowuje β -równość.
- $T_1 \equiv V_1 S_1$ oraz $T_2 \equiv V_2 S_2$
Dowód przeprowadzamy analogicznie do przypadku z typem funkcji.

\square

Fakt 2. Algorytm $unify_\beta$ działający na konstruktorach typów nie zawierających zmiennych schematowych zwraca pusty unifikator jeśli konstruktory są β równe, w przeciwnym przypadku zawodzi (zawraca *fail*).

Fakt 3. Jeżeli z algorytmu usuniemy pierwszy przypadek sprawdzający β -równość konstruktorów, otrzymamy algorytm równoważny.

4.3. Rekonstrukcja typów

4.3.1. Algorytm W

Dla rodzajów:

$$\begin{aligned}
kindof(\Gamma \vdash X :: ?) &= \\
&\text{let } \Omega \hat{X}_1 \dots \hat{X}_n.K = \Gamma(X) \text{ in} \\
&\text{let } \hat{Y}_1, \dots, \hat{Y}_n = \text{fresh} \text{ in} \\
&\quad ([], [\hat{X}_1 := \hat{Y}_1, \dots, \hat{X}_n := \hat{Y}_n]K) \\
kindof(\Gamma \vdash T_1 \rightarrow T_2 :: ?) &= \\
&\text{let } (\sigma_1, K_1) = kindof(\Gamma \vdash T_1 :: ?) \text{ in} \\
&\text{let } (\sigma_2, K_2) = kindof(\sigma_1 \Gamma \vdash \sigma_1 T_1 :: ?) \text{ in} \\
&\text{let } \rho_1 = unify(\sigma_2 K_1, *) \text{ in} \\
&\text{let } \rho_2 = unify(\rho_1 K_2, *) \text{ in} \\
&\quad (\rho_2 \circ \rho_1 \circ \sigma_2 \circ \sigma_1, *) \\
kindof(\Gamma \vdash \forall X :: K.T :: ?) &= \\
&\text{let } (\sigma, K_T) = kindof(\Gamma, X :: K \vdash T :: ?) \text{ in} \\
&\text{let } \rho = unify(K_T, *) \text{ in} \\
&\quad (\rho \circ \sigma, *) \\
kindof(\Gamma \vdash \lambda X :: K.T :: ?) &= \\
&\text{let } (\sigma, K_T) = kindof(\Gamma, X :: K \vdash T :: ?) \text{ in} \\
&\quad (\sigma, K \Rightarrow K_T) \\
kindof(\Gamma \vdash T_1 T_2 :: ?) &= \\
&\text{let } (\sigma_1, K_1) = kindof(\Gamma \vdash T_1 :: ?) \text{ in} \\
&\text{let } (\sigma_2, K_2) = kindof(\sigma_1 \Gamma \vdash \sigma_1 T_2 :: ?) \text{ in} \\
&\text{let } \hat{X} = \text{fresh} \text{ in} \\
&\text{let } \sigma_3 = unify(\sigma_2 K_1, K_2 \Rightarrow \hat{X}) \text{ in} \\
&\quad (\sigma_3 \circ \sigma_2 \circ \sigma_1, \sigma_3 \hat{X}) \\
kindof(\Gamma \vdash \bar{X} :: ?) &= \\
&\quad ([], *)
\end{aligned}$$

Dla typów:

$$\begin{aligned}
typeof(\Gamma \vdash x :: ?) &= \\
&\text{let } \Omega \bar{X}_1 \dots \bar{X}_n \hat{X}_1 \dots \hat{X}_m.T = \Gamma(x) \text{ in} \\
&\text{let } \bar{Y}_1, \dots, \bar{Y}_n, \hat{Y}_1, \dots, \hat{Y}_m = \text{fresh} \text{ in} \\
&\quad ([], [\bar{X}_1 := \bar{Y}_1, \dots, \bar{X}_n := \bar{Y}_n, \hat{X}_1 := \hat{Y}_1, \dots, \hat{X}_m := \hat{Y}_m]T) \\
typeof(\Gamma \vdash \lambda x : T.t :: ?) &= \\
&\text{let } (\sigma_1, K) = kindof(\Gamma \vdash T :: ?) \text{ in} \\
&\text{let } \sigma_2 = unify(K, *) \text{ in} \\
&\text{let } (\sigma_3, T_t) = typeof(\sigma_2 \sigma_1(\Gamma, x : T) \vdash \sigma_2 \sigma_1 t :: ?) \text{ in} \\
&\quad (\sigma_3 \circ \sigma_2 \circ \sigma_1, \sigma_3 \sigma_2 \sigma_1 T \rightarrow T_t)
\end{aligned}$$

$$\begin{aligned}
& \text{typeof}(\Gamma \vdash t_1 t_2 :?) = \\
& \quad \text{let } (\sigma_1, T_1) = \text{typeof}(\Gamma \vdash t_1 :?) \text{ in} \\
& \quad \text{let } (\sigma_2, T_2) = \text{typeof}(\sigma_1 \Gamma \vdash \sigma_1 t_2 :?) \text{ in} \\
& \quad \text{let } \bar{X} = \text{fresh} \text{ in} \\
& \quad \text{let } (\sigma_3, T_3) = \text{unify}_\beta(\sigma_2 T_1, T_2 \rightarrow \bar{X}) \text{ in} \\
& \quad \quad (\sigma_3 \circ \sigma_2 \circ \sigma_1, \sigma_3 \bar{X}) \\
& \text{typeof}(\Gamma \vdash \lambda X :: K.t :?) = \\
& \quad \text{let } (\sigma, T) = \text{typeof}(\Gamma, X :: K \vdash t :?) \text{ in} \\
& \quad \quad (\sigma \setminus X, \forall X :: \sigma K.T) \\
& \text{typeof}(\Gamma \vdash t[T] :?) = \\
& \quad \text{let } (\sigma_1, T_t) = \text{typeof}(\Gamma \vdash t :?) \text{ in} \\
& \quad \text{let } (\sigma_2, K) = \text{kindof}(\sigma_1 \Gamma \vdash \sigma_1 T ::?) \text{ in} \\
& \quad \text{let } \bar{X} = \text{fresh} \text{ in} \\
& \quad \text{let } Y = \text{fresh} \text{ in} \\
& \quad \text{let } (\sigma_3, T'_t) = \text{unify}_\beta(\sigma_2 T_t, \forall Y :: K.\bar{X}) \text{ in} \\
& \quad \quad (\sigma_3 \circ \sigma_2 \circ \sigma_1, [Y := \sigma_3 \sigma_2 \sigma_1 T] \sigma_3 \bar{X}) \\
& \text{typeof}(\Gamma \vdash \text{let } x = t_1 \text{ in } t_2 :?) = \\
& \quad \text{let } (\sigma_1, T_1) = \text{typeof}(\Gamma \vdash t_1 :?) \text{ in} \\
& \quad \text{let } \{\bar{X}_1, \dots, \bar{X}_n\} = F\bar{T}V(T_1) \setminus F\bar{T}V(\Gamma) \text{ in} \\
& \quad \text{let } \{\hat{X}_1, \dots, \hat{X}_m\} = \widehat{FKV}(T_1) \setminus \widehat{FKV}(\Gamma) \text{ in} \\
& \quad \text{let } (\sigma_2, T_2) = \text{typeof}(\sigma_1 \Gamma, x : \Omega \bar{X}_1 \dots \bar{X}_n \hat{X}_1 \dots \hat{X}_m.T_2 \vdash \sigma_1 t_2 :?) \text{ in} \\
& \quad \quad (\sigma_2 \circ \sigma_1, T_2) \\
& \text{typeof}(\Gamma \vdash \text{tlet } X = T \text{ in } t :?) = \\
& \quad \text{let } (\sigma_1, K) = \text{kindof}(\Gamma \vdash T ::?) \text{ in} \\
& \quad \text{let } \{\hat{X}_1, \dots, \hat{X}_m\} = \widehat{FKV}(K) \setminus \widehat{FKV}(\Gamma) \text{ in} \\
& \quad \text{let } (\sigma_2, T_t) = \text{typeof}(\sigma_1 \Gamma, X :: \Omega \hat{X}_1 \dots \hat{X}_m.K \vdash \sigma_1 t :?) \text{ in} \\
& \quad \quad (\sigma_2 \circ \sigma_1, T_t)
\end{aligned}$$

5. Własności i dowody

5.1. Inne własności F_ω

Definicja 4. Reguły przepisywania typów w systemie F_ω w wersji Curry'ego standardowe, oprócz:

$$\frac{\Gamma \vdash M : \forall X \sigma}{\Gamma \vdash M : \text{nf}(\sigma[\bar{X} := \tau])}$$

Nierozstrzygalne są problemy:

- sprawdzania typu: dane Γ, M, τ , Czy $\Gamma \vdash M : \tau$
- typowalność: dane M , Czy $\exists \Gamma \tau. \Gamma \vdash M : \tau$

5.2. pare słów o rozszerzeniach

6. Praktyczne zastosowanie

7. Podsumowanie

Literatura

[1] Pierce,