

Metody Realizacji Języków Programowania

Analiza semantyczna II

Marcin Benke

MIM UW

29 listopada 2010

Analiza semantyczna

- Analiza nazw
- **Analiza zgodności typów**
- Identyfikacja operacji

Po co nam typy?

- *A type system is a syntactic method for automatically checking the absence of certain erroneous behaviors by classifying program phrases according to the kinds of values they compute.* — Benjamin Pierce.
- *Type structure is a syntactic discipline for enforcing levels of abstraction.* - John Reynolds. — J.C. Reynolds.
- Potrzeba rozróżnienia pomiędzy różnymi rodzajami obiektów; operacje wykonywane na napisach są inne od wykonywanych na liczbach.
- Typy pozwalają na uniknięcie pewnych błędów w programie, mogą zapewniać niezmienniki.
- Typy pozwalają na klasyfikację obiektów. Możliwość definiowania nowych typów zwiększa siłę wyrazu języka.

Po co nam typy?

Zależnie od systemu, kontrola typów może zapobiec

- Zastosowaniu funkcji do niewłaściwej liczby argumentów
- Zastosowaniu funkcji całkowanej do napisu
- Użyciu niezadeklarowanych zmiennych
- Użyciu niezainicjalizowanych zmiennych
- Funkcjom, które nie dają wyniku (a powinny)
- Dzieleniu przez zero
- Wyjściu poza zakres tablicy
- Algorytmom sortowania, które źle sortują
- ...

Dlaczego nie?

- W każdym (rozstrzygalnym) systemie typów są programy, które są dobrze zdefiniowane, ale nie są poprawne typowo, np
`length ["hello", 256, False]`
- Czasem jesteśmy zmuszeni napisać program większy lub wolniejszy niż byśmy chcieli aby dopasować go do systemu typów.

Typowanie dynamiczne

- Kontrola typów w czasie wykonania
- Daje programiście większą elastyczność, ale nie za darmo.
- Skrajnym przypadkiem jest assembler: bardzo elastyczny, ale żadnych zabezpieczeń.
- Pozwala na pisanie funkcji, które zachowują się różnie dla różnych typów wejścia, np:

```
def inv(x):  
    if type(x) == int:  
        return -x  
    elif type(x) == bool:  
        return not(x)  
    elif type(x) == string:  
        return reverse(x)  
    else:  
        return None
```

Typowanie dynamiczne

- Każda wartość niesie informację o swoim typie.
- Te informacje są przeważnie dostępne dla programisty (np **instanceof** w Javie).
- Każda operacja sprawdza typy swoich argumentów.
- Niezgodność typów \rightsquigarrow błąd wykonania (często: wyjątek).
- Przykłady języków: Lisp, Python, Smalltalk
- Wiele języków łączy typowanie statyczne i dynamiczne, np.
 - ▶ Haskell: `toDynamic :: Typable a => a -> Dynamic`
 - ▶ C#:

```
public Object[] DataRow.ItemArray{ get; set; }
int id = (int) row.ItemArray[0];
```

Typowanie statyczne

- Kontrola typów w czasie kompilacji.
- Niepoprawne typowo programy są odrzucane (przy typowaniu dynamicznym błędy typowe mogą zostać długo niewykryte).
- Mniejsza elastyczność (o ile — to zależy od systemu typów)
- Nowoczesne systemy typów dopuszczają:
 - ▶ przeciążanie
 - ▶ polimorfizm
 - ▶ kontrolowane typowanie dynamiczne
- Prawie takie same możliwości co przy typowaniu dynamicznym.
- Silne systemy typów pozwalają wyrazić niemal dowolne własności programów.

Silne i słabe typowanie

- Silne typowanie (kontroler typów ma władzę)
 - ▶ Niezgodność typów uniemożliwia uruchomienie programu
 - ▶ Gwarantuje bezpieczeństwo typowe w trakcie wykonania
 - ▶ Problemy gdy system typów nie jest zbyt ekspresywny (np. Pascal).
- Słabe typowanie (programista ma władzę)
 - ▶ Programista może obejść kontrolę typów
 - ▶ Żadnych gwarancji
 - ▶ Przykłady:
 - ★ Java: `(String) vector.get(1)`
 - ★ C: `(int *)123`

Przykłady typów i typowań

Asembler:

```
section .rodata
hello: db "Hello, World!", 10
```

C:

```
int scandir (
    const char *dir, struct dirent ***namelist,
    int (*selector) (struct dirent*),
    int (*cmp) (const void *, const void *))
```

C++:

```
const int*const Method3(const int*const&)const;
```

Java:

```
public interface Comparable {  
    public int compareTo(Object o);  
}
```

Haskell:

```
class (Eq a) => Ord a where  
    compare          :: a -> a -> Ordering  
    (<), (<=), (>=), (>) :: a -> a -> Bool
```

Alonzo:

```
sort : {a:Set} -> (o:Ord a) -> (xs:List a)  
      -> Sorted o xs
```

Systemy typów

- Przyjrzymy się teraz kilku systemom typów.
- System typów składa się z reguł postaci

$$\frac{A_1 \quad \dots \quad A_n}{B}$$

znaczących “jeśli A_1 i \dots i A_n to B ”.

Prosty system typów

Typy:

$$\tau ::= \mathbf{int} \mid \mathbf{bool}$$

Wyrażenia:

$$e ::= n \mid b \mid e_1 + e_2 \mid e_1 = e_2 \mid \mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2$$

Reguły:

$$\frac{}{n : \mathbf{int}} \quad \frac{}{b : \mathbf{bool}}$$

$$\frac{e_1 : \mathbf{int} \quad e_2 : \mathbf{int}}{e_1 + e_2 : \mathbf{int}} \quad \frac{e_1 : \mathbf{int} \quad e_2 : \mathbf{int}}{e_1 = e_2 : \mathbf{bool}}$$

$$\frac{e_0 : \mathbf{bool} \quad e_1 : \tau \quad e_2 : \tau}{\mathbf{if} \ e_0 \ \mathbf{then} \ e_1 \ \mathbf{else} \ e_2 : \tau}$$

Wyprowadzanie typów

Aby wykazać, że wyrażenie e ma typ τ możemy skonstruować *wyprowadzenie typu* (dowód w naszym systemie typów).

$$\frac{\frac{1 : \mathbf{int} \quad 2 : \mathbf{int}}{1 + 2 : \mathbf{int}} \quad 3 : \mathbf{int}}{(1 + 2) + 3 : \mathbf{int}}$$

$$\frac{\frac{1 : \mathbf{int} \quad 0 : \mathbf{int}}{1 = 0 : \mathbf{bool}} \quad 1 : \mathbf{int} \quad 2 : \mathbf{int}}{\mathbf{if } 1 = 0 \mathbf{ then } 1 \mathbf{ else } 2 : \mathbf{int}}$$

$$\frac{\frac{\mathbf{true} : \mathbf{bool} \quad \mathbf{false} : \mathbf{bool}}{\mathbf{if true then false else true} : \mathbf{bool}} \quad \frac{1 : \mathbf{int} \quad 2 : \mathbf{int}}{1 + 2 : \mathbf{int}} \quad 3 : \mathbf{int}}{\mathbf{if (if true then false else true) then } 1 + 2 \mathbf{ else } 0 : \mathbf{int}}$$

Zmienne

- Rozszerzmy nasz język o zmienne:

$$e ::= x \mid n \mid b \mid e_1 + e_2 \mid e_1 = e_2 \mid \text{if } e_0 \text{ then } e_1 \text{ else } e_2$$

- Typ zmiennej zależy od kontekstu, rozszerzymy zatem nasze reguły typowania o informacje o kontekście (środowisko).
- Będziemy używać notacji

$$\Gamma \vdash e : \tau$$

znaczącej “w środowisku Γ , wyrażenie e ma typ τ ”.

- Środowisko przypisuje zmiennym typy, tzn. jest zbiorem par $(x : \tau)$, gdzie x jest zmienną zaś τ typem.

Reguły typowania w kontekście

Stałe mają z góry ustalone typy:

$$\overline{\Gamma \vdash n : \mathbf{int}} \quad \overline{\Gamma \vdash b : \mathbf{bool}}$$

Typy zmiennych odczytujemy ze środowiska:

$$\overline{\Gamma(x : \tau) \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 + e_2 : \mathbf{int}} \quad \frac{\Gamma \vdash e_1 : \mathbf{int} \quad \Gamma \vdash e_2 : \mathbf{int}}{\Gamma \vdash e_1 = e_2 : \mathbf{bool}}$$

$$\frac{\Gamma \vdash e_0 : \mathbf{bool} \quad \Gamma \vdash e_1 : \tau \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \mathbf{if } e_0 \mathbf{ then } e_1 \mathbf{ else } e_2 : \tau}$$

Kontrola typów w językach imperatywnych

Rozważmy mały język imperatywny:

$$e ::= x \mid n \mid b \mid e_1 + e_2 \mid e_1 = e_2$$
$$s ::= x := e \mid \mathbf{while} \ e \ \mathbf{do} \ s \mid s; s$$

Wprowadzimy nowy osąd dla programów

$$\Gamma \vdash_P s$$

o znaczeniu “w środowisku Γ , program s jest poprawny. Niektóre reguły będą używać zarówno \vdash jak \vdash_P , np.

$$\frac{\Gamma \vdash x : \tau \quad \Gamma \vdash e : \tau}{\Gamma \vdash_P x := e}$$

czy

$$\frac{\Gamma \vdash_E e : \text{bool} \quad \Gamma \vdash_P p}{\Gamma \vdash_P \mathbf{while} \ e \ \mathbf{do} \ p}$$

Deklaracje

Możemy uznać deklarację jako rodzaj instrukcji oraz regułę

$$\frac{\Gamma(x : \tau) \vdash_P p}{\Gamma \vdash_P \mathbf{var} x : \tau; p}$$

inną możliwością jest wprowadzenie nowego typu osądu, \vdash_D :

$$\Gamma \vdash_D (\mathbf{var} x : \tau) : \Gamma(x : \tau)$$

$$\frac{\Gamma \vdash_D ds : \Gamma' \quad \Gamma' \vdash_P p}{\Gamma \vdash_P ds; p}$$

Można też pozwolić instrukcjom na modyfikację środowiska.
Deklaracje i instrukcje mogą być wtedy swobodnie przeplatane:

$$\frac{\Gamma \vdash_P s : \Gamma' \quad \Gamma' \vdash_P p : \Gamma''}{\Gamma \vdash_P s; p : \Gamma''}$$

Kontrola typów w językach funkcyjnych

Typy:

$$\tau ::= \text{int} \mid \text{bool} \mid \tau_1 \rightarrow \tau_2$$

Wyrażenia:

$$E ::= x \mid n \mid b \mid e_1 e_2 \mid \lambda(x:\tau).e \mid e_1 + e_2 \mid e_1 = e_2 \mid \\ \text{if } e_0 \text{ then } e_1 \text{ else } e_2$$

Reguły typowania

$$\frac{\Gamma(x:\tau) \vdash e : \rho}{\Gamma \vdash \lambda(x:\tau).e : \tau \rightarrow \rho}$$
$$\frac{\Gamma \vdash e_1 : \tau \rightarrow \rho \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \rho}$$

Przykłady

$$\frac{\frac{x : \text{int} \vdash x : \text{int} \quad x : \text{int} \vdash 1 : \text{int}}{x : \text{int} \vdash x + 1 : \text{int}}}{\vdash \lambda(x : \text{int}).x + 1 : \text{int} \rightarrow \text{int}} \quad \vdash 7 : \text{int}$$
$$\frac{}{\vdash (\lambda(x : \text{int}).x + 1) 7 : \text{int}}$$

$$\frac{x : \text{int}, y : \text{int} \vdash x : \text{int} \quad x : \text{int}, y : \text{int} \vdash y : \text{int}}{x : \text{int}, y : \text{int} \vdash x + y : \text{int}}$$
$$\frac{x : \text{int} \vdash \lambda(y : \text{int}).x + y : \text{int} \rightarrow \text{int}}{\vdash \lambda(x : \text{int}).\lambda(y : \text{int}).x + y : \text{int} \rightarrow \text{int} \rightarrow \text{int}}$$

Uwaga: ze względów praktycznych, wygodnie jest przepisać regułę aplikacji tak:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

Reguły typowania są sterowane składnią (tzn. jest dokładnie jedna reguła dla każdej produkcji). Możemy zatem łatwo je zaprogramować:

```
check env (EInt n) = return TInt
check env (EBool b) = return TBool
check env (EVar x) = lookup x env
check env (ELam v t e) = do
    t' <- check ((v,t):env) e
    return (t :-> t')
check env (EApp e1 e2) = do
    (t1 :-> t) <- check env e1
    t2 <- check env e2
    guard (t1 == t2)
    return t
check env (EPlus e1 e2) = do
    TInt <- check env e1
    TInt <- check env e2
    return TInt
```

Rekonstrukcja typów

- Jeśli typy identyfikatorów nie są znane, musimy *zrekonstruować* pasujące typy.
- Reguły typowania pozostają te same; reguła dla funkcji odpowiada zmienionej składni:

$$\frac{\Gamma(x : \tau) \vdash e : \rho}{\Gamma \vdash \lambda x. e : \tau \rightarrow \rho}$$

co prowadzi do problemu: skąd wziąć dobre τ ?

- Możemy go rozwiązać, czyniąc τ niewiadomą (zmienną typową). Proces typowania da nam typ wraz z układem ograniczeń (w tym przypadku równań)
- Przy każdym użyciu reguły aplikacji

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

dodajemy do układu równanie $\tau_1 = \tau_2$.

Przykłady rekonstrukcji typów

Możemy podobnie jak w Haskellu traktować $a + b$ jako aplikację $(+)$ a b .

$$\frac{\frac{x : \tau_x \vdash x : \tau_x \quad x : \tau_x \vdash 1 : \text{int}}{x : \tau_x \vdash x + 1 : \text{int}} \quad \{\tau_x = \text{int}\}}{\vdash \lambda x. x + 1 : \tau_x \rightarrow \text{int}} \quad \vdash 7 : \text{int}}{\vdash (\lambda x. x + 1) 7 : \text{int}}$$

z niemal trywialnym układem równań $\{\tau_x = \text{int}\}$.

Przykłady rekonstrukcji typów

Podobnie możemy uzyskać

$$\vdash (\lambda f. \lambda x. f(fx))(\lambda y. y) \ 7 : \tau'_f$$

Z równaniami:

$$\tau_f = \tau_x \rightarrow \tau'_f \tag{1}$$

$$\tau_f = \tau'_f \rightarrow \tau'_f \tag{2}$$

$$\tau_f \rightarrow (\tau_x \rightarrow \tau'_f) = (\tau_y \rightarrow \tau_y) \rightarrow (\tau_x \rightarrow \tau'_f) \tag{3}$$

$$\tau_x \rightarrow \tau'_f = \text{int} \rightarrow \tau'_f \tag{4}$$

Rozwiązywanie równań: unifikacja

Otrzymane układy możemy rozwiązywać niemal tak samo jak każde inne: przez upraszczanie.

W naszym przykładzie możemy uprościć równanie (4)

$$\tau_x \rightarrow \tau'_f = \text{int} \rightarrow \tau'_f$$

do

$$\tau_x = \text{int}$$

i podstawić int za x w pozostałych, otrzymując

$$\tau_f = \text{int} \rightarrow \tau'_f$$

$$\tau_f = \tau'_f \rightarrow \tau'_f$$

$$\tau_f \rightarrow (\text{int} \rightarrow \tau'_f) = (\tau_y \rightarrow \tau_y) \rightarrow (\text{int} \rightarrow \tau'_f)$$

$$\tau_x = \text{int}$$

$$\tau_f = \text{int} \rightarrow \tau'_f \quad (5)$$

$$\tau_f = \tau'_f \rightarrow \tau'_f \quad (6)$$

$$\tau_f \rightarrow (\text{int} \rightarrow \tau'_f) = (\tau_y \rightarrow \tau_y) \rightarrow (\text{int} \rightarrow \tau'_f) \quad (7)$$

$$\tau_x = \text{int} \quad (8)$$

Dalej możemy połączyć (5) z (6) otrzymując

$$\text{int} \rightarrow \tau'_f = \tau'_f \rightarrow \tau'_f$$

co może być uproszczone do

$$\tau'_f = \text{int}.$$

Po podstawieniu int za τ'_f , mamy

$$\tau_f = \text{int} \rightarrow \text{int} \quad (9)$$

$$\tau'_f = \text{int} \quad (10)$$

$$\tau_f \rightarrow (\text{int} \rightarrow \text{int}) = (\tau_y \rightarrow \tau_y) \rightarrow (\text{int} \rightarrow \text{int}) \quad (11)$$

$$\tau_x = \text{int} \quad (12)$$

Upraszczając (11) i podstawiając τ_f mamy

$$\text{int} \rightarrow \text{int} = \tau_y \rightarrow \tau_y$$

skąd ostatecznie

$$\tau_f = \text{int} \rightarrow \text{int} \quad (13)$$

$$\tau'_f = \text{int} \quad (14)$$

$$\tau_y = \text{int} \quad (15)$$

$$\tau_x = \text{int} \quad (16)$$

Opisany proces rozwiązywania równań nazywamy *unifikacją*. W przypadku sukcesu wynikiem jest *podstawienie*.

Fakt: unifikacja może być zastosowana do rozwiązywania równań na termach nad dowolną sygnaturą. Rozstrzygalna w czasie liniowym.

Kiedy unifikacja zawodzi

Unifikacja zawodzi, gdy napotka jedno z poniższych:

- Równanie postaci:

$$k_1 = k_2$$

gdzie k_1 i k_2 są różnymi stałymi

- Równanie postaci (k — stała):

$$k = t \rightarrow t'$$

- Równanie postaci

$$x = t$$

gdzie x — zmienna, zaś t zawiera x ale różny od x .

Na przykład, próba wyprowadzenia typu dla $\lambda x.xx$ prowadzi do

$$\tau_x = \tau_x \rightarrow \rho.$$

Ten term nie jest typowalny (w naszym systemie).

Polimorfizm

Z drugiej strony, układ równań może mieć więcej niż jedno rozwiązanie. W efekcie możemy wyprowadzić więcej niż jeden typ dla danego wyrażenia. Na przykład, mamy

$$\vdash \lambda x. x : \tau \rightarrow \tau$$

dla każdego typu τ !

Dla opisu tego zjawiska możemy wprowadzić nową postać typu: $\forall \alpha. \tau$, gdzie α jest zmienną typową, oraz dwie nowe reguły:

$$\frac{\Gamma \vdash e : \tau}{\Gamma \vdash e : \forall \alpha. \tau} \quad \alpha \notin FV(\Gamma) \qquad \frac{\Gamma \vdash e : \forall \alpha. \tau}{\Gamma \vdash e : \tau[\rho/\alpha]}$$

$(\tau[\rho/\alpha])$ oznacza typ τ z ρ podstawionym za α .

Polimorfizm — przykłady i smutna konstatacja

Możemy wyprowadzić

$$\vdash \lambda x.x : \forall \alpha. \alpha \rightarrow \alpha$$

Także $\lambda x.xx$ staje się typowalne:

$$\vdash \lambda x.xx : \forall \beta (\forall \alpha. \alpha \rightarrow \alpha) \rightarrow \beta \rightarrow \beta$$

Niestety nowy system nie jest już sterowany składnią: nowe reguły nie odpowiadają żadnym konstrukcjom składniowym i nie wiemy kiedy je stosować. Okazuje się, że rekonstrukcja w tym systemie jest **nierozstrzygalna**.

Płytki polimorfizm

Rekonstrukcja typów jest rozstrzygalna jeśli wprowadzimy pewne ograniczenie: kwantyfikatory są dopuszczalne tylko na najwyższym poziomie oraz mamy specjalną składnię dla wiązań polimorficznych:

$$\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma(x : \forall \vec{\alpha}. \tau_1) \vdash e : \tau}{\Gamma \vdash \mathbf{let} \ x = e_1 \ \mathbf{in} \ e : \tau}$$

Taki system jest często wystarczający w praktyce. Na przykład możemy zastąpić konstrukcję **if** funkcją

$$\mathit{if_then_else_} : \forall \alpha. \mathit{bool} \rightarrow \alpha \rightarrow \alpha \rightarrow \alpha$$

Jest on również podstawą systemów dla ML i Haskellu (choć ten ostatni jest znacznie bardziej skomplikowany).

Podtypy

Jeśli klasa B dziedziczy po C , każdy obiekt klasy B może być użyty w miejscu, gdzie spodziewany jest obiekt klasy C .

Można to sformalizować przy pomocy pojęcia *podtypu* (podobnego do pojęcia podzbioru):

$$\frac{\Gamma \vdash e : B \quad B \leq C}{\Gamma \vdash e : C}$$

Można też przepisać regułę aplikacji tak:

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leq \tau_1}{\Gamma \vdash e_1 e_2 : \tau}$$

Przy sprawdzaniu typów przy każdym użyciu reguły aplikacji sprawdzamy, że nierówność $\tau_1 \leq \tau_2$ zachodzi, przy rekonstrukcji dodajemy do układu nierówność do zbioru ograniczeń (i w efekcie rozwiązujemy układy nierówności zamiast równań).

Przeciążanie

- Funkcje polimorficzne działają w ten sam sposób niezależnie od typu argumentów.
- Przeciążanie z kolei oznacza, że jeden symbol funkcyjny (lub operator) oznacza różne funkcje dla różnych typów argumentów.
- Podczas kontroli typów przeciążone symbole są zastępowane przez ich warianty odpowiednie dla typów argumentów.
- W systemie typów możemy to wyrazić następująco:

$$\Gamma \vdash e \rightsquigarrow e' : \tau$$

co oznacza “w środowisku Γ , wyrażenie e ma typ τ i jest przekształcane do e' ”.

Równość

- Nawet w językach, które nie wspierają przeciążania jawnie, operator równości jest w istocie przeciążony.
- W istocie na przykład równość dla napisów musi być zrealizowana inaczej niż dla liczb.
- W naszym języku możemy dopuścić równość dla typów `int` i `bool` i dodać następujące reguły transformacji:

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : \text{int} \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : \text{int}}{\Gamma \vdash e_1 = e_2 \rightsquigarrow \text{eqInt } e'_1 e'_2 : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : \text{bool} \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : \text{bool}}{\Gamma \vdash e_1 = e_2 \rightsquigarrow \text{eqBool } e'_1 e'_2 : \text{bool}}$$

gdzie `eqInt` i `eqBool` są wbudowanymi operacjami równości dla odpowiednich typów.

Konwersje typów

Czasami (zwłaszcza dla typów numerycznych) zachodzi potrzeba konwersji — zamiany wartości jednego typu na odpowiadającą mu wartość innego typu, np:

```
int r = 20000;  
int x = int(3.14159 * double(r));
```

NB wartości typu `int` są reprezentowane inaczej niż `double` i konwersja musi być rzeczywiście dokonana w czasie wykonania programu.

Niektóre języki wstawiają konwersje (zwane wtedy czasem koercjami) automatycznie, pozwalając pisać

```
int x = 3.14159 * r;
```

Takie wstawianie koercji możemy zrealizować np

$$\frac{\Gamma \vdash e_1 \rightsquigarrow e'_1 : \mathbf{int} \quad \Gamma \vdash e_2 \rightsquigarrow e'_2 : \mathbf{double}}{\Gamma \vdash e_1 + e_2 \rightsquigarrow \mathit{int2double}(e_1)' + e'_2 : \mathbf{double}}$$

I to by było na tyle

Bonus

$$\frac{f:\tau_f \vdash f:\tau_f \quad x:\tau_x \vdash x:\tau_x}{f:\tau_f, x:\tau_x \vdash fx:\tau'_f} \quad (1)$$

$$\frac{f:\tau_f, x:\tau_x \vdash f:\tau_f \quad f:\tau_f, x:\tau_x \vdash fx:\tau'_f}{f:\tau_f, x:\tau_x \vdash f(fx):\tau'_f} \quad (2)$$

$$\frac{f:\tau_f \vdash \lambda x.f(fx) : \tau_x \rightarrow \tau'_f}{\vdash \lambda f.\lambda x.f(fx) : \tau_f \rightarrow \tau_x \rightarrow \tau'_f} \quad \frac{y:\tau_y \vdash y:\tau_y}{\vdash \lambda y.y : \tau_y \rightarrow \tau_y} \quad (3)$$

$$\frac{\vdash (\lambda f.\lambda x.f(fx))(\lambda y.y) : \tau_x \rightarrow \tau'_f \quad \vdash 7 : \text{int}}{\vdash (\lambda f.\lambda x.f(fx))(\lambda y.y) 7 : \tau'_f} \quad (4)$$

Z równaniami:

$$\tau_f = \tau_x \rightarrow \tau'_f \quad (17)$$

$$\tau_f = \tau'_f \rightarrow \tau'_f \quad (18)$$

$$\tau_f \rightarrow (\tau_x \rightarrow \tau'_f) = (\tau_y \rightarrow \tau_y) \rightarrow (\tau_x \rightarrow \tau'_f) \quad (19)$$

$$\tau_x \rightarrow \tau'_f = \text{int} \rightarrow \tau'_f \quad (20)$$