Metody Realizacji Języków Programowania

Analiza semantyczna

Marcin Benke

15 października 2018

Analiza semantyczna

- Analiza nazw
  - Czy x jest zadeklarowane przed użyciem?
  - Która deklaracja x obowiazuje w danym miejscu programu?
  - Czy jakieś nazwy są zadeklarowane a nie używane?
- Analiza zgodności typów
  - Czy wyrażenie e jest poprawne typowo?
  - Jakiego typu jest e?
  - Czy funkcja zawsze zwraca wartość typu zgodnego z zadeklarowanym?
- Identyfikacja operacji
  - Jaką operację reprezentuje $+$ wyrażeniu $a + b$?

Odpowiedzi na te pytania mogą wymagać informacji nielokalnych — *kontekstowych*. Nie są to własności bezkontekstowe.

Czy analiza semantyczna jest potrzebna?


Create account Log in

WIKIPEDIA
The Free Encyclopedia

Main page
Contents
Featured content
Current events
Random article
Donate to Wikipedia
Wikipedia store

Interaction
Help
About Wikipedia
Community portal
Recent changes
Contact page

Tools
What links here

Article Talk Read Edit View history Search

Dynamic programming language

From Wikipedia, the free encyclopedia

Script error: The module returned a value. It is supposed to return an export table. Script error: The module returned a value. It is supposed to return an export table.

Dynamic programming language is a term used in computer science to describe a class of high-level programming languages which, at runtime, execute many common programming behaviors that static programming languages perform during compilation. These behaviors could include extension of the program, by adding new code, by extending objects and definitions, or by modifying the type system. Although similar behaviours can be emulated in nearly any language, with varying degrees of difficulty, complexity and perfomance costs, dynamic languages provide direct tools to make use of them. Many of these features were first implemented as native features in the Lisp programming language.

Most dynamic languages are also dynamically typed, but not all are. Dynamic languages are frequently (but not always) referred to as "scripting languages", although the term "scripting language" in its narrowest sense refers to languages specific to a given run-time environment.

Contents [hide]
1 Examples
2 Implementation

Gramatyki atrybutywne

Wygodnym narzędziem opisu reguł kontekstowych są *gramatyki atrybutywne*

Gramatyka atrybutywna

$$AG = \langle G, A, R \rangle$$

G — gramatyka bezkontekstowa, A — zbiór atrybutów,
R — zbiór reguł atrybutowania

Niech A(X) — zbiór atrybutów symbolu X; X.a oznacza atrybut a symbolu X.

Dla produkcji $p : X_0 \to X_1 \dots X_n$ definiujemy reguły atrybutowania

$$R(p) = \{X_i.a \leftarrow f_{i,a}(X_j.b \dots X_k.c) \mid 0 \leqslant i \leqslant n, a \in A(X_i)\}$$

Well defined Attribute Grammar

Mając drzewo struktury chcemy dla każdego wierzchołka X wyznaczyć wartości wszystkich atrybutów zgodnie z regułami atrybutowania.

Definicja (WAG)

Gramatyka atrybutywna jest **dobrze zdefiniowana** jeśli dla każdego drzewa struktury zgodnego z tą gramatyką można w sposób jednoznaczny wyznaczyć wartości wszystkich atrybutów.

Nieważne "jak", ważne, że "można".
Zagrożenia: brak reguły, sprzeczne reguły, cykl

Atrybuty syntetyzowane i dziedziczone

Dla produkcji $p : X_0 \to X_1 \dots X_n$ zbiorem **definiujących wystąpień atrybutów** jest

$$AF(p) = \{X_i.a \mid X_i.a \leftarrow f(\cdots) \in R(p)\}$$

- Atrybut $X.a$ jest **syntetyzowany**, jeśli istnieje produkcja $p : X \to \alpha$ i $X.a \in AF(p)$ (czyli zależy od poddrzewa)

- Atrybut $X.a$ jest **dziedziczony**, jeśli istnieje produkcja $q : Y \to \alpha X \beta$ i $X.a \in AF(q)$ (czyli zależy od otoczenia)

Oznaczenia: $AS(X)$ — atrybuty syntetyzowane X, $AI(X)$ — atrybuty dziedziczone X.

Dla symboli terminalnych mówimy o **atrybutach wbudowanych**.

Przykład — atrybut syntetyzowany

Konwencja: jeśli dany symbol występuje więcej niż raz w danej produkcji, jego wystąpienia numerujemy.

Atrybuty: $E.val$ — syntetyzowane, $n.val$ — wbudowany

$E_0 \to E_1{+}E_2 \,\{E_0.val \leftarrow E_1.val{+}E_2.val\}$ $E_0 \to E_1{*}E_2\{E_0.val \leftarrow E_1.val{*}E_2.val\}$ $E_0 \to n\,\{E_0.val \leftarrow n.val\}$

Zadanie programistyczne 1

Dana lista liczb. Obliczyć jej średnią.

```
-- avg xs = sum xs / length xs

avg xs = sum / len where
  (sum,len) = foldr cons (0,0) xs
  cons x (sum',len') = (x+sum',1+len')

foldr :: (a → b → b) → b → [a] → b
foldr cons nil = go where
  go []     = nil
  go (x:xs) = cons x (go xs)
```

Przykład — atrybut syntetyzowany

Konwencja: jeśli dany symbol występuje więcej niż raz w danej produkcji, jego wystąpienia numerujemy.

Atrybuty: $L.len$, $L.sum$ — syntetyzowane, $num.val$ — wbudowany

$$
\begin{array}{ll}
P \to L & \{P.avg = L.sum/L.len\} \\
L_0 \to nil & \{L_0.len \leftarrow 0,\ L_0.sum \leftarrow 0\} \\
L_0 \to n : L_1 & \{L_0.len \leftarrow 1 + L_1.len,\ L_0.sum = n.val + L_1.sum\}
\end{array}
$$

```
avg :: [Float] → Float
avg xs = sum / len where
 (sum,len) = foldr cons nil xs
 nil = (0.0, 0.0)
 cons x (sum',len') = (x+sum', 1+len')
```

Zadanie programistyczne 2

Dana lista liczb. Obliczyć jej wariancję.

```
var xs = foldr cons 0 xs / length xs where
  cons x xs = (x - listavg)**2
  listavg = avg xs
```

Jak to zrobić za jednym przejściem listy?

Przykład — atrybut dziedziczony

Do dotychczasowych reguł dodajemy: $L.avg$ — atrybut dziedziczony $L.var$, $P.var$ — atrybuty syntetyzowane

$$
\begin{array}{ll}
P \to L & \{P.var = L.var,\ L.avg = P.avg\} \\
L_0 \to nil & \{L_0.len \leftarrow 0,\ L_0.sum \leftarrow 0\} \\
L_0 \to n : L_1 & \{L_1.avg \leftarrow L_0.avg,\ L_0.var \leftarrow (n.val - L_0.avg)^2 + L_1.var\}
\end{array}
$$

```
var :: [Float] → Float
var xs = d2 / len where
  (sum,len,d2) = foldr cons nil xs listavg
  listavg = sum / len
  nil avg = (0.0,0,0.0)
  cons x fs avg = let
     (s,l,d2) = fs avg
   in (s+x,l+1,(x-avg)**2+d2)x
```

Przykład — atrybut dziedziczony

$D \to TL\,\{L.typ \leftarrow D.typ;\ D.typ \leftarrow T.typ\}$ $T \to \text{int}\,\{T.typ \leftarrow \text{int}\}$ $T \to \text{real}\,\{T.typ \leftarrow \text{real}\}$ $L_0 \to L_1, \text{id}\ \{L_1.typ \leftarrow L_0.typ,\ id.typ \leftarrow L_0.typ\}$ $L \to \text{id}\ \{id.typ \leftarrow L.typ\}$

Atrybuty:

- $T.typ$, $D.typ$ — syntetyzowany
- $L.typ$ — dziedziczony
- $id.typ$ — dziedziczony

Gramatyki zupełne

Gramatyka jest zupełna, jeśli dla każdego symbolu X spełnione są warunki:

1. dla każdej produkcji $p : X \to \alpha$ mamy $AS(X) \subseteq AF(p)$,
2. dla każdej produkcji $q : Y \to \alpha X \beta$ mamy $AI(X) \subseteq AF(q)$,
3. $AS(X) \cup AI(X) = A(X)$,
4. $AS(X) \cap AI(X) = \emptyset$.

Katamorfizmy

- `foldr` można uogólnić na wszelkie drzewa struktury; w ogólności mówimy o *katamorfizmach*
- na przykład

```
data Tree a = Leaf | Br (Tree a) a (Tree a)

foldTree :: (b→a→b→b) → b → Tree a → b
foldTree bf lf = go where
  go Leaf       = lf
  go (Br l x r) = bf (go l) x (go r)

count :: Tree a → Int
count = foldTree (λx y z → x + 1 + z) 0
```

- w języku leniwym atrybuty można zwykle wyliczyć przy pomocy katamorfizmu gdzie

$$b = (Inh_1 \to Inh_2 \to \cdots \to Inh_m \to (Syn_1, \ldots, Syn_n))$$

Łatwe klasy gramatyk dobrze zdefiniowanych

Gramatyka S-atrybutowana:

- wszystkie atrybuty są syntetyzowane
- wyliczanie atrybutów od liści do korzenia — dobrze łączy się z analizą wstępującą

Gramatyka L-atrybutowana:

- atrybuty moga być syntetyzowane bądź dziedziczone
- atrybuty dziedziczone zależą tylko od rodzica i rodzeństwa na lewo
- można wyliczyć przechodząc drzewo struktury DFS

Tablica symboli

- Opis wszystkich bytów (zmienych, funkcji, typów, klas, atrybutów, metod,...) wystepujących w programie.
- Musi mieć narzuconą strukturę (mechanizm wyszukiwania), odzwierciedlającą reguły wiązania identyfikatorów w danym języku.
- Opis bytu:
  - rodzaj definicji
  - inne informacje zależne od rodzaju
- Byty mogą być wzajemnie powiązane.

Struktura języka a struktura tablicy symboli

Niektóre konstrukcje językowe narzucające strukturę tablicy symboli:

- zagnieżdzanie (struktura blokowa)
- dziedziczenie
- sumowanie (moduły)

```
import java.util.*
class A {
  int a,b;
  class B extends A {
    B() {}
    int f() {
      String a;
    }
}
```

Zasięg i zakres

Zasięg definicji identyfikatora to obszar programu, w którym możemy użyć identyfikatora w zdefiniowanym znaczeniu. Nie musi być ciągły.

Zakres to konstrukcja składniowa, z którą mogą być związane definicje identyfikatorów (funkcja, blok, itp.)

Przykład

```
void f() {
   int a;
   a = g();
   {
     string a;
     b = a;
   }
   h(a,b);
}
```

Zasięg deklaracji `int a` jest zaznaczony na czerwono. Jest ona związana z zakresem funkcji f.

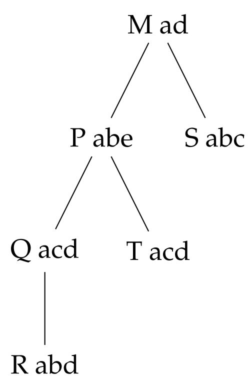
Struktura blokowa

```
module M;
  var a,d : int;
  type t = ...
```

```
procedure P;
  var a,b,e : int
  procedure Q;
    var a,c,d : t;
    procedure R;
      var a,b,d : real;
  end
  procedure T;
    var a,c,d : int;
  end
procedure S;
  var a,b,c : int;
  d := e+1; // Gdzie są definicje d i e?
...
```

Drzewo zagnieżdżeń

Problem: analizujemy węzeł drzewa struktury, np przypisanie `d:=e+1`. Gdzie są definicje `d` i `e`?



Metoda I: stos tablic symboli

Wyszukiwanie:

- przeszukaj zakresy od bieżącego do znalezienia lub do końca,
- jeżeli nie znaleziono, to dodaj fikcyjną definicję dla uniknięcia kaskady błędów.

Wejście do zakresu:

- połóż na stos nową tablicę symboli,
- umieść w niej definicje związane z tym zakresem

Wyjście z zakresu:

- zdejmij ze stosu ostatnią tablicę symboli

Metoda II: tablica stosów

Dla każdego identyfikatora tworzymy osobny stos odwołań do jego definicji

Niezmiennik: w trakcie analizy, dla każdego identyfikatora na szczycie stosu jest odsyłacz do aktualnej definicji (lub stos pusty).

Wejście do zakresu: przechodzimy listę definicji związanych z zakresem i wkładamy odsyłacze do nich na odpowiednie stosy.

Wyjście z zakresu: przechodzimy ponownie listę definicji i zdejmujemy odsyłacze ze stosów.

W porównaniu z Metodą I nieco więcej pracy na granicach zakresów, ale za to szybsze wyszukiwanie.

Zagadka

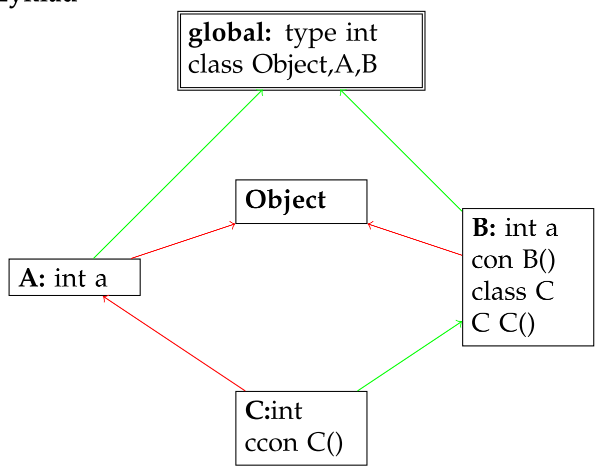
```
class A {
char a;
A() { a = 'A';}
}
class B {
    char a;
    B() { a = 'B'; }
    class C extends A {
        public char c;
        C() { c = a; }
    }
    C C() { return new C(); }
}
...
B b = new B(); B.C c = b.C();
```

Jaką wartość ma `c.c` ?

Dziedziczenie

- Jak widać z powyższego przykładu, dziedziczenie nieco komplikuje wyszukiwanie.
- Przy pojedynczym dziedziczeniu możemy przy wchodzeniu do zakresu podklasy wkładać na stos(y) definicje z nadklasy.
- Innym rozwiązaniem jest modyfikacja metody I: zamiast stosu - graf acykliczny tablic symboli.
- Każda tablica ma dowiązanie do tablic ewentualnych nadklas i zakresu obejmującego.

Przykład



Java odwiedza najpierw czerwoną krawędź.

Systemy typów

System typów — zbiór typów i reguł wnioskowania o typach

Reguły są zwykle wyrażane w postaci

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

oznaczającej "jeśli A_1 i ... i A_n to możemy wnioskować B".

Używamy też notacji

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

znaczącej "w środowisku Γ, wyrażenie e ma typ τ".

Środowisko przypisuje zmiennym typy, tzn. jest zbiorem par $(x : \tau)$.

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:

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

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

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

Wyprowadzanie typów

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

$$\frac{\dfrac{1 : \textbf{int} \quad 2 : \textbf{int}}{1 + 2 : \textbf{int}} \qquad 3 : \textbf{int}}{(1+2) + 3 : \textbf{int}}$$

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

Wyprowadzanie typów działa w tym przypadku od liści do korzenia.

Zmienne

Rozszerzmy nasz język o zmienne:

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

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:

$$\frac{}{\Gamma \vdash n : \textbf{int}} \qquad \frac{}{\Gamma \vdash b : \textbf{bool}}$$

Typy zmiennych odczytujemy ze środowiska:

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

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

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

Kierunki przepływu danych

Synteza typów:

- jaki typ ma dane wyrażenie: $\vdash e : ?$
- dane płyną w górę drzewa
- skojarzenie: atrybut syntetyzowany

$$\Gamma \vdash e \Rightarrow t$$

Kontrola typów:

- czy wyrażenie ma dany typ t?
- dane płyną w dół drzewa
- skojarzenie: atrybut dziedziczony

$$\Gamma \vdash e \Leftarrow t$$

Kierunki przepływu danych

W miarę potrzeby możemy łączyć te dwa kierunki

Przejście od syntezy do kontroli jest trywialne:

$$\frac{\Gamma \vdash e \Rightarrow t}{\Gamma \vdash e \Leftarrow t}$$

W drugą stronę zwykle potrzebujemy deklaracji typu:

$$\frac{\Gamma \vdash e \Leftarrow t}{\Gamma \vdash (e : t) \Rightarrow t}$$

Kontrola typów w językach imperatywnych

Rozważmy mały język imperatywny:

$$\begin{aligned} e &::= x \mid n \mid b \mid e_1 + e_2 \mid e_1 = e_2 \\ s &::= x := e \mid \textbf{while}\ e\ \textbf{do}\ s \mid s ; s \end{aligned}$$

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.

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

czy

$$\frac{\Gamma \vdash e \Leftarrow \mathrm{bool} \quad \Gamma \vdash_P p}{\Gamma \vdash_P \textbf{while } e \textbf{ do } p}$$

Deklaracje

Możemy uznać deklarację jako rodzaj instrukcji oraz regułę

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

inną mozliwością jest wprowadzenie nowego typu osądu, $\vdash_D$:

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

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

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

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

Auto

Dla deklaracji z inicjalizatorami możemy wprowadzić odpowiednik `auto` z C++11:

$$\frac{\Gamma \vdash e \Rightarrow \tau}{\Gamma \vdash_D (\textbf{auto } x = e) \Rightarrow \Gamma(x : \tau)}$$

Kontrola typów w językach funkcyjnych

Typy:

$$\tau ::= \mathrm{int} \mid \mathrm{bool} \mid \tau_1 \to \tau_2$$

Wyrażenia:

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

Reguły typowania

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

$$\frac{\Gamma \vdash e_1 : \tau \to \rho \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash e_1 e_2 : \rho}$$

Kierunek przepływu danych

Tu użyteczny może się okazać system dwukierunkowy, np.

Funkcja:

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

Aplikacja:

$$\frac{\Gamma \vdash e_1 \Rightarrow (\tau \to \rho) \quad \Gamma \vdash e_2 \Leftarrow \tau}{\Gamma \vdash e_1 e_2 \Rightarrow \rho}$$

Przykładowa implementacja

```
findType :: Env → Exp → CM Type
findType env (EInt n)   = return TInt
findType env (EVar x)   = envType x env
findType env (ELam v t e) = do
        t' ← findType ((v,t):env) e
        return (t :→ t')
findType env (EApp e1 e2) = do
        (t1 :→ t) ← findType env e1
        checkType env e2 t1
        return t
findType env (EPlus e1 e2) = do
        checkType env e1 TInt
        checkType env e2 TInt
        return TInt
```

Przykładowa implementacja

```
checkType :: Env → Exp → Type → CM ()
checkType env e@(ELam v t b) (t1 :→ t2) =  do
    if t == t1 then checkType ((v,t):env) b t2 else
        throwError $unwords ["The arg of", show e,"is not
            of type",show  t]
checkType env e t = do
```

```
  t' ← findType env e
  unless (t'==t) $ typeError e t t'

typeError e t t' = throwError $ unwords [
  "Couldn't match expected type", show t,
  "against inferred type", show t',
  "in the expression", show e
  ]
```

Rekonstrukcja typów

- Jeśli typy nie są znane, trzeba *zrekonstruować* pasujące typy.
- Reguły typowania pozostają te same; reguła dla funkcji odpowiada zmienionej składni:
  $$\frac{\Gamma(x:\tau) \vdash e : \rho}{\Gamma \vdash \lambda x.e : \tau \to \rho}$$
  co prowadzi do problemu: skąd wziąć dobre τ?
- Możemy uczynić τ niewiadomą. Proces typowania da nam typ wraz z układem równań
- Przy każdym użyciu reguły aplikacji
  $$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$
  dodajemy do układu równanie $\tau_1 = \tau_2$.

Rozwiązywanie równań: unifikacja

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

W każdym kroku mamy układ równań E i podstawienie S (na początku $S = \emptyset$, koniec gdy $E = \emptyset$)

- Równanie $t_1 \to t_2 = u_1 \to u_2$ zastępujemy przez parę równań
  $$t_1 = u_1$$
  $$t_2 = u_2$$
- Równania postaci $x = x$ usuwamy.
- Gdy E jest postaci $x = t; F$, przy czym $x \notin FV(t)$ to przyjmujemy $E' = F[x := t]$, $S' = [x := t] \circ S$

Kiedy unifikacja zawodzi

Unifikacja zawodzi, gdy napotka jedno z poniższych:

- Równanie postaci (k_1 i k_2 są różnymi stałymi)

$$k_1 = k_2$$

- Równanie postaci (k — stała):

$$k = t \to t'$$

- Równanie postaci

$$x = t$$

  gdzie x — zmienna a t zawiera x ale różny od x.

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

$$\tau_x = \tau_x \to \rho.$$

Ten term nie jest typowalny (w tym 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

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

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:

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

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

Polimorfizm — przykłady i smutna konstatacja

Możemy wyprowadzić

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

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

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

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 typów 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

$$\mathit{if\_then\_else\_} : \forall\alpha.\mathrm{bool} \to \alpha \to \alpha \to \alpha$$

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

Podtypy

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

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

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

Można też przepisać regułę aplikacji tak:

$$\frac{\Gamma \vdash e_1 : \tau_1 \to \tau \quad \Gamma \vdash e_2 : \tau_2 \quad \tau_2 \leqslant \tau_1}{\Gamma \vdash e_1 e_2 : \tau}$$

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

Przeciążanie

- Funkcje polimorficzne działają niezależnie od typu argumentów.
- Przeciążanie oznacza, że jeden symbol funkcyjny (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:

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

  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:

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : \text{int} \quad \Gamma \vdash e_2 \leadsto e_2' : \text{int}}{\Gamma \vdash e_1 = e_2 \leadsto \texttt{eqInt}\ e_1'\ e_2' : \text{bool}}$$

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : \text{bool} \quad \Gamma \vdash e_2 \leadsto e_2' : \text{bool}}{\Gamma \vdash e_1 = e_2 \leadsto \texttt{eqBool}\ e_1'\ e_2' : \text{bool}}$$

  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

$$\frac{\Gamma \vdash e_1 \leadsto e_1' : \textbf{int} \quad \Gamma \vdash e_2 \leadsto e_2' : \textbf{double}}{\Gamma \vdash e_1 + e_2 \leadsto \mathtt{int2double}(e_1') + e_2' : \textbf{double}}$$