PARADIGME DE PROGRAMARE

Curs 7

Sinteză de tip.

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Sistem de tipuri

Sistem de tipuri = set de mecanisme și reguli valabile într-un limbaj de programare, privind organizarea, construirea, manevrarea tipurilor de date acceptate în limbaj

Atribuții (nu se ocupă neapărat de toate)

- Asocierea de tipuri constructelor din limbaj
- Definirea de noi tipuri
- Decizia asupra echivalenței / compatibilității tipurilor
- Sinteza (inferența) de tip

Avantaje și limitări

Avantaje

- Detectare timpurie a
  - Erorilor (+ mai mare acuratețe în identificarea sursei)
  - Codului inutil sau invalid
- Declarații de tip – abstractizare, documentare (care rămâne consistentă cu modificările efectuate ulterior)
- Eficiență (se pot optimiza operații pentru anumite tipuri)

Limitări

- Rejectează cod care nu ar produce probleme
- Renunță la anumite facilități (ex: Haskell nu permite liste eterogene)
- Tiparea statică necesită ori adnotări (declarații) de tip, ori sinteză (inferență) de tip

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Sinteză (inferență) de tip

- Tipul unei expresii este sintetizat în funcție de
  - **Tipul componentelor** expresiei (ex: tipul expresiei 2 + 2)
  - **Contextul** lexical al expresiei (ex: tipul lui x în contextul ['a' .. x])
- Reprezentarea tipului – expresie de tip, care reprezintă o combinație de
  - **Constante de tip** (tipurile primitive: Bool, Char, Int, etc.)
  - **Variabile de tip** (cu semnificația „orice (expresie de) tip”)
  - **Constructori de tip** ((,), [ ], ->, constructori definiți cu „data”)

Sinteză de tip = **determinarea automată a tipului unei expresii, prin combinarea constantelor, variabilelor și constructorilor de tip după reguli bine stabilite (reguli de sinteză de tip)**

Reguli de sinteză de tip

Relație de tipare: e :: T

- Se citește „expresia t are tipul T”
- O expresie e este **bine tipată** ⇔ există un tip T astfel încât e :: T

Regulă de sinteză de tip

$$\frac{P_1, P_2, \ldots P_n}{C} \text{ (id_regulă)}$$

P_i = premise (0 sau mai multe relații de tipare)

C = concluzie (o relație de tipare)

Reguli de sinteză de tip – Exemple

$$\frac{}{\text{True, False} :: \text{Bool}} \ (TBool)$$

$$\frac{expr1 :: \text{Bool} \quad expr2 :: \text{a} \quad expr3 :: \text{a}}{\text{if } expr1 \text{ then } expr2 \text{ else } expr3 :: \text{a}} \ (TIf)$$

$$\frac{}{0, 1, 2 \ \ldots :: \text{Int}} \ (TInt)$$

$$\frac{expr1 :: \text{Int} \quad expr2 :: \text{Int}}{expr1 + expr2 :: \text{Int}} \ (T+)$$

$$\frac{var :: \text{a} \quad expr :: \text{b}}{\backslash var \rightarrow expr :: \text{a} \rightarrow \text{b}} \ (TAbs)$$

$$\frac{fun :: \text{a} \rightarrow \text{b} \quad expr :: \text{a}}{fun \ expr :: \text{b}} \ (TApp)$$

valori booleene

funcții

întregi

Observații

- **Tipul** unei expresii este **unic**
- Regulile de tipare funcționează în ambele sensuri (vom observa că în sinteza de tip pornim de la concluzie către premise)
  - De la **premise către concluzie**
    Exemplu: Dacă expr1 :: Int și expr2 :: Int, atunci expr1 + expr2 :: Int
  - De la concluzie către premise
    Exemplu: Dacă expr1 + expr2 :: Int, atunci expr1 :: Int și expr2 :: Int

Algoritm de sinteză de tip

- **Identifică regula aplicabilă**, prin pattern match între
  - Expresia care trebuie tipată
  - Concluziile regulilor de sinteză de tip
- **Aplică regula**, obținând constrângerile asupra componentelor expresiei
- **Pentru fiecare componentă în parte, reia algoritmul**, până când nu mai este nimic de descompus în părți componente
- **Tipul expresiei rezultă din aplicarea tuturor constrângerilor** obținute la pașii anteriori

Sinteză de tip – Exemple

Care este tipul funcției f: **f g = g 3 + 1 ⇔ f = \g -> g 3 + 1**

Constrângeri

$$\frac{var :: \mathrm{a} \quad expr :: \mathrm{b}}{\backslash var \to expr :: \mathrm{a} \to \mathrm{b}} \ (TAbs)$$

$$\frac{g :: \mathrm{a} \quad g\ 3 + 1 :: \mathrm{b}}{f :: \mathrm{a} \to \mathrm{b}} \ (TAbs)$$

$$\frac{expr1 :: \mathrm{Int} \quad expr2 :: \mathrm{Int}}{expr1 + expr2 :: \mathrm{Int}} \ (T+)$$

$$\frac{g\ 3 :: \mathrm{Int} \quad 1 :: \mathrm{Int}}{g\ 3 + 1 :: \mathrm{Int}} \ (T+)$$

b = Int

$$\frac{fun :: \mathrm{a} \to \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \ (TApp)$$

$$\frac{g :: \mathrm{c} \to \mathrm{Int} \quad 3 :: \mathrm{c}}{g\ 3 :: \mathrm{Int}} \ (TApp)$$

a = c ->Int

$$\frac{}{0, 1, 2\ \ldots :: \mathrm{Int}} \ (TInt)$$

$$\frac{}{1, 3\ :: \mathrm{Int}} \ (TInt)$$

c = Int

Așadar **f :: a -> b ⇔ f :: (c -> Int) -> Int ⇔ f :: (Int -> Int) -> Int**

Sinteză de tip – Exemple

Care este tipul funcției fix: **fix f = f (fix f)** ⇔ **fix = \f -> f (fix f)**

Constrângeri

$$\frac{var :: \mathrm{a} \quad expr :: \mathrm{b}}{\backslash var \rightarrow expr :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{f :: \mathrm{a} \quad f\ (fix\ f) :: \mathrm{b}}{fix :: \mathrm{a} \rightarrow \mathrm{b}} \; (TAbs)$$

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \; (TApp)$$

$$\frac{f :: \mathrm{c} \rightarrow \mathrm{b} \quad (fix\ f) :: \mathrm{c}}{f\ (fix\ f) :: \mathrm{b}} \; (TApp)$$

a = c -> b

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \; (TApp)$$

$$\frac{fix :: \mathrm{d} \rightarrow \mathrm{c} \quad f :: \mathrm{d}}{fix\ f :: \mathrm{c}} \; (TApp)$$

a = d, b = c

Așadar **fix :: a -> b** ⇔ **fix :: (c -> b) -> b** ⇔ **fix :: (b -> b) -> b**

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

(++)	[a] -> [b] -> [c]
zipWith (ex: zipWith (+) [1,2] [3,4] = [4,6])	(a -> a -> a) -> [a] -> [a] -> [a]
unzip (ex: unzip [(1,2), (3,4)] = ([1,3], [2,4]))	[(a,b)] -> ([a], [b])
[take, drop]	[Int -> [a] -> [a]]
words (ex: words "wait for it" = ["wait", "for", "it"])	String -> String
(odd . fst)	(Int, b) -> Bool
map ("01" :) . reverse	[[Char]] -> [[Char]]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

(++)	[a] -> [b] -> [c]
zipWith (ex: zipWith (+) [1,2] [3,4] = [4,6])	(a -> a -> a) -> [a] -> [a] -> [a]

Nu, nu pot concatena liste cu elemente de tipuri diferite!

Ex: [1,2,3] ++ [(+),(-)]

Corect: [a] -> [a] -> [a]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

zipWith (ex: zipWith (+) [1,2] [3,4] = [4,6])	(a -> a -> a) -> [a] -> [a] -> [a]
unzip (ex: unzip [(1,2), (3,4)] = ([1,3], [2,4]))	[(a,b)] -> ([a], [b])

Nu, nu e necesar ca cele două liste să aibă elemente de același tip.

Ex: zipWith (:) [1,2] [[3],[4]]

Corect: (a -> b -> c) -> [a] -> [b] -> [c]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

unzip (ex: unzip [(1,2), (3,4)] = ([1,3], [2,4]))	[(a,b)] -> ([a], [b])
[take, drop]	[Int -> [a] -> [a]]

Ex: unzip [(1,'a'), (2,'b'), (3,'c')]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

[take, drop]	[Int -> [a] -> [a]]
words (ex: words "wait for it" = ["wait", "for", "it"])	String -> String

Da, atenție la faptul că signatura indică o listă de funcții de același tip.

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

Nu, rezultatul este o listă de String-uri, nu o listă de caractere (String ~ [Char]).

Corect: String -> [String]

Alternativ, se putea scrie: words :: [Char] -> [[Char]]

words (ex: words "wait for it" = ["wait", "for", "it"])	String -> String
(odd . fst)	(Int, b) -> Bool

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

Da:

fst :: (a,b) -> a odd :: Integral a => a -> Bool

Ex: (odd . fst) (1, [])

(odd . fst)	(Int, b) -> Bool
map ("01" :) . reverse	[[Char]] -> [[Char]]

Antrenament: Expresii și signaturi

Exercițiu: Pentru expresia din stânga este corectă signatura din dreapta?

Nu, fiecare element din listă trebuie să fie nu un String, ci o listă de String-uri.

"o1" : element înseamnă că "o1" are tipul a și element are tipul [a].

Ex: (map ("o1":) . reverse) [["So", "many"], ["Strings"]]

Corect: [[String]] -> [[String]] sau [[[Char]]] -> [[[Char]]]

map ("o1" :) . reverse [[Char]] -> [[Char]]

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Siguranță de tip

Există **siguranță de tip** atunci când expresiile bine tipate „nu o pot lua pe căi greșite” în timpul evaluării, adică atunci când se respectă proprietățile:

- **Progres**
  - O expresie bine tipată e :: T nu se blochează:
    - Ori este o valoare
    - Ori se poate reduce conform unei reguli de evaluare
- **Conservare**
  - Un pas de evaluare asupra unei expresii bine tipate e :: T produce o expresie bine tipată e’ :: T

Scop: prevenirea erorilor cauzate de discrepanțele dintre tipurile așteptate și cele procesate.

Test

Scrieți un list comprehension în Haskell care reprezintă mulțimea (așadar fără duplicate) tuturor tripletelor pitagoreice de numere naturale de maxim 3 cifre.

Obs: Numerele a, b, c formează un triplet pitagoreic dacă pot fi laturile unui triunghi dreptunghic (a^2 + b^2 = c^2).

Ex: 3, 4, 5 formează un triplet pitagoreic întrucât 9 + 16 = 25

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Unificare

Unificarea a 2 expresii de tip = **găsirea celei mai generale substituții**

$$S = \{e_1 / t_1, e_2 / t_2, \ldots e_n / t_n\}$$

pentru variabilele de tip t_i din cele 2 expresii astfel încât, în urma substituției, cele 2 expresii de tip devin una și aceeași

- Se folosește în sinteza de tip în pasul de **aplicare a constrângerilor**

Exemplu

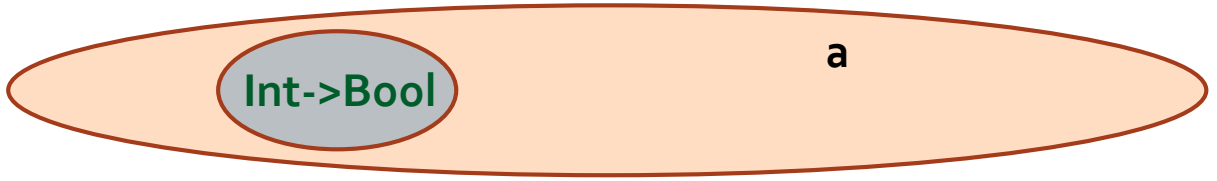
- Din sinteza de tip avem constrângerile a = b -> [d] și a = Int -> c
  → b -> [d] trebuie să unifice cu Int -> c, altfel sinteza de tip va eșua
- Există S = {Int / b, [d] / c} a.î. b -> [d] și Int -> c se intersectează în Int -> [d]
- Există și S' = {Int / b, [Int] / c, Int / d}, rezultând în Int -> [Int], dar S este **mai generală**!

Mai multe exemple

- e1 = a, e2 = Int -> Bool
- e1 = (a, [a]), e2 = ([b], c)
- e1 = Int, e2 = Float

Mai multe exemple

- e1 = a, e2 = Int -> Bool S = {Int -> Bool / a} $e_{unificată}$ = Int -> Bool

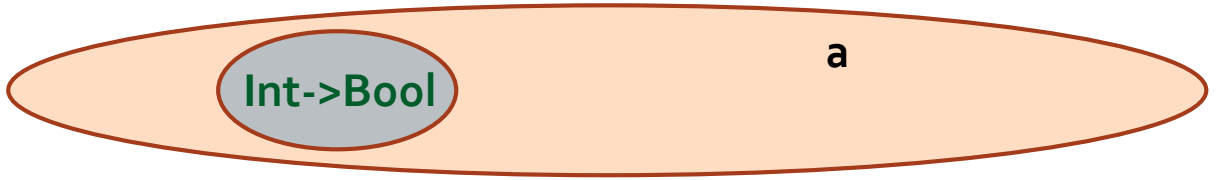


- e1 = (a, [a]), e2 = ([b], c)

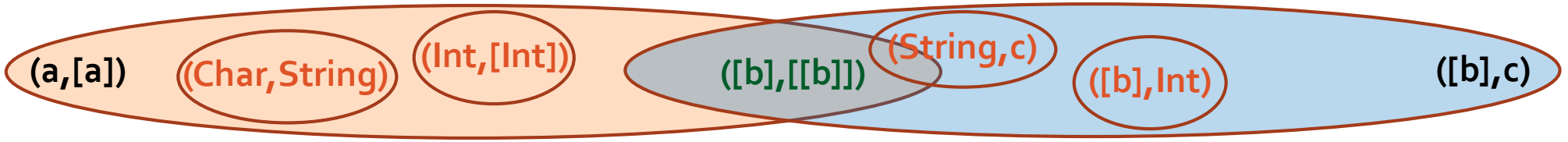
- e1 = Int, e2 = Float

Mai multe exemple

- e1 = a, e2 = Int -> Bool S = {Int -> Bool / a} $e_{unificată}$ = Int -> Bool



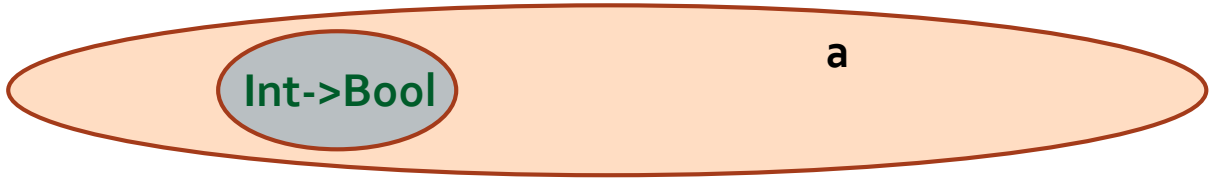
- e1 = (a, [a]), e2 = ([b], c) S = {[b] / a, [[b]] / c} $e_{unificată}$ = ([b], [[b]])



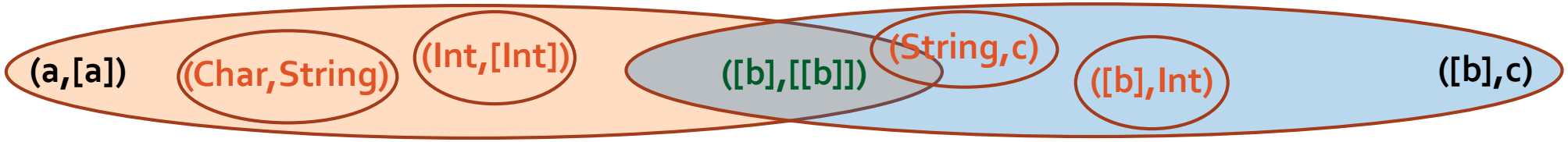
- e1 = Int, e2 = Float

Mai multe exemple

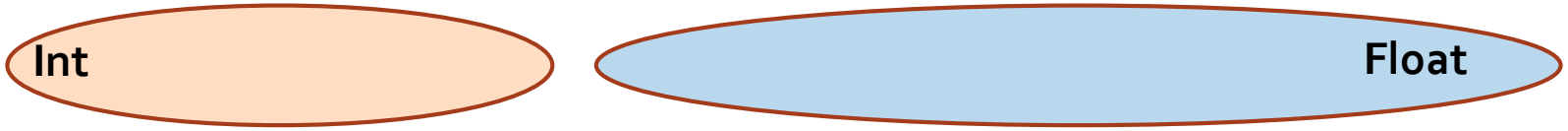
- e1 = a, e2 = Int -> Bool S = {Int -> Bool / a} $e_{unificată}$ = Int -> Bool



- e1 = (a, [a]), e2 = ([b], c) S = {[b] / a, [[b]] / c} $e_{unificată}$ = ([b], [[b]])



- e1 = Int, e2 = Float Nu unifică (nu se fac conversii implicite)!



Unificare la sinteza de tip – Exemplu

Care este tipul funcției f: **f x = x x** ⇔ **f = \x -> x x**

Constrângeri

$$\frac{var :: \mathrm{a} \quad expr :: \mathrm{b}}{\backslash var \rightarrow expr :: \mathrm{a} \rightarrow \mathrm{b}} \ (TAbs)$$

$$\frac{x :: \mathrm{a} \quad x\ x :: \mathrm{b}}{f :: \mathrm{a} \rightarrow \mathrm{b}} \ (TAbs)$$

$$\frac{fun :: \mathrm{a} \rightarrow \mathrm{b} \quad expr :: \mathrm{a}}{fun\ expr :: \mathrm{b}} \ (TApp)$$

$$\frac{x :: \mathrm{c} \rightarrow \mathrm{b} \quad x :: \mathrm{c}}{x\ x :: \mathrm{b}} \ (TApp)$$

a = c -> b = c

a unifică cu c -> b, dar c -> b nu unifică cu c.
(c = c -> b = c -> b -> b = c -> b -> b -> b ... rezultă un tip infinit, care nu poate fi sintetizat)

Întrucât unificarea eșuează (găsește o inconsistență), definiția lui f generează **eroare**:
Occurs check: cannot construct the infinite type: t1 ~ t1 -> t

Reguli folosite în algoritmul de unificare

- O **variabilă de tip** `a` unifică cu o **expresie de tip** `e` (S = {`e` / `a`}) dacă și numai dacă
  - `e = a` sau
  - `e` nu conține `a` (occurs check – pentru a evita tipuri infinite)

- **2 constante de tip** unifică (S = { }) dacă și numai dacă sunt egale

- **2 construcții de tip** unifică ($S = S_1 \cup \ldots \cup S_n$) dacă și numai dacă sunt aplicații ale aceluiași constructor de tip asupra câte n argumente care unifică recursiv (prin substituțiile $S_1 \ldots S_n$)

Unificare la sinteza de tip – Implementare

- Algoritmul de sinteză de tip pe bază de unificare folosește
  - O **stivă de constrângeri** de forma **`e1 = e2`** – se umple în timpul algoritmului de sinteză de tip
  - O **substituție S** (mulțime de asocieri între variabile de tip și expresii de tip) – inițial vidă
- La fiecare iterație a pasului de unificare
  - Se extrage o constrângere `e1 = e2` din stivă
  - Dacă `e1` unifică cu `e2` prin substituția S_{12}, se aplică S_{12}
    - asupra stivei și
    - asupra substituției S
  - Altfel, unificarea eșuează → sinteza de tip eșuează → eroare de tip

Exemplu

Stiva	Substituția

`a = b -> c`
`a = d -> d`
`b = Int`	

`b -> c = d -> d` `b = Int`	`a = b -> c`

Exemplu

Stiva	Substituția

`a = b -> c`
`a = d -> d`
`b = Int`	

`b -> c = d -> d` `b = Int`	`a = b -> c`

`b = d`
`c = d`
`b = Int`
	`a = b -> c`

Exemplu

Stiva	Substituția
a = b -> c a = d -> d b = Int	
b -> c = d -> d b = Int	a = b -> c
b = d c = d b = Int	a = b -> c
c = d d = Int	a = d -> c b = d

Exemplu

Stiva	Substituția
a = b -> c a = d -> d b = Int	
b -> c = d -> d b = Int	a = b -> c
b = d c = d b = Int	a = b -> c
c = d d = Int	a = d -> c b = d
d = Int	a = d -> d b = d c = d

Exemplu

Stiva	Substituția
a = b -> c a = d -> d b = Int	
b -> c = d -> d b = Int	a = b -> c
b = d c = d b = Int	a = b -> c
c = d d = Int	a = d -> c b = d
d = Int	a = d -> d b = d c = d
	a = Int -> Int b = Int c = Int d = Int

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Determinarea tipului expresiilor / funcțiilor

length ([]++[1,2]) + length ([]++"abcd") **Constrângeri**

(T+) ⟶ expr :: Int ⇔ length ([]++[1,2]) :: Int, length ([]++"abcd") :: Int

(TLen) ⟶ []++[1,2] :: [a], []++"abcd" :: [b]

(T++) ⟶ [] :: [a], [1,2] :: [a], [] :: [b], "abcd" :: [b]

(T[]) ⟶ 1,2 :: a, 'a','b','c','d' :: b

(TInt, TChar) ⟶ 1,2 :: Int, 'a','b','c','d' :: Char a = Int, b = Char

[] :: [Int] și [] :: [Char] ok ([] e polimorfic – cele 2 liste vide nu reprezintă aceeași valoare)

Etapa de unificare reușește și ea (nu avem două constrângeri asupra aceleiași variabile), deci expresia are tipul Int conform T+.

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")

Constrângeri

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd") **Constrângeri**

(T+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd") **Constrângeri**

(T+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int

(TLen) ⟶ xs++[1,2] :: [a], xs++"abcd" :: [b]

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd") **Constrângeri**

(T+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int

(TLen) ⟶ xs++[1,2] :: [a], xs++"abcd" :: [b]

(T++) ⟶ xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b] [a] = [b]

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd") **Constrângeri**

(T+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int

(TLen) ⟶ xs++[1,2] :: [a], xs++"abcd" :: [b]

(T++) ⟶ xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b] [a] = [b]

(T[]) ⟶ 1,2 :: a, 'a','b','c','d' :: b

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")	**Constrângeri**
(T+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int	
(TLen) ⟶ xs++[1,2] :: [a], xs++"abcd" :: [b]	
(T++) ⟶ xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b]	[a] = [b]
(T[]) ⟶ 1,2 :: a, 'a','b','c','d' :: b	
(TInt, TChar) ⟶ 1,2 :: Int, 'a','b','c','d' :: Char	a = Int, b = Char

Determinarea tipului expresiilor / funcțiilor

length (xs++[1,2]) + length (xs++"abcd")	**Constrângeri**
(T+) ⟶ expr :: Int ⇔ length (xs++[1,2]) :: Int, length (xs++"abcd") :: Int	
(TLen) ⟶ xs++[1,2] :: [a], xs++"abcd" :: [b]	
(T++) ⟶ xs :: [a], [1,2] :: [a], xs :: [b], "abcd" :: [b]	[a] = [b]
(T[]) ⟶ 1,2 :: a, 'a','b','c','d' :: b	
(TInt, TChar) ⟶ 1,2 :: Int, 'a','b','c','d' :: Char	a = Int, b = Char

În etapa de unificare: [Int] = [Char] ⟶ Int = Char care nu unifică, deci tipul expresiei nu poate fi sintetizat și vom obține o eroare de tip.

Determinarea tipului expresiilor / funcțiilor

curry id

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry id

Constrângeri

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, id :: a

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) $\longrightarrow$ expr :: b $\Leftrightarrow$ curry :: a -> b, id :: a

$$\frac{}{\text{curry} \;::\; ((c, d) \to e) \to c \to d \to e} \; (TCurry)$$

Constrângeri

a = (c,d) -> e, b = c -> d -> e

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, id :: a

$$\frac{}{\text{curry} \;::\; \big((\text{c},\text{d}) \to \text{e}\big) \to \text{c} \;\to \text{d} \;\to \text{e}} \;(TCurry)$$

$$\frac{}{\text{id} \;::\; \text{f} \;\to \text{f}} \;(TId)$$

Constrângeri

a = (c,d) -> e, b = c -> d -> e

a = f -> f

Determinarea tipului expresiilor / funcțiilor

curry id

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, id :: a

$$\frac{}{\text{curry} \;::\; ((c, d) \to e) \to c \to d \to e} \; (TCurry)$$

$$\frac{}{\text{id} \;::\; f \to f} \; (TId)$$

Constrângeri

a = (c,d) -> e, b = c -> d -> e

a = f -> f

În etapa de unificare: a = (c,d) -> e = f -> f ⟶ e = f = (c,d) ⟶ b = c -> d -> (c,d), deci expresia are tipul c -> d -> (c,d) conform TApp.

Determinarea tipului expresiilor / funcțiilor

uncurry id

Constrângeri

Determinarea tipului expresiilor / funcțiilor

uncurry id

Constrângeri

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

$$\frac{}{\text{uncurry} \;::\; (c \to d \to e) \to (c, d) \;\to e} \; (TUncurry)$$

Constrângeri

a = c -> d -> e, b = (c,d) -> e

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

$$\frac{}{\text{uncurry} \;::\; (c \to d \to e) \to (c, d) \to e} \; (TUncurry)$$

$$\frac{}{\text{id} \;::\; f \to f} \; (TId)$$

Constrângeri

a = c -> d -> e, b = (c,d) -> e

a = f -> f

Determinarea tipului expresiilor / funcțiilor

uncurry id

(TApp) ⟶ expr :: b ⇔ uncurry :: a -> b, id :: a

$$\frac{}{\text{uncurry} \;::\; (c \to d \to e) \to (c, d) \to e} \; (TUncurry)$$

$$\frac{}{\text{id} \;::\; f \to f} \; (TId)$$

Constrângeri

a = c -> d -> e, b = (c,d) -> e

a = f -> f

În etapa de unificare: a = c -> d -> e = f -> f ⟶ c = f = d -> e ⟶ b = (d -> e, d) -> e, deci expresia are tipul (d -> e, d) -> e conform TApp.

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

Constrângeri

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

(ExAnterior) ⟶ curry id :: f -> g -> (f,g)

Constrângeri

a = f -> g -> (f,g)

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

(ExAnterior) ⟶ curry id :: f -> g -> (f,g)

$$\frac{}{\text{curry} \;::\; \big((\text{c},\text{d}) \to \text{e}\big) \to \text{c} \;\to \text{d} \;\to \text{e}} \;(TCurry)$$

Constrângeri

a = f -> g -> (f,g)

a = (c,d) -> e, b = c -> d -> e

Determinarea tipului expresiilor / funcțiilor

curry (curry id)

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, curry id :: a

(ExAnterior) ⟶ curry id :: f -> g -> (f,g)

$$\frac{}{\text{curry} \;::\; \big((c, d) \to e\big) \to c \;\to d \;\to e} \quad (TCurry)$$

Constrângeri

a = f -> g -> (f,g)

a = (c,d) -> e, b = c -> d -> e

În etapa de unificare: a = f -> g -> (f,g) = (c,d) -> e ⟶ f = (c,d), e = g -> (f,g) = g -> ((c,d),g) ⟶ b = c -> d -> g -> ((c,d), g), deci expresia are tipul c -> d -> g -> ((c,d), g) conform TApp.

Determinarea tipului expresiilor / funcțiilor

curry uncurry

Constrângeri

Determinarea tipului expresiilor / funcțiilor

curry uncurry

Constrângeri

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

$$\frac{}{\text{uncurry} \;::\; (c \to d \to e) \to (c, d) \;\to e} \;(TUncurry)$$

Constrângeri

a = (c -> d -> e) -> (c,d) -> e

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) $\longrightarrow$ expr :: b $\Leftrightarrow$ curry :: a -> b, uncurry :: a

$$\frac{}{\text{uncurry} \;::\; (c \to d \to e) \to (c, d) \to e} \;(TUncurry)$$

$$\frac{}{\text{curry} \;::\; \big((f, g) \to h\big) \to f \to g \to h} \;(TCurry)$$

Constrângeri

a = (c -> d -> e) -> (c,d) -> e

a = (f,g) -> h, b = f -> g -> h

Determinarea tipului expresiilor / funcțiilor

curry uncurry

(TApp) ⟶ expr :: b ⇔ curry :: a -> b, uncurry :: a

$$\frac{}{\text{uncurry} \;::\; (c \to d \to e) \to (c, d) \to e} \;(TUncurry)$$

$$\frac{}{\text{curry} \;::\; \big((f, g) \to h\big) \to f \to g \to h} \;(TCurry)$$

Constrângeri

a = (c -> d -> e) -> (c,d) -> e

a = (f,g) -> h, b = f -> g -> h

În etapa de unificare: a = (c -> d -> e) -> (c,d) -> e = (f,g) -> h ⟶ (printre altele) c -> d -> e = (f, g) care nu unifică (sunt aplicații ale unor constructori de tip diferiți), deci tipul expresiei nu poate fi sintetizat și vom obține o eroare de tip.

Exercițiu: Încercați și curry curry, uncurry curry, uncurry uncurry (nu toate dau eroare).

Sinteză de tip – Cuprins

- Sistem de tipuri
- Sinteză de tip
- Siguranță de tip
- Unificare
- Exemple
- Mesaje de eroare

Parse error

- programul nu respectă **sintaxa Haskell**
- se indică locul în care a apărut un simbol neașteptat
- eroarea e la locul indicat sau undeva înainte

```
f : Int -> [a] -> Int
f a b = a + length b
```

curs7.hs:1:9: parse error on input `->'

```
f x = x ++ [1; 2; 3]
```

curs7.hs:1:14: parse error on input `;'

```
f x =
      | null x = 0
      | otherwise = 1
```

curs7.hs:2:9: parse error on input `|'

Parse error

- programul nu respectă **sintaxa Haskell**
- se indică locul în care a apărut un simbol neașteptat
- eroarea e la locul indicat sau undeva înainte

```
f : Int -> [a] -> Int
f a b = a + length b

f x = x ++ [1; 2; 3]

f x =
      | null x = 0
      | otherwise = 1
```

curs7.hs:1:9: parse error on input `->'
Înlocuiește cu ::

curs7.hs:1:14: parse error on input `;'
Înlocuiește cu ,
curs7.hs:2:9: parse error on input `|'
Șterge = dinainte de |

Not in scope

- compilatorul **nu recunoaște o variabilă / o funcție / un constructor**
- fie am folosit o variabilă / o funcție / un constructor care nu există (posibil typo)
- fie trebuie să importăm modulul în care se află ace(a)sta

```
f a x
      | null x = cons a x
      | otherwise = f (a+1) (tail x)
```

Not in scope: `cons'

```
f a x
      | null x = a
      | oterwise = f (a+1) (tail x)
```

Not in scope: `oterwise'
Perhaps you meant `otherwise'

```
f x = sort (take 10 x)
```

Not in scope: `sort'

Not in scope

- compilatorul **nu recunoaște o variabilă / o funcție / un constructor**
- fie am folosit o variabilă / o funcție / un constructor care nu există (posibil typo)
- fie trebuie să importăm modulul în care se află ace(a)sta

```
f a x
      | null x = cons a x
      | otherwise = f (a+1) (tail x)
```

Not in scope: `cons'
Înlocuiește cu (:)

```
f a x
      | null x = a
      | oterwise = f (a+1) (tail x)
```

Not in scope: `oterwise'
Perhaps you meant `otherwise'

```
f x = sort (take 10 x)
```

Not in scope: `sort'

Caută funcția (ex: hoogle.haskell.org) și vezi ce modul trebuie importat (aici import Data.List)

The function <> is applied to <> arguments

- **am aplicat o funcție pe mai multe sau mai puține argumente decât așteaptă**
- fie am pus ceva în plus / minus
- fie n-am pus parantezele bine

```
import Data.List
f x = sort (take 10 x) (<)

f x = if null x then 0 else 1 + f tail x
```

The function `sort' is applied to two arguments,
but its type `[a] -> [a]' has only one

...

Probable cause: `tail' is applied to too few arguments

The function <> is applied to <> arguments

- **am aplicat o funcție pe mai multe sau mai puține argumente decât așteaptă**
- fie am pus ceva în plus / minus
- fie n-am pus parantezele bine

```
import Data.List
f x = sort (take 10 x) (<)

f x = if null x then 0 else 1 + f tail x
```

The function `sort' is applied to two arguments,
but its type `[a] -> [a]' has only one
Șterge (<)
...
Probable cause: `tail' is applied to too few arguments
Pune paranteze la tail: f (tail x)

Couldn’t match expected type with actual type

- fie nu suntem în concordanță cu signatura (ori corectăm funcția, ori signatura)
- fie **am aplicat o funcție pe tipuri incompatibile cu ce așteaptă funcția**
  - posibil să fi greșit ordinea argumentelor

```
f :: String -> String
f = head
```

Couldn't match type `Char' with `[Char]'

```
f = 'A' ++ "BC"
```

Couldn't match expected type `[Char]' with actual type `Char'

```
f = (:) "BC" 'A'
```

Couldn't match expected type `[[Char]]' with actual type `Char'
In the second argument of `(:)', namely 'A'

Couldn’t match expected type with actual type

- fie nu suntem în concordanță cu signatura (ori corectăm funcția, ori signatura)
- fie **am aplicat o funcție pe tipuri incompatibile cu ce așteaptă funcția**
  - posibil să fi greșit ordinea argumentelor

```
f :: String -> String
f = head
```

Couldn't match type `Char' with `[Char]'
Fie f :: String -> Char, fie f = (: []) . head

```
f = 'A' ++ "BC"
```

Couldn't match expected type `[Char]' with actual type `Char’
Fie 'A' : "BC", fie "A" ++ "BC"

```
f = (:) "BC" 'A'
```

Couldn't match expected type `[[Char]]' with actual type `Char'
In the second argument of `(:)', namely ‘A’
Schimb ordinea: (:) 'A' "BC"

Non-exhaustive patterns

- o funcție definită cu pattern matching sau gărzi **nu acoperă toate valorile tipului**
- fie am uitat anumite valori
- fie avem un typo la numele funcției (și Haskell pur și simplu consideră că este altă funcție)

```
f x
    | x < 0 = -1
    | x > 0 = 1
```

```
*Main> f 0
*** Exception: curs7.hs:(1,1)-(3,19): Non-exhaustive patterns in function f
```

(dacă avem warning-urile activate, primim warning la compilare)

```
mySum [] = 0
mySUm (x:xs) = x + mySum xs
```

```
*Main> mySum [1..5]
*** Exception: ... Non-exhaustive patterns ... mySum
```

Non-exhaustive patterns

- o funcție definită cu pattern matching sau gărzi **nu acoperă toate valorile tipului**
- fie am uitat anumite valori
- fie avem un typo la numele funcției (și Haskell pur și simplu consideră că este altă funcție)

```
f x
        | x < 0 = -1
        | x > 0 = 1
```

*Main> f 0

*** Exception: curs7.hs:(1,1)-(3,19): Non-exhaustive patterns in function f

(dacă avem warning-urile activate, primim warning la compilare)

Adaugă | otherwise = …

```
mySum [] = 0
mySUm (x:xs) = x + mySum xs
```

*Main> mySum [1..5]

*** Exception: … Non-exhaustive patterns … mySum

Corectează mySUm în mySum

Rezumat

Sinteză de tip
Regulă de sinteză de tip

Algoritm de sinteză de tip

Siguranță de tip

Unificare
O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip

Algoritm de sinteză de tip

Siguranță de tip

Unificare
O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip

Siguranță de tip

Unificare
O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip

Unificare
O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)
Unificare
O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)
Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice
O variabilă unifică cu
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)
Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice
O variabilă unifică cu: orice, cât timp occurs check este OK
2 constante unifică
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)
Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice
O variabilă unifică cu: orice, cât timp occurs check este OK
2 constante unifică: dacă sunt identice
2 construcții unifică

Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)
Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice
O variabilă unifică cu: orice, cât timp occurs check este OK
2 constante unifică: dacă sunt identice
2 construcții unifică: dacă folosesc același constructor de tip pe argumente care unifică la rândul lor
Structuri folosite de algoritmul de unificare

Rezumat

Sinteză de tip: determinarea tipului unei expresii pe baza regulilor de sinteză de tip
Regulă de sinteză de tip: pe baza unor relații de tipare (premise) se obține o relație de tipare (concluzie)
Algoritm de sinteză de tip: potrivește concluziile relațiilor de sinteză de tip cu componentele expresiei până ajunge la componente elementare
Siguranță de tip: progres (expresia bine tipată nu se blochează), conservare (rămâne bine tipată după evaluare)
Unificare: cea mai generală substituție conform căreia 2 expresii de tip devin identice
O variabilă unifică cu: orice, cât timp occurs check este OK
2 constante unifică: dacă sunt identice
2 construcții unifică: dacă folosesc același constructor de tip pe argumente care unifică la rândul lor
Structuri folosite de algoritmul de unificare: stiva de constrângeri, substituția