Rust i napredni tipski sustavi

9. Rust - Još malo o tipskim sustavima

Ivan Radiček

22. svibnja 2021.

Plan

- Prošli puta:
  - Uvod u tipske sustave
- Danas:
  - Pregled (diskusija) završnog zadatka
  - Par Rust sitnica
  - Prevođenje u bytecode
  - Razni egzotični tipski sustavi

Rust: kloniranje unutar `Option`-a

```
1  fn clone_borrow(x : Option<&String>) -> Option<String> {
2    // ???
3  }
4
5  fn main() {
6    let s1 = "hello_world".to_string();
7    let s2 = clone_borrow(Some(&s1));
8    println!("{:?} {:?}", s1, s2)
9  }
```

[Igralište]

Rust: kloniranje unutar `Option`-a

- Koristi `clippy`
- Vidi: https://doc.rust-lang.org/std/iter/struct.Cloned.html

```
1  fn clone_borrow(x : Option<&String>) -> Option<String> {
2    /*match x {
3      Some(s) => Some(s.clone()),
4      None => None,
5    }*/
6    x.cloned()
7  }
8
9  fn main() {
10   let s1 = "hello_world".to_string();
11   let s2 = clone_borrow(Some(&s1));
12   println!("{:?} {:?}", s1, s2)
13 }
```

[Igralište]

Rust: `match` i posuđivanje iza pametnog pokazivača

```rust
#[derive(Debug)]
enum E {
  A(String),
  B(String),
}

fn string_length(s: &Box<E>) -> usize {
  match s {
    E::A(s) => s.len(),
    E::B(s) => s.len(),
  }
}

fn main() {
  let s = Box::new(E::A("hello_world".to_string()));
  let l = string_length(&s);
  println!("{:?} {}", s, l)
}
```

[Igralište]

Rust: `match` i posuđivanje iza pametnog pokazivača

- Vidi: https://doc.rust-lang.org/std/borrow/trait.Borrow.html

```
1  use std::borrow::Borrow;
2
3  #[derive(Debug)]
4  enum E { ... }
5
6  fn string_length(s: &Box<E>) -> usize {
7    match s.borrow() {
8      E::A(s) => s.len(),
9      E::B(s) => s.len(),
10   }
11 }
12
13 fn main() {
14   let s = Box::new(E::A("hello_world".to_string()));
15   let l = string_length(&s);
16   println!("{:?} {}", s, l)
17 }
```

[Igralište]

VM model

- Prevođenje funkcijskog jezika (baziranu na tipovima) u VM instrukcije

Virtualni stroj

Sastoji se od liste **instrukcija**, **stacka** ($\mathcal{S}$), **heapa** ($\mathcal{H}$), **registra sa trenutnim rezultatom** ($\mathcal{R}$). U svakom koraku rada, stroj izvodi jednu instrukciju. Instrukcije se izvode sekvencijalno, osim ako instrukcija sama promijeni redoslijed izvođenja (npr. pozivanje funkcije). Svaka instrukcija na neki način mijenja gore navedene elemente stroja.
Jedina **vrijednost** na stacku, heapu ili u registru je **cijeli broj**, koji može biti i **adresa u memoriji** (slično kao i prava računala).

Instrukcije - aritmetika

Konstanta

`const` n stavlja broj n (konstantu) u registar $\mathcal{R}$, tj. poslje izvođenja imamo $\mathcal{R} = n$.

Zbrajanje

`add` A_1 A_2 - zabraja argument A_1 i A_2 i sprema rezultat u registar vrijednosti $\mathcal{R}$. Slično možemo definirati i za druge operacije (množenje, dijeljenje, usporedbu, ...).

Argument

Argument A može biti ili broj n ili element na stacku $\&n$ (element odmaknut n mjesta od vrha stacka)

- `add 2 3` $\Rightarrow \mathcal{R} = 5$
- `add 2 &3` sa $\mathcal{S} = 5, 34567, 345134, 10, \ldots \Rightarrow \mathcal{R} = 12$

Instrukcije - stack

Dodavanje vrijednosti na stack

`push` vrijednost iz registra $\mathcal{R}$ postavlja na vrh stack. Drugim riječima, ako je prije izvođenja instrukcije `push` imamo: $\mathcal{S} = S$ i $\mathcal{R} = n$, onda nakon izvođenja imamo $\mathcal{S} = n, S$

Uklanjanje vrijednosit sa stacka

`pop` i `pop` k uklanjaju jednu ili k vrijednosti sa stacka. Ako prije izvođenja imamo $\mathcal{S} = n_1, \ldots, n_k, S$, onda nakon izvođenja imamo $\mathcal{S} = S$.

Dohvaćanje vrijednosti sa stacka

`get` &k dohvaća k-ti element (počevši od vrha) stacka. Ako je stack $\mathcal{S} = n_1, \ldots, n_k, S$, onda nakon izvođenja imamo $\mathcal{R} = n_k$

Prijevod: varijable i let-izraz

$$\frac{\Gamma = x_0 : \tau_0, \ldots, x_k : \tau \ldots}{\Gamma \vDash x : \tau \mid \texttt{get}\ k}$$

$$\frac{\Gamma \vDash e_1 : \tau' \mid l_1 \qquad x : \tau, \Gamma \vDash e_2 : \tau \mid l_2}{\Gamma \vDash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : \tau \mid l_1, \texttt{push}, l_2, \texttt{pop}}$$

Prijevod: aritmetika

$$\frac{\Gamma \vDash int : e_1 \mid a_1 \qquad \Gamma \vDash int : e_1 \mid a_2}{\Gamma \vDash e_1 + e_2 : int \mid \texttt{add}\ a_1\ a_2}$$

Na primjer, sljedeći izraz (program) . . .

```
1  let x = 3 in
2  let y = x + 5 in
3  x + y
```

. . . prevodimo u:

```
const 3
push
add &0 5
push
add &1 &0
pop
pop
```

(ili optimizacija: `pop 2`)

Heap memorija

Kreiranje novog bloka u memoriji

`mkblock` k kreira novi (nezauzeti) blok u heap memoriji duljine k, potom uzima i uklanja zadnjih k vrijednosti sa stacka i kopira ih u novu heap memoriju. Nova adresa je spremljena je u registar. Preciznije, ako prije izvođenja imamo $\mathcal{S} = n_1, \ldots, n_k, S$ i $\mathcal{H} = H$, onda nakon izvođenja imamo $\mathcal{S} = S$, $heap = a \to n_1, (a+1) \to n_2, \ldots, (a+k-1) \to n_k, H$ i $\mathcal{R} = a$ (pretpostavljamo da su adrese, $a, \ldots, a+k-1$ slobodne).

Dohvaćanje heap memorije

`load` k, ukoliko imamo $\mathcal{R} = a$ učitava adresu $a + k$ iz heapa ($\mathcal{H}$) i sprema vrijednost u $\mathcal{R}$.

n-torke

$$\frac{\Gamma \vDash s_1 : \tau_1 \mid i_1 \qquad \ldots \qquad \Gamma \vDash s_k : \tau_k \mid i_k}{\Gamma \vDash (s_1, \ldots, s_k) : (\tau_1, \ldots, \tau_k) \mid i_1, \texttt{push}, \ldots, i_k, \texttt{push}, \texttt{mkblock}\ k}$$

$$\frac{\Gamma \vDash x : (\tau_1, \ldots, \tau_k) \mid i \qquad 1 \leq j \leq k}{\Gamma \vDash x.j : \tau_i \mid i, \texttt{load}\ j}$$

Skok

- VM izvodi jednu po jednu naredbu u nizu
- Naredbe skoka mijenjaju taj tok

Promjena tijeka izvođenja

`jmp` k prebacuje tok na k – tu naredbu u nizu (ta se izvodi sljedeća). U pisanju naredbi ovično ne pišemo k direktno već u obliku neke oznake (*label*) ℓ koja se kasnije automatski prevodi u određeni indeks u nizu naredbi.

Uvijetna promjena tijeka izvođenja

`jmpz` k prebacuje tok na k – tu ako je trenutna vrijednost $\mathcal{R} = 0$. Slično `jmpn` prebacuje tok ako je trenutna vrijednost $\mathcal{R} \neq 0$. Ukoliko uvijeti nisu ispunjeni, tijek izvođena se nastavlja kao da tih naredbi nema.

Grananje

$$\frac{}{\Gamma \vDash \texttt{true} : \mathit{bool} \mid \texttt{const } 1} \qquad \frac{}{\Gamma \vDash \texttt{false} : \mathit{bool} \mid \texttt{const } 0}$$

$$\frac{\Gamma \vDash x : \mathit{bool} \mid i \qquad \Gamma \vDash e_1 : \tau \mid I_1 \qquad \Gamma \vDash e_2 : \tau \mid I_2}{\Gamma \vDash \texttt{if } x \texttt{ then } e_1 \texttt{ else } e_2 : \tau \mid i, \texttt{jmpz } \ell_f, I_1, \texttt{jmp } \ell_e, \ell_f : I_2, \ell_e :}$$

Liste

$$\frac{}{\Gamma \vDash \texttt{nil} : \mathit{List}[\tau] \mid \texttt{const } 0}$$

$$\frac{\Gamma \vDash x : \tau \mid i_1 \qquad \Gamma \vDash y : \mathit{List}[\tau] \mid i_2}{\Gamma \vDash x :: y : \mathit{List}[\tau] \mid i_1, \texttt{push}, i_2, \texttt{push}, \texttt{mkblock } 2}$$

$$\frac{\begin{array}{c}\Gamma \vDash x : \mathit{List}[\tau'] \mid i \\ \Gamma \vDash e_1 : \tau \mid l_1 \qquad h : \tau', t : \mathit{List}[\tau'] \Gamma \vDash e_2 : \tau \mid l_2\end{array}}{\begin{array}{c}\Gamma \vDash \texttt{match } x \texttt{ with nil} \Rightarrow e_1,\ h :: t \Rightarrow e_2 : \tau \mid \\ i, \texttt{jmpn } \ell_t, l_1, \texttt{jmp } \ell_e, \ell_t : \texttt{load } 1, \texttt{push}, i, \texttt{load } 0, \texttt{push}, l_2, \ell_e :\end{array}}$$

Funkcije

- Ideja je da se funkcija sastoji od:
  - Adrese gdje počinje kod tijela funkcije
  - Dodatne memorije koju funkcija zahvaća, izvan svojega scopea
- Konkretno, funkcija je blok na heapu, duljine $m + 2$, gdje je m duljina dodatne memorije:

a_f	m	n_1	$\cdots$	n_m

Poziv funkcije

`call` poziva funkciju, gdje $\mathcal{R}$ sadrži pokazivač na opis funkcije na heapu. Odnosno, ako je $\mathcal{R} = a$ i $\mathcal{H} = a \to a_f, (a+1) \to m, (a+2) \to n_1, \ldots, (a+m+2) \to n_m, H$, tada $\mathcal{S} = a_{\text{pc}}, a, n_1, \ldots, n_m, S$ (gdje je a_{pc} povratna adresa, odnosno pokazivač na instrukciju nakon trenutne), a tok izvođenja *skače* na adresu a_f.

Funkcije

Poziv funkcije

`call` poziva funkciju, gdje $\mathcal{R}$ sadrži pokazivač na opis funkcije na heapu. Odnosno, ako je $\mathcal{R} = a$ i $\mathcal{H} = a \to a_f, (a+1) \to m, (a+2) \to n_1, \ldots, (a+m+2) \to n_m, H$, tada $\mathcal{S} = a_{\text{pc}}, a, n_1, \ldots, n_m, S$ (gdje je a_{pc} povratna adresa, odnosno pokazivač na instrukciju nakon trenutne), a tok izvođenja *skače* na adresu a_f.

Povratak iz funkcije

`ret` k se vraća iz funkcije koja ima k parametara i dodatne memorije - instrukcija počisti sve sa stacka vezano uz poziv funkcije te vraća tok kontrole pozivatelju funkcije. Odnosno, ako je prije instrukcije, $\mathcal{S} = a_{\text{pc}}, a, n_1, \ldots, n_k, S$, tada nakon izvođenja instrukcije imamo $\mathcal{S} = S$ i sljedeća naredba koja se izvodi je na adresi a_{pc}. (Povratna vbrijednost je u registru $\mathcal{R}$.)

Prevođenje funkcije i poziva

$$\frac{\begin{array}{c} y_1, \ldots, y_c = \mathit{freevars}(\lambda x_1, \ldots, x_p.\ e1) \\ \Gamma' = - : (), f : (\tau_1, \ldots, \tau_p) \to \tau_r, y_1 : \Gamma(y_1), \ldots, y_c : \Gamma(y_c), x_1 : \tau_1, \ldots, x_p : \tau_p \\ \Gamma' \vDash e_2 : \tau_r \mid I_f \qquad f : (\tau_1, \ldots, \tau_p) \to \tau_r, \Gamma \vDash e_2 : \tau \mid I \end{array}}{\begin{array}{c} \Gamma \vDash \texttt{let}\ f\ (x_1, \ldots, x_p) = e_1\ \texttt{in}\ e_2 : \tau \mid \quad \texttt{jmp}\ \ell, \ell_f : I_f, \texttt{ret}\ (p + c), \\ \ell : \texttt{get}\ \&(|y_c|), \texttt{push}, \ldots, \texttt{get}\ \&(|y_1|), \texttt{push}, \\ \texttt{const}\ c, \texttt{push}, \texttt{const}\ \ell_f, \texttt{push}, \texttt{mkblock}\ (c + 2), \texttt{push}, I, \texttt{pop} \end{array}}$$

$$\frac{\begin{array}{c} \Gamma \vDash f : (\tau_1, \ldots, \tau_m) \to \tau \mid i_f \\ \Gamma \vDash x_1 : \tau_1 \mid \texttt{get}\ \&o_1 \qquad \ldots \qquad \Gamma \vDash x_m : \tau_m \mid \texttt{get}\ \&o_m \end{array}}{\begin{array}{c} \Gamma \vDash f\ (x_1, \ldots, x_m) : \tau \mid \texttt{get}\ \&o_1, \texttt{push}, \ldots, \texttt{get}\ \&(o_m + m - 1), \texttt{push}, \\ i_f, \texttt{call} \end{array}}$$

Primjer prijevoda (moguće krivi :))

```
1  let add (n) =
2    let f (x) = n + x
3    in f
4  in
5  let g = add (5) in
6  g(10)
```

```
                jmp ℓ_add'
ℓ_add :         jmp ℓ_f'
ℓ_f :           add &2 &3          # (n + x)
                ret 2
ℓ_f' :          get &2             # (učitaj n)
                push
                const 1            # (duljina dodatne memorije)
                push
                const (ℓ_f)
                push
                mkblock 3          # (kreiranje f bloka)
                push
                get &0
                pop 1              # (!!!)
                ret 1              # (vraćanje f iz add)
ℓ_add' :        const 0            # (duljina dodatne memorije)
                push
                const (ℓ_add)
                push
                mkblock 2          # (kreiranje add bloka)
                push
                const 5
                push
                get &1             # (dohvaćanje f-a)
                call               # (poziv f-a s 5)
                push               # (spremanje g vrijednosti)
                const 10
                push
                get &1             # (dohvaćanje g-a)
                call               # (poziv g-a s 10)
                pop 1              # (čišćenje - 1 let)
```

Primjer Pythona in OCamla

```python
def main():
  def add(n):
    def f(x): return n + x
    return f

  g = add(5)
  add(10)

import dis
dis.dis(main)
```

```
let h ()  =
  let add n =
    let f x = n + x in f
  in
  let g = add 5 in
  g 10

let () =
  print_endline (string_of_int (h ()))
```

```
ocamlc -c -dinstr prog.ml
```

Tip mjernih jedinice

- Uz običan brojčani tip (npr. `float`), dodajemo informaciju mjerne jedinice (npr. $kg \cdot m^{-1} \cdot s^{-2}$)
- Tako npr. funkcija koja prima `f64<m/s>`, ne može primiti `f64<m/s^2>`
- Ta dodatna informacija nestaje kod izvođenja (postoji samo kod kompajliranja)
- U jeziku F# postoji ugrađeno, a u nekima se može definirati pomoću trikova

Trikovi za implementaciju

- Tip se može definirati kao vektor (lista) eksponenata SI jedinica
- Npr. uzmimno podskup SI jedinica: kg, m, s:
  - $kg = \langle 1, 0, 0 \rangle = kg^1 \cdot m^0 \cdot s^0$
  - $m = \langle 0, 1, 0 \rangle$
  - $a = \frac{m}{s^2} = m \cdot s^{-2} = \langle 0, 1, -2 \rangle$
  - $x : \langle n_1, n_2, n_3 \rangle \cdot y : \langle m_1, m_2, m_3 \rangle = \langle n_1 + m_1, n_2 + m_2, n_3 + m_3 \rangle$
  - Slično za druge operacije
- U Rustu postoji crate `dimensioned` [1], koji koristi traitove (za definiranje operacija), makroe (za definiranje svih mogućih veza između tipova) i *brojčane* trikova nad tipovima (crate `typenum` [2])
- Navodno se može implementirati i u C++-u

[1] https://github.com/paholg/dimensioned
[2] https://docs.rs/typenum/1.13.0/typenum/

Duljina liste

- Većina sljedećih stvari nije implementirana u mainstream jezicima
- Tipu liste dodamo parametar duljine (isto kao i mjerne jedinice), pa imamo $List[\tau, n]$ za listu gdje su elementi tipa τ i ima duljinu n.
- Duljina se transformira sa sljedeća dva pravila:

$$\frac{\Gamma \vDash e_1 : \tau \qquad \Gamma \vDash e_2 : List[\tau, n]}{\Gamma \vDash e_1 :: e_2 : List[\tau, n+1]}$$

$$\frac{\Gamma \vDash e : List[\tau_\ell, n] \qquad \Gamma \vDash e_1 : \tau \qquad \Gamma, h : \tau_\ell, t : List[\tau_\ell, n-1] \vDash e_1 : \tau}{\Gamma \vDash \texttt{match}\ e\ \texttt{with}\ \texttt{nil} \Rightarrow e_1, \quad h :: t \Rightarrow e_2 : \tau}$$

Primjeri duljine liste

Spajanje dvije liste

```
1  let append (l1: List[T, n],
2              l2: List[T, m]) -> List[T, m] =
3    match l1 with
4      nil => l2     (* : List[T, 0 + m]  *)
5      h::t =>
6        let tt : List[T, n - 1 + m] = append (t, l2) in
7        h::tt
```

- Spajanjem dviju listi dobijemo novu listu duljine zbroja duljina ulaznih listi

Dodajmo duljini i uvjete

```
1  let tail (l : List[T, n | n > 0]) -> List[T, n - 1] =
2    match l with
3      nil => !, (* Nemoguć slučaj  *)
4      h::t => t
```

Kompleksnost izvođenja

- Tipu funkcije dodajemo kompleksnost, odnosno (apstraktan!) broj koraka izvođenja, npr. $(\tau_1, \ldots, \tau_n) \xrightarrow{n} \tau$ ili $(\tau_1, \ldots, \tau_n) \xrightarrow{2+n^3} \tau$

```
1  let insert (x : T, l : List[T, n]) -(n)-> List[T, n+1] =
2    match l with
3      nil => [x],
4      h::t => if x < h then x::h::t else h::(insert, x, t)
5      in
6    let isort (l : List[T, n]) -(n * n)-> List[T, n] =
7      match l with
8        nil => []
9        h::t => insert(h, isort(l))
```

Stvarni sustavi za kompleksnost

- U literaturi postoje sustavi koji mogu automatski izvesti takav tip
- Npr. kod niže (OCaml sintaksa) unijeti u sučelje na: https://www.raml.co/interface/

```
1  let rec insert (x:int) l =
2    match l with
3      | [] -> [x]
4      | h::t -> if x < h then (x::h::t) else h::(insert x t)
5
6  let rec isort l =
7    match l with
8      | [] -> []
9      | h::t -> insert h (isort t)
```

- Moguće dodati još preciznije logike tipovima koja bi omogućila još preciznije određivanje kompleksnosti

Sub-strukturni tipovi

- Podsjetimo se pravila za konstruiranje para

$$\frac{\Gamma \vDash e_1 : \tau \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash (e_1, e_2) : (\tau_1, \tau_2)}$$

Sub-strukturni tipovi

- Podsjetimo se pravila za konstruiranje para
- I jedna konkretne instance

$$\frac{\Gamma \vDash e_1 : \tau \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash (e_1, e_2) : (\tau_1, \tau_2)} \qquad \frac{}{x : \texttt{String} \vDash (x, x) : (\texttt{String}, \texttt{String})}$$

Sub-strukturni tipovi

- Podsjetimo se pravila za konstruiranje para
- I jedna konkretne instance

$$\frac{\Gamma \vDash e_1 : \tau \qquad \Gamma \vDash e_2 : \tau}{\Gamma \vDash (e_1, e_2) : (\tau_1, \tau_2)} \qquad \frac{}{x : \texttt{String} \vDash (x, x) : (\texttt{String}, \texttt{String})}$$

- U Rustu ovo očito ne funkcionira ovako jednostavno!

- Problem je što *"slobodno dijelimo"* okolinu Γ
- Možemo i ograničiti ($\circ$ znači da okoline ne dijele varijable/resurse/memoriju)

$$\frac{\Gamma_1 \vDash e_1 : \tau_1 \qquad \Gamma_2 \vDash e_2 : \tau_2}{\Gamma_1 \circ \Gamma_2 \vDash (e_1, e_2) : (\tau_1, \tau_2)}$$

- Samo jedan korisnik varijable: slična ideja kao u Rustu (moguće dodati i reference ili posuđivanje i sl.)
- Dodavanjem dodatnih pravila vezanih uz okolinu možemo kontrolirati korištenje resursa (memorije u ovom slučaju)
- Vidi općenito:
  https://en.wikipedia.org/wiki/Substructural_type_system

Završne misli

- Programiranje je građenje apstrakcija pomoću alata **programskog jezika**
- Tipovi su jedan od alata i pomažu nam kod organizacija, paze na nas da ne progriješimo, ...
- Postoje razno razni tipovi i tipski sustavi