Rust

niezwykła wydajność
niezwykłe bezpieczeństwo
niezwykłe piękno

M. Kotwica

11.05.2017

M. Kotwica

KNI Kernel

ACK Cyfronet AGH

Porządek rzeczy

1. tożsamość
   - 1.1 ze zmiennymi jest coś nie tak?
   - 1.2 co daje zmiennej ciągłość
   - 1.3 przekształcanie
2. właściciele
   - 2.1 z referencjami jest coś nie tak?
   - 2.2 pożyczenia
   - 2.3 przekazania
3. funkcyjność
   - 3.1 ze stanem jest coś nie tak?
   - 3.2 mut-var-const-val vs const-var-mut-val
   - 3.3 wpływ stałego stanu na wydajność

Porządek rzeczy

Dlaczego Rust, nawet dla początkujących?

Zaplecze:

- Cargo (menadżer pakietów)
- rustc --explain
- dużo podpowiedzi kompilatora
- komentarze dokumentacji
- duże community

Programista:

- zasady zgodne z dobrymi praktykami
- elementy funkcyjności
- jawna obsługa błędów

Algorytmik:

- bezkosztowe cechy wysokopoziomowe
- oparty o wyrażenia
- inferencja typów w obrębie funkcji

Inżynier oprogramowania:

- system modułów
- generyczność i rozszerzalność
- system typów oparty o cechy

Imperative approach (C++)

```
class precvector {
  precvector
};

precvector change_precision(precvector vec) {

}
```

Generic approach (C++ STL)

```
{
  auto el_new = vec.begin();
  for(auto& el_old : vec) {
    if(pred(el_old)) {
      *el_new = std::move(el_old);
      ++el_new;
    }
  }
  vec.resize(el_new - vec.begin());
}
```

Functional approach (C++ STL and Rust)

C++ STL

```
vec.erase(
  remove_if(vec.begin(),
            vec.end(),
            [](int x) { !pred(x); }),
  vec.end()
);
```

Rust

```
vec = vec.into_iter()
         .filter(pred)
         .collect();
```

Pros of functional approach

- highly generic
- no intermediate state visible
- no invariant analysis needed
- often easily composable:

```
let promoters = students
    .into_iter()
    .filter_map(|s| s.promoter)    // if has promoter, yield promoter
    .unique()                      // remove any duplicates
    .sorted_by(|p| p.surname);     // sort yielded promoters by surname
                                   //   and place them in a vector
```

What makes the variable itself

Immutable value variable binding

```
var a: List = 1 :: 3 :: Nil
a = 4 :: a
a = (5 :: 1 :: Nil) ++ a
```

1. can be reassigned
2. the value itself is immutable
3. similar to ‘T const *‘ in C++

Mutable value constant binding

```
val b = new ListBuffer[Int]()
b += 3
b += (1, 4)
b ++= Seq(1, 5)
```

1. cannot be reassigned
2. the value itself is mutable
3. similar to ‘const T *‘ in C++