Rust

niezwykła wydajność
niezwykłe bezpieczeństwo
niezwykłe piękno

M. Kotwica

30.03.2017

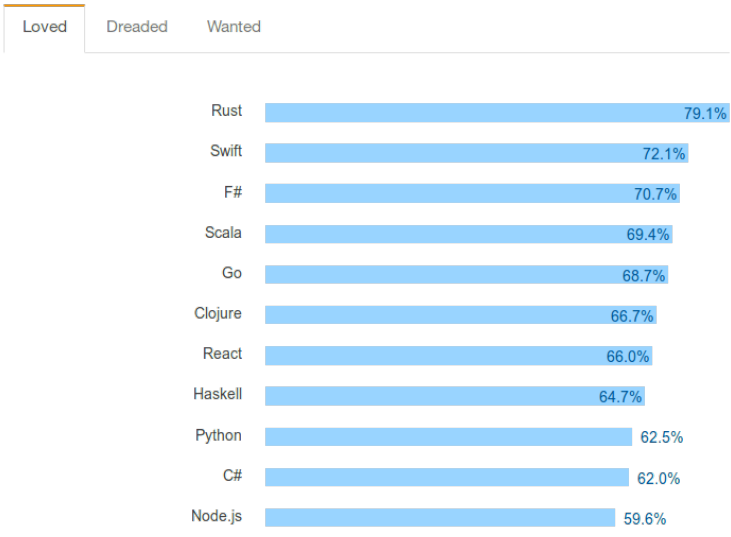
Część I: wprowadzenie

M. Kotwica

KNI Kernel

ACK Cyfronet AGH

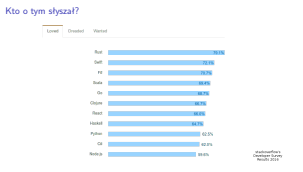
Kto o tym słyszał?



stackoverflow's Developer Survey Results 2016

Rust

Kto o tym słyszał?

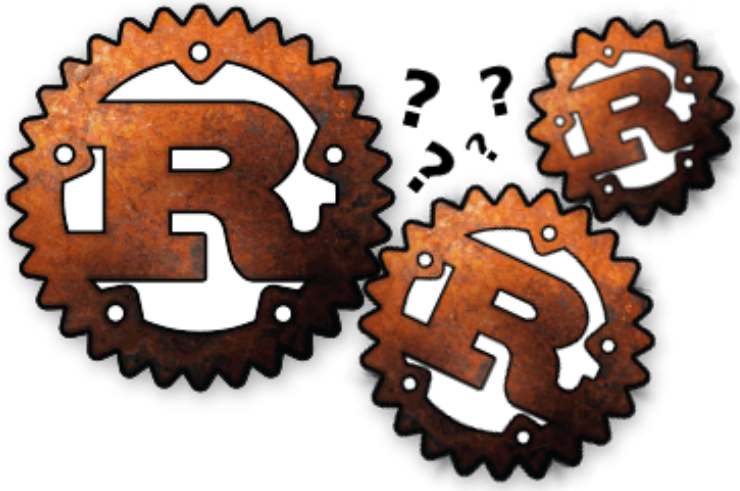


maj 2015 wersja 1.0 Rusta
kwiecień 2016 ta ankieta na stackoverflow

Podobny sukces przeżywały Swift i Go, wspierane przez ogromne korporacje. Rust miał wsparcie Mozilli (obecnie już go nie ma, jest całkowicie community-based).

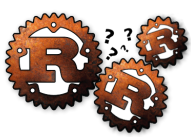
Kompilator oparty o LLVM, otwartoźródłowy.

Ale dlaczego tak?



- menadżer pakietów
- wygodne testy
- użyteczne linty
- bogata semantyka
- ipo i cross-inlining
- system pożyczeń
- iteratory
- destrukturyzacja
- cechy
- generyki
- makra
- wtyczki

Ale dlaczego tak?



- menadżer pakietów
- wygodne testy
- użyteczne linty
- bogata semantyka
- ipo i cross-inlining
- system pożyczeń
- iteratory
- destrukturyzacja
- cachy
- generyki
- makra
- wtyczki

Linty to podpowiedzi kompilatora. Rust jest jednym z niewielu języków, w których można napisać pakiet, który można zaimportować w swoim pakiecie jako wtyczkę i w ten sposób dodać nowe linty.

Clippy — popularny pakiet lintów. Sprawdza konwencje i dobre praktyki. Dostępny jako pakiet-wtyczka oraz jako osobne polecenie ‘cargo clippy‘.

Istnieje REPL Rusta, rusti.

Rusta można podpiąć pod Visual Studio oraz QtCreator.

Cargo

- tworzenie projektów
- budowa
- testy
- benchmarki
- dystrybucja
- szukanie
- dodatkowe podpowiedzi

cargo new	tworzy nowy projekt (w tym tworzy katalogi)
cargo init	inicjalizuje projekt w istniejącym katalogu
cargo build	buduje projekt włącznie z zależnościami
cargo run	uruchamia plik wykonywalny projektu
cargo test	wykonuje wszystkie testy
cargo publish	publikuje projekt na https://crates.io/

nazwa	**położenie**	**importy**	**co testuje**
Doc-test	dokumentacja	środowisko elementu	element
Fun-test	osobna funkcja	jak reszta modułu	moduł
Mod-test	moduł testowy	dowolne jak moduł	poddrzewo modułów
Crate-test	pakiet testowy	jak pakiet kliencki	cały pakiet

Przykład praktyczny

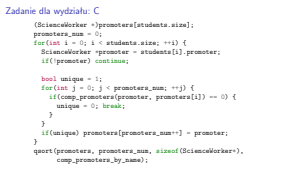
Proszę zebrać listę promotorów wszystkich studentów utworzoną w porządku alfabetycznym.

Jak się za to zabrać:

- z listy wszystkich studentów...
- ...wyciągnąć promotorów
- każdy promotor brany raz
- porządek alfabetyczny (nazwisko, imię)

Zadanie dla wydziału: C

```
(ScienceWorker *)promoters[students.size];
promoters_num = 0;
for(int i = 0; i < students.size; ++i) {
  ScienceWorker *promoter = students[i].promoter;
  if(!promoter) continue;

  bool unique = 1;
  for(int j = 0; j < promoters_num; ++j) {
    if(comp_promoters(promoter, promoters[i]) == 0) {
      unique = 0; break;
    }
  }
  if(unique) promoters[promoters_num++] = promoter;
}
qsort(promoters, promoters_num, sizeof(ScienceWorker*),
      comp_promoters_by_name);
```

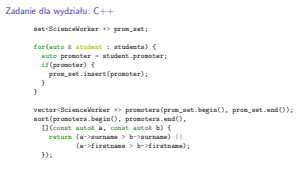


W zamieszczonym kodzie zakładamy, że funkcje `comp_promoters` oraz `comp_promoters_by_name` są dostępne. Cały kod zająłby więc jeszcze węcej miejsca.

Jak widać kod jest poszatkowany względem funkcjonalności i mało czytelny. Dobrze byłoby go opatrzeć kilkoma komentarzami.

Zadanie dla wydziału: C++

```
set<ScienceWorker *> prom_set;

for(auto & student : students) {
  auto promoter = student.promoter;
  if(promoter) {
    prom_set.insert(promoter);
  }
}

vector<ScienceWorker *> promoters(prom_set.begin(), prom_set.end());
sort(promoters.begin(), promoters.end(),
  [](const auto& a, const auto& b) {
    return (a->surname > b->surname) ||
           (a->firstname > b->firstname);
  });
```

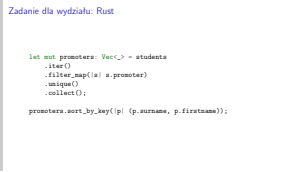


Przykład jest napisany w C++14, przed tym standardem w funkcji anonimowej należałoby jawnie podać typy wejściowe.

Jest znacznie lepiej niż w C. Zbiór wyodrębnił funkcjonalność unikalności, a komparator sortowania jest deklarowany inline. Mimo to kod jest dość długi.

Zadanie dla wydziału: Rust

```
let mut promoters: Vec<_> = students
    .iter()
    .filter_map(|s| s.promoter)
    .unique()
    .collect();

promoters.sort_by_key(|p| (p.surname, p.firstname));
```



Końcówkę można rozwiązać jeszcze na dwa sposoby:

pakiet		**cecha**		**metoda**	**działanie**
itertools	::	Itertools	::	sorted	zbiera iterator do posortowanego wektora
lazysort	::	SortedBy	::	sorted	leniwie sortuje iterator

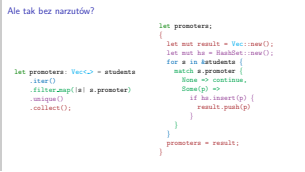
Oba pozwalają pozbyć się mutowalności zmiennej promoters.

Kod jest bardzo zwarty, nieposzatkowany.

Ale tak bez narzutów?

```
let promoters: Vec<_> = students
    .iter()
    .filter_map(|s| s.promoter)
    .unique()
    .collect();
```

```
let promoters;
{
  let mut result = Vec::new();
  let mut hs = HashSet::new();
  for s in &students {
    match s.promoter {
      None => continue,
      Some(p) =>
        if hs.insert(p) {
          result.push(p)
        }
    }
  }
  promoters = result;
}
```



Iteratory Rusta są operate o cross-inline'owane adaptatory. Kolejne struktury danych są tworzone w oparciu o poprzednie i dokładają kolejne kawałki kodu. Na końcu kod jest inline'owany i optymalizowany, co znosi koszty znane np. z Javy czy C#.

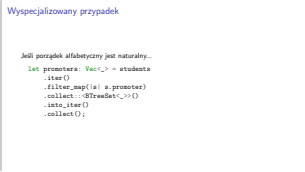
Wiązania niemutowalne: działają inaczej niż np. w C++ — można im przypisać wartość z opóźnieniem, nie można jedynie modyfikować już przypisanej wartości.

Dzięki silnej inferencji typów, żaden kontener z przykładu nie deklaruje, jaki typ przechowuje, ponieważ implikuje to typ elementów w nich umieszczanych. Wiązanie `promoters` w ogóle nie ma deklarowanego typu, ponieważ jest implikowany przez przypisanie mu wartości zmiennej `result`.

Wyspecjalizowany przypadek

Jeśli porządek alfabetyczny jest naturalny...

```
let promoters: Vec<_> = students
    .iter()
    .filter_map(|s| s.promoter)
    .collect::<BTreeSet<_>>()
    .into_iter()
    .collect();
```



Jednym z efektów generyczności iteratorów jest, że możliwa jest konwersja każdego iterowalnego kontenera do każdego kontenera, który można stworzyć iteracyjnie. Działa to nawet jeśli autorzy kontenerów nie wiedzieli nawzajem o swoich kontenerach.

Przykład pokazuje też, że metoda `.collect()` jest przeładowana po typie wynikowym (oraz wejściowym) oraz że można wybrać dowolny jej wariant. Jest znana jako jedna z najbardziej uniwersalnych metod w bibliotece standardowej Rusta.

Iteratory

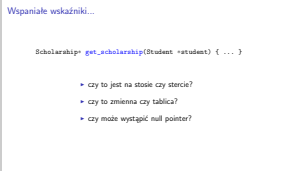
- zwięzłość
- generyczność
- rekonfigurowalność
- ochrona przed błędami
- inferencja typów
- cross-inlining
- nigdy więcej spaghetti!

Na warsztaty przewidziane jest używanie iteratorów oraz napisanie funkcji iteratorowej, tj. takiej, która przyjmuje iterator i zwraca typ niebędący iteratorem (np. `.collect()` zwraca kontener, `.sum()` zwraca skalar).

Wspaniałe wskaźniki...

```
Scholarship* get_scholarship(Student *student) { ... }
```

- czy to jest na stosie czy stercie?
- czy to zmienna czy tablica?
- czy może wystąpić null pointer?

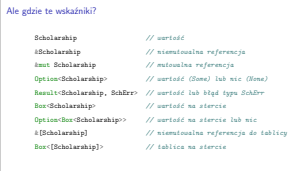


Sygnatura funkcji w języku C mówi o formie argumentów i wyniku, ale nie o ich znaczeniu. Wskaźnik może mieć bardzo wiele znaczeń, z których każde jest sensowne, a niektóre bywają sprzeczne.

Jednym z ważniejszych problemów pozostaje możliwość zwrócenia wskaźnika pustego (null pointer).

Ale gdzie te wskaźniki?

```
Scholarship                  // wartość
&Scholarship                 // niemutowalna referencja
&mut Scholarship             // mutowalna referencja
Option<Scholarship>          // wartość (Some) lub nic (None)
Result<Scholarship, SchErr>  // wartość lub błąd typu SchErr
Box<Scholarship>             // wartość na stercie
Option<Box<Scholarship>>     // wartość na stercie lub nic
&[Scholarship]               // niemutowalna referencja do tablicy
Box<[Scholarship]>           // tablica na stercie
```

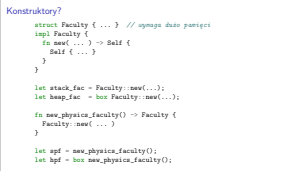


W Ruście wprawdzie występują czyste wskaźniki, jednak ich użycie jest ograniczone do bloków niebezpiecznych (unsafe), używa się ich rzadko. W normalnym użyciu zamiast wskaźników występują ich odpowiedniki semantyczne. Część z nich (np. `Option<Box<_>>`) ma nawet taką samą implementację.

Nie tylko sygnatury funkcji mówią szczegółowo o tym, jak należy interpretować zwracany wynik lub przyjmowany argument, ale również język broni przed ich niewłaściwym użyciem, np. `Option<Box<_>>` nie podlega arytmetyce wskaźników, ponieważ jest opcjonalnym unikalnym wskazaniem na obiekt, nie zaś tablicę.

Konstruktory?

```
struct Faculty { ... }  // wymaga dużo pamięci
impl Faculty {
  fn new( ... ) -> Self {
    Self { ... }
  }
}

let stack_fac = Faculty::new(...);
let heap_fac  = box Faculty::new(...);

fn new_physics_faculty() -> Faculty {
  Faculty::new( ... )
}

let spf = new_physics_faculty();
let hpf = box new_physics_faculty();
```

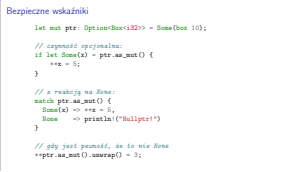


Typ `Box<_>` ukazuje kolejną cechę Rusta różniącą go od C++ — brak konstruktorów. De facto każda rutyna, która na końcu tworzy dany obiekt może być traktowana jak jego konstruktor. Czasami może zostać w podobny sposób zoptymalizowana rutyna, która zwraca obiekt tworzony w dowolnej części swojego ciała.

Opisywane zjawisko widać w braku kopii przy umieszczaniu na stercie wyniku działania konstruktora niezależnie od tego, czy jest metodą statyczną `::new()` (lub dowolną inną metodą) czy funkcją.

Bezpieczne wskaźniki

```
let mut ptr: Option<Box<i32>> = Some(box 10);

// czynność opcjonalna:
if let Some(x) = ptr.as_mut() {
    **x = 5;
}

// z reakcją na None:
match ptr.as_mut() {
  Some(x) => **x = 5,
  None    => println!("Nullptr!")
}

// gdy jest pewność, że to nie None
**ptr.as_mut().unwrap() = 3;
```



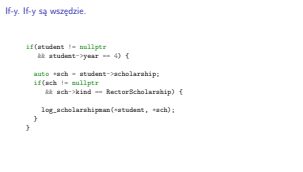
`Option<Box<_>>` udostępnia funkcjonalność podobną jak często wskaźnik — opcjonalnego odwołania do obiektu na stercie, ale wymusza jawne sprawdzanie przypadku wskazania pustego (`if let` lub `match`) lub jawne zignorowanie tego przypadku (`.unwrap()`).

Mutowalność zmiennej `Option<Box<T>>` pozwala podstawić pod nią nową wartość opcjonalną, np. `None`. Metoda `.as_mut()` pozwala uzyskać wartość typu `Option<&mut Box<T>>`. Po odpakowaniu jedną z wymienionych już wcześniej metod uzyskiwana jest wartość typu `&mut Box<T>`. Tę można zdereferować do `&mut T`, a następnie pod nią kolejną dereferencją podstawić nową wartość `T`.

Choć obsługa jest dość złożona, daje precyzyjną kontrolę dostępu oraz gwarantuje destrukcję wartości, zapobiegając wyciekowi pamięci.

If-y. If-y są wszędzie.

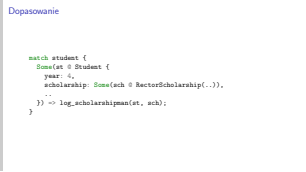
```
if(student != nullptr
   && student->year == 4) {

  auto *sch = student->scholarship;
  if(sch != nullptr
     && sch->kind == RectorScholarship) {

    log_scholarshipman(*student, *sch);
  }
}
```



W języku C++ często potrzeba sprawdzić wiele warunków sformułowanych dla kolejnych pól jakiejś struktury danych. Sprawdzenie takie musi być wykonywane ręcznie, poprzez kombinację if-ów oraz wiązań. Daje to kod mało zwarty i czytelny.

Dopasowanie

```
match student {
  Some(st @ Student {
    year: 4,
    scholarship: Some(sch @ RectorScholarship(..)),
    ..
  }) => log_scholarshipman(st, sch);
}
```



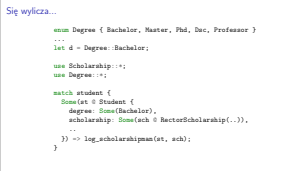
Rust udostępnia funkcjonalność zwaną destrukturyzacją. Pozwala ona rozkładać obiekty na prostsze zmienne składnią dokładnie odwrotną niż przy ich konstrukcji: po lewej znajduje się wzorzec oraz nazwy wiązań, po lewej zaś kod, który je wykorzystuje.

W destrukturyzacji i dopasowaniach występuje kilka symboli specjalnych:

symbol	znaczenie
`_`	dowolne pole
`..`	dowolna liczba pól
`@`	wiązanie wzorca
`|`	alternatywa
`()`	grupowanie
`if`	dodatkowe warunki

Się wylicza...

```
enum Degree { Bachelor, Master, Phd, Dsc, Professor }
...
let d = Degree::Bachelor;

use Scholarship::*;
use Degree::*;

match student {
  Some(st @ Student {
    degree: Some(Bachelor),
    scholarship: Some(sch @ RectorScholarship(..)),
    ..
  }) => log_scholarshipman(st, sch);
}
```



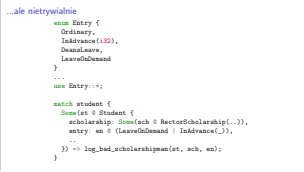
Wyliczenia w Ruście działają nie jak `enum` w C, ale `enum class`, tj. tworzą nowy zakres nazw oraz nie są niejawnie kompatybilne z typami liczbowymi.

Elementy wyliczenia można zaimportować tak samo jak elementy modułu.

Wyliczenia są często stosowane w dopasowaniu wzorców.

...ale nietrywialnie

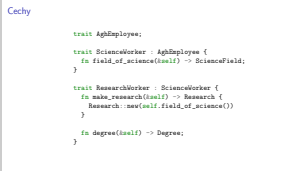
```
enum Entry {
  Ordinary,
  InAdvance(i32),
  DeansLeave,
  LeaveOnDemand
}
...
use Entry::*;

match student {
  Some(st @ Student {
    scholarship: Some(sch @ RectorScholarship(..)),
    entry: en @ (LeaveOnDemand | InAdvance(_)),
    ..
  }) => log_bad_scholarshipman(st, sch, en);
}
```



W przeciwieństwie do `enum class` C++, wyliczenia Rusta mogą mieć swoje pola, które również mogą być wykorzystywane w dopasowaniu wzorców. Z punktu widzenia teorii typów są to typy sumacyjne, zaś ich niskopoziomową implementacją jest tzw. bezpieczna unia, tj. unia, która zawiera flagę informującą, który wariant jest w danym momencie aktywny.

Cechy

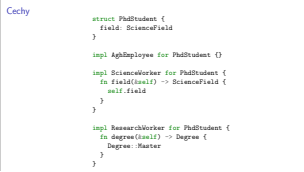
```
trait AghEmployee;

trait ScienceWorker : AghEmployee {
  fn field_of_science(&self) -> ScienceField;
}

trait ResearchWorker : ScienceWorker {
  fn make_research(&self) -> Research {
    Research::new(self.field_of_science())
  }

  fn degree(&self) -> Degree;
}
```



Cechy są odpowiednikiem typów abstrakcyjnych. Mogą służyć za etykiety, udostępniać pewne metody, a nawet domyślne implementacje. Prócz tego mogą w nich występować typy oraz składowe statyczne. Cechy mogą dziedziczyć po sobie. Typ spełniający daną cechę musi spełniać cechy, po których dziedziczy ta cecha oraz implementować wszystkie żądane przez nie metody, dla których nie dostarczyły implementacji domyślnej.

Cechy

```
struct PhdStudent {
  field: ScienceField
}

impl AghEmployee for PhdStudent {}

impl ScienceWorker for PhdStudent {
  fn field(&self) -> ScienceField {
    self.field
  }
}

impl ResearchWorker for PhdStudent {
  fn degree(&self) -> Degree {
    Degree::Master
  }
}
```



Implementacja cechy dla danego typu jest separowana od implementacji innych cech. Powoduje to, że ten sam typ może wypełniać kilka cech dostarczających metod o tej samej nazwie, a nawet sygnaturze.

Metody cechy mogą być użyte tylko kiedy zostanie ona zaimportowana w danym module. Jeśli mimo to występuje konflikt nazw — wystarczy sprecyzować, o metodę jakiej cechy chodzi. Najważniejszą zaletą jest, że implementacja danej cechy dla dowolnego typu nie wpływa na moduły, które nie importują tej cechy.

Dodatkowo cechę można zaimplementować jedynie w pakiecie definiującym dany typ lub daną cechę. Dzięki temu nie można arbitralnie dodawać znanych cech do znanych typów.

A już na kolejnym wykładzie

- zarządzanie własnością
- komórki
- generyki
- ograniczanie
- metaprogramowanie

Tak że Rusta to wy szanujcie!

Graphic by zen3ger from Rust announcements