Rust z perspektywy programisty C++

Rust to stosunkowo nowy gracz na rynku systemowych języków programowania. Choć jego historia sięga roku 2006, to o prawdziwej popularności można mówić dopiero w kontekście początku obecnej dekady. Został on wtedy wzięty pod skrzydła organizacji Mozilla, a niedługo później jego kompilator osiągnął poziom pozwalający mu skompilować samego siebie. Przez kolejne kilka lat Rust ulegał burzliwym zmianom, podczas których wiele oferowanych funkcjonalności znacznie się zmieniało, a nawet znikało na zawsze. Okres ten zakończył się 15 maja 2015 roku, kiedy ujrzała światło dzienne jego wersja 1.0. W tym artykule podjęto próbę użycia tego języka przez kompletnego nowicjusza, którego jedynym – w tym momencie – atutem jest całkiem niezła znajomość języka C++.

ZAŁOŻENIA WSTĘPNE

Aby poznać nowy język lub framework poprzez utworzenie w nim projektu, należy ten projekt dobrać tak, aby:

- był prosty, nie będąc jednocześnie zbyt prostym (np. gra „zgadnij liczbę" jest zbyt prosta),
- ukazywał sensowne i nietrywialne zastosowanie języka/frameworka, najlepiej kilku różnych jego elementów,
- nie był on nadmiernie skomplikowany lub rozbudowany (np. serwer HTTP 2).

By spełnić te warunki, autor zdecydował się utworzyć bibliotekę do raportowania błędów w oprogramowaniu do serwisu Backtrace I/O[1]. Dzięki temu sprawdzone w użyciu zostaną co najmniej:

- klient HTTP, wraz z obsługą HTTPS,
- obsługa formatu JSON,
- dołączanie zewnętrznych bibliotek do programów,
- elastyczność wbudowanych typów danych.

Za system testowy wybrany został Arch Linux na architekturze x86_64, czyli 64-bitowej. Instalacja [0] przebiegła bezproblemowo i była kwestią zainstalowania menadżera `rustup`, a następnie użycia go do równie bezproblemowej instalacji kompilatora i bibliotek języka (Listing 0).

Listing 0. Instalacja Rust na Arch Linuksie

```
> sudo pacman -S rustup
resolving dependencies...
looking for conflicting packages...

Packages (1) rustup-1.13.0-3

Total Download Size:   1.72 MiB
Total Installed Size:  7.02 MiB

:: Proceed with installation? [Y/n]
:: Retrieving packages...
 rustup-1.13.0-3-x86_64
(1/1) checking keys in keyring
(1/1) checking package integrity
(1/1) loading package files
(1/1) checking for file conflicts
(1/1) checking available disk space
:: Processing package changes...
(1/1) installing rustup
You may need to run rustup update stable
and possibly also rustup self upgrade-data
Optional dependencies for rustup
    lldb: rust-lldb script
    gdb: rust-gdb script [installed]
:: Running post-transaction hooks...
(1/1) Arming ConditionNeedsUpdate...

> rustup install stable
info: syncing channel updates for 'stable-x86_64-unknown-linux-gnu'
info: latest update on 2018-08-02, rust version 1.28.0 (9634041f0
2018-07-30)
info: downloading component 'rustc'
 68.6 MiB /  68.6 MiB (100 %)   6.4 MiB/s ETA:   0 s
info: downloading component 'rust-std'
 51.3 MiB /  51.3 MiB (100 %)   7.0 MiB/s ETA:   0 s
info: downloading component 'cargo'
info: downloading component 'rust-docs'
  9.4 MiB /   9.4 MiB (100 %)   7.8 MiB/s ETA:   0 s
info: installing component 'rustc'
info: installing component 'rust-std'
info: installing component 'cargo'
info: installing component 'rust-docs'

  stable-x86_64-unknown-linux-gnu installed - rustc 1.28.0 (9634041f0
2018-07-30)

> rustup default stable
info: using existing install for 'stable-x86_64-unknown-linux-gnu'
info: default toolchain set to 'stable-x86_64-unknown-linux-gnu'

  stable-x86_64-unknown-linux-gnu unchanged - rustc 1.28.0 (9634041f0
2018-07-30)

> rustup component add rustfmt-preview
info: downloading component 'rustfmt-preview'
info: installing component 'rustfmt-preview'
> rustc -V
rustc 1.28.0 (9634041f0 2018-07-30)
```

Do zapoznania się z samym językiem na początku w pełni wystarcza darmowa książka *The Rust Programming Language*, dostępna na stronie internetowej [1]. Można ją uznać za kompletny tutorial podstaw języka, z objaśnieniami dostatecznymi nawet dla osób, które wcześniej nie zajmowały się programowaniem.

1. Ku przejrzystości intencji – autor jest pracownikiem wymienionej firmy, jednak w jego opinii artykuł ten nie nosi znamion artykułu sponsorowanego. Utworzony kod jest otwartoźródłowy, usługa udostępnia darmowy plan i jest wartościowa dla twórców aplikacji.

PIERWSZE WRAŻENIA

Jedną z pierwszych rzeczy, które można zauważyć, czytając o języku Rust i pisząc w nim proste programy, jest jego kompletność oraz brak bałaganu spowodowanego utrzymywaniem kompatybilności wstecznej z decyzjami podjętymi niekiedy dekady temu – jak to ma miejsce np. w C++, D, PHP, Pythonie czy Rubym. Z jednej strony nie powinno to być zaskakujące – wersja stabilna języka ma zaledwie trzy lata – ale z drugiej autor nie napotkał w swojej krótkiej eksploracji decyzji, która sprawiałaby wrażenie nieprzemyślanej.

Typy wbudowane

Jedną z najbardziej rzucających się w oczy decyzji projektowych jest nazewnictwo typów liczbowych. Przyjęto bardzo lubianą przez autora konwencję krótkich, ale jednoznacznych nazw (Tabela 0). Warto zauważyć, że typ `char` jest typem 32-bitowym reprezentującym „wartość skalarną Unicode" [2], niemającym bezpośredniego przełożenia na C/C++. Przedstawione to zostało w Listingu 1, którego kod jest w pełni poprawny, a jego działanie zdefiniowane.

Rust	C/C++	C/C++ Zależne od architektury/ kompilatora	D
bool	bool	bool	bool
i8	int8_t	char	byte
i16	int16_t	short	short
i32	int32_t	int	int
i64	int64_t	long	long
i128			cent
u8	uint8_t	unsigned char	ubyte
u16	uint16_t	unsigned short	ushort
u32	uint32_t	unsigned int	uint
u64	uint64_t	unsigned long	ulong
u128			ucent
f32		float	float
f64		double	double
char			dchar
usize	size_t	size_t	size_t
isize	ptrdiff_t	ptrdiff_t	ptrdiff_t

Tabela 0. Porównanie typów w językach Rust, C, C++ i D

Listing 1. Przykład użycia wysokich znaków Unicode wraz z typem char. Źródło: [3]

```
fn main() {
    let c = 'z';
    let z = 'ℤ';
    let heart_eyed_cat = '😻';
}
```

Domyślne stałe

Dzięki temu, że twórcy języka nie byli obarczeni ciężarem w postaci starych jego wersji, mogli wprowadzić zasadę, że wszystkie zadeklarowane obiekty są stałe, chyba że są jawnie zadeklarowane jako mutowalne (za pomocą kwalifikatora `mut`).

Listing 2. Domyślna niezmienność obiektów [4]

```
fn main() {
    let answer = 42;
    // answer = 0; // error
    let mut question = "!";
    question = "?"; // ok!
}
```

Moduły i biblioteki

Moduły są jedną z największych bolączek nowoczesnego C++. Rust traktuje je jak pierwszoplanowych obywateli języka. Utworzenie własnej biblioteki eksportującej moduł jest tak proste jak napisanie `cargo new --lib` w konsoli.

Same moduły z powodzeniem przyjmują nazwy z systemu plików: jeśli istnieje plik o nazwie modułu, który importujemy, to jest on rozpoznawany jako ten moduł. W przeciwnym wypadku powinien istnieć katalog o tej nazwie, wewnątrz którego znajduje się główny plik modułu `lib.rs` oraz ewentualne inne moduły – i tak rekursywnie.

Przykład zawierający moduły `mylib`, `mylib::foobinator`, `mylib::foobinator::helper` oraz `mylib::barinator` znajduje się w Listingu 3.

Listing 3. Struktura katalogów modułów Rust

```
mylib
├── barinator.rs
├── foobinator
│   ├── helper.rs
│   └── lib.rs
└── lib.rs
```

Borrow checker

Autor nie zwrócił na niego specjalnej uwagi, gdyż nie miał z nim na początku swojej przygody problemów. Niewątpliwie jest to jednak jedna z funkcjonalności języka, która okryta jest swego rodzaju złą sławą.

Borrow checker sprawdza statycznie (w czasie kompilacji), czy wszystkie zmienne, wskaźniki i referencje dostępne w kodzie są poprawne i bezpieczne w użyciu. Na przykład czy nie prowadzą do zmiennych, które zostały już zwolnione, czas ich życia się skończył lub nie prowadzą do nich jednocześnie mutowalne i niemutowalne referencje.

Ten ostatni przypadek jest przedstawiony w Listingu 4. Próba wypisania `a`, gdy `b` ma pożyczoną referencję do `a`, powoduje błąd kompilacji.

Listing 4. Borrow checker w akcji [5]

```
fn main() {
    let mut a = 42;

    {
        let b = &mut a;
        //println!("{}", a); // error!
    }

    println!("{}", a);
}
```

Brak klas i klasycznego dziedziczenia

Język Rust nie posiada typów, które można określić mianem klas w rozumieniu podejścia obiektowego. Jego jedyne złożone typy danych to *tuple*, struktura i tablica (Listing 5).

Listing 5. Typy złożone w języku Rust [6]

```
struct foo {
    key: String,
    value: i32,
}

fn main() {
    let tuple = ("answer", 42);
    let arr = [1, 2, 3];
    let struct_ = foo {
        key: String::from("answer"),
        value: 42,
    };
}
```

Rust nie oferuje konceptu takiego jak dziedziczenie, ale jest on z powodzeniem zastępowany przez *traity*, które można rozumieć jak interfejsy pozwalające na domyślne implementacje:

Listing 6. Traity i domyślne implementacje funkcji [7]

```
trait HasArea<T> {
    fn area(&self) -> T;
}

trait HasName {
    fn name(&self) -> String {
        String::from("Unknown")
    }
}

struct Circle {
    radius: f64,
}

impl HasArea<f64> for Circle {
    fn area(&self) -> f64 {
        self.radius * self.radius * 3.1415926535
    }
}

impl HasName for Circle {}

fn main() {
    let c = Circle { radius: 10.0 };
    println!("{}", c.area());
    println!("{}", c.name());
}
```

Generyki i inferencja typów

Generyki w języku Rust mają bardzo zbliżoną syntaktykę do C++: po nazwie np. struktury lub funkcji parametry znajdują się w nawiasach ostrych: na przykład `fn foo<T>(x : &T) -> i32`. Bardzo ciekawe jest jednak odłożenie w czasie samej dedukcji typu generycznego (implementacja algorytmu Hildney-Milner [8]). W C++, D czy C# musi być on znany w momencie utworzenia obiektu lub wywołania funkcji. Rust pozwala na dokonanie tego dopiero w momencie, gdy zostanie on faktycznie użyty.

W Listingu 7 przedstawiono kod w języku Rust wykorzystujący tę własność.

W Listingu 8 zaprezentowano hipotetyczny analogiczny kod w C++. Jest on jednak niepoprawny, ponieważ typ zmiennej `vec` musi być znany w momencie jej deklaracji.

Listing 7. Opóźniona inferencja typów w języku Rust [9]

```
fn main() {
    let mut vec = Vec::new();
    vec.push(42); // dopiero tutaj odbywa się
                  // dedukcja typu
}
```

Listing 8. Hipotetyczny kod w C++

```
#include <vector>

int main()
{
    std::vector vec;
    vec.push_back(42); // dopiero tutaj byłby dedukowany
                       // std::vector<int> vec
}
```

Formatowanie stringów

Rust oferuje wygodne i bezpieczne formatowanie stringów za pomocą makra `format!` (lub wypisywanie bezpośrednie za pomocą `println!`). *Z perspektywy programisty C++* jest to nieocenione ułatwienie. Przykład użycia i rozszerzenia za pomocą *traitu* `std::fmt::Display` znajduje się w Listingu 9.

Listing 9. Przykład formatowania stringów w języku Rust [A]

```
use std::fmt;

struct KeyValue<T>
where
    T: fmt::Display,
{
    key: String,
    value: T,
}

impl<T> fmt::Display for KeyValue<T>
where
    T: fmt::Display,
{
    fn fmt(&self, f: &mut fmt::Formatter) -> fmt::Result {
        write!(f, "({}, {})", self.key, self.value)
    }
}

fn main() {
    let kv = KeyValue {
        key: String::from("answer"),
        value: 42,
    };
    println!("{}", kv);
}
```

Koncepty

Nie sposób nie zauważyć, że `where` w Listingu 9 pozwala na sprawdzenie, czy typ generyczny spełnia wymagania zadeklarowanego *traita* (który tutaj można traktować podobnie do deklaracji interfejsu). W C++ na coś podobnego powinny pozwalać koncepty, ale jak ich nie było, tak nie ma…

Nadpisywanie stałych

Tytuł sekcji jest trochę na wyrost, ponieważ Rust pozwala tylko na przesłanianie nazw obiektów obiektami o tej samej nazwie, ale o innej wartości. Jest to świetne rozwiązanie, gdy zachodzi potrzeba kilkustopniowej obsługi jakiejś wartości. Zamiast kilku stałych o nazwach `value_str`, `value_int` itd., używana jest tylko jedna nazwa i tylko tak długo, jak jest potrzebna.

Listing A. Przesłanianie stałych [B]

```
fn get_data() -> String {
    String::from("42")
}

fn main() {
    let value = get_data();
    let value = value.parse::<i32>().unwrap();
    // value = 42; // error!
    println!("{:04}", value);
}
```

Bez returna

Jeśli ostatnie wyrażenie w funkcji nie kończy się średnikiem (czyli jest to *expression*, a nie *statement* – choć oba po polsku tłumaczy się tak samo), to jest ono traktowane tak, jakby zostało użyte słowo `return`.

Listing B. Zwracanie wartości z funkcji bez instrukcji return [C]

```
use std::ops::Mul;

fn square<T>(v: T) -> T::Output
where
    T: Mul + Copy,
{
    v * v
}

fn main() {
    println!("{:.2}", square(6.480745));
}
```

IMPLEMENTACJA BIBLIOTEKI

Po opanowaniu podstaw można przejść do utworzenia biblioteki. Pierwszym krokiem jest utworzenie projektu:

Listing C. Utworzenie projektu

```
> cargo new --lib backtrace-rust
     Created library `backtrace-rust` project
> tree backtrace-rust
backtrace-rust
├── Cargo.toml
└── src
    └── lib.rs

1 directory, 2 files
```

Następnie można uzupełnić plik *Cargo.toml* o podstawowe dane (Listing D). Warto zwrócić uwagę na to, że eksportowany moduł (w kategorii `lib`) używa podkreślnika zamiast myślnika.

Listing D. Początkowa zawartość pliku Cargo.toml

```
[package]
name = "backtrace-rust"
version = "0.1.0"
authors = ["KrzaQ <krzaq@krzaq.cc>"]

[lib]
name = "backtrace_rust"

[dependencies]
```

Standardowymi typami zwracanym w języku Rust są `Result<T,E>` i `Option<T>`, ale z perspektywy programisty ich obsługa jest normą.

Twórca aplikacji powinien jednak być zainteresowany, kiedy wywołane zostanie `panic` – błąd, który oznacza, że program zrobił coś niepożądanego i zaraz zakończy pracę. Można podpiąć swój *handler*, czyli procedurę do obsługi takiego błędu, za pomocą funkcji `std::panic::set_hook` [D]. Nie można jednak za jej pomocą obsłużyć błędu i wrócić do wykonania programu – służy ona jedynie do przechwycenia informacji o błędzie i odpowiedniego ich zapisania. Jej użycie przedstawiono w Listingu E.

Listing E. Użycie std::panic::set_hook [E]

```
fn main() {
    std::panic::set_hook(Box::new(|panic_info|{
        println!("My panic: {:?}", panic_info);
    }));

    panic!("Answer was not {}", 42);
}
```

Listing F. Wynik działania programu z Listingu E [formatowanie: red]

```
My panic: PanicInfo {
    payload: Any,
    message: Some(Answer was not 42),
    location: Location {
        file: "src/main.rs",
        line: 6,
        col: 5
    }
}
```

Aby jednak funkcja ta była przydatna do raportowania błędów do serwisu oferowanego przez Backtrace I/O, powinna ona spełniać założenia przez dokumentację API [F]. Według niej dane raportowanego błędu powinny trafić za pomocą protokołu HTTP(S) pod unikalny dla użytkownika adres, a parametrem zapytania powinien być token służący do wysyłania błędów.

Adres do wysyłki jest taki sam jak konta użytkownika, z portem o numerze 6098. Adres autora to https://krzaq.sp.backtrace.io:6098. O tym, jak utworzyć swój token, można przeczytać w [10].

W celu zapamiętania i przekazania tych danych do *handlera* utworzona zostanie osobna funkcja `backtrace_rust::register_error_handler`. Trochę wyprzedzając implementację, funkcja ta przyjmie jeszcze jeden argument – domknięcie wywoływane w momencie obsługi błędu. Dzięki temu użytkownik może samodzielnie zadecydować o przekazaniu dodatkowych danych umożliwiających kategoryzację błędu. W Listingu 10 przedstawiono implementację tej funkcji.

Listing 10. Implementacja backtrace_rust::register_error_handler

```
pub fn register_error_handler<T>(
    url: &str, token: &str, user_handler: T)
where
    T: 'static + Send + Sync +
        Fn(&mut Report, &PanicInfo) -> (),
{
    let submission_target = SubmissionTarget {
        token: String::from(token),
        url: String::from(url),
    };

    std::panic::set_hook(Box::new(move |panic_info| {
        sender::submit(
            &submission_target,
            panic_info,
            &user_handler
        );
    }));
}
```

Warto zwrócić uwagę na zapis dyrektywy `where`, łączącej za pomocą operatora plus cztery *traity*, które muszą zostać spełnione przez typ `T`.

Typ `PanicInfo` to widziany wyżej `std::panic::PanicInfo`, a deklaracje `SubmissionTarget` i `Report` przedstawione są w listingu poniżej.

Listing 11. Implementacja SubmissionTarget i Report

```
use std::collections::HashMap;
use std::panic::PanicInfo;

#[derive(Debug, Clone)]
pub struct SubmissionTarget {
    token: String,
    url: String,
}

#[derive(Debug, Clone, Default)]
pub struct Report {
    pub annotations: HashMap<String, String>,
    pub attributes: HashMap<String, String>,
}
```

Można zauważyć, że funkcja `register_error_handler` jedynie przekazuje informacje dalej, do funkcji `sender::submit()`. Jej implementacja jest omówiona poniżej.

W tym momencie należy ponownie przyjrzeć się danym oczekiwanym przez API Backtrace I/O i pozyskać je.

Backtrace – czyli ścieżka wywołań i metadane tych wywołań pobierane są za pomocą `error_chain::Backtrace` z paczki (ang. *crate*) `error_chain`.

Listing 12. Uzyskanie backtrace

```
extern crate error_chain;

//...

let bt = error_chain::Backtrace::new();
```

Informacja o wersji pobierana jest z `rustc_version_runtime`:

Listing 13. Pobranie wersji języka Rust

```
extern crate rustc_version_runtime;

// ...

let version = rustc_version_runtime::version();
let version = format!(
    "{}.{}",
    version.major,
    version.minor
);
```

Obecny timestamp:

Listing 14. Pobranie obecnej liczby sekund od początku ery Uniksa

```
use std::time;

fn get_timestamp() -> u64 {
    let now = time::SystemTime::now();
    now.duration_since(
        time::UNIX_EPOCH
    ).unwrap().as_secs()
}
```

Przekazanie użytkownikowi (a konkretniej: *handlerowi* przekazanemu przez użytkownika) danych do uzupełnienia:

Listing 15. Wywołanie handlera użytkownika

```
let mut r = Report {
    ..Default::default()
};
user_handler(&mut r, _p);
```

Wygenerowanie UUID:

Listing 16. Generowanie UUID

```
extern crate uuid;

// ...

uuid::Uuid::new_v4().to_string()
```

I finalnie utworzenie obiektu JSON z tych danych. Autor zauważa, że pomimo iż Rust jest językiem nastawionym na niskopoziomowy kod wynikowy, czytelność i wygoda generowania JSON-a jest na równi z językami skryptowymi, takimi jak Ruby czy Python. C++, a nawet D zostają daleko w tyle.

Listing 17. Tworzenie obiektu JSON

```
let payload = json!({
    "uuid": uuid::Uuid::new_v4().to_string(),
    "timestamp": get_timestamp(),
    "lang": "Rust",
    "langVersion": version,
    "agent": "backtrace-rust",
    "agentVersion": "0.0.0",
    "mainThread": "main",
    "annotations": r.annotations,
    "attributes": r.attributes,
    "threads": {
        "main": {
            "name": "main",
            "fault": true,
            "stack": stack
        }
    }
});
```

W tym momencie pozostaje tylko wysłanie zapytania do serwera Backtrace I/O:

Listing 18. Wysłanie zapytania HTTP

```
let url = format!(
    "{}/api/post?format=json&token={}",
    st.url,
    st.token
);

let client = reqwest::Client::new();

let resp = client
    .post(&url)
    .json(&payload)
    .send();

match resp {
    Ok(x) => println!("{:?}", x),
    Err(error) => println!("{:?}", error),
}
```

Cały moduł `sender` przedstawiony jest w Listingu 19:

Listing 19. Cały moduł sender.rs

```
extern crate error_chain;
extern crate reqwest;
extern crate rustc_version_runtime;
extern crate serde_json;
extern crate uuid;

use std::collections::HashMap;
use std::panic::PanicInfo;
use std::time;

use Report;
use SubmissionTarget;

pub fn submit<T>(
    st: &SubmissionTarget,
```

```
    _p: &PanicInfo,
    user_handler: T)
where
    T: 'static + Send + Sync +
        Fn(&mut Report, &PanicInfo) -> (),
{
    let bt = error_chain::Backtrace::new();

    let version = rustc_version_runtime::version();
    let version = format!(
        "{}.{}",
        version.major,
        version.minor
    );

    let mut r = Report {
        ..Default::default()
    };
    user_handler(&mut r, _p);

    let mut stack = Vec::new();

    for x in bt.frames() {
        for y in x.symbols() {
            let line = match y.lineno() {
                Some(x) => x.to_string(),
                None => String::new(),
            };

            let filename = match y.filename() {
                Some(x) => String::from(match x.to_str() {
                    Some(w) => w,
                    None => "",
                }),
                None => String::new(),
            };

            let addr = match y.addr() {
                Some(x) => format!("{:p}", x),
                None => String::new(),
            };

            let name = match y.name() {
                Some(x) => x.to_string(),
                None => String::new(),
            };

            let mut elem = HashMap::new();
            elem.insert(String::from("line"), line);
            elem.insert(String::from("library"), filename);
            elem.insert(String::from("address"), addr);
            elem.insert(String::from("funcName"), name);
            stack.push(elem);
        }
    }

    let payload = json!({
        "uuid": uuid::Uuid::new_v4().to_string(),
        "timestamp": get_timestamp(),
        "lang": "Rust",
        "langVersion": version,
        "agent": "backtrace-rust",
        "agentVersion": "0.0.0",
        "mainThread": "main",
        "annotations": r.annotations,
        "attributes": r.attributes,
        "threads": {
            "main": {
                "name": "main",
                "fault": true,
                "stack": stack
            }
        }
    });

    let url = format!(
        "{}/api/post?format=json&token={}",
        st.url,
        st.token
    );

    let client = reqwest::Client::new();

    let resp = client
        .post(&url)
        .json(&payload)
        .send();

    match resp {
        Ok(x) => println!("{:?}", x),
        Err(error) => println!("{:?}", error),
    }
}

fn get_timestamp() -> u64 {
    let now = time::SystemTime::now();
    now.duration_since(
        time::UNIX_EPOCH
    ).unwrap().as_secs()
}
```

Autor zauważa, że w kodzie użytych zostało dużo zewnętrznych bibliotek od osób trzecich. Ich dodanie do projektu było trywialne i sprowadzało się do dopisania ich jako zależności w pliku *Cargo.toml*.

Listing 1A. Cargo.toml po implementacji

```
[package]
name = "backtrace-rust"
version = "0.1.0"
authors = ["KrzaQ <krzaq@krzaq.cc>"]

[lib]
name = "backtrace_rust"

[dependencies]
error-chain = "0.12"
rustc_version_runtime = "0.1.3"
serde_json = "1.0"
uuid = { version = "0.7.0-beta", features = ["v4"] }
reqwest = "0.8.8"
```

Użycie zaimplementowanej biblioteki

Na potrzeby przykładu tworzymy osobny projekt, do którego podłączymy raportowanie błędów do Backtrace I/O.

Listing 1B. Utworzenie projektu

```
> cargo new art
     Created binary (application) `art` project
> tree art
art
├── Cargo.toml
└── src
    └── main.rs

1 directory, 2 files
```

Dodanie zależności `backtrace-rust` na razie ma charakter względnej ścieżki, ale po dodaniu paczki (*crate*) do globalnego repozytorium możliwe by było użycie dokładnie takie jak innych paczek.

Listing 1C. Dodanie zależności od backtrace-rust

```
[package]
name = "art-test"
version = "0.1.0"
authors = ["KrzaQ <krzaq@krzaq.cc>"]

[dependencies]
num_cpus = "1.0"
backtrace-rust = { path = "../backtrace-rust" }
```

Zarejestrowanie handlera w funkcji `main` testowego programu:

Listing 1D. Główny plik programu testującego bibliotekę

```
extern crate backtrace_rust;
extern crate num_cpus;

use backtrace_rust::Report;
use std::panic::PanicInfo;

fn main() {
    backtrace_rust::register_error_handler(
        "https://krzaq.sp.backtrace.io:6098",
        "ad02f3f944c...2c2a17afb1c003",
        |r: &mut Report, _| {

            let cpus = num_cpus::get();
```

```
            let cpus = cpus.to_string();


            r.attributes.insert(
                String::from("cpu.cores"),
                cpus
            );
        },
    );

    println!("Hello, world!");
    panic!("{:?}", 42);
}
```

Wywołanie:

Listing 1E. Uruchomienie programu testowego

```
> cargo run
# 106 linii kompilowania zależności
   Compiling backtrace-rust v0.1.0
       (file:///home/krzaq/code/prog_rust/backtrace-rust)
   Compiling art-test v0.1.0
       (file:///home/krzaq/code/prog_rust/art-test)
   Finished dev [unoptimized + debuginfo] target(s) in 37.60s
     Running `target/debug/art`
Hello, world!
```

Efekt

Po wyświetleniu szczegółów projektu w serwisie Backtrace I/O można zauważyć nowe raporty błędów (Rysunek 0). Widok szczegółowy pozwala dodatkowo na weryfikację konkretnych przypadków, wraz z wglądem w przesłane atrybuty lub stos wywołań w momencie wystąpienia błędu (Rysunek 1).

PODSUMOWANIE

Podczas pisania tego artykułu celem przyświecającym autorowi było zapoznanie się z językiem Rust poprzez wykonanie projektu o niezerowej przydatności i skomplikowaniu. Cel ten został osiągnięty. Projekt dostępny jest na platformie GitHub [11].

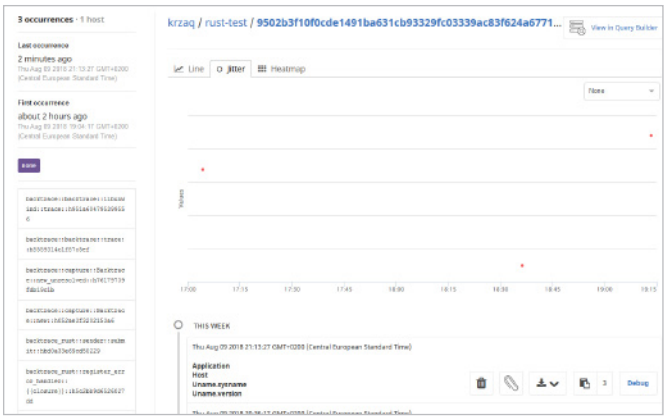
Choć jeden projekt – i to relatywnie niewielki – to zdecydowanie zbyt mało, aby wyrażać się pewnikami, to jednak korzystanie z języka Rust pozostawiło autora z mieszanymi uczuciami.

Z jednej strony dało się zaobserwować przemyślany język, którego części składają się w spójną całość, a którego konwencja wymusza wręcz stały styl programowania. Znacząco ułatwiało to późniejsze czytanie kodu – zarówno swojego, jak i cudzego. Ponadto dołączanie zewnętrznego kodu jest równie trywialne jak w Rubym lub Pythonie.

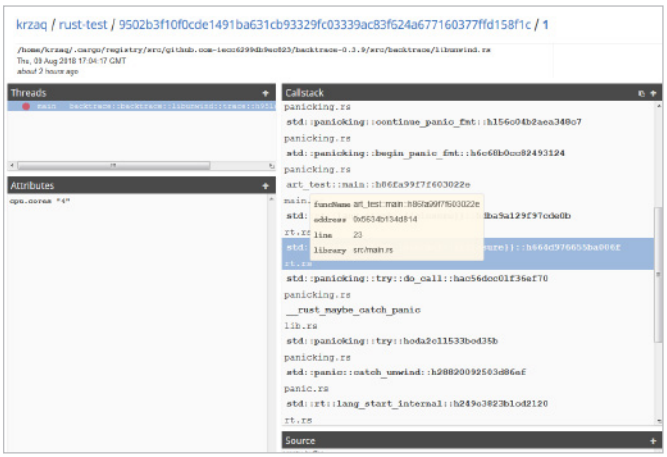
Z drugiej strony w wielu miejscach wymogi języka wydają się nader restrykcyjne. Przykładem może tu być funkcja `square` z Listingu B. Zajmuje ona 8 linii, z czego właściwie wszystkie kompilator powinien móc sobie wydedukować. W języku D analogiczna funkcja zostałaby zapisana jako `auto square = a => a * a`.

Ostatecznie jednak autor przyznaje, że pisanie w języku Rust dało mu znacznie więcej radości niż przysporzyło frustracji, a powody tych irytacji mogą zniknąć wraz z postępującą znajomością języka.

W opinii autora jest to zdecydowanie język warty poświęcenia większej uwagi. Być może następny artykuł o tym języku będzie już na poziomie wyższym niż podstawowy.



Rysunek 0. Widok ogólny projektu



Rysunek 1. Widok szczegółowy błędu

Bibliografia

[0]: https://wiki.archlinux.org/index.php/rust
[1]: https://doc.rust-lang.org/book/2018-edition/index.html
[2]: http://www.unicode.org/glossary/#unicode_scalar_value
[3]: https://doc.rust-lang.org/book/2018-edition/ch03-02-data-types.html
[4]: https://goo.gl/a62nZ1
[5]: https://goo.gl/Qr1fuP
[6]: https://goo.gl/rC96cw
[7]: https://goo.gl/29u8dH
[8]: https://en.wikipedia.org/wiki/Hindley%E2%80%93Milner_type_system
[9]: https://goo.gl/Edqnss
[A]: https://goo.gl/8eUW35
[B]: https://goo.gl/8iHqE8
[C]: https://goo.gl/1sPBQR
[D]: https://doc.rust-lang.org/1.7.0/std/panic/fn.set_handler.html
[E]: https://goo.gl/XnjrNy
[F]: https://api.backtrace.io/
[10]: https://help.backtrace.io/troubleshooting/what-is-a-submission-token
[11]: https://github.com/KrzaQ/backtrace-rust

Paweł "KrzaQ" Zakrzewski
https://dev.krzaq.cc

Absolwent Automatyki i Robotyki na Zachodniopomorskim Uniwersytecie Technologicznym. Pracuje jako Software Engineer w Backtrace I/O. Programowaniem interesuje się od dzieciństwa, jego ostatnie zainteresowania to C++ i metaprogramowanie.