AGH
Akademia Górniczo-Hutnicza im. Stanisława Staszica w Krakowie
Wydział Fizyki i Informatyki Stosowanej

Praca magisterska

Gabriel Górski

kierunek studiów: **informatyka stosowana**

specjalność: **grafika komputerowa i przetwarzanie obrazów**

Rust jako nowoczesny język programowania systemów wbudowanych

Opiekun: **dr inż. Krzysztof Świentek**

Kraków, listopad 2020

Oświadczenie studenta

Uprzedzony(-a) o odpowiedzialności karnej na podstawie art. 115 ust. 1 i 2 ustawy z dnia 4 lutego 1994 r. o prawie autorskim i prawach pokrewnych (t.j. Dz. U. z 2018 r. poz. 1191 z późn. zm.): „Kto przywłaszcza sobie autorstwo albo wprowadza w błąd co do autorstwa całości lub części cudzego utworu albo artystycznego wykonania, podlega grzywnie, karze ograniczenia wolności albo pozbawienia wolności do lat 3. Tej samej karze podlega, kto rozpowszechnia bez podania nazwiska lub pseudonimu twórcy cudzy utwór w wersji oryginalnej albo w postaci opracowania, artystyczne wykonanie albo publicznie zniekształca taki utwór, artystyczne wykonanie, fonogram, wideogram lub nadanie.", a także uprzedzony(-a) o odpowiedzialności dyscyplinarnej na podstawie art. 307 ust. 1 ustawy z dnia 20 lipca 2018 r. Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.) „Student podlega odpowiedzialności dyscyplinarnej za naruszenie przepisów obowiązujących w uczelni oraz za czyn uchybiający godności studenta.", oświadczam, że niniejszą pracę dyplomową wykonałem(-am) osobiście i samodzielnie i nie korzystałem(-am) ze źródeł innych niż wymienione w pracy.

Jednocześnie Uczelnia informuje, że zgodnie z art. 15a ww. ustawy o prawie autorskim i prawach pokrewnych Uczelni przysługuje pierwszeństwo w opublikowaniu pracy dyplomowej studenta. Jeżeli Uczelnia nie opublikowała pracy dyplomowej w terminie 6 miesięcy od dnia jej obrony, autor może ją opublikować, chyba że praca jest częścią utworu zbiorowego. Ponadto Uczelnia jako podmiot, o którym mowa w art. 7 ust. 1 pkt 1 ustawy z dnia 20 lipca 2018 r. — Prawo o szkolnictwie wyższym i nauce (Dz. U. z 2018 r. poz. 1668 z późn. zm.), może korzystać bez wynagrodzenia i bez konieczności uzyskania zgody autora z utworu stworzonego przez studenta w wyniku wykonywania obowiązków związanych z odbywaniem studiów, udostępniać utwór ministrowi właściwemu do spraw szkolnictwa wyższego i nauki oraz korzystać z utworów znajdujących się w prowadzonych przez niego bazach danych, w celu sprawdzania z wykorzystaniem systemu antyplagiatowego. Minister właściwy do spraw szkolnictwa wyższego i nauki może korzystać z prac dyplomowych znajdujących się w prowadzonych przez niego bazach danych w zakresie niezbędnym do zapewnienia prawidłowego utrzymania i rozwoju tych baz oraz współpracujących z nimi systemów informatycznych.

.....
(czytelny podpis)

Kraków, 30. października 2020

Tematyka pracy magisterskiej i praktyki dyplomowej Gabriela Górskiego, studenta drugiego roku studiów drugiego stopnia na kierunku informatyka stosowana, specjalności grafika komputerowa i przetwarzanie obrazów

Temat pracy magisterskiej: **Rust jako nowoczesny język programowania systemów wbudowanych**

Opiekun pracy:	dr inż. Krzysztof Świentek
Recenzenci pracy:	dr inż. Jakub Moroń
Miejsce praktyki dyplomowej:	WFiIS AGH, Kraków

Program pracy magisterskiej i praktyki dyplomowej

1. Omówienie realizacji pracy magisterskiej z opiekunem.
2. Zebranie i opracowanie literatury dotyczącej tematu pracy.
3. Praktyka dyplomowa:
   - Nauka samego języka.
   - Eksploracja możliwości Rusta.
   - Stworzenie prototypu projektu wprowadzającego w wykorzystanie praktyczne języka.
4. Przekształcenie prototypu w pełnoprawny projekt, który empirycznie zweryfikuje ekosystem Rusta w kontekście systemów wbudowanych.
5. Porównanie Rusta względem obecnie stosowanych rozwiązań/technologii.
6. Wyklarowanie mocnych i słabych stron Rusta.
7. Analiza zebranych wniosków, ich omówienie i zatwierdzenie przez opiekuna.
8. Opracowanie redakcyjne pracy.

Termin oddania w dziekanacie: 30. października 2020

....

(podpis kierownika katedry) (podpis opiekuna)

Na kolejnych dwóch stronach proszę dołączyć kolejno recenzje pracy popełnione przez Opiekuna oraz Recenzenta (wydrukowane z systemu MISIO i podpisane przez odpowiednio Opiekuna i Recenzenta pracy). Papierową wersję pracy (zawierającą podpisane recenzje) proszę złożyć w dziekanacie celem rejestracji co najmniej na tydzień przed planowaną obroną.

Spis treści

Abstract

Embedded systems are ubiquitous in our lives even though we tend not to notice them. They can be found in most of the modern appliances including washing machines or fridges, but they are also used in very critical and demanding environments like aircraft or spacecraft equipment. Currently, the most dominant programming languages in that field are C and C++. Their enormous flexibility and close-to-hardware characteristics give a programmer a lot of power which, very often, gets recklessly overused and therefore leads to dangerous faults in software. According to Microsoft Security Response Center, over 70% of reported CVEs are memory safety issues [1][2]. It is worth noting that in case of embedded systems, especially those responsible for driving critical pieces of hardware, potential consequences of memory safety violations can be very severe, even deadly. Rust, the programming language, aims to address these concerns by providing means of compile-time correctness verification disallowing memory safety violations and data races by design. Next to this undoubtedly most distinctive language feature, Rust also provides whole suite of valuable constructs and abstractions, such as algebraic data types, iterators and macros, useful in assembling safe and correct solutions in environments with limited resources. Apart from the language itself, Rust's value also lies in its ecosystem which among the others includes standardized means of project and dependency management; thus, prioritizing coherent experience of developers which stands in opposition to distributed, UNIX-like nature of C/C++ ecosystems. As for the embedded systems specific features, Rust introduces hardware-as-a-state-machine paradigm which allows driver implementers to make peripherals misusage unrepresentable. In order to evaluate Rust's claims and promises, author decided to create a project, weather station, that makes use of different external peripherals relying on variety of different interfaces with RTOS as its backbone. As a result, Rust proves to be a sound contestant against C/C++ language family that is worth considering, especially in embedded system applications; both in casual and enterprise projects.

Wstęp

Systemy wbudowane (ang. *embedded systems*) stanowią integralną część niezliczonej liczby sprzętów. Są wykorzystywane w wielu urządzeniach ułatwiających nam życie codzienne takich jak pralki, lodówki czy samochody, ale również w przemyśle, inżynierii wojskowej, a nawet projektach podboju kosmosu. Nie ma jednej precyzyjnej definicji systemu wbudowanego; mimo to w bardzo ogólny sposób można powiedzieć, że typowo jest to zespół sprzętu komputerowego, oprogramowania i innych części, które jako całość mają pełnić z góry zdefiniowaną funkcję [3, s. 166]. Jednym z elementów odróżniających systemy wbudowane od komputerów osobistych jest właśnie nacisk na silne ograniczenie wielofunkcyjności i brak możliwości przeprogramowywania przez użytkowników końcowych [4, s. 2].

Systemy wbudowane, w tym także ogólnie rozumiane systemy operacyjne, stanowią szczególnie wymagający segment świata IT (ang. *information technology*), ponieważ obok standardowych umiejętności rozwiązywania problemów algorytmicznych i logicznych, programista musi dysponować dostatecznie drobiazgową wiedzą na temat sprzętu. W przeciwnym wypadku bardzo szybko natknie się na problemy wynikające z nieostrożnego zarządzania pamięcią czy niewłaściwej interakcji z peryferiami i innymi elementami; nawet tak prymitywny komponent, jak przycisk może sprawić nie lada kłopotu niedoświadczonemu inżynierowi. Istnieje obecnie wiele środowisk (np. *Keil MDK* [5]; *GNU ARM Embedded toolchain* [6]) pozwalających na budowanie aplikacji na systemy wbudowane, które różnią się pod względem jakości, łatwości i sposobu użycia, poziomu wsparcia dla platform docelowych czy dostarczanych narzędzi. Nie mniej we wszystkich niezmiennie od wielu lat dominującym językiem programowania pozostaje C wspierany okazjonalnie przez C++.

Kluczową cechą zarówno języka C, jak i C++ jest pozostawienie programiście pełnej swobody w kwestii podejmowanych decyzji; także takich, które bez wątpienia zaowocują poważnymi problemami. O ile istnieją sytuacje, w których taki poziom kontroli jest użyteczny, o tyle okazuje się, że bardzo często działa on na szkodę programisty oraz jakości tworzonego programu. Wg raportu *Microsoft Security Response Center*, aż 70% luk bezpieczeństwa naprawianych przez *Microsoft* jest związane z naruszeniami ochrony pamięci [1][2]. Mając na względzie fakt, że w przypadku systemów wbudowanych zazwyczaj nie ma żadnej warstwy pośredniej między tworzoną aplikacją a sprzętem, która by go w jakikolwiek sposób zabezpieczała i często brakuje sprzętowych mechanizmów ochrony pamięci, wady języka C (oraz C++) stają się tym dotkliwsze.

Język programowania Rust stara się wyjść naprzeciw tym potrzebom, balansując między adekwatnym poziomem kontroli koniecznym przy programowaniu systemowym a gwarantowanym bezpieczeństwem. Dokonuje tego poprzez weryfikację poprawności kodu źródłowego w czasie kompilacji pod względem m.in. naruszeń kontroli pamięci czy potencjalnych wyścigów danych, unikając, jeśli to tylko możliwe, niepożądanych kosztów obliczeniowych podczas działania aplikacji. Aby to osiągnąć, Rust wprowadza odpowiednio skonstruowane abstrakcje i składnię, za pomocą których kompilator jest w stanie rzeczoną analizę poprawności wykonać.

Początek historii Rusta datuje się na rok mniej więcej 2006 [7], a wersja 1.0 została wydana zaledwie pięć lat temu, w roku 2015 [8]. W porównaniu do swoich konkurentów sprawia pod tym względem wrażenie bardzo niedojrzałego; język C++ ma bowiem ponad 30 lat [9] lub 20, gdyby traktować jako punkt odniesienia jego pierwszą standaryzację [10], a początki języka C sięgają końca lat 60. [11]. Mimo tego Rust cieszy się rosnącym zainteresowaniem zarówno wśród programistów, jak i biznesu. W ramach corocznej ankiety serwisu *Stack Overflow* Rust święci triumfy jako najbardziej uwielbiany i pożądany język już piąty rok z rzędu [12], a liczba korporacji starających się aplikować Rusta jako pierwszorzędne rozwiązanie w swoich produktach systematycznie wzrasta [13]. Tutaj na szczególną uwagę zasługuje fakt, że Rust uzyskał wstępną aprobatę opiekunów jądra Linux, w tym Linusa Torvaldsa, w kwestii jego rozwijania z wykorzystaniem rzeczonego języka; tj. m.in. tworzenia nowych komponentów i sterowników [14]. Świadczy to o tym, że Rust, mimo swojego młodego wieku, stał się dojrzałym językiem programowania i traktowany jest bardzo poważnie.

Celem tej pracy jest przeanalizowanie języka Rust oraz okalającego go ekosystemu przez pryzmat usprawnień, jakie wprowadza do pracy programisty systemów wbudowanych i sprawdzenie, czy stanowi on sensowną alternatywę do języków C i C++. Pierwszy rozdział manuskryptu skupia się na samym języku i jego mechanizmach; w szczególności na wartościowych elementach syntaktycznych. Następnie charakteryzowany jest ekosystem języka Rust wraz z omówieniem specyfiki jego zastosowania w systemach wbudowanych. Na końcu przedstawiony i omówiony jest projekt, stworzony na potrzeby tej pracy, który ma sprawdzić Rusta „w boju" w zastosowaniu do realizacji kompletnej praktycznej aplikacji i stanowić fundament pod ostateczne wnioski.

Autor pragnie zaznaczyć, że praca skupia się na wykorzystaniu Rusta w systemach wbudowanych korzystających przede wszystkim z mikroprocesorów z rodziny *ARM Cortex M*; z tego powodu w pracy często ma miejsce skrót myślowy gdzie autor przez mikrokontroler rozumie mikrokontroler wyposażony w jeden z omawianych procesorów, o ile nie zostało powiedziane inaczej. Decyzja ta wynika przede wszystkim z faktu, że posiadają one najwyższej jakości wsparcie ekosystemu Rust z tytułu swojej powszechności, choć należy zaznaczyć, że wsparcie dla alternatywnych mikroarchitektur (RISC-V, MIPS, PowerPC) także istnieje [15] i jest systematycznie rozwijane.

1 Rust jako język

Rust czerpie inspirację z wielu języków programowania. Jednym z istotnych źródeł składni są języki C i C++ z których zaczerpnięto m.in. granice wyrażenie blokowego wyznaczanego przez klamry, instrukcje kończące się średnikiem czy symbol ampersandu (&) pozwalający na pobranie referencji. Dużą rolę w kształtowaniu zarówno jego składni, jak i własności odegrały także klasyczne języki funkcyjne: SML, OCaml czy Haskell [16]. Tutaj za przykład mogą posłużyć powszechne dla paradygmatu funkcyjnego algebraiczne typy danych czy dopasowanie do wzorca [17, s. 211]. Szczególnie wartościowe z perspektywy systemów wbudowanych funkcjonalności Rusta zostaną omówione w kolejnych podrozdziałach.

1.1 Semantyka własności

Dla programisty dowolnego języka kod z listingu 1. powinien być relatywnie czytelny. U samej góry rzuca się w oczy definicja funkcji `main`. Wewnątrz jej ciała widzimy definicję zmiennej, skopiowanie wartości jednej zmiennej do drugiej, a na końcu wypisanie wartości tej pierwszej na ekran. Mimo tego, że sam kod źródłowy wydaje się trywialny, skrywa on w sobie kilka wartych uwagi elementów.

```
1 fn main() {
2     let original = 123;
3     let moved_to = original;
4
5     println!("{}", original);
6 }
```

Listing 1: Rust – kopiowanie wartości

Po pierwsze, zmienne są domyślnie niemodyfikowalne. Dla porównania, w językach C i C++ zachowanie domyślne jest odwrotne. Stąd, w Rustcie programista ma do dyspozycji słowo kluczowe `mut`, a w C++ słowo kluczowe `const`. Wartość tego podejścia tkwi w leniwej ludzkiej naturze, w ramach której człowiek zazwyczaj wykonuje tylko te czynności, które są w danym momencie konieczne dla osiągnięcia celu. W przypadku gdy programista potrzebuje modyfikowalnych zmiennych, z tytułu konieczności dołoży słowo kluczowe `mut`, ponieważ kompilator przeszkodzi mu w dalszej pracy. W języku C++ to programista musi zadbać o zachowanie dyscypliny w użyciu słowa kluczowego `const`. Już ten drobny zabieg socjotechniczny czyni Rusta bezpieczniejszym językiem.

Po drugie, warto zaznaczyć, że Rust wspiera dedukcję typów. Oznacza to, że kompilator jest w stanie wydedukować typ definiowanej zmiennej na podstawie tego, co stoi po prawej stronie znaku przypisania i tym samym jawna deklaracja typu jest nadmiarowa. Co ważne, typowanie jest statyczne, co oznacza, że raz nadanego typu nie da się zmienić, nawet jeżeli zmienna jest modyfikowalna jak na listingu 2. Próba kompilacji daje czytelny komunikat o błędzie (rys. 1.).

```rust
fn main() {
    let mut variable = 123;
    variable = "hello world";
}
```

Listing 2: Rust – błąd kompilacji przy zmianie typu

```
error[E0308]: mismatched types
 --> examples/non-compile-change-type.rs:3:16
  |
3 |     variable = "hello world";
  |                ^^^^^^^^^^^^^ expected integer, found `&str`
```

Rysunek 1: Wynik próby skompilowania kodu z listingu 2.

Po trzecie, przykład z listingu 1. pokazuje kopiowanie wartości zmiennej, które całkiem naturalne dla języków takich jak C i C++, nie jest w Rustcie zupełnie oczywiste. Taki sposób przypisywania jednej zmiennej do drugiej nazywany jest tutaj kopiowaniem własności i stanowi element obecnej w Rustcie bardziej ogólnej koncepcji własności i jej przekazywania pomiędzy zmiennymi.

Aby lepiej zrozumieć ten mechanizm, rozpatrzmy dla porównania listing 3. Na pierwszy rzut oka, jest on identyczny z listingiem 1. Nowym elementem jest wykorzystanie instancji struktury w miejscu typu prostego. W rezultacie tej zmiany kod przestaje się kompilować i przy próbie użycia oryginalnej zmiennej zgłaszany jest błąd (rys. 2.); uprzednio kompilator nie protestował przy użyciu elementu kopiowanego.

```rust
struct Struct {
    struct_member: i32,
}

fn main() {
    let original = Struct { struct_member: 123 };
    let moved_to = original;

    println!("{}", original.struct_member); // Błąd kompilacji
}
```

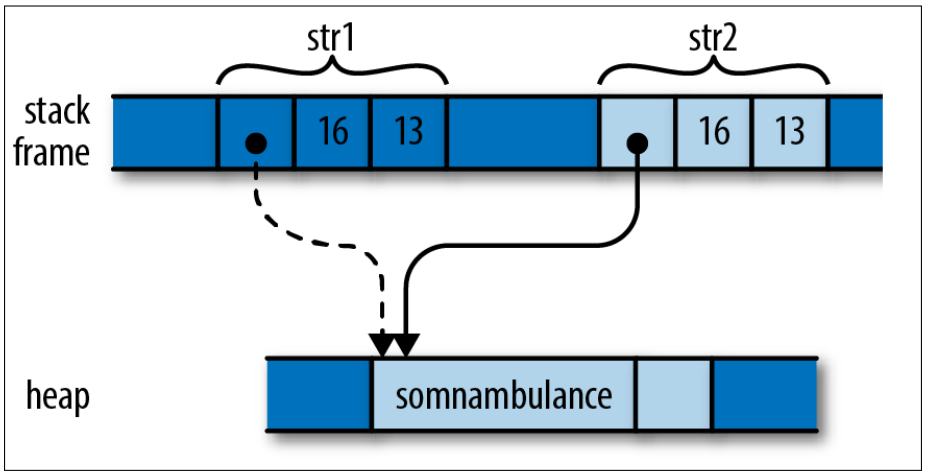
Listing 3: Rust – semantyka przenoszenia

```
error[E0382]: borrow of moved value: `original`
 --> examples/move-semantics-fail.rs:9:20
  |
6 |     let original = Struct { struct_member: 123 };
  |         -------- move occurs because `original` has type `Struct`, which does not implement the `Copy` trait
7 |     let moved_to = original;
  |                    -------- value moved here
8 |
9 |     println!("{}", original.struct_member);
  |                    ^^^^^^^^^^^^^^^^^^^^^^ value borrowed here after move
```

Rysunek 2: Wynik próby skompilowania kodu z listingu 3.

Żeby to wyjaśnić, konieczne jest wprowadzenie koncepcji własności i opisanie jej znaczenia dla języka. Kopiowanie własności, alternatywnie noszące miano semantyki kopiowania (ang. *copy semantics*), jest podejściem domyślnym dla języka C++. W przypadku języka Rust domyślnie ma miejsce przenoszenie własności (semantyka przenoszenia; ang. *move semantics*), co objawia się błędem kompilacji (rys. 2.) na listingu 3. Równocześnie z przepisaniem wartości ze zmiennej `original` do `moved_to`, została również przeniesiona własność nad obiektem. Z tego powodu, późniejsze użycie zmiennej `original` jest operacją niedozwoloną.

Z perspektywy sprzętowej, przeniesienie własności sprowadza się do skopiowania instancji struktury. Brak możliwości użycia oryginału po stworzeniu fizycznej kopii wydaje się nadgorliwością ze strony kompilatora, ale tylko ze względu na prostotę przykładu. W praktyce struktury często składają się z elementów składowych będącymi referencjami do innych obiektów. Na przykład `String` czy `Vec<T>` (wektor) posiadają w swojej definicji referencję do obszaru pamięci zaalokowanego na stercie (rys. 3.). W takiej sytuacji kopia (a właściwie płytka kopia) zawierałaby referencję do tego samego obiektu (np. bufora) co oryginał. Dzielenie stanu między oryginał a kopię jest zdecydowanie zachowaniem niepożądanym i potencjalnie prowadzącym do błędów. To tłumaczy zakaz używania oryginalnego obiektu po przeniesieniu własności.



Rysunek 3: Przykładowa wizualizacja konstrukcji przez przeniesienie własności z `str1` do `str2`
Źródło: [17, s. 87]

Alternatywą dla kopii płytkiej jest stworzenie głębokiej kopii obiektu. Ta jednak jest z natury operacją złożoną obliczeniowo i pamięciowo. Z tego powodu, aby uniknąć niejawności kosztownych operacji, w miejscu potencjalnie niechcianej i łatwo uchodzącej uwadze (zwyczajne przypisanie) operacji kopiowania własności (głęboka kopia), domyślnie ma miejsce przeniesienie własności (płytka kopia z blokadą użycia obiektu kopiowanego) [17]. W przypadku lekkich struktur agregujących typy proste warto rozważyć przełączenie ich na semantykę kopiowania.

Aby zmienić domyślne zachowanie i pozwolić na kopiowanie własności struktury, należy udekorować definicję struktury adnotacją `derive` z parametrami `Copy` i `Clone`. Technicznie rzecz ujmując, żądamy od kompilatora, aby stworzył domyślną implementację cech (ang. *traits*) `Copy` i `Clone`. Cechy jako mechanizm języka zostaną omówione później, natomiast tutaj mogą być one rozumiane jako znacznik modyfikujący zachowanie struktury. Po tej zmianie każda kopia struktury stworzona poprzez przypisywanie do nowej zmiennej jest niezależna i można jej swobodnie używać (listing 4.).

```rust
#[derive(Copy, Clone)]
struct Struct {
    struct_member: i32,
}

fn main() {
    let original = Struct { struct_member: 123 };
    let copied_to = original;

    println!("{}", original.struct_member);
}
```

Listing 4: Listing 3. z wprowadzonym *copy semantics*

Typy proste (ang. *primitive types*) opisujące proste wartości takie jak np. liczby całkowite (`u8`, `u16`, ... `i8`, `i16`, ...), znaki typu unikod (`char`), liczby zmiennoprzecinkowe (`f32`, `f64`) itp. automatycznie implementują wspomniane cechy `Clone` i `Copy`. To tłumaczy, dlaczego listing 1. poprawnie się kompiluje.

Alternatywnie możemy zaimplementować tylko cechę `Clone`, wymuszając jawność kopiowania poprzez konieczność zawołania metody `clone` (listing 5.). Implementacja cechy `Copy` sprawia, że kopia w ramach cechy `Clone` zachodzi niejawnie.

```rust
#[derive(Clone)]
struct Struct {
    struct_member: Vec<i32>,
}

fn main() {
    let original = Struct {
        struct_member: vec![1, 2, 3],
    };
    // `vec!` to makro tworzące nową instancję obiektu
    // `Vec<T>` wypełnioną podanymi wartościami
    let copied_to = original.clone();

    println!("{:?}", original.struct_member);
}
```

Listing 5: Rust – jawne klonowanie struktury

Warto zaznaczyć, że typ może implementować cechę `Copy` tylko i wyłącznie w sytuacji, gdy jego wszystkie elementy składowe również implementują tę cechę. Przykładowo na listingu 6., typ `Vec<i32>` implementuje tylko cechę `Clone`, powodując błąd kompilacji (rys. 4.).

```rust
#[derive(Copy, Clone)] // Błąd kompilacji
struct Struct {
    struct_member: Vec<i32>,
}

fn main() {
    let original = Struct {
        struct_member: vec![1, 2, 3],
    };
    let copied_to = original;

    println!("{:?}", original.struct_member);
}
```

Listing 6: Rust – próba niewłaściwego użycia cechy

```
error[E0204]: the trait `Copy` may not be implemented for this type
 --> examples/copy-semantics-3.rs:1:10
  |
1 | #[derive(Copy, Clone)] // Błąd kompilacji
  |          ^^^^
2 | struct Struct {
3 |     struct_member: Vec<i32>,
  |     ---------------------- this field does not implement `Copy`
  |
```

Rysunek 4: Wynik próby skompilowania kodu z listingu 6.

Dla porównania, na listingu 7. znajduje się, możliwie zbliżony pod względem logicznym do listingu 3., kod źródłowy w języku C++. Standard C++11 wprowadza semantykę przenoszenia z wykorzystaniem tzw. prawostronnych referencji (`T&&`) [18]. Funkcja `std::move` ma za zadanie wykonać rzutowanie dowolnego typu referencji do referencji prawostronnej. Po tej operacji kompilator wybierze w linii 21. zawołanie konstruktora przenoszącego (przyjmującego parametr typu `T&&`), ponieważ ten ma pierwszeństwo dla typu `T&&` przed konstruktorem kopiującym (przyjmującym typ `const T&`). Taka zmiana pozwala programistom na wyrażenie intencji w kwestii oczekiwanego zachowania. Wraz z jej wprowadzeniem rozszerzono odpowiednio typy z biblioteki standardowej C++. Semantyka przenoszenia dla typów takich jak `std::vector<T>` i `std::string` polega na odpowiednim przepisaniu wskaźnika na bufor do konstruowanego obiektu, podczas gdy do tej pory możliwe było jedynie wykonanie pełnej, głębokiej kopii.

```cpp
#include <iostream>

struct Struct {
    Struct(int v) : struct_member(v) {}
    Struct(const Struct& copy_source) :
        struct_member(copy_source.struct_member) {
        std::cout << "Copying" << std::endl;
    }
    Struct(Struct&& move_source) :
        struct_member(move_source.struct_member) {
        move_source.struct_member = 0;
        std::cout << "Moving" << std::endl;
    }

    int struct_member;
};

int main() {
    auto original = Struct { 123 };
    auto copied_to = original;
    auto moved_to = std::move(original);

    std::cout << original.struct_member << std::endl;
}
```

Listing 7: C++ – semantyka przenoszenia

```
Copying
Moving
0
```

Rysunek 5: Wynik działania programu z listingu 7.

Tutaj warto zauważyć kilka różnic między podejściem zaproponowanym w Rustcie, a tym w C++. Po pierwsze, jak już wspomniano w poprzednich akapitach, ze względu na bagaż kompatybilności wstecznej C++ wymusza domyślnie semantykę kopiowania, która może umknąć uwadze programisty i doprowadzić do niepożądanych, nadmiarowych alokacji[1]. Po drugie kompilator nie zaprotestuje przeciwko skompilowaniu kodu z listingu 7. tak jak to ma miejsce w Rustowym odpowiedniku; linia 23. jest poprawna w rozumieniu standardu języka.

Wynik działania programu widoczny na rysunku 5. potwierdza wywołania odpowiednich konstruktorów. Jako że kompilator C++ nie śledzi wykorzystania obiektów w takim stopniu, jak ma to miejsce w przypadku kompilatora Rusta, ten pierwszy nie może zabronić dalszego używania *opróżnionego* obiektu. Z tego powodu programista musi samodzielnie zapewnić w projekcie klasy, że po rzeczonym *opróżnieniu* oryginalna instancja klasy jest bezpieczna w użytkowa-

[1]Opcjonalne słowo kluczowe `explicit` przy deklaracji konstruktora kopiującego pozwala na uniemożliwienie niejawnych kopiowań; tutaj spowodowałoby ono błąd kompilacji w linii 20.

niu. W przypadku analizowanego listingu zostało to zaznaczone poprzez symboliczne wpisanie wartości `0` do pola struktury przenoszonej. W przypadku klasy szablonowej `std::vector<T>` standard gwarantuje, że po wyniesieniu własności obiekt ma być pusty, ale detale implementacyjne są pozostawione implementującemu bibliotekę [19].

1.2 Referencje i pożyczanie

Referencje w Rustcie (`&T`/`&mut T`) są bardzo podobną konstrukcją językową do surowych wskaźników znanych z języka C i C++ (`const T*`/`T*`). Na listingu 8. widzimy intuicyjny schemat działania: pobranie adresu i dereferencja w miejscu użycia.

```
fn main() {
    let original = 123;
    let reference = &original;

    println!("{}", *reference);
}
```

Listing 8: Rust – referencje

Referencje w języku Rust łączą cechy wskaźników i referencji z języka C++. Z tymi pierwszymi łączy je konieczność jawnego używania operatora dereferencji (*) dla typów prostych (jak na listingu 8.). Natomiast z tymi drugimi automatyczna dereferencja w przypadku dostępu do elementów składowych struktur (metod i pól) co pokazuje listing 9.

```
struct Struct {
    struct_member: i32,
}

impl Struct {
    fn print_member(&self) {
        println!("{}", self.struct_member);
    }
}

fn main() {
    let obj = Struct { struct_member: 123 };
    let reference = &obj;
    println!("{}", reference.struct_member); // OK: Brak operatora dereferencji
    reference.print_member(); // OK: Brak operatora dereferencji
    (*reference).print_member(); // OK: Jawny operator dereferencji
}
```

Listing 9: Rust – automatyczna dereferencja

Referencje są podstawą mechaniki pożyczania własności (ang. *borrowing*), która wraz z uprzednio opisanym systemem kopiowania i przenoszenia własności tworzy fundament dla pozostałych funkcjonalności języka. Przykładowo, własność nad obiektem można przenieść tylko w sytuacji, gdy referencje do obiektu, z którego własność jest wynoszona przestały istnieć.

Wg tzw. reguł mieszania referencji, na jeden obiekt może jednocześnie wskazywać

- wiele referencji niemodyfikowalnych (`&T`) **lub**;
- jedna referencja modyfikowalna (`&mut T`).

Modyfikowalność lub jej brak odnosi się tutaj do możności modyfikacji obiektu wskazywanego. Sposób działania omawianych reguł ukazany jest kolejno na listingach 10. i 11.

```rust
fn main() {
    let variable = 123;
    let const_ref_1 = &variable;
    let const_ref_2 = &variable;
    let const_ref_3 = &variable;
    borrow_const_refs(const_ref_1, const_ref_2, const_ref_3);
}

fn borrow_const_refs(a: &i32, b: &i32, c: &i32) {}
```

Listing 10: Rust – wiele niemodyfikowalnych referencji

```rust
fn main() {
    let mut variable = 123;
    let const_ref_1 = &variable;
    let const_ref_2 = &variable;
    let const_ref_3 = &variable;
    let ref_1 = &mut variable;

    borrow_const_refs(const_ref_1, const_ref_2, const_ref_3); // Błąd kompilacji
}

fn borrow_const_refs(a: &i32, b: &i32, c: &i32) {}
```

Listing 11: Rust – złamanie reguły mieszania referencji

```
error[E0502]: cannot borrow `variable` as mutable because it is also borrowed as immutable
 --> examples/borrowing-2.rs:6:17
  |
3 |     let const_ref_1 = &variable;
  |                       --------- immutable borrow occurs here
...
6 |     let ref_1 = &mut variable;
  |                 ^^^^^^^^^^^^^ mutable borrow occurs here
7 |
8 |     borrow_const_refs(const_ref_1, const_ref_2, const_ref_3); // Błąd kompilacji
  |                       ----------- immutable borrow later used here
```

Rysunek 6: Wynik próby skompilowania kodu z listingu 11.

Listing 12. wydaje się odstępstwem od tych reguł, ale tylko powierzchownie. Kompilator stara się być możliwe elastyczny. Zauważamy bowiem, że ostatnie użycie niemodyfikowalnych referencji (tj. zawołanie `borrow_const_refs`) następuje przed zdefiniowaniem modyfikowalnej referencji. Kompilator zrównuje ostatnie użycie referencji z końcem jej istnienia, co można logicznie uzasadnić poprzez podzielenie ciała funkcji `main` na dwa bloki jak na listingu 13.

```rust
fn main() {
    let mut variable = 123;
    let const_ref_1 = &variable;
    let const_ref_2 = &variable;
    let const_ref_3 = &variable;
    borrow_const_refs(const_ref_1, const_ref_2, const_ref_3);

    let ref_1 = &mut variable;
    borrow_mut_ref(ref_1);
}

fn borrow_const_refs(a: &i32, b: &i32, c: &i32) {}
fn borrow_mut_ref(a: &mut i32) {}
```

Listing 12: Rust – specyficzny przypadek z pozornym mieszaniem typów referencji

```rust
fn main() {
    let mut variable = 123;
    {
        let const_ref_1 = &variable;
        let const_ref_2 = &variable;
        let const_ref_3 = &variable;
        borrow_const_refs(const_ref_1, const_ref_2, const_ref_3);
    }
    {
        let ref_1 = &mut variable;
        borrow_mut_ref(ref_1);
    }
}

fn borrow_const_refs(a: &i32, b: &i32, c: &i32) {}
fn borrow_mut_ref(a: &mut i32) {}
```

Listing 13: Rust – podzielenie listingu 12. na bloki kodu

Aby rozjaśnić użyteczność tych reguł, posłużmy się praktycznym przykładem z listingu 14. Na początku tworzymy instancję klasy `Vec<i32>` i wypełniamy przykładowymi liczbami. Najpierw definiujemy referencję do pierwszego elementu wektora, po czym dokładamy jeszcze jeden element na jego koniec. Programista C++ mający doświadczenie z takim typem kontenera powinien natychmiast zauważyć potencjalny problem. Wraz z dołożeniem kolejnego elementu do wektora powstaje ryzyko, że bufor został na nowo zaalokowany, zawartość przekopiowana, a referencja `const_ref` zaczyna wskazywać na niepoprawny obszar pamięci. Kompilator nie pozwoli na skompilowanie takiego kodu, ponieważ zostały złamane reguły pożyczania własności. Komunikat o błędzie kompilacji widoczny jest na rysunku 7.

```rust
fn main() {
    let mut vector = vec![1, 2, 3, 4, 5];

    let const_ref = &vector[0];

    vector.push(6);

    // !!! Co się stało z `const_ref` w przypadku realokacji `vector`?
    borrow_const_ref(const_ref); // Błąd kompilacji
}

fn borrow_const_ref(a: &i32) {}
```

Listing 14: Rust – próba użycia zinwalidowanej referencji

```
error[E0502]: cannot borrow `vector` as mutable because it is also borrowed as immutable
 --> examples/borrowing-4.rs:6:5
  |
4 |     let const_ref = &vector[0];
  |                      ------ immutable borrow occurs here
5 |
6 |     vector.push(6);
  |     ^^^^^^^^^^^^^^ mutable borrow occurs here
...
9 |     borrow_const_ref(const_ref); // Błąd kompilacji
  |                      --------- immutable borrow later used here
```

Rysunek 7: Wynik próby skompilowania kodu z listingu 14.

Aby uzasadnić zachowanie kompilatora, prześledźmy jego rozumowanie krok po kroku. Metoda `push` klasy `Vec<i32>` posiada sygnaturę `fn push(&mut self, value: T)`, natomiast operator nawiasów kwadratowych jest lukrem składniowym dla metody `index` o sygnaturze `fn index(&self, index: T) -> &U`. Metody w strukturach są po prostu funkcjami, które jako pierwszy parametr przyjmują obiekt (w tym przypadku jako referencję) `self`. Jako że sięgnięcie do elementu przechowywanego w wektorze polega na pożyczeniu wektora[2] (linia 4.; listing 14.), to próba zawołania metody `push` oznacza operację pożyczenia z modyfikacją (`mut`) i jest złamaniem drugiej reguły pożyczania owocującym błędem kompilacji. Świadczy to o spójności projektu języka i pozwala wykorzystać wprowadzone reguły dotyczące pożyczania.

Alternatywnym przykładem jest błąd początkujących programistów C++, polegający na rozszerzaniu wektora w czasie iteracji po nim, widoczny na listingu 15. Kod ten poprawnie się kompiluje, ale w rozumieniu standardu C++ stanowi zachowanie niezdefiniowane (ang. *undefined behavior*; *UB*).

[2]Przechodniość pożyczania własności wynika z ogólniejszego mechanizmu okresów życia (ang. *lifetime*), który zostanie częściowo omówiony w rozdziale 1.5.

```cpp
#include <vector>
#include <iostream>

int main() {
    std::vector<int> vector = { 1, 2, 3, 4, 5 };

    for(auto& el : vector) {
        vector.push_back(1); // Zachowanie niezdefiniowane!
    }
}
```

Listing 15: C++ – iteracja po rozszerzanym wektorze; zachowanie niezdefiniowane

Dla porównania symetryczny przykład z listingu 16. się nie skompiluje (rys. 8.), ponieważ w ramach Rustowych reguł pożyczania nie da się takiej operacji wyrazić.

```rust
fn main() {
    let mut vector = vec![1, 2, 3, 4, 5];

    for el in &vector {
        vector.push(1); // Błąd kompilacji
    }
}
```

Listing 16: Rust – symetryczny przykład względem listingu 15.

```
error[E0502]: cannot borrow `vector` as mutable because it is also borrowed as immutable
 --> examples/iterating-over-vector.rs:5:9
  |
4 |     for el in &vector {
  |               -------
  |               |
  |               immutable borrow occurs here
  |               immutable borrow later used here
5 |         vector.push(*el); // Błąd kompilacji
  |         ^^^^^^^^^^^^^^^^ mutable borrow occurs here
```

Rysunek 8: Wynik próby skompilowania kodu z listingu 16.

Podsumowując, koncepcja kopiowania, przenoszenia i pożyczania własności pozwala programiście nie tylko na unikanie operacji niebezpiecznych, prowadzących do zachowań niezdefiniowanych, ale stanowi także zestaw narzędzi umożliwiających zabezpieczanie funkcji i klas przed niepoprawnym użyciem.

1.3 Dopasowanie do wzorca i typ wyliczeniowy

Istotną rolę w Rustcie odgrywa zaczerpnięty z języków funkcyjnych mechanizm dopasowania do wzorca (ang. *pattern matching*). W najprostszym przypadku działa on w bardzo podobny sposób do instrukcji `switch case` znanej z języków C i C++, co pokazuje listing 17. Warto zwrócić uwagę na symbol podkreślenia _ w linii 7. oznaczający wszystkie pozostałe wartości co powoduje pełne pokrycie zakresu zmiennej.

```rust
fn main() {
    let variable = 123;
    match variable {
        1 => println!("One"),
        2 => println!("Two"),
        3 => println!("Three"),
        _ => println!("Everything else!"),
    }
}
```

Listing 17: Rust – dopasowanie do wzorca z użyciem literałów

W praktyce możliwości dopasowania do wzorca są znacznie bardziej wyrafinowane. W przypadku literałów programista może dopasować się do kilku wymienionych wartości w jednej gałęzi lub do całego przedziału co zobrazowano na listingu 18.

```rust
fn main() {
    let variable = 123;
    match variable {
        0 => println!("Zero"),
        1 | 2 | 3 => println!("One or Two or Three"),
        4..=100 => println!("Between 4 and 100 inclusive"),
        _ => println!("Everything else!"),
    }
}
```

Listing 18: Rust – dopasowanie do wzorca z użyciem literałów dla przedziałów

Z pomocą operatora `@` możliwe jest uzyskanie dostępu (powiązanie; ang. *binding*) do dopasowanej wartości wewnątrz gałęzi instrukcji `match` (listing 19.).

```rust
fn main() {
    let variable = 123;
    match variable {
        n @ 0 => println!("Zero: {}", n),
        n @ 1 | n @ 2 | n @ 3 => println!("Small value: {}", n),
        n @ 4..=100 => println!("Big value: {}", n),
        n => println!("All others: {}", n),
    }
}
```

Listing 19: Rust – dopasowanie do wzorca z powiązaniem wartości

Co istotne, kompilator wymusza obsłużenie wszystkich możliwych wartości podczas dopasowywania, a w przypadku nachodzących na siebie przedziałów generuje ostrzeżenie. Z tego powodu zakomentowanie jednego z przedziałów (listing 20.) spowoduje błąd kompilacji widoczny na rysunku 9.

```rust
        // n => println!("All others: {}", n),
```

Listing 20: Rust – modyfikacja listingu 19.

```
error[E0004]: non-exhaustive patterns: `std::i32::MIN..=-1i32` and `101i32..=std::i32::MAX` not covered
 --> examples/pattern-matching-4.rs:3:11
  |
3 |     match variable {
  |           ^^^^^^^^ patterns `std::i32::MIN..=-1i32` and `101i32..=std::i32::MAX` not covered
  |
  = help: ensure that all possible cases are being handled, possibly by adding wildcards or more match arms
  = note: the matched value is of type `i32`
```

Rysunek 9: Wynik próby skompilowania listingu 19. z modyfikacją z listingu 20.

Podobnie jak C i C++, Rust dysponuje typem wyliczeniowym (ang. *enumerated type*; *enum*). Listing 21. pokazuje go w najprostszej postaci.

```rust
enum Enum {
    A,
    B,
    C,
    D,
}

fn main() {
    let a = Enum::A;
    let b = Enum::B;
}
```

Listing 21: Rust – typ wyliczeniowy

Mimo że na pierwszy rzut oka wszystko wydaje się oczywiste, podobnie jak w przypadku dopasowania do wzorca, `enum` skrywa w sobie bogactwo możliwości. Formalnie typ wyliczeniowy w Rustcie jest implementacją typu sumy w ramach wspomnianych we wstępie algebraicznych typów danych [17, s. 211], które w językach niefunkcyjnych często określa się mianem dyskryminowanej lub oznaczonej unii. Dla programistów języków funkcyjnych kod z listingu 22. powinien wyglądać znajomo.

```rust
enum MyEnum {
    StringType(String),
    ArrayType([i32; 3]),
    TupleType((i32, f64, [i32; 2])),
    JustAnotherVariant,
}

fn matcher(variant: MyEnum) -> String {
    use MyEnum::*;
    match variant {
        StringType(value) => format!("string!: {}", value),
        ArrayType(value) => format!("array!: {:?}", value),
        TupleType((intValue, doubleValue, arrayValue)) => {
            format!("tuple!: {}, {}, {:?}", intValue, doubleValue, arrayValue)
        }
        JustAnotherVariant => format!("nothing to destructure :("),
    }
    // `format!` to makro tworzące nową instancję
    // obiektu `String` korzystając z formattera.
}

fn main() {
    let someEnumVariant = MyEnum::TupleType((123, 1.4_f64, [1, 2]));
    let matchingResultString = matcher(someEnumVariant);

    println!("Final string after matching: {}", matchingResultString);
}
```

Listing 22: Rust – `enum` jako dyskryminowana unia

Dla porównania listing 23. zawiera taki sam kod napisany w języku F#. Nawet nie rozumiejąc dokładnie obu przykładów, można zauważyć wyraźne podobieństwo składni w obu listingach[3].

[3]Podobieństwo wynika z faktu, że F# i Rust dzielą zbiór języków, z których czerpią inspirację (m.in. OCaml, Haskell, ML). Autor wybrał F#, ponieważ zna go najlepiej ze wszystkich wymienionych w pracy języków funkcyjnych.

```
type MyEnum =
    | StringType of string
    | ArrayType of int []
    | TupleType of int * float * int []
    | JustAnotherVariant

let matcher (variant: MyEnum): string =
    match variant with
    | StringType (value) -> sprintf "string!: %A" value
    | ArrayType (value) -> sprintf "array!: %A" value
    | TupleType (intValue, floatValue, arrayValue) ->
        sprintf "tuple!: %A, %A, %A" intValue floatValue arrayValue
    | JustAnotherVariant -> sprintf "nothing to destructure :("

[<EntryPoint>]
let main argv =
    let someEnumVariant = MyEnum.TupleType(123, 1.4, [| 1; 2; 3 |])
    let matchingResultString = matcher someEnumVariant

    printfn "Final string after matching: %A" matchingResultString
    0
```

Listing 23: F# – analogiczny przykład do listingu 22.

Listing 22. pokazuje, że instrukcja `match` obok literałów typów prostych wspiera także dopasowywanie do typu wyliczeniowego. Tutaj bardzo użyteczna okazuje się technika destrukturyzacji, pozwalająca rozkładać typy złożone na elementy składowe. W linii 13. mamy do czynienia z podwójną destrukturyzacją. Najpierw rozkładany jest wariant typu wyliczeniowego `MyEnum` do wartości typu krotki, a następnie krotka rozkładana jest na pojedyncze zmienne. Pozwala to połączyć proces dopasowywania wariantów z wyciąganiem zagnieżdżonych w nich elementów do użytku w kolejnych gałęziach instrukcji.

Wspomnianej wcześniej destrukturyzacji, obok krotek i enumów, podlegają także struktury, co pokazuje listing 24. Dla wzbogacenia przykładu struktura zawiera w sobie pole typu krotki. Notacja z linii 15–20 przypominająca odwrócone przypisanie struktury pozwala na rozłożenie nie tylko pól struktury na zmienne, ale także zagnieżdżonych struktur i krotek. W istocie jest to skomplikowana deklaracja trzech zmiennych: `integer`, `local_integer` i `local_integer_2`. Za pomocą zarezerwowanych symboli znaku podkreślenia `_` i dwóch kropek `..` można odpowiednio zignorować jedną konkretną wartość lub wszystkie pozostałe. Mechanizm ten jest spójny z opisywanymi już technikami zarządzania własnością. W omawianym listingu własność nad strukturą została wyniesiona wraz z destrukturyzacją i nie jest możliwe jej ponowne użycie.

```rust
struct Struct {
    another_integer: i32,
    integer: i32,
    float: f32,
    tuple_of_tuples: (i32, (i32, i32, i32), f32, i32),
}

fn main() {
    let s = Struct {
        tuple_of_tuples: (1, (2, 3, 4), 5_f32, 6),
        another_integer: 123,
        integer: 123,
        float: 123_f32,
    };
    let Struct {
        integer, // Lukier składniowy dla: `integer: integer`
        another_integer: local_integer,
        tuple_of_tuples: (_, (_, local_integer_2, ..), ..),
        ..
    } = s;

    println!(
        "{}, {}, (_, (_, {}, _), _, _)",
        integer, local_integer, local_integer_2
    );
}
```

Listing 24: Rust – destrukturyzacja struktury

Tak jak wiele innych dostępnych w Rustcie instrukcji sterujących (`if`; `if let`; `for`; `while` itd.), `match` jest w istocie wyrażeniem, które ewaluuje się do konkretnej wartości. A skoro właściwie każda konstrukcja języka jest wyrażeniem dającym wartość, to można zrezygnować ze słowa kluczowego `return`. Takie podejście (ang. *expression-oriented language* [20]), zaczerpnięte z języków funkcyjnych (np. Haskell, Lisp), dostarcza dodatkowych możliwości w odniesieniu do kompozycji kodu i przepływu kontroli programu. Na listingu 22. każda z gałęzi wyrażenia `match` zwraca wartość typu `String`, a ono samo jest ostatnim wyrażeniem w ciele funkcji `matcher`. Kompilator weryfikuje typy wyrażeń zwracanych w każdej z gałęzi, tym samym uniemożliwiając użycie dwóch różnych typów. Przy próbie popełnienia takiego błędu (listing 25.) kompilator generuje komunikat widoczny na rysunku 10.

```rust
    match variant {
        StringType(value) => 123, // Błąd kompilacji
        ArrayType(value) => format!("array!: {:?}", value),
        TupleType((intValue, doubleValue, arrayValue)) => {
            format!("tuple!: {}, {}, {:?}", intValue, doubleValue, arrayValue)
        }
        JustAnotherVariant => format!("nothing to destructure :("),
```

Listing 25: Rust – zmodyfikowany fragment listingu 22.

```
error[E0308]: mismatched types
  --> examples/enum-6.rs:11:30
   |
8  | fn matcher(variant: MyEnum) -> String {
   |                               ------ expected `std::string::String` because of return type
...
11 |         StringType(value) => 123, // Błąd kompilacji
   |                              ^^^
   |                              |
   |                              expected struct `std::string::String`, found integer
   |                              help: try using a conversion method: `123.to_string()`
```

Rysunek 10: Wynik próby skompilowania listingu 22. z modyfikacją z listingu 25.

Opisane w tym rozdziale techniki można łączyć, co pokazuje listing 26. Pierwsza gałąź dopasowania odpowiada tylko takim punktom dwuwymiarowym, których wartości `x` i `y` są w przedziale $[-20, 20]$. Druga gałąź narzuca takie obostrzenie tylko i wyłącznie na wartość `y`, tym samym będąc ogólniejszą od pierwszej. Trzecia akceptuje wszystkie pozostałe wartości dla wariantu typu wyliczeniowego `TwoDim`, a czwarta resztę.

```rust
enum Point {
    TwoDim { x: i32, y: i32 },
    ThreeDim { x: i32, y: i32, z: i32 },
}

fn main() {
    let point = Point::TwoDim { x: 1, y: 2 };
    let s = match point {
        Point::TwoDim {
    //      Co rozłożyć ze struktury
    //      |  Do jakiej lokalnej zmiennej przypisać
    //      |  |          Przedziały dopasowywania
    //      |  |          |
            x: local_x @ -20..=20,
            y: loc_y @ -20..=20
        } => format!("Close 2D! x: {}; y: {}", local_x, loc_y),
        Point::TwoDim {
            x, // Lukier składniowy: `x: x`
            y: y @ -20..=20
        } => format!("Close only 'y' 2D! x: {}; y: {}", x, y),
        Point::TwoDim {
            x, y
        } => format!("Far 2D! x: {}; y: {}", x, y),
        _ => format!("Not implemented"),
    };

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

Listing 26: Rust – połączenie technik dopasowania do wzorca

Dopasowanie następuje w takiej kolejności, w jakiej zdefiniowane są gałęzi. Niepoukładanie gałęzi wyrażenia `match` w kolejności od najbardziej szczególnego do najbardziej ogólnego zaowocuje ostrzeżeniem kompilatora o nieosiągalnym wzorcu. Przykładowo, zamieniając miejscami pierwszą i drugą gałąź (listing 27.), dostaniemy wynik kompilacji jak na rysunku 11.

```
9         Point::TwoDim {
10            x,
11            y: y @ -20..=20
12        } => format!("Close only 'y' 2D! x: {}; y: {}", x, y),
13        Point::TwoDim { // Ostrzeżenie! Nieosiągalna gałąź
14            x: local_x @ -20..=20,
15            y: loc_y @ -20..=20,
16        } => format!("Close 2D! x: {}; y: {}", local_x, loc_y),
```

Listing 27: Rust – odwrócenie kolejności gałęzi na listingu 26.

```
warning: unreachable pattern
  --> examples/pattern-matching-complex-destructuring-with-wrong-branch-order.rs:13:9
   |
13 | /         Point::TwoDim { // Ostrzeżenie! Nieosiągalna gałąź
14 | |             x: local_x @ -20..=20,
15 | |             y: loc_y @ -20..=20,
16 | |         } => format!("Close 2D! x: {}; y: {}", local_x, loc_y),
   | |_________^
   |
   = note: `#[warn(unreachable_patterns)]` on by default
```

Rysunek 11: Ostrzeżenie kompilatora przy złej kolejności gałęzi

C i C++ bardzo blado wypadają w porównaniu z Rustem w tej materii. Podobnie działający typ sumy można osiągnąć poprzez wykorzystanie wzorca dyskryminowanej unii przez strukturę danych `union`. Przykładowa, prymitywna implementacja rzeczonego wzorca w C++ znajduje się na listingu 28. Ma ona kilka poważnych wad.

```cpp
#include <cstdint>
#include <iostream>
#include <string>
#include <tuple>

struct MyEnum {
    enum {
        StringType,
        ArrayType,
        TupleType,
        JustAnotherVariant,
    } type;

    union {
        std::string string_type;
        std::array<int32_t, 3> array_type;
        std::tuple<int32_t, double, std::array<int32_t, 2>> tuple_type;
    };

    ~MyEnum() {
        switch(type) {
            case StringType:
                string_type.~basic_string();
        }
    }
};

int main() {
    MyEnum someEnumVariant {
        .type = MyEnum::TupleType,
        .tuple_type =
        std::make_tuple<int, double, std::array<int, 2>>(123, 1.4, {1, 2}),
    };

    switch(someEnumVariant.type) {
        case MyEnum::TupleType:
            std::cout << std::get<0>(someEnumVariant.tuple_type) << " ";
            std::cout << std::get<1>(someEnumVariant.tuple_type);
            std::cout << std::endl;
            break;
        default:
            break;
    }
}
```

Listing 28: C++ – przykładowa implementacja wzorca dyskryminowanej unii

Po pierwsze, informacja o typie danych przechowywanych wewnątrz struktury `MyEnum` jest elementem osobnym od samych danych. Oznacza to, że należy zachować ostrożność przy tworzeniu i używaniu instancji takiej struktury lub dołożyć dodatkowych starań w jej projekcie by była ona bezpieczniejsza w użytkowaniu.

Po drugie, implementacja takiego wzorca w bezpieczny sposób jest zadaniem trudnym. Co prawda struktura z listingu 28. próbuje naśladować enum z listingu 22., ale ze względu na to, że jednym z dostępnych wariantów jest `std::string` posiadający nietrywialny destruktor [21], kompilator wymaga od nas jego ręcznej implementacji dla całej struktury `MyEnum`.

Biorąc pod uwagę rosnącą złożoność implementacji i możliwość popełnia błędu, w C++ warto rozważyć sięgnięcie po sprawdzone abstrakcje. Jedną z dostępnych opcji jest biblioteka Boost wraz z klasą `boost::variant<T..>`, która implementuje omawiany wzorzec [22]. W bibliotece standardowej C++17 i nowszej jest ona wbudowana jako `std::variant<T..>`. Autor pracy nie ma doświadczenia ze środowiskiem Boost ani z idiomami najnowszych standardów języka C++, jednak bez posiadania takowego można stwierdzić, że bezpieczna abstrakcja w języku C++ nie jest w stanie zastąpić programiście funkcjonalności i gwarancji dostarczanych przez Rusta ze względu na charakterystykę języka, w którym jest ona napisana. Ponadto, minusem może być fakt, że jest to dodatkowa (w przypadku Boosta), zewnętrzna zależność dla projektu, którą zarządzanie może być kłopotliwe.

Podsumowując, w Rustcie dopasowanie do wzorca wraz z towarzyszącą mu destrukturyzacją i potężnym typem wyliczeniowym mają ogromny potencjał, który ogranicza jedynie wyobraźnia programisty. Należy zaznaczyć, że ten rozdział nie wyczerpuje tematu i autor gorąco zachęca do jego dalszej eksploracji [17, s. 221-231].

1.4 Cechy i programowanie generyczne

Cechy pojawiły się po raz pierwszy w rozdziale 1.1[4] pełniąc tam rolę znacznika zmieniającego zachowanie semantyki języka. Ich zastosowanie jest jednak znacznie szersze; pozwalają bowiem na definiowanie wspólnych wzorców zachowań dla różnych typów danych. Elementem charakterystycznym cech, który odróżnia je od znanych z paradygmatu obiektowego interfejsów, jest możliwość implementowania ich dla typów już istniejących. W wypadku konwencjonalnych języków obiektowych, takich jak C++, C# czy Java, operacja dodania do klasy wsparcia dla nowego interfejsu wymagałaby zmiany jej definicji. Listing 29. pokazuje przykład prostej cechy definiowanej słowem kluczowym `trait` i implementującej ją struktury.

[4]Patrz cechy `Copy` i `Clone` na listingach 4, 5 i 6.

```rust
trait SimpleTrait {
    fn do_a_simple_thing(&self);
}

struct Struct {
    integer: i32
}

impl SimpleTrait for Struct {
    fn do_a_simple_thing(&self) {
        println!("Struct.integer is {}", self.integer);
    }
}

impl SimpleTrait for i32 {
    fn do_a_simple_thing(&self) {
        println!("me (i32) is {}", self);
    }
}

fn main() {
    let value = 123;
    let s = Struct { integer: 123 };

    value.do_a_simple_thing();
    s.do_a_simple_thing();
}
```

Listing 29: Rust – przykład cechy i implementacji

Cechy są także kluczową częścią programowania generycznego (ang. *generic programming*), które stanowi bardzo ważną część Rusta, choć jest także dostępne w wielu innych językach programowania. Przykładowo, C++ implementuje ten paradygmat poprzez mechanikę szablonów (ang. *templates*). Pozwala ono na tworzenie ogólniejszych abstrakcji, które są w stanie obsłużyć różne typy danych. Dzięki cechom możliwe jest nakładanie ograniczeń na parametry generyczne; użyteczność tej techniki powinna stać się jaśniejsza w czasie lektury tego podrozdziału. Listing 30. pokazuje przykład generycznej funkcji i struktury w Rustcie; możliwe jest także tworzenie generycznych typów wyliczeniowych. Czytelnicy, którzy mieli już do czynienia z analogicznymi konstrukcjami z innych języków, z łatwością doszukają się podobieństw.

```rust
struct Struct<T> {
    value: T
}

fn generic_function<T>(value: T) {}

fn main() {
    let s1 = Struct { value: 123 };
    let s2 = Struct { value: "To jest tekst" };

    generic_function(s1);
    generic_function(s2);
    generic_function(123);
    generic_function("To jest tekst");
}
```

Listing 30: Rust – przykładowe konstrukcje generyczne

Reguły opisane w poprzednich rozdziałach wciąż obowiązują. Przykładowo, dwukrotne zawołanie funkcji `generic_function` z parametrem `s1` jest zabronione, ze względu na przeniesienie własności w linii 11. Taka próba (listing 31.) spowoduje znany czytelnikowi błąd kompilacji (rysunek 12.).

```rust
    generic_function(s1); // Błąd kompilacji
```

Listing 31: Rust – modyfikacja listingu 30.; dodatkowa linia

```
error[E0382]: use of moved value: `s1`
  --> examples/simple-generics-use-after-move.rs:15:22
   |
8  |     let s1 = Struct { value: 123 };
   |         -- move occurs because `s1` has type `Struct<i32>`, which does not implement the `Copy` trait
...
11 |     generic_function(s1);
   |                      -- value moved here
...
15 |     generic_function(s1);
   |                      ^^ value used here after move
```

Rysunek 12: Wynik próby skompilowania listingu 30. z modyfikacją z listingu 31.

W taki sposób wykorzystane parametry generyczne mają ograniczoną użyteczność. Przykładowo, programista nie jest w stanie zawołać żadnej metody na obiekcie typu T (listing 32.; komunikat o błędzie na rysunku 13.), ponieważ sygnatura funkcji akceptuje dowolny typ; w szczególności taki który wybranych metod czy pól nie posiada. Takie zachowanie kontrastuje ze znanymi z języka C++ szablonami. Te ze względu na swoją charakterystykę przypominają makra, których poprawność jest weryfikowana w miejscu użycia na zasadzie naiwnego rozwijania tekstu.

```rust
fn generic_function<T>(value: T) {
    value.some_method(); // Błąd kompilacji
}
```

Listing 32: Rust – modyfikacja listingu 30.; rozszerzone ciało funkcji

```
error[E0599]: no method named `some_method` found for type parameter `T` in the current scope
 --> examples/simple-generics-not-bounded-method-call.rs:6:11
  |
6 |     value.some_method(); // Błąd kompilacji
  |           ^^^^^^^^^^^ method not found in `T`
```

Rysunek 13: Wynik próby skompilowania listingu 30. z modyfikacją z listingu 32.

Aby interakcja z generycznymi obiektami była w przypadku Rusta możliwa, należy wcześniej rozszerzyć definicję struktury czy funkcji poprzez związanie parametru cechą (ang. *trait bound*), która rzeczone metody deklaruje. Jest to możliwe z pomocą notacji pokazanej na listingu 33. Zaletą takiego podejścia jest możliwość uściślenia kontraktu, jaki typy wejściowe muszą spełniać. Restrykcje stawiane przez funkcje i struktury są widoczne na poziomie ich sygnatur; dzięki temu są bardziej przejrzyste niż w przypadku szablonów w C++.

```rust
struct Struct<T: Copy + Debug> { // Wiązanie może dotyczyć kilku cech naraz
    value: T
}
// lub alternatywnie
struct Struct<T>
where
    T: Copy + Debug
{
    value: T
}
```

Listing 33: Rust – wiązanie parametrów generycznych cechami

Rozważmy listing 34., który wykorzystuję cechę i strukturę z listingu 29. Sygnatura funkcji `generic_function` z linii 23. oznacza, że przyjmuje ona argument dowolnego typu tak długo, jak implementuje on dwie cechy: `Debug` i `SimpleTrait`. Wymagamy implementacji tych cech dla parametru wejściowego, ponieważ chcemy na obiekcie `value` zawołać metodę cechy `SimpleTrait` (linia 24.) oraz wypisać jej wartość na ekran w trybie *Debug* (z użyciem formatera `"{:?}"`; linia 25.). Makro `println`[5] zostanie w czasie kompilacji rozwinięte do zawołań

[5]Makra jako konstrukt językowy powinny być zrozumiałe dla programistów C czy C++; zostaną częściowo omówione w rozdziale 1.6.2.

odpowiednich funkcji związanych z obsługą wejścia i wyjścia; m.in. funkcji `Debug::fmt`. Stąd wynika konieczność związania typu zmiennej `value` także z cechą `Debug`[6]. Jest ona implementowana automatycznie przez typy proste i część typów złożonych z biblioteki standardowej (np. `Vec`).

```rust
trait SimpleTrait {
    fn do_a_simple_thing(&self);
}

struct Struct {
    integer: i32
}

impl SimpleTrait for Struct {
    fn do_a_simple_thing(&self) {
        println!("Struct.integer is {}", self.integer);
    }
}

impl SimpleTrait for i32 {
    fn do_a_simple_thing(&self) {
        println!("me (i32) is {}", self);
    }
}

use core::fmt::Debug; // Import do lokalnej przestrzeni nazw

fn generic_function<T: Debug + SimpleTrait>(value: T) {
    value.do_a_simple_thing(); // Wymagana cecha `SimpleTrait`
    println!("Debugged value: {:?}", value); // Wymagana cecha `Debug`
}

fn main() {
    generic_function(123);
    generic_function(Struct { integer: 123 }); // Błąd kompilacji
    generic_function(321);
}
```

Listing 34: Rust – związanie parameteru generycznego cechą

[6]Obok cechy `Debug`, dostępna jest także cecha `Display`, służąca do formatowania obiektów w trybie zwyczajnym, jednakże różnica semantyczna między oboma tymi cechami, jak i dalsze detale opisujące działanie formaterów pozostawione są czytelnikowi do samodzielnej eksploracji [23, s. 90-92][17, s. 413-424].

```
error[E0277]: `Struct` doesn't implement `Debug`
  --> examples/simple-generics-bounded-method-call.rs:30:22
   |
23 | fn generic_function<T: Debug + SimpleTrait>(value: T) {
   |                        ----- required by this bound in `generic_function`
...
30 |     generic_function(Struct { integer: 123 }); // Błąd kompilacji
   |                      ^^^^^^^^^^^^^^^^^^^^^^ `Struct` cannot be formatted using `{:?}`
   |
   = help: the trait `Debug` is not implemented for `Struct`
   = note: add `#[derive(Debug)]` or manually implement `Debug`
```

Rysunek 14: Wynik próby skompilowania kodu z listingu 34.

Listing 34. nie kompiluje się, ponieważ zawołanie funkcji w linii 30. nie spełnia kontraktu wyspecyfikowanego w jej sygnaturze; typ `Struct` nie implementuje cechy `Debug` (rysunek 14.). Podobnie jak w przypadku cechy `Copy`[7], możemy uzupełnić definicję struktury `Struct` stosownym atrybutem (listing 35), aby zażądać od kompilatora domyślnej implementacji.

```
5  #[derive(Debug)]
6  struct Struct {
7      integer: i32
8  }
```

Listing 35: Rust – korekta listingu 34.

```
me (i32) is 123
Debugged value: 123
Struct.integer is 123
Debugged value: Struct { integer: 123 }
me (i32) is 321
Debugged value: 321
```

Rysunek 15: Wynik działania programu z modyfikacją z listingu 35.

Warto zaznaczyć, że bloki implementacyjne (konstrukcje `impl TTrait for T` oraz `impl T`) też mogą być generyczne (listing 36.). W szczególności blok można zaimplementować tylko dla konkretnych typów (listing 37.). Podobna technika jest dostępna w języku C++ poprzez specjalizację szablonów. Błąd kompilacji z rysunku 16. pokazuje, że metoda `some_method` istnieje tylko dla konkretnego typu `Struct<i32>`. Co intuicyjne, rzeczone typy muszą spełniać podczas specjalizacji wiązanie narzucone przez samą strukturę; w tym przypadku implementować cechę `Display`, którą typ `i32` (listing 37.; linia 7.) implementuje.

[7] Patrz cechy `Copy` i `Clone` na listingach 4, 5 i 6.

```rust
use std::fmt::Display;

//            Definicja parametru generycznego
//            |
struct Struct<T: Display> {
    value: T // <- Użycie parametru generycznego
}

//   Definicja parametru generycznego
//   |                  Użycie parametru generycznego
//   |                  |
impl<T: Display> Struct<T> {
    fn some_method(&self) {
        println!("Struct.value is {}", self.value);
    }
}

fn main() {
    let s1 = Struct { value: 123 };
    let s2 = Struct { value: "Test string" };
    s1.some_method();
    s2.some_method();
}
```

Listing 36: Rust – generyczny blok implementacyjny

```rust
use std::fmt::Display;

struct Struct<T: Display> {
    value: T
}

impl Struct<i32> {
    fn some_method(&self) {
        println!("Struct.value is {}", self.value);
    }
}

fn main() {
    let s1 = Struct { value: 123 };
    let s2 = Struct { value: "Test string" };
    s1.some_method();
    s2.some_method(); // Błąd kompilacji
}
```

Listing 37: Rust – specjalizowany generyczny blok implementacyjny

```
error[E0599]: no method named `some_method` found for struct `Struct<&str>` in the current scope
  --> examples/generic-impl-block-specialised.rs:17:8
   |
3  | struct Struct<T: Display> {
   | ------------------------ method `some_method` not found for this
...
17 |     s2.some_method(); // Błąd kompilacji
   |        ^^^^^^^^^^^ method not found in `Struct<&str>`
```

Rysunek 16: Wynik próby skompilowania kodu z listingu 37.

Przykład z listingu 38. pokazuje kreatywne wykorzystanie generycznych metod wewnątrz generycznych bloków implementacyjnych, pozwalających w tym przypadku na mieszanie typów danych reprezentujących współrzędne punktu. Wynik działania tego programu prezentuje rysunek 17.

```rust
//           Definicja parametru generycznego
//           |
struct Point<T, U> {
    x: T, // --- Użycie parametru generycznego
    y: U, // -^
}

//   Definicja parametru generycznego
//   |           Użycie parametru generycznego
//   |           |
impl<T, U> Point<T, U> {
    fn mixup<V, W>(self, other: Point<V, W>) -> Point<T, W> {
        Point {
            x: self.x,
            y: other.y,
        }
    }
}

fn main() {
    let p1 = Point { x: 5, y: 10.4 };
    let p2 = Point { x: "Hello", y: 'c' };
    let p3 = p1.mixup(p2);

    println!("p3.x = {}, p3.y = {}", p3.x, p3.y);
}
```

Listing 38: Rust – generyczna metoda w generycznym bloku implementacyjnym
Źródło: [23, s. 180]

```
p3.x = 5, p3.y = c
```

Rysunek 17: Wynik działania programu z listingu 38.

Jednymi z kluczowych struktur danych korzystających z poznanych do tej pory technik są generyczne typy wyliczeniowe `Result` i `Option`. Pozwalają one programiście na wyrażenie scenariusza, w którym wartości może nie być. Są zdefiniowane w sposób widoczny na listingu 39.

```rust
enum Result<T, E> {
    Ok(T),
    Err(E)
}

enum Option<T> {
    Some(T),
    None
}
```

Listing 39: Rust – typy `Result` i `Option`

Zaletą wynikającą z ich stosowania jest wysoka ekspresyjność. Przykładowo, sygnatura metody w Javie zwracająca obiekt typu złożonego jest zwodnicza, ponieważ wartość zwracana może być równa `null`; bez wniknięcia w implementację nie można tego ocenić. W Rustcie programista może jasno wyrazić niepożądane scenariusze poprzez system typów. Jeśli funkcja zwracająca typ `T` może zawieść i zaowocować błędem (np. błąd połączenia do bazy danych[8]), można przyjąć jako typ zwracany `Result<T, E>`, gdzie `E` to wybrany typ obiektu opisujący błąd. Jeśli funkcja z definicji może, ale nie musi zwrócić konkretnej wartości (np. brak rekordu w bazie danych), warto zastosować typ zwracany `Option<T>`. Trzymając się przykładu z zapytaniem do bazy danych warto rozważyć typ zwracany funkcji jako kompozycji powyższych typów; np. `Result<Option<T>, E>`.

Programiści języka C++ zaznajomieni z metaprogramowaniem i szablonami, doskonale znają zalety płynące ze stosowania technik programowania uogólnionego. Możliwość tworzenia zgrabnych abstrakcji wielokrotnego użytku jest nieoceniona. Ich koszt leży w rozmiarze pliku wykonywalnego i czasie kompilacji; nie wpływają jednak na szybkość działania programów. Usprawnienia wprowadzone przez Rusta względem języka C++ pozwalają na tworzenie jeszcze bezpieczniejszych interfejsów programistycznych dzięki m.in. zaprzęgnięciu mechanizmu cech do ograniczania generycznych parametrów wejściowych funkcji i struktur.

1.5 Okresy życia

W rozdziale 1.2 omówiono, wysokopoziomowo, referencje i semantykę pożyczania własności, aby nakreślić czytelnikowi sens istnienia i użyteczność wprowadzanych przez Rusta reguł. U podstaw rzeczonej semantyki leżą tytułowe okresy życia (ang. *lifetimes*). Zaznajomienie się z nimi jest bardzo pomocne w zrozumieniu dalszych, często mniej intuicyjnych konsekwencji ich istnienia; na przykład przechodniości pożyczonych własności (listing 14.; s. 14), które w skrajnych

[8]Do obsługi błędów krytycznych służy mechanizm panikowania (makro `panic!`), które w zależności od konfiguracji może przerwać działanie wątku, programu lub, co częste w przypadku środowiska wbudowanego, zatrzymanie go w pętli nieskończonej. Więcej informacji czytelnik może znaleźć w źródłach [17, s. 145-147].

przypadkach mogą generować długie, nieoczywiste łańcuchy zależności [24, s. 11]. Oprócz tego wiele komunikatów kompilatora tłumaczy błędne użycie referencji odwołując się do właśnie tego konstruktu. Początkującemu programiście Rusta ten element przysparza zazwyczaj sporo kłopotów i frustracji.

Każdy obiekt w Rustcie posiada okres życia. Konstrukt ten jako koncepcja jest obecny w informatyce od bardzo dawna; przykładowo zmienna zdefiniowana w ramce stosu funkcji kończy swoje życie wraz z momentem, gdy rzeczona funkcja zwraca. Rust traktuje okresy życia dosłownie i wykorzystuje je do weryfikacji poprawności kodu źródłowego za pomocą mechanizmu kontroli pożyczeń własności (ang. *borrow checking*). Dzięki niemu kompilator jest w stanie sprawdzić, czy referencja w momencie dereferencji wskazuje na obiekt, który wciąż istnieje; komentarz na listingu 40. wyjaśnia sposób rozumowania kompilatora w takim przypadku. Rzeczony mechanizm pozwala między innymi na wprowadzenie w życie obostrzeń omówionych w rozdziałach 1.1 i 1.2.

```
fn main() {
    let x;
    {
        let y = 1; // --------------------------------*
        //                                            |
        //                                            *-- Czas życia `y`
        //                                            |
        x = &y; // -- Czas życia `x`:                 |
        //             żyje co najwyżej tyle co `y`   |
        //             |               |              |
        //             |               V              |
    } //          (koniec) ---<--- więc ---<---- (koniec)

    println!("{}", *x); // Błąd kompilacji; `x` nie żyje wystarczająco długo
}
```

Listing 40: Rust – próba użycia nieprawidłowej referencji z komentarzem

```
error[E0597]: `y` does not live long enough
  --> examples/lifetimes-not-enclosed.rs:8:13
   |
8  |         x = &y;
   |             ^^ borrowed value does not live long enough
...
12 |     }
   |     - `y` dropped here while still borrowed
13 |
14 |     println!("{}", *x);
   |                     -- borrow later used here
```

Rysunek 18: Wynik próby skompilowania kodu z listingu 40.

Listing 40. pokazuje okresy życia na przykładzie zmiennych lokalnych funkcji. Zgodnie z komentarzem zawartym na listingu, referencja kończy swoje życie wraz z obiektem referowanym i jej późniejsze użycie nie jest przez kompilator akceptowalne. W takim scenariuszu analiza poprawności jest stosunkowo prosta; potencjalnie możliwe byłoby rozbudowanie kompilatora języka C++ o podobny mechanizm. Niestety, złożoność relacji między referencjami często wymaga dodatkowych informacji, do których wyrażenia konieczna jest odpowiednia składnia. Aby temu zaradzić, Rust wprowadza nowy element składniowy w postaci specyfikatorów okresów życia (ang. *lifetime specifiers*).

Zanim je zbadamy, przeanalizujmy pośredni przypadek w postaci listingu 41. Na pierwszy rzut oka wszystko wygląda znajomo. Zamiast przepisywać referencję wprost (jak na listingu 40.), przekazujemy ją dodatkowo poprzez funkcję. Funkcja ta nie robi niczego użytecznego; po prostu natychmiast zwraca argument wejściowy. Komunikat o błędzie z rysunku 19. wskazuje, że kompilator w jakiś sposób dostrzega związek między przekazaną do funkcji referencją a referencją zwracaną; tak jak w przypadku z prostym pożyczeniem własności. Aby to wyjaśnić, konieczne jest uzupełnienie funkcji `noop` o jawne specyfikatory okresów życia (listing 42.). W ogólności przypadek ten zalicza się do grupy standardowych, automatycznie dedukowanych przez kompilator sygnatur funkcji korzystających z referencji [23, s. 201-204]; w tym sensie użycie specyfikatorów jest nadmiarowe, a listingi 41. i 42. są w pełni równoważne.

```rust
fn main() {
    let x;
    {
        let y = 1;
        x = noop(&y);
    };

    println!("{}", *x);
}

fn noop(input: &i32) -> &i32 {
    input
}
```

Listing 41: Rust – przyjęcie i natychmiastowe zwrócenie referencji

```
error[E0597]: `y` does not live long enough
 --> examples/passthrough-reference.rs:5:18
  |
5 |         x = noop(&y);
  |                  ^^ borrowed value does not live long enough
6 |     };
  |     - `y` dropped here while still borrowed
7 |
8 |     println!("{}", *x);
  |                     -- borrow later used here
```

Rysunek 19: Wynik próby skompilowania kodu z listingu 41.

Komentarz na listingu 42. stara się ukazać logikę, w ramach której kompilator dokonuje analizy poprawności. W linii 16. można zauważyć nowinkę w postaci wspomnianego specyfikatora okresu życia. Przez argument lub specyfikator okresu życia rozumiemy token zaczynający się od symbolu ' zakończonego nazwą. Tradycyjnie, jako nazwy stosuje się kolejne małe litery alfabetu łacińskiego, choć może to być dowolny identyfikator.

```
1  fn main() {
2      let x;
3      {
4          let y = 1;   // --->--*--->--- Czas życia 'a (zmiennej `y`) --->---*
5  //                           |                                              |
6          x = noop(&y); // --<--*                                              |
7  //      | \-----*             | 'a                                           |
8  //      \       \   'b        |                                              |
9      } //(koniec)  *---<---*   *------------>----------------------*    (koniec)
10 //                         |                                      |
11 //                         *---<---*                              *------>-----*
12     println!("{}", *x); //   'b   |                                        'a  |
13 }   //                             *- Czas życia 'b (zmiennej `x`) -*          |
14 //                                    'b: żyje co najwyżej 'a       |          |
15 //                                                                  |          |
16 fn noop<'z>(input: &'z i32) -> &'z i32 { //                         | 'b       |
17     input //          |           |                                  |          |
18     //                |        'b |     Tłumaczenie:                 |          |
19 }   //                |           *->-  dla każdej referencji wejściowej       |
20 //                 'a |                 o okresie życia 'z, referencja         |
21 //                   |                 wyjściowa żyje co najwyżej 'z          |
22 //                   |                                                  'a  |
23 //                   *------------------------------<-------------------------*
```

Listing 42: Rust – listing 41. uzupełniony o jawne okresy życia i komentarz

Mając w pamięci rozdział 1.4 możemy zauważyć podobieństwo zapisu okresów życia do notacji stosowanej przy parametrach generycznych. Nie jest ono przypadkowe; semantyka obu konstrukcji jest zbliżona, a okresy życia można uznać za rodzaj parametrów generycznych. O ile w przypadku standardowym programista wyraża uogólnienie definicji funkcji czy struktury względem wybranej rodziny typów, o tyle specyfikator okresu życia ma na celu wskazanie relacji między referencjami a elementami okalającymi (w szczególności innymi referencjami). Dla funkcji oznacza to wskazanie związku między okresami życia parametrów wejściowych a wartościami zwracanymi (komentarz z listingu 42.). W przypadku definicji struktur (a także typów wyliczeniowych, krotek itd.) ma to na celu związanie okresu życia rzeczonej struktury z przechowywaną w jej wnętrzu referencją lub referencjami co pokazuje listing 43.

```rust
struct Struct<'a, T> { // --*
    //                       |- Tłumaczenie:
    value: &'a T // --------*  dla każdego pola `value` o typie T i okresie
}   //                        życia 'a, instancja struktury `Struct` żyje
//                            co najwyżej 'a

fn main() {
    let i = 123;
    let string = String::from("This is text");

    let s1 = Struct { value: &i };
    let s2 = Struct { value: &string };
}
```

Listing 43: Rust – generyczna struktura z referencją

Język nie pozwoli na doprowadzenie do sytuacji, w której instancja struktury przeżywa przechowywaną w jego wnętrzu referencję (listing 44.; komunikat o błędzie na rysunku 20.).

```rust
struct Struct<'a, T> {
    value: &'a T
}

fn main() {
    let s1 = bad_struct();
}

fn bad_struct<'a>() -> Struct<'a, String> {
    let string = String::from("This is text");
    Struct { value: &string } // Błąd kompilacji
}
```

Listing 44: Rust – błędne wpisanie krótkożyjącej referencji do struktury

```
error[E0515]: cannot return value referencing local variable `string`
  --> examples/struct-generic-over-lifetime-outlived-ref-via-function.rs:11:5
   |
11 |     Struct { value: &string } // Błąd kompilacji
   |     ^^^^^^^^^^^^^^^^-------^^
   |     |               |
   |     |               `string` is borrowed here
   |     returns a value referencing data owned by the current function
```

Rysunek 20: Wynik próby skompilowania kodu z listingu 44.

Co więcej, przykład z listingu 44. pokazuje, że próba zwrócenia referencji do elementu znajdującego się na stosie bieżącej funkcji jest operacją niewyrażalną. Skoro funkcja nie posiada żadnych argumentów wejściowych, z którymi zwracana struktura mogłaby się okresem życia związać, oznacza to, że struktura może posiadać referencję co najwyżej do jakiegoś elementu na stosie danej funkcji i tym samym jej zwrócenie nie ma sensu.

Wyjątkiem od wyżej omówionego rozumowania stanowią referencje do zasobów, które istnieją przez cały czas działania programu. Dla nich w Rustcie zarezerwowany jest specjalny specyfikator okresu życia o nazwie `'static`. Do takich zaliczają się m.in. zmienne globalne, ale także stałe tekstowe i literały liczbowe, co pokazuje listing 45. Przy okazji zasobów statycznych, warto wspomnieć o typie stałych tekstowych, które w Rustcie definiuje się jako `&'static str`[9]. Jego odpowiednikiem w języku C jest typ `const char *`.

```rust
struct Struct<'a, T> {
    value: &'a T
}

fn main() {
    let a = 123; // ------------- Okres życia: 'a ---------*
    //                                                    |
    let s1 = ok_struct_generic_over_lifetime(&a); // ------*
    let s2 = ok_struct_generic_over_lifetime(&123); // Okres życia: 'static
    // let s3 = ok_struct_only_static(&a); // Błąd kompilacji
    let s4 = ok_struct_only_static(&123);
}

fn ok_struct_only_static(value: &'static i32) -> Struct<'static, i32> {
    Struct { value }
}

fn ok_struct_generic_over_lifetime<'a>(value: &'a i32) -> Struct<'a, i32> {
    Struct { value }
}
```

Listing 45: Rust – zasoby o statycznym okresie życia

W przypadku, jeśli funkcja posiada wiele argumentów wejściowych związanych pomiędzy sobą i elementem zwracanym tym samym okresem życia, będzie on żył przez okres życia najkrócej żyjącego elementu wejściowego. Tu za przykład może posłużyć komunikat o błędzie kompilacji listingu 46. pokazany na rysunku 21.

[9]Typ `str`, obok wycinków (ang. *slices*; `&[T]`) i obiektów cechowych (ang. *trait objects*; `&dyn T_Trait`) zalicza się do typów o nieznanym rozmiarze (ang. *Dynamically Sized Types; DSTs*). Nie ma możliwości stworzenia instancji takich typów na stosie; zawsze wymagają dostępu poprzez referencję/wskaźnik. Opierają się na tzw. ciężkich wskaźnikach (ang. *fat pointers*), tj. wskaźnik + metadane. DST wykraczają poza zakres tej pracy, nie mniej autor zachęca czytelnika do ich eksploracji [17, s. 62-69; 238-240][23, s. 75-81; 375-381; 441-443]

```rust
fn main() {
    let out;
    let a = 1;
    {
        let b = 2;
        {
            let c = 3;
            {
                out = foo(&a, &b, &c);
                // Okres życia `out`: 'out == min_of('a, 'b, 'c)
                println!("{}", *out); // Ok!        |
            } //                                    |
            println!("{}", *out); // Ok!           |
        } // 'out: (koniec) <- 'c (koniec) <-------*
        println!("{}", *out); // Błąd kompilacji
    }
    println!("{}", *out); // Błąd kompilacji
}

fn foo<'a>(a: &'a i32, b: &'a i32, c: &'a i32) -> &'a i32 {
    if a < b {
        a
    }
    else if a > b {
        b
    }
    else {
        c
    }
}
```

Listing 46: Rust – wiele parametrów funkcji związanych wspólnym okresem życia

```
error[E0597]: `b` does not live long enough
  --> examples/multiple-input-references-in-function.rs:9:31
   |
9  |                 out = foo(&a, &b, &c);
   |                               ^^ borrowed value does not live long enough
...
16 |     }
   |     - `b` dropped here while still borrowed
17 |     println!("{}", *out); // Błąd kompilacji
   |                     ---- borrow later used here

error[E0597]: `c` does not live long enough
  --> examples/multiple-input-references-in-function.rs:9:35
   |
9  |                 out = foo(&a, &b, &c);
   |                                   ^^ borrowed value does not live long enough
...
14 |         } // 'out: (koniec) <- 'c (koniec) <-------*
   |         - `c` dropped here while still borrowed
15 |         println!("{}", *out); // Błąd kompilacji
   |                         ---- borrow later used here
```

Rysunek 21: Wynik próby skompilowania kodu z listingu 46.

Identyczne rozumowanie dotyczy struktur (także typów wyliczeniowych, krotek itd.). Okres życia struktury posiadającej wiele pól związanych tym samym specyfikatorem okresu życia jest co najwyżej taki jak najkrócej żyjącego elementu składowego (listing 47. i rysunek 22).

```rust
#[derive(Debug)]
struct Struct<'a> {
    a: &'a i32,
    b: &'a i32,
    c: &'a i32,
}

fn main() {
    let out;
    let a = 1;
    {
        let b = 2;
        {
            let c = 3;
            {
                out = Struct { a: &a, b: &b, c: &c };
                // Okres życia `out`: 'out == min_of('a, 'b, 'c)
                println!("{:?}", out); // Ok!       |
            } //                                    |
            println!("{:?}", out); // Ok!          |
        } // 'out: (koniec) <- 'c (koniec) <-------*
        println!("{:?}", out); // Błąd kompilacji
    }
    println!("{:?}", out); // Błąd kompilacji
}
```

Listing 47: Rust – wiele pól w strukturze związanych wspólnym okresem życia

```
error[E0597]: `b` does not live long enough
  --> examples/multiple-references-in-struct.rs:16:42
   |
16 |                 out = Struct { a: &a, b: &b, c: &c };
   |                                          ^^ borrowed value does not live long enough
...
23 |     }
   |     - `b` dropped here while still borrowed
24 |     println!("{:?}", out); // Błąd kompilacji
   |                       --- borrow later used here

error[E0597]: `c` does not live long enough
  --> examples/multiple-references-in-struct.rs:16:49
   |
16 |                 out = Struct { a: &a, b: &b, c: &c };
   |                                                 ^^ borrowed value does not live long enough
...
21 |         } // 'out: (koniec) <- 'c (koniec) <-------*
   |         - `c` dropped here while still borrowed
22 |         println!("{:?}", out); // Błąd kompilacji
   |                           --- borrow later used here
```

Rysunek 22: Wynik próby skompilowania kodu z listingu 47.

Wspomniana na początku przechodniość reguł mieszania referencji wynika bezpośrednio z okresów życia i mechanizmu kontroli pożyczeń własności. Dzięki nim pobranie referencji do wartości przechowywanej w wektorze (listing 14.; s. 14) daje taki efekt, jakby pożyczenie własności dotyczyło samego wektora. Omawianą już sygnaturę `fn index(&self, index: T) -> &U` można rozwinąć do postaci `fn index<'a>(&'a self, index: T) -> &'a U`[10]; dzięki temu widać powiązanie między wektorem (`&'a self`), a zwracaną referencją do elementu składowego (`&'a U`).

Warto zaznaczyć, że o ile te koncepcje mogą być przytłaczające na początku nauki, wraz z praktyką przychodzi głębsze ich zrozumienie. Wymuszane przez nie techniki ostatecznie okazują się czymś, co w języku takim jak C++ można nazwać dobrymi praktykami, z dodatkowym atutem w postaci błędu kompilacji w wypadku ich naruszenia.

1.6 Inne istotne elementy

W tym rozdziale zostaną opisane elementy Rusta, które są warte uwagi, ale z różnych powodów nie zasługują na osobny, rozbudowany rozdział.

1.6.1 Iteratory

Podobnie jak w innych językach programowania, również w Rustcie dostępna jest mechanika iteratorów. Jest to technika pozwalająca na stosowanie tzw. wartościowania leniwego (ang. *lazy evaluation*). W Pythonie jest ona realizowana poprzez generatory oraz iteratory; w C# poprzez interfejs `IEnumerable<T>`. Oznacza to, że właściwa iteracja zachodzi dopiero w momencie, gdy jej wynik jest potrzebny. Sama ilość iteracji jest z zasady minimalna; nie ma mowy o nadmiarowych, naiwnych przebiegach pętli. Iteratory, ze względu na swoją specyfikę można komponować, co rozbudowuje wachlarz ich możliwości. Listing 48. buduje krótki wiersz wg prostego schematu widocznego na rysunku 23. Podstawą wiersza jest tablica słów kluczowych. Jego celem jest poglądowe ukazanie możliwości i zwięzłości zapisu iteratorów.

[10] Podobnie jak w uprzednio, metody struktur należą do rodziny sygnatur automatycznie dedukowanych.

```rust
fn main() {
    let coll = [
        "nail", "shoe", "horse", "rider", "message", "battle", "kingdom",
    ];
    let poem = coll
        .iter()
        .zip(coll.iter().skip(1))
        .map(|(a, b)| format!("For want of a {} the {} was lost.", a, b))
        .chain(
            coll.iter()
                .take(1)
                .map(|v| format!("And all for the want of a {}.", v)),
        )
        .collect::<Vec<_>>()
        .join("\n");
    println!("{}", poem);
}
```

Listing 48: Rust – przykładowa kompozycja iteratorów

```
For want of a nail the shoe was lost.
For want of a shoe the horse was lost.
For want of a horse the rider was lost.
For want of a rider the message was lost.
For want of a message the battle was lost.
For want of a battle the kingdom was lost.
And all for the want of a nail.
```

Rysunek 23: Wynik działania programu z listingu 48.

Większość wywołań funkcji w tym kodzie ma na celu jedynie zdefiniowanie oczekiwanego zachowania; na tym etapie nie dochodzi jeszcze do iteracji. W efekcie przykładowa konstrukcja taka jak

`list.iter().zip(list.iter().skip(3)).map(|el| el + 2).chain(list.iter())`

zwraca iterator złożony z wielu iteratorów o typie (w przybliżeniu)

`Chain<Map<Zip<Iter, Skip<Iter>>>, Iter>`

gdzie zewnętrzny iterator (z perspektywy typu) iteruje po iteratorze lub iteratorach wewnętrznym.

Podstawą iteratorów (w najprostszym przypadku; iteratory mogą być także dwukierunkowe) jest metoda `Iterator::next(&mut self) -> Option<T>`, która pobiera kolejny element. Jeśli zwróci ona wariant `Option<T>::None`, oznacza to, że iterator został wyczerpany i nie można go już użyć. Metody takie jak `collect()` czy `count()` konsumują cały iterator za pomocą metody `next()`, aby odpowiednio zebrać elementy do nowo zaalokowanej kolekcji lub sprawdzić ich ilość. Nic nie stoi na przeszkodzie, by tworzyć nieskończone iteratory, np. do generowania liczb pseudolosowych. Detale związane z ich użytkowaniem pozostawione są czytelnikowi do dalszej eksploracji w innych źródłach [17, rozdział 15].

Iteratory dostarczają programiście zestawu wszechstronnych abstrakcji do manipulowania kolekcjami czy strumieniami. Ich używanie nie wiąże się z żadnym nieoczekiwanym kosztem[11] i z tego powodu będą bardzo wartościowe także przy pracy w środowisku o ograniczonych zasobach, jakim są m.in. systemy wbudowane.

1.6.2 Makra

W Rustcie makra, w odróżnieniu od preprocesora C/C++, stanowią część języka i są świadome jego struktury. Przykładowo, ciała makr muszą zawierać poprawny w sensie składniowym kod źródłowy, a parametry wejściowe do makr są metazmiennymi, czyli zmiennymi których typem jest rodzaj tokenu drzewa składniowego (ang. *abstract syntax tree*; *AST*). Oprócz tego Rust wprowadza makra proceduralne, które w formie funkcji lub atrybutu mogą przyjmować jako parametr fragment drzewa składniowego Rusta, modyfikować go, a następnie zwracać. Owe makra pozwalają na tworzenie bibliotek opartych na technikach metaprogramistycznych, gdzie programista korzysta z wprowadzonego przez bibliotekę języka domenowego (ang. *domain specific language*; *DSL*), a kompilator generuje odpowiedni kod źródłowy Rusta. W przypadku systemów wbudowanych powstało wiele użytecznych, korzystających z tych technik bibliotek; jedna z nich stanowi podstawę stworzonego w ramach pracy przykładowego projektu (biblioteka `cortex-m-rtic`; rozdział 4.1). Makra w Rustcie są potężne i ich pełny opis mógłby bez problemu zająć osobną, pełnoprawną pozycję literaturową; autor zachęca do dalszego poszerzania wiedzy na ich temat w zacytowanych źródłach [17, rozdział 20].

1.6.3 *Unsafe* Rust

Wszystkie omawiane dotychczas w pracy techniki i mechanizmy dotyczą podzbioru języka określanego mianem bezpiecznego Rusta (ang. *Safe Rust*). Korzystanie z niego gwarantuje brak ryzyka związanego z zajściem zachowania niezdefiniowanego, które jest jednym z fundamentalnych zagrożeń znanych z operujących bezpośrednio na pamięci języków programowania systemowego[12]. Niestety, w pewnych sytuacjach techniki przewidziane w ramach bezpiecznego Rusta są niewystarczające przy rozwiązywaniu niektórych problemów; na przykład przy interakcji z surowym rejestrem sprzętowym wymagającym dereferencji pozornie arbitralnego adresu zmapowanego na sprzęt.

Aby temu zaradzić, twórcy języka stworzyli słowo kluczowe `unsafe` wprowadzające podzbiór języka określanego mianem niebezpiecznego Rusta (ang. *Unsafe Rust*), którego najważniejszą funkcjonalnością jest umożliwienie dereferencji surowych wskaźników (`*const T`/`*mut T`) w obrębie bloku opatrzonego rzeczonym słowem (przykład na listingu 49.). Wskaźniki w Ru-

[11] Nieoczekiwanym tj. wynikającym z samych iteratorów. Listing 48. zawiera alokacje pamięci, ze względu na wykorzystanie makra `format` oraz metod `Iterator::collect` i `Vec::join`.

[12] Z wyjątkiem sytuacji, w której program lub biblioteka nie ma charakterystyki solidności; rozwinięte w dalszej części rozdziału.

stcie odróżnia od referencji brak śledzonego w czasie kompilacji okresu życia, w wyniku czego kompilator nie jest w stanie ocenić poprawności programu i musi zaufać programiście. Blok `unsafe` podkreśla jawność niebezpiecznych punktów w programie i wspomaga rozdzielanie niskopoziomowych detali implementacyjnych od wysokopoziomowych, bezpiecznych interfejsów użytkownika.

```
fn main() {
    unsafe {
        *(0xdead_beef as *mut u32) = 123; // Zachowanie niezdefiniowane
    }
}
```

Listing 49: Rust – zachowanie niezdefiniowane

Korzystanie z niebezpiecznego Rusta, jak nazwa wskazuje, wymaga dodatkowej ostrożności ze strony programisty. Standardy jakości przyjęte przez społeczność języka wymagają od deweloperów ścisłej weryfikacji czy dowolne użycie tworzonych przez nich wysokopoziomowych abstrakcji (korzystających w implementacji z `unsafe`) nie doprowadzą do zachowania niezdefiniowanego. Mówi się, że program czy biblioteka pozbawiona tego typu błędów implementacyjnych posiada charakterystykę tzw. solidności (ang. *soundness*). Co istotne, pozostałe obostrzenia bezpiecznego Rusta dalej obowiązują w bloku `unsafe`.

Na szczęście, w większości sytuacji nie ma konieczności korzystania z niebezpiecznego Rusta. W przypadku popularnych modeli systemów wbudowanych (o mikroarchitekturze *ARM*, w szczególności opartych na rodzinie *Cortex M*) programista ma do dyspozycji gotowe rozwiązania opakowujące znaczną część niskopoziomowych detali w bezpieczne abstrakcje; przykładowo rejestry sprzętowe dostępne są pod postacią bezpiecznych w użyciu struktur. Nie mniej jednak czytelnik powinien być świadomym istnienia tego typu elementów języka w razie, gdyby zaszła potrzeba wykonania niewykonalnych lub bardzo trudnych w bezpiecznym Rustcie operacji.

2 Rust jako ekosystem

Opis języka Rust nie byłby kompletny bez omówienia towarzyszącego mu ekosystemu. Na użyteczność języka programowania składa się nie tylko jego składnia, ale także narzędzia, istniejące biblioteki czy choćby nomenklatura, której znajomość pozwala na sprawniejsze rozwiązywanie napotykanych problemów i wyszukiwanie dodatkowych informacji.

2.1 Instalacja środowiska

Na stronie domowej projektu *Rust* [25] można znaleźć odpowiednie instalatory dla większości popularnych platform. Instalatory są obsługiwane z wiersza poleceń i przystępnie prowadzą użytkownika przez cały proces. Po poprawnie przeprowadzonej instalacji, w katalogu użytkownika powinny znajdować się katalogi `.cargo` oraz `.rustup`, a zmienna środowiskowa `PATH` powinna zostać uzupełniona o ścieżkę `<katalog-domowy>/.cargo/bin`.

2.2 Toolchain i jego dystrybucja

Toolchain określa zestaw bazowych narzędzi prekompilowanych na wybraną platformę. W przypadku Rusta najważniejsze z nich to:

- menedżer środowiska *rustup*;
- kompilator *rustc*;
- menedżer projektów i zależności *cargo*.

Toolchainy są dystrybuowane w ramach 3 kanałów: *stable*, *beta* oraz *nightly*. Ten ostatni posiada najczęstszy harmonogram wydawniczy i tylko on pozwala na korzystanie z nieustabilizowanych funkcjonalności podlegającym ciągłym zmianom[13]. Stopniowo, wraz z upływem czasu wprowadzane poprawki i funkcjonalności stają się dostępne w kanale *beta*, a ostatecznie lądują w *stable*, gdzie stabilność API jest zagwarantowana. Kanał *stable* powinien stanowić rozsądny wybór dla początkujących programistów Rusta.

Komendy `rustup toolchain ...` oraz `rustup default <toolchain>` pozwalają odpowiednio na zarządzanie (dodawanie, usuwanie) zainstalowanymi toolchainami (w celu zmiany kanału dystrybucji lub wybranie alternatywy jeśli dana platforma na to pozwala) oraz wybieranie domyślnego.

[13] Konieczne jest także użycie atrybutu `feature(nazwa_funkcjonalności)` na poziomie *crate'u* (*crate*; patrz rozdział 2.4); dzięki temu nie jest możliwe przypadkowe użycie funkcjonalności niedostępnych poza kanałem *nightly*.

W przypadku systemu *MS Windows* w wersji 64-bitowej dostępne są dwa toolchainy:

- oparty na *MSVC* `<kanał>-x86_64-pc-windows-msvc`[14];
- oparty o *MinGW* `<kanał>-x86_64-pc-windows-gnu`.

Rozróżnienie wynika z niekompatybilności binarnych i ma szczególne znaczenie przy korzystaniu z prekompilowanych bibliotek na maszynie gospodarza. Pierwszy wymaga zainstalowanego *Visual C++ Build tools*, który zawiera brakujący konsolidator `LINK.EXE`. Drugi jest samowystarczalny, stąd na potrzeby ćwiczeń i pisania prostych, opartych o wiersz poleceń aplikacji warto wybrać właśnie jego. Wybór między tymi dwoma toolchainami nie robi różnicy aplikacjom budowanym na systemy wbudowane.

Dla systemów operacyjnych opartych o jądro *Linux* sprawy mają się nieco prościej. Zazwyczaj najczęściej używanym toolchainem jest `<kanał>-x86_64-unknown-linux-gnu`, jako że znaczna większość dystrybucji[15] domyślnie korzysta z *glibc* jako implementacji biblioteki standardowej C. Dystrybucje dostarczające domyślnie *musl libc*[16] mają do dyspozycji toolchain `<kanał>-x86_64-unknown-linux-musl`.

2.3 Target

Kolejnym istotnym elementem nomenklatury Rusta jest koncepcja targetu. Opisuje on platformę docelową, z jaką ma być kompatybilny wynikowy kod maszynowy. Aby to osiągnąć, kompilator Rusta nie wymaga rekompilacji z innymi opcjami tak, jak ma to miejsce w przypadku *gcc*[17]; kompilator znajdujący się w dowolnym toolchainie wspiera automatycznie wszystkie aktualnie obsługiwane platformy docelowe, choć niektóre warianty kompilacji skrośnej mogą być chwilowo nieosiągalne (próba linkowania z niedostępnymi na maszynie gospodarza bibliotekami)[18].

Z targetem ściśle związana jest implementacja biblioteki standardowej, która w Rustcie dzieli się na dwie części: *libcore* oraz *libstd*. Ta pierwsza nie czyni żadnych założeń wobec platformy, na której będzie używana. Zawiera ona podstawowe cechy i struktury niewymagające systemu operacyjnego, alokacji pamięci, systemu plików itd. Biblioteka *libstd* stanowi nadzbiór *libcore* i udostępnia narzędzia pozwalające na interakcję z systemem operacyjnym. W zależności od targetu dostępne są obie biblioteki lub, tak jak ma to miejsce w przypadku systemów wbudowanych, jedynie *libcore*. Aby wyłączyć automatyczne linkowanie do pełnej biblioteki standardowej, należy użyć atrybutu `no_std` na poziomie *crate'u* (*crate*; patrz rozdział 2.4); jest to konieczne w przypadku projektów na systemy wbudowane. Co intuicyjne, takie projekty mogą korzystać tylko i wyłącznie z bibliotek również opatrzonych takim atrybutem.

[14] Przykładem właściwego toolchaina jest `stable-x86_64-pc-windows-msvc`.
[15] Debian, Ubuntu, Fedora, Red Hat, CentOS, Arch Linux itd.
[16] Alpine Linux, Void Linux itd.
[17] Kompilator języka C należący do kolekcji kompilatorów GNU (ang. *GNU Compiler Collection*; *GCC*)
[18] W niektórych ekosystemach zdarzają się też kombinacje zupełnie nieosiągalne (ekosystem *Apple*; patrz [15])

Tabela 1. zawiera kilka przykładowych targetów. Do ich nazywania stosuje się tzw. *target-triple* [26]; konwencję stworzoną podczas prac nad *GNU Autoconf*. W kolumnie *libstd* symbol ♦ oznacza dostępność pełnego *libstd*, natomiast symbol ◊ jedynie *libcore*. Symbol w kolumnie *toolchain* oznacza, że w ramach danej platformy docelowej dostępny jest toolchain pozwalający na budowanie projektów Rustowych. Przewijające się w opisach słowo *bare* odnosi się do programowania bezpośrednio na sprzęcie (ang. *bare-metal*), bez warstwy pośredniej w postaci np. systemu operacyjnego.

target	libstd	toolchain	opis
`x86_64-apple-darwin`	♦	♦	64-bit OSX (10.7+, Lion+)
`x86_64-pc-windows-gnu`	♦	♦	64-bit MinGW (Windows 7+)
`x86_64-pc-windows-msvc`	♦	♦	64-bit MSVC (Windows 7+)
`x86_64-unknown-linux-gnu`	♦	♦	64-bit Linux (2.6.32+, glibc 2.11+)
`aarch64-pc-windows-msvc`	♦		ARM64 Windows MSVC
`aarch64-unknown-linux-gnu`	♦	♦	ARM64 Linux (4.2, glibc 2.17)
`aarch64-unknown-linux-musl`	♦		ARM64 Linux with MUSL
`aarch64-unknown-none`	◊		Bare ARM64, hardfloat
`thumbv6m-none-eabi`	◊		Bare Cortex-M0, M0+, M1
`thumbv7m-none-eabi`	◊		Bare Cortex-M3
`thumbv7em-none-eabi`	◊		Bare Cortex-M4, M7
`thumbv7em-none-eabihf`	◊		Bare Cortex-M4F, M7F
`riscv32i-unknown-none-elf`	◊		Bare RISC-V (RV32I ISA)
`riscv32imac-unknown-none-elf`	◊		Bare RISC-V (RV32IMAC ISA)
`riscv32imc-unknown-none-elf`	◊		Bare RISC-V (RV32IMC ISA)
`riscv64gc-unknown-linux-gnu`	♦	♦	RISC-V Linux (4.20, glibc 2.29)
`riscv64gc-unknown-none-elf`	◊		Bare RISC-V (RV64IMAFDC ISA)
`riscv64imac-unknown-none-elf`	◊		Bare RISC-V (RV64IMAC ISA)

Tabela 1: Zestawienie targetów względem toolchainów i implementacji bibliotek standardowych
Źródło: [15]

Programistów systemów wbudowanych opartych o rodzinę procesorów *ARM Cortex M* zainteresują targety zaczynające się od przyrostka `thumb`. Do zarządzania targetami służy polecenie `rustup target ...` pozwalające na dodawanie i usuwanie wsparcia dla wybranego targetu w aktualnie używanym toolchainie[19].

[19]Kompilator wspiera z góry wszystkie platformy docelowe; mimo to konieczne jest użycie narzędzia `rustup` w celu pobrania prekompilowanej biblioteki standardowej dla pożądanego targetu.

2.4 Cargo, *crate* i struktura projektów

Obok kompilatora, instalacja Rusta dostarcza menedżer projektu i zależności o nazwie Cargo. Do zarządzania projektem Cargo wykorzystuje pliki konfiguracyjne w formacie TOML (ang. *Tom's Obvious, Minimal Language*). Projekty w Rustcie noszą miano *crate'ów* czyli skrzynek[20]. *Crate* może być biblioteką lub samodzielną aplikacją; listing 50. pokazuje strukturę obu typów projektów w najprostszej postaci.

```
bin
|-- Cargo.toml
`-- src
    `-- main.rs
lib
|-- Cargo.toml
`-- src
    `-- lib.rs
```

Listing 50: Struktura katalogów najprostszej aplikacji i biblioteki

W pliku TOML możemy definiować zależności zewnętrzne (opublikowane w oficjalnym, publicznym repozytorium `crates.io`; przy pierwszej próbie zbudowania projektu Cargo pobierze brakujące zasoby z Internetu) lub wewnętrzne (np. referencję do *crate'u* bibliotecznego w innym katalogu). Możliwości związane z konfiguracją projektów są bardzo szerokie; więcej informacji można znaleźć w literaturze poświęconej Cargo [27].

Dla porównania C i C++ nie posiadają ustandaryzowanego systemu zarządzania projektami czy zależnościami; istnieje wiele konkurencyjnych systemów o zróżnicowanych możliwościach i zakresie kompetencji: *GNU Autoconf* [28], *GNU Make* [29], *CMake*[30], *Bazel* [31], *Meson* [32], *Conan*[21] [33], *Keil µVision* [34]. Różnorodność systemów można postrzegać jako zaletę lub wadę. Z jednej strony łatwiej dopasować narzędzia do własnych potrzeb, a z drugiej trudniej przekazać swój kod komuś innemu, jeśli użyliśmy mało popularnych rozwiązań. Nie wspominając o zwiększonej trudności zrozumienia ekosystemu z perspektywy osób początkujących.

[20]Mimo tego, że jeden *crate* w Rustcie może się składać z wielu katalogów i plików (modułów), stanowi on jako całość pojedynczą jednostkę kompilacyjną (odpowiednik pojedynczego pliku obiektowego).

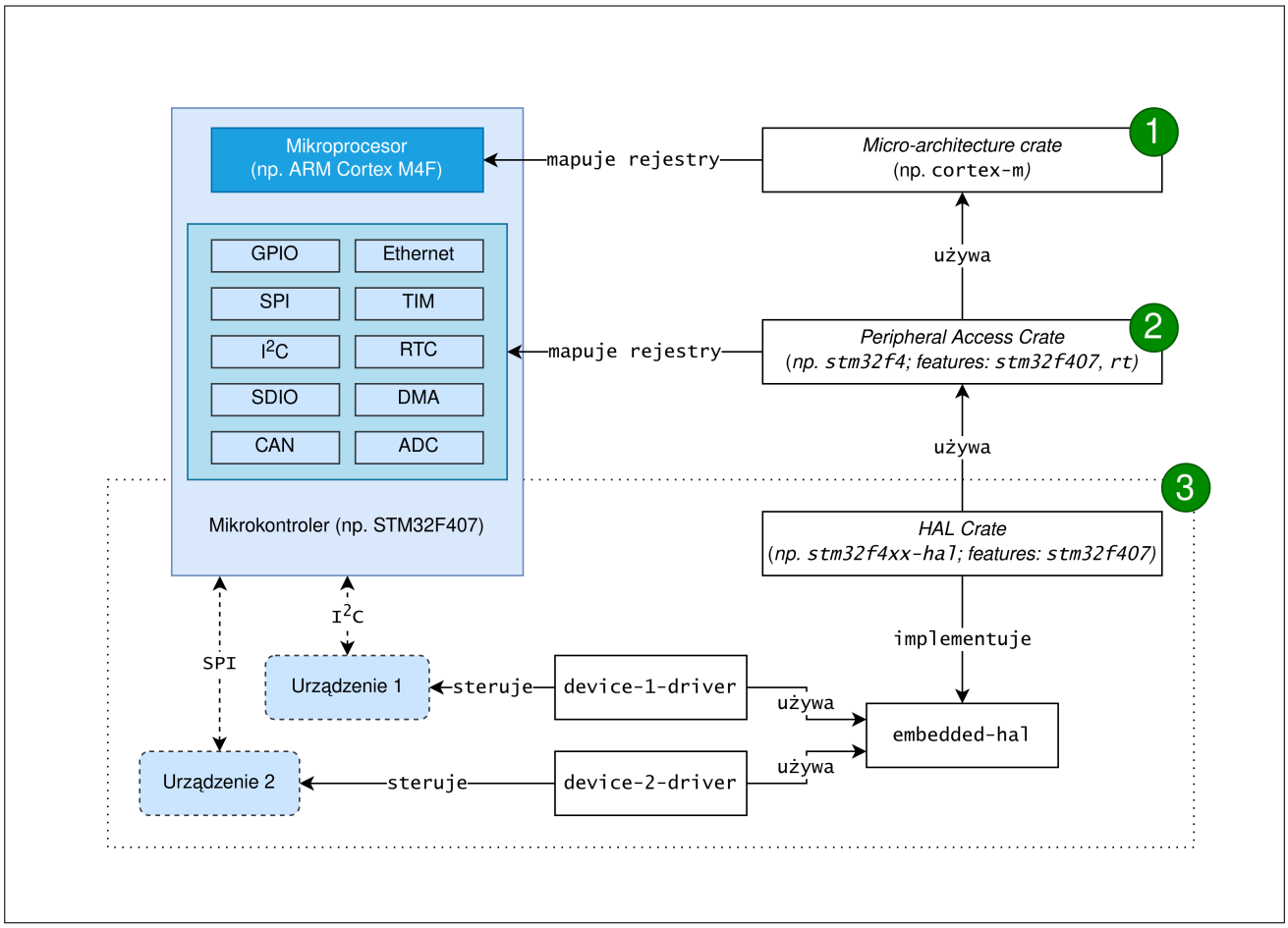
[21]Tylko tworzenie pakietów i zarządzanie zależnościami.

3 Rust w systemach wbudowanych

Poprzednie rozdziały stanowiły wprowadzenie do ogólnego ekosystemu Rustowego. Ten skupi się na omówieniu pracy z Rustem w kontekście systemów wbudowanych.

3.1 Abstrakcje

Grupa robocza *Rust Embedded*[22] ma za zadanie ulepszanie ekosystemu w zakresie systemów wbudowanych. W szczególności odpowiada ona za czynienie Rusta przystępniejszym poprzez utrzymywanie poradników, dokumentacji oraz moderację wprowadzanych przez społeczność innowacji w kluczowych abstrakcjach i bibliotekach. Podstawowe, obecnie stosowane abstrakcje ukazane są na rysunku 24. [35]



Rysunek 24: Podstawowe abstrakcje *Rust Embedded*

[22]Kontrola nad projektem języka Rust jest sprawowana przez społeczność; wybierane demokratycznie osoby decyzyjne znajdują się w zespołach o zróżnicowanych odpowiedzialnościach. Grupy robocze (ang. *Working Groups*; *WGs*) moderują rozwój różnych segmentów ekosystemu.

3.1.1 μAC

Micro-architecture crate (rysunek 24., nr 1 w zielonym kole), nazywany w dalszej części pracy μAC, mapuje rejestry sprzętowe danego mikroprocesora. Przykładowo, w przypadku procesora *ARM Cortex M* daje on dostęp do licznika SysTick oraz podstawowych przerwań. Wszystkie mikrokontrolery używające tego samego mikroprocesora, korzystają także z tego samego μAC.

3.1.2 PAC

PAC (ang. *Peripheral Access Crate*; rysunek 24., nr 2 w zielonym kole) to *crate* uzupełniający mapy rejestrów dostępnych z poziomu μAC o te specyficzne dla danego mikrokontrolera. Jest on generowany za pomocą narzędzia `svd2rust` z wykorzystaniem plików SVD[23] (ang. *System View Description*). Dla popularnych mikrokontrolerów manualne generowanie PACów nie jest konieczne; są one zazwyczaj dostępne w publicznym repozytorium bibliotek. W przypadku mikrokontrolerów ST, PACi dystrybuowane w ramach projektu *stm32-rs* wspierają całą jedną rodzinę urządzeń (np. *STM32{F1, F2, F3, F4..}XX*) na raz. W momencie podpięcia zależności należy wskazać, który mikrokontroler nas interesuje poprzez mechanizm funkcjonalności (ang. *features*; przykład na listingu 51.). Jest to technika wprowadzająca znaną z języka C kompilację warunkową.

```
[dependencies]
stm32f4 = { features = ["stm32f407"], version = "0.11.0" }
```

Listing 51: Przykład użycia zależności z poziomu pliku `Cargo.toml`

3.1.3 HAL

PAC i μAC są abstrakcjami niskopoziomowymi; dostarczają one jedynie zaadresowane rejestry peryferiów, przy których manipulowaniu, tak jak w języku C, konieczna jest drobiazgowa znajomość dokumentacji i ostrożność. HAL (ang. *Hardware Abstraction Layer*) w rozumieniu Rusta ma na celu oddalenie się w pewnym stopniu od sprzętu na rzecz uogólnienia i zwiększenia wygody użytkowania.

Na HALa w Rustcie (rysunek 24., nr 3 w zielonym kole) składają się dwa komponenty: uogólniony interfejs w postaci *crate'u* `embedded-hal` [36] oraz jego implementacje zależne od platform docelowych (*HAL Crate*). Ten pierwszy definiuje cechy (np. `i2c::Write`), a drugi w miarę możliwości je implementuje. Pozwala to na tworzenie niezależnych od mikrokontrolera sterowników do peryferiów zewnętrznych. Sterownik oczekuje na wejściu typów implementujących wybrane cechy z `embedded-hal`a, które programista dostarcza w momencie jego inicjalizacji.

[23] Są to pliki w formacie XML dostarczane przez producenta sprzętu i w ogólności służą do uzupełniania konfiguracji debuggera o adresację rejestrów zmapowanych na pamięć.

3.1.4 Dalsze inicjatywy i problemy

Obecnie prace nad usprawnieniami w tym rejonie Rusta są bardzo aktywne i powstają liczne propozycje dalszych inicjatyw i nowych projektów (np. `embedded-time` [37], `embedded-dma` [38], `embedded-nal` [39]) mających na celu ustandaryzowanie abstrakcji powszechnego użytku i tym samym ułatwienie tworzenia niezależnych od sprzętu sterowników. Część z nich jest już dojrzała (jak wspomniany `embedded-hal`), inne znajdują się na etapie projektowania lub prototypowania. Tempo rozwoju jest dynamiczne i wraz z upływem czasu można spodziewać się stopniowej poprawy sytuacji i stabilizowania się kolejnych bibliotek.

Co ważne, ekosystem Rusta nie jest wolny od problemów. Każda abstrakcja ma granice swojej uniwersalności i Rust nie jest tu wyjątkiem. Przykładowo, wiele istniejących sterowników do urządzeń niefortunnie zakłada, że są unikalnymi właścicielami przekazywanych im zasobów. Stanowi to spore utrudnienie w przypadku takich magistral komunikacyjnych jak I^2C, które z natury są współdzielone. Istnieją rozwiązania tego problemu; jednym z nich jest biblioteka `shared-bus`, pozwalająca na współdzielenie magistrali bez wprowadzania zmian w samych sterownikach. Mimo tego, ze względu ze wykorzystanie specyficznych bibliotek w projecie stacji pogodowej dzielenie zasobów wciąż pozostało kłopotliwe (więcej w rozdziale 4. na stronie 66.).

Warto również wspomnieć, że poziom wsparcia różni się między mikrokontrolerami różnych producentów. O ile PACi można łatwo wygenerować dla dowolnego urządzenia, o tyle implementacje `embedded-hal` i innych *crate'ów* są na różnym poziomie zaawansowania. W momencie pisania tej pracy najpełniej wspierane są urządzenia *STMicroelectronics* [40].

3.1.5 Inne biblioteki pomocnicze

Standardowym podziałom opisanym w tym rozdziale wymykają się w szczególności dwa *crate'y*.

- `bare-metal` – *crate* opisujący powszechne abstrakcje użyteczne podczas programowania blisko sprzętu; obecnie są to typy `Mutex` i `CriticalSection`.
- `cortex-m-rt` – *crate* zawierający kod startowy dla mikrokontrolerów opartych o mikroprocesor ARM Cortex M; dostarcza ogólne skrypty linkera, a także definiuje między innymi tablicę wektorów przerwań; w tym najważniejszy, czyli `ResetHandler`.

3.2 Peryferia jako maszyny stanów

Zarządzanie peryferiami wewnętrznymi w systemach wbudowanych opiera się na odczytywaniu lub wpisywaniu odpowiednich wartości do rejestrów sprzętowych. Rzeczone zarządzanie wymaga znajomości peryferium; część rejestrów służy tylko do odczytu lub zapisu, a ustawienia mogą ze sobą konfliktować.

Rust stara się zaprząc system typów do przedstawiania peryferiów jako maszyny stanów, gdzie każdy stan odpowiada osobnemu typowi; metody zdefiniowane w ramach tych typów zwracają obiekty innego typu, reprezentując w ten sposób zmianę stanu [41]. Technika ta jest stosowana przez wysokopoziomowe abstrakcje *Rust Embedded*; w szczególności implementacje `embedded-hal`. Przykładowo, gdy peryferium nie jest zainicjalizowane, typ obiektu go reprezentującego posiada jedynie metody pozwalające na jego inicjalizację. Typ obiektu zwracanego z takiej funkcji inicjalizacyjnej odpowiadał będzie innemu stanowi maszyny i tym samym będzie dostarczał innego zestawu metod.

Listing 52. stanowi fragment implementacji peryferium `GPIOA` z *crate'u* `stm32f4xx-hal`[24]. Część z widocznych sygnatur znajduje się w blokach implementacyjnych cech, inne dotyczą bezpośrednio struktur. Często pojawiają się implementacje specjalizowane (linie 18, 21, 24 itd.), które pozwalają na dodawanie metod w obrębie zawężonej rodziny stanów. W linii 2. możemy zauważyć tajemniczo wyglądający typ `PhantomData<T>`; w kodzie intensywnie stosującym programowanie generyczne jest on bardzo powszechny. Jego jedynym celem jest przyjęcie podanego w definicji parametru generycznego `MODE`, ponieważ Rust nie pozwala na wiszące, nieużyte parametry generyczne; typ ten ma rozmiar 0, tak jak górująca nad nim struktura `PA5<T>`. Instancje tych typów nigdy w kodzie maszynowym nie powstaną i służą jedynie jako znaczniki dostępnych w danym momencie funkcji.

[24]Po rozwinięciu makr, usunięciu ciał metod i drobnych uproszczeniach. Oprócz tego zostały pominięte definicje typów `Input<T>` czy `OpenDrain`, które są trywialne. Wersja `0.8.3`.

```rust
pub struct PA5<MODE> {
    _mode: PhantomData<MODE>,
}
impl<MODE> PA5<MODE> {
    pub fn into_alternate_af0(self) -> PA5<Alternate<AF0>> { /* .. */ }
    pub fn into_alternate_af1(self) -> PA5<Alternate<AF1>> { /* .. */ }
    pub fn into_alternate_af2(self) -> PA5<Alternate<AF2>> { /* .. */ }
    // ..
    pub fn into_alternate_af0_open_drain(self) -> PA5<AlternateOD<AF0>> { /* .. */ }
    pub fn into_alternate_af1_open_drain(self) -> PA5<AlternateOD<AF1>> { /* .. */ }
    pub fn into_alternate_af2_open_drain(self) -> PA5<AlternateOD<AF2>> { /* .. */ }
    // ..
    pub fn into_floating_input(self) -> PA5<Input<Floating>> { /* .. */ }
    pub fn into_pull_down_input(self) -> PA5<Input<PullDown>> { /* .. */ }
    pub fn into_pull_up_input(self) -> PA5<Input<PullUp>> { /* .. */ }
    pub fn into_open_drain_output(self) -> PA5<Output<OpenDrain>> { /* .. */ }
    pub fn into_push_pull_output(self) -> PA5<Output<PushPull>> { /* .. */ }
    pub fn into_analog(self) -> PA5<Analog> { /* .. */ }
}
impl<MODE> PA5<Output<MODE>> {
    pub fn set_speed(self, speed: Speed) -> Self { /* .. */ }
}
impl PA5<Output<OpenDrain>> {
    pub fn internal_pull_up(&mut self, on: bool) { /* .. */ }
}
impl<MODE> PA5<Alternate<MODE>> {
    pub fn set_speed(self, speed: Speed) -> Self { /* .. */ }
    pub fn internal_pull_up(self, on: bool) -> Self { /* .. */ }
}
impl<MODE> PA5<Alternate<MODE>> {
    pub fn set_open_drain(self) -> PA5<AlternateOD<MODE>> { /* .. */ }
}
impl<MODE> OutputPin for PA5<Output<MODE>> {
    fn set_high(&mut self) -> Result<(), Infallible> { /* .. */ }
    fn set_low(&mut self) -> Result<(), Infallible> { /* .. */ }
}
impl<MODE> StatefulOutputPin for PA5<Output<MODE>> {
    fn is_set_high(&self) -> Result<bool, Infallible> { /* .. */ }
    fn is_set_low(&self) -> Result<bool, Infallible> { /* .. */ }
}
impl<MODE> InputPin for PA5<Output<MODE>> {
    fn is_high(&self) -> Result<bool, Infallible> { /* .. */ }
    fn is_low(&self) -> Result<bool, Infallible> { /* .. */ }
}
impl<MODE> InputPin for PA5<Input<MODE>> {
    fn is_high(&self) -> Result<bool, Infallible> { /* .. */ }
    fn is_low(&self) -> Result<bool, Infallible> { /* .. */ }
}
```

Listing 52: Uproszczony pseudokod Rusta – implementacja pinu `PA5` peryferium `GPIOA`

Aby zobaczyć, jak te techniki wpływają na pracę przy projekcie, rozważmy listing 53. będący fragmentem kodu inicjalizującego `USART1` zmodyfikowanym tak, by używać niepoprawnych pinów.

```rust
let gpioa = cx.device.GPIOA.split();
let (serial_transmitter, serial_receiver) = {
    let tx = gpioa.pa7.into_alternate_af7(); // Zamiast poprawnego pa9
    let rx = gpioa.pa8.into_alternate_af7(); // Zamiast poprawnego pa10
    let mut serial = Serial::usart1( // Błąd kompilacji
        cx.device.USART1,
        (tx, rx),
        Config {
            baudrate: 115_200.bps(),
            ..Config::default()
        },
        clocks,
    )
    .unwrap();
    serial.listen(Event::Rxne);
    serial.split()
};
```

Listing 53: Wycinek kodu inicjalizującego `USART1` z projektu stacji pogodowej (patrz rozdział 4)

```
error[E0277]: the trait bound `PA7<Alternate<stm32f4xx_hal::gpio::AF7>>: PinTx<stm32f4xx_hal::stm32::USART1>` is not satisfied
  --> src/main.rs:65:30
   |
65 |             let mut serial = Serial::usart1(
   |                              ^^^^^^^^^^^^^^ the trait `PinTx<stm32f4xx_hal::stm32::USART1>` is not implemented for `PA7<Alternate<
stm32f4xx_hal::gpio::AF7>>`
   |
   = note: required because of the requirements on the impl of `stm32f4xx_hal::serial::Pins<stm32f4xx_hal::stm32::USART1>` for `(PA7<Al
ternate<stm32f4xx_hal::gpio::AF7>>, PA8<Alternate<stm32f4xx_hal::gpio::AF7>>)`
   = note: required by `Serial::<stm32f4xx_hal::stm32::USART1, PINS>::usart1`

error[E0277]: the trait bound `PA8<Alternate<stm32f4xx_hal::gpio::AF7>>: PinRx<stm32f4xx_hal::stm32::USART1>` is not satisfied
  --> src/main.rs:65:30
   |
65 |             let mut serial = Serial::usart1(
   |                              ^^^^^^^^^^^^^^ the trait `PinRx<stm32f4xx_hal::stm32::USART1>` is not implemented for `PA8<Alternate<
stm32f4xx_hal::gpio::AF7>>`
   |
   = note: required because of the requirements on the impl of `stm32f4xx_hal::serial::Pins<stm32f4xx_hal::stm32::USART1>` for `(PA7<Al
ternate<stm32f4xx_hal::gpio::AF7>>, PA8<Alternate<stm32f4xx_hal::gpio::AF7>>)`
   = note: required by `Serial::<stm32f4xx_hal::stm32::USART1, PINS>::usart1`
```

Rysunek 25: Wynik próby skompilowania projektu z modyfikacją z listingu 53.

Komunikat o błędzie z rysunku 25. tłumaczy problem od szczegółu do ogółu; w celu zbudowania intuicji czytelnika przeanalizujemy problem w odwrotną stronę, zaczynając od notatek na dole. Wszystkie poniżej wymienione informacje można znaleźć w automatycznie generowanej z kodu dokumentacji, do której odpowiednie odniesienia znajdują się w każdym z punktów.

- Sygnatura metody `Serial::<USART1, PINS>::usart1` ([42]; listing 54.) narzuca na generyczny typ `PINS` obostrzenie: musi implementować cechę `Pins<USART1>`.

```
//                V---------- implementacja dla konkretnego typu peryferium `USART1`
impl<PINS> Serial<USART1, PINS> {
//   ^--------------------^-- dla każdego typu `PINS`
    pub fn usart1(
        usart: USART1,
        pins: PINS,
        config: Config,
        clocks: Clocks
    ) -> Result<Self, InvalidConfig> where PINS: Pins<USART1>,
} //                                      ^------ spełniającego obostrzenie
```

Listing 54: Sygnatura `Serial::<USART1, PINS>::usart1`

- Cecha `Pins<USART>` jest implementowana ([43]; listing 55.) dla każdego typu `USART`, dla każdej dwuelementowej krotki spełniającej obostrzenie: typ pierwszego elementu krotki musi implementować cechę `PinTx<USART>`, typ drugiego elementu krotki musi implementować cechę `PinRx<USART>`.

```
//   V-------------------V----- dla każdego typu `USART`
impl<USART, TX, RX> Pins<USART> for (TX, RX) where
//          ^---^--------------------^---^---- dla każdego typu `TX` i~`RX`
    TX: PinTx<USART>, // ---\
    RX: PinRx<USART>, // ------- spełniającego obostrzenie
```

Listing 55: Implementacja cechy `Pins<USART>` (bez ciała)

- Zarówno typ obiektu `tx` (`PA7<Alternate<AF7>>`), jak i `rx` (`PA8<Alternate<AF7>>`) nie implementują ([44][45]; listing 56., 57.) odpowiednio cech `PinTx<USART1>` i `PinRx<USART1>`.

```
impl PinTx<UART4> for PA0<Alternate<AF8>>
impl PinTx<UART4> for PC10<Alternate<AF8>>
impl PinTx<UART4> for NoTx
impl PinTx<UART5> for PC12<Alternate<AF8>>
impl PinTx<UART5> for NoTx
impl PinTx<USART1> for PA9<Alternate<AF7>> // USART1
impl PinTx<USART1> for PB6<Alternate<AF7>> // USART1
impl PinTx<USART1> for NoTx                // USART1
impl PinTx<USART2> for PA2<Alternate<AF7>>
impl PinTx<USART2> for PD5<Alternate<AF7>>
impl PinTx<USART2> for NoTx
impl PinTx<USART3> for PB10<Alternate<AF7>>
impl PinTx<USART3> for PC10<Alternate<AF7>>
impl PinTx<USART3> for PD8<Alternate<AF7>>
impl PinTx<USART3> for NoTx
impl PinTx<USART6> for PC6<Alternate<AF8>>
impl PinTx<USART6> for PG14<Alternate<AF8>>
impl PinTx<USART6> for NoTx
```

Listing 56: Implementacje cechy `PinTx<USART>` (bez ciała)

```
impl PinRx<UART4> for PA1<Alternate<AF8>>
impl PinRx<UART4> for PC11<Alternate<AF8>>
impl PinRx<UART4> for NoRx
impl PinRx<UART5> for PD2<Alternate<AF8>>
impl PinRx<UART5> for NoRx
impl PinRx<USART1> for PA10<Alternate<AF7>> // USART1
impl PinRx<USART1> for PB7<Alternate<AF7>>  // USART1
impl PinRx<USART1> for NoRx                 // USART1
impl PinRx<USART2> for PA3<Alternate<AF7>>
impl PinRx<USART2> for PD6<Alternate<AF7>>
impl PinRx<USART2> for NoRx
impl PinRx<USART3> for PB11<Alternate<AF7>>
impl PinRx<USART3> for PC11<Alternate<AF7>>
impl PinRx<USART3> for PD9<Alternate<AF7>>
impl PinRx<USART3> for NoRx
impl PinRx<USART6> for PC7<Alternate<AF8>>
impl PinRx<USART6> for PG9<Alternate<AF8>>
impl PinRx<USART6> for NoRx
```

Listing 57: Implementacje cechy `PinRx<USART>` (bez ciała)

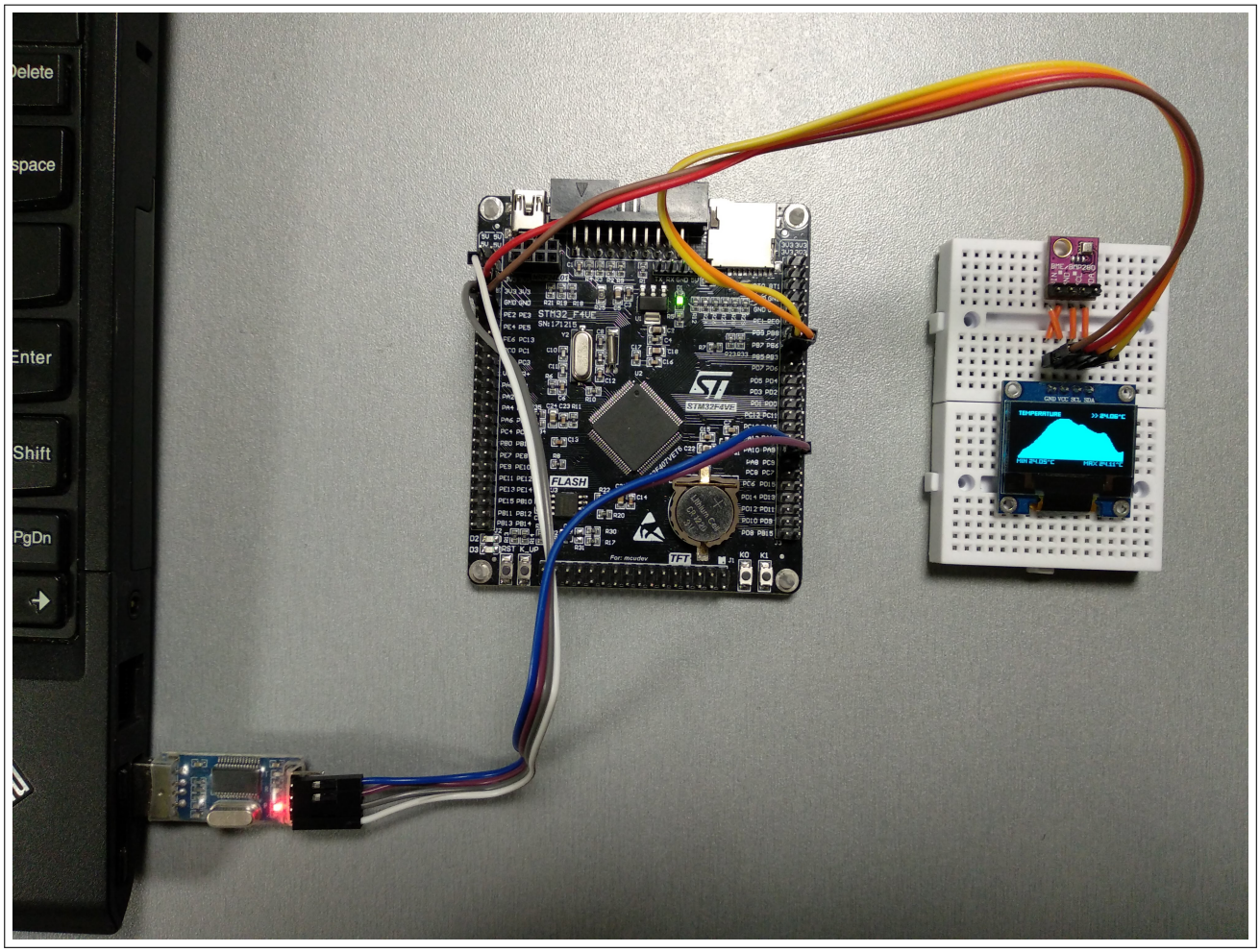
Listingi 56. oraz 57. pokazują, że dzięki systemowi typów jesteśmy w stanie wyrazić zależności sprzętowe; np. pin odbiorczy dla peryferium `USART1` może znajdować się tylko na pinach `PA10` i `PB7` w trybie *Alternate Function* numer 7 itd.

Podsumowując, przemyślana hierarchia typów odpowiadająca zachowaniu peryferium umożliwia wygenerowanie weryfikowalnego w czasie kompilacji interfejsu programistycznego co przy rosnącej złożoności zarówno sprzętu, jak i aplikacji go obsługujących wydaje się wartością trudną do przecenienia.

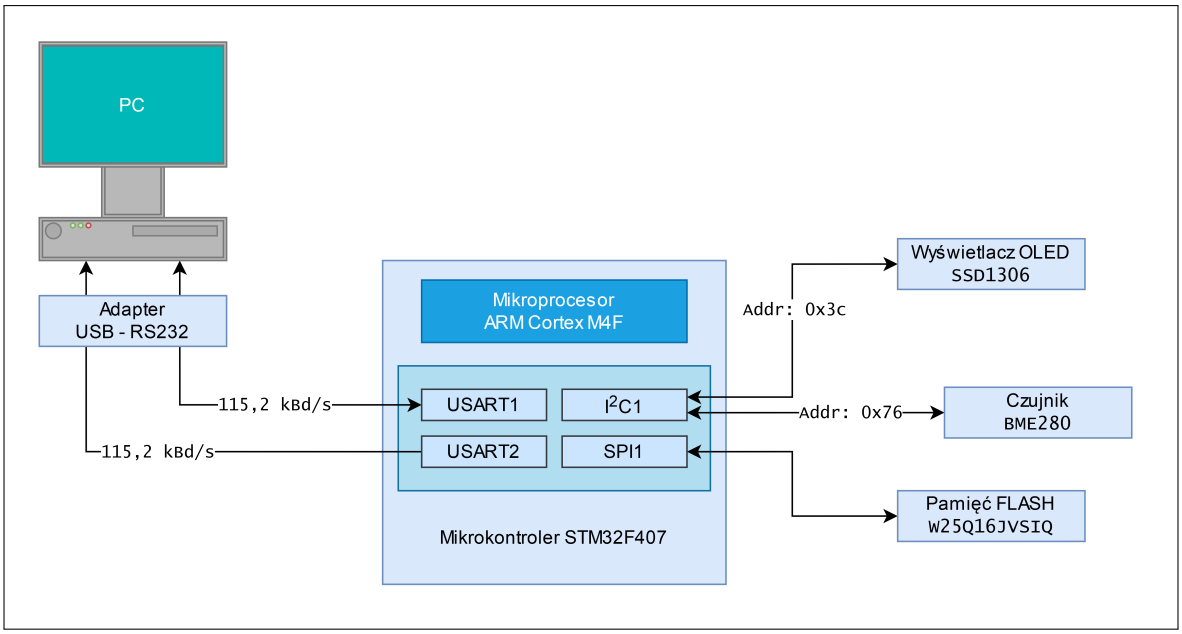
4 Przykładowy projekt – stacja pogodowa

Wcześniejsze rozdziały skupiały się na walorach języka i ekosystemu Rustowego z teoretycznego punktu widzenia. Aby w praktyce pokazać jego przydatność w oprogramowywaniu systemów wbudowanych, z pomocą mikrokontrolera *STM32F407* została zbudowana stacja pogodowa. Mierzy ona zmiany temperatury, ciśnienia i wilgotności w czasie.

Zdjęcie zmontowanego układu w czasie pracy przedstawia rysunek 26., natomiast jego elektryczny schemat blokowy zamieszczono na rysunku 27. Sercem stacji jest, komunikujący się z mikrokontrolerem poprzez interfejs I^2C, czujnik BME280 realizujący wszystkie wymienione pomiary. Towarzyszący mu wyświetlacz współdzieli z nim magistralę I^2C i prezentuje na bieżąco wyniki. Natomiast, korzystająca z interfejsu SPI pamięć FLASH wykorzystywana jest do przechowywania konfiguracji stacji. Oprócz tego aplikacja komunikuje się także z komputerem korzystając z interfejsu UART (z wykorzystaniem konwertera do USB). Tymi kanałami realizowana jest kontrola urządzenia oraz raportowanie stanu aplikacji (tzw. logi). Na etapie prac programistycznych wykorzystywany był także programator (z debuggerem) *ST-LINK/V2*.

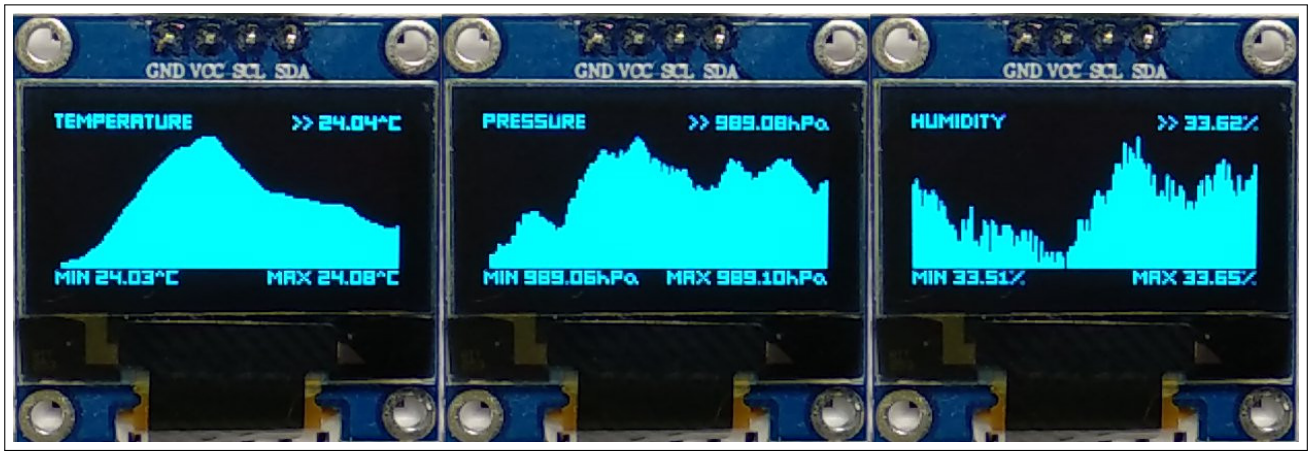


Rysunek 26: Stacja pogodowa podczas działania



Rysunek 27: Schemat blokowy stacji na poziomie peryferiów

Wynik pomiaru jest wyświetlany na monochromatycznym ekranie OLED (patrz rysunek 28.) w postaci wartości odpowiednio ostatniego (prawy górny róg ekranu), maksymalnego (prawy dolny róg ekranu) i minimalnego pomiaru (lewy dolny róg ekranu) w obrębie przesuwającego się okna czasowego; wszystkie dostępne w aplikacji widoki: temperatura, ciśnienie i wilgotność zostały pokazane na wspomnianym rysunku. Rezultatom liczbowym towarzyszy wykres, gdzie najbardziej aktualny wynik pomiaru znajduje się po prawej stronie. W zależności od ustawienia stacji widoki mogą się cyklicznie zmieniać lub pozostawać statyczne.

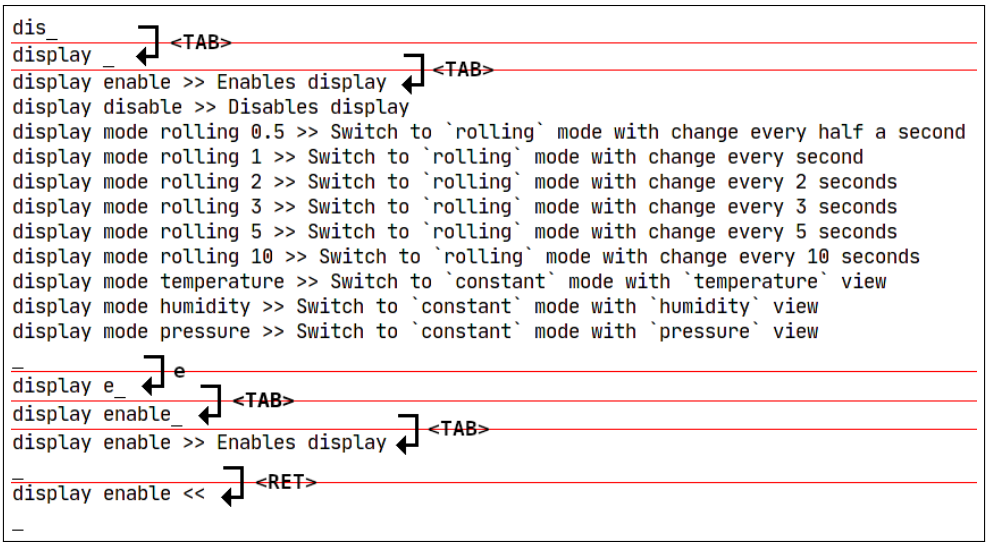


Rysunek 28: Dostępne widoki stacji pogodowej

Działanie aplikacji jest konfigurowalne poprzez przypominający powłokę systemową wiersz poleceń zaimplementowany na bazie interfejsu UART; przewiduje on pomoc i podpowiadanie składni. Pozwala on na: ustawienie w trybie zmiennym lub stałym wyświetlania obsługiwanych parametrów, wyłączenie ekranu czy wykonywania pomiarów w trybie ciągłym, a także

zapisanie bieżącej konfiguracji do pamięci FLASH. Podobnie jak w przypadku popularnych powłok, użytkownik za pomocą klawisza tabulacji może zażądać próby uzupełnienia wpisanej frazy, a klawiszem Enter próby wykonania reprezentowanej przez nią komendy; oprócz tego bufor wiersza poleceń można wyczyścić za pomocą kombinacji Ctrl + C. Między kolejnymi wciśnięciami klawisza cały ekran jest samoczynnie odświeżany.

Na rysunku 29. widoczny jest przykładowy scenariusz użycia. Czerwone linie rozdzielają momenty, między którymi ekran został w całości przerysowany, natomiast zawinięte strzałki oznaczają wciśnięcie odpowiednio klawisza tabulacji (<TAB>), 'e' lub Enter (<RET>). Wciśnięcie klawisza tabulacji w sytuacji, gdy próba autouzupełnienia daje niejednoznaczne wyniki (tak jak po drugim wciśnięciu <TAB> na rysunku 29.) powoduje wyświetlenie pomocy w postaci wylistowanych dostępnych komend zaczynających się od już wpisanej frazy wraz z ich opisem.



Rysunek 29: Przykładowy scenariusz użycia wiersza poleceń

Został również przewidziany mechanizm raportowania wypisujący wykonywane przez mikrokontroler zadania; korzysta on z osobnego interfejsu UART. Rysunek 30. pokazuje przykładowy raport z działania aplikacji; warto zauważyć, że do serializacji obiektów reprezentujących komendy zostały wykorzystane implementacje opisanej w uprzednich rozdziałach cechy `Debug`.

```
flash_handler:    PopulateConfigs
sensor_handler:   ConfigUpdate(Overwrite(SensorConfig { enabled: true }))
display_handler:  ConfigUpdate(Overwrite(DisplayConfig { enabled: true, view: Humidity, mode: Rolling(ThreeSeconds) }))
display_handler_scheduler
display_handler:  Redraw
sensor_handler_scheduler
sensor_handler:   MeasurementRun
display_handler:  Redraw
sensor_handler_scheduler
sensor_handler:   MeasurementRun
display_handler:  Redraw
sensor_handler_scheduler
sensor_handler:   MeasurementRun
display_handler:  Redraw
display_handler_scheduler
display_handler:  Redraw
sensor_handler_scheduler
sensor_handler:   MeasurementRun
display_handler:  Redraw
```

Rysunek 30: Raport z działania aplikacji po resecie urządzenia

Co istotne, projekt aplikacji jest modularny i pozwala na uruchamianie poszczególnych funkcjonalności na zasadzie podsystemów; w skrajnym przypadku wyłączone mogą być prawie wszystkie i urządzenie nasłuchuje jedynie komunikatów wiersza poleceń. W takiej sytuacji stację pogodową można przywrócić do życia, korzystając z wiersza poleceń i zmieniając ustawienia aplikacji. Warto zwrócić uwagę, że sama zmiana ustawień nie sprawia, że jest ona permanentna. Aby utrzymać zachowanie stacji po resecie mikrokontrolera, należy w osobnym kroku wykonać polecenie zapisania konfiguracji do pamięci FLASH. W przypadku gdy nie ma żadnych ustawień w pamięci, które aplikacja byłaby w stanie odczytać, używane są parametry domyślne.

4.1 Wykorzystane oprogramowanie

Podczas tworzenia projektu autor wykorzystał zintegrowane środowisko programistyczne *IntelliJ IDEA* [46] firmy *JetBrains* z wtyczką *IntelliJ Rust* [47]. Sprawowało się ono najlepiej ze wszystkich dostępnych w czasie tworzenia projektu środowisk programistycznych; zarówno pod względem jakości podpowiadanej składni, inteligentnej refaktoryzacji, jak i możliwości skakania po kodzie źródłowym. Należy zaznaczyć jednak, że rozwijany obecnie przez społeczność *Rust Analyzer* [48], stanowiący samodzielny, integrowalny z różnymi edytorami tekstu serwer językowy, może w najbliższym czasie wyprzedzić pod względem jakości i funkcjonalności *IntelliJ Rust*. Wsparcie debuggera GDB [49] dla Rusta również stoi na wysokim poziomie nie mniej jednak warto upewnić się, czy korzystamy z jego najnowszej wersji; te zawierają liczne łatki poprawiające doświadczenie użytkowania w Rustcie (np. bardziej estetyczne wyświetlanie złożonych typów). Do obsługi sprzętowego debuggera został wykorzystany program *OpenOCD* [50].

Jeśli chodzi o wykorzystane w projekcie biblioteki, na szczególną uwagę zasługują sterowniki czujnika oraz ekranu implementujące, zgodnie z duchem opisanym w rozdziale 3.1, *crate'y* `embedded-hal` oraz `embedded-graphics` pozwalając tym samym na ich wszechstronne użycie w obrębie różnych mikrokontrolerów. Oprócz tego kluczową rolę w tworzeniu aplikacji odegrał sterujący funkcjonowaniem całego programu system czasu rzeczywistego RTIC.

RTIC (ang. *Real-Time Interrupt-driven Concurrency*) jest systemem czasu rzeczywistego umożliwiającym tworzenie asynchronicznych, statycznie zdefiniowanych, kolejkowalnych zadań (ang. *tasks*), które mogą być uruchamiane bezpośrednio jako część logiki aplikacji (w trybie natychmiastowym lub planowanym z wyprzedzeniem wybranej liczby cykli zegara) albo za pomocą uprzednio skonfigurowanych przerwań sprzętowych. Zadania mogą mieć przydzielone statyczne priorytety, pozwalając na wzajemne wywłaszczanie. System jest zaprojektowany tak, że uniemożliwia powstanie wyścigu danych (ang. *data race*) oraz zakleszczeń; próba wywołania takowych nie daje się w RTICu wyrazić. Osiągnięcie takiego stanu rzeczy było możliwe dzięki charakterstyce Rusta, w którym jest on notabene napisany; w szczególności mowa tu o systemie makr oraz weryfikowanej w czasie kompilacji poprawności programu. Ponadto nie wymaga on dynamicznej alokacji pamięci, a koszta obliczeniowe z tytułu wykorzystania systemu są minimalne [51][52].

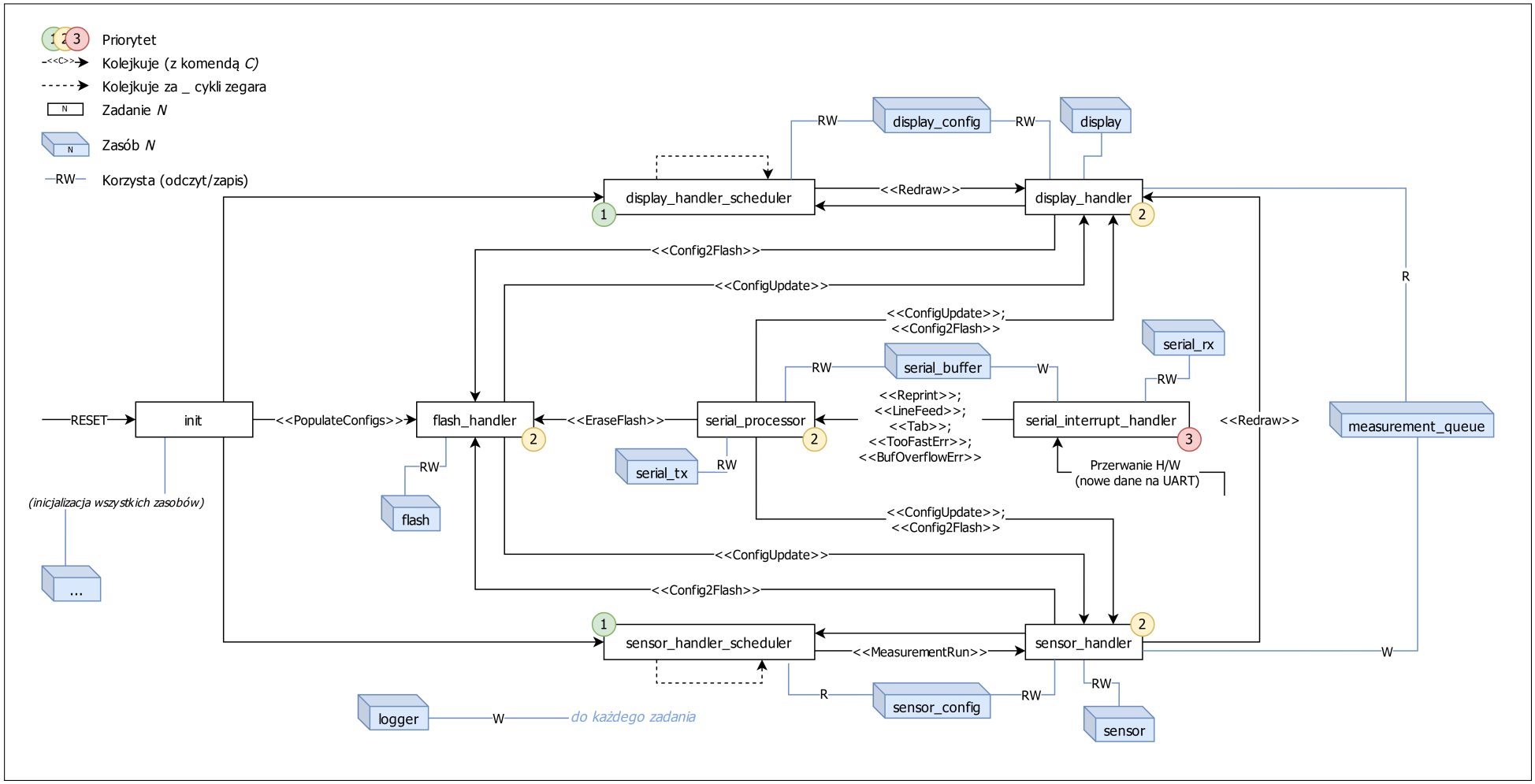
4.2 Budowa aplikacji

W przypadku omawianego projektu RTIC pozwolił na podzielenie funkcjonalności aplikacji na osobne, kolejkowalne zadania. Na rysunku 31. znajduje się schemat blokowy przedstawiający relacje między zaimplementowanymi typami zadań i wykorzystywanymi przez nie zasobami; co ważne, diagram nie pokazuje konkretnego scenariusza, a jedynie jakie zadania mogą wywołać inne i z jaką parametryzacją. Podczas kolejkowania zadania możliwe jest przekazanie towarzyszącej mu wiadomości (ang. *message*), która w projekcie została wykorzystana do wyspecyfikowania komendy i tym samym oczekiwanego od zadania zachowania; wiadomości oznaczone są na diagramie nawiasami `<<>>`.

Zadanie `init` jest zadaniem specjalnym, startowanym po resecie urządzenia, przewidzianym na inicjalizację peryferiów i innych zasobów (np. struktur danych i innych obiektów dzielonych między zadaniami). Poza nim wszystkie zadania posiadają przydzielony priorytet reprezentowany przez liczbę naturalną, gdzie większa wartość oznacza wyższy priorytet. Przykładowo, zadanie `serial_interrupt_handler`, odpowiadające za obsługę przerwania sprzętowego z tytułu nadchodzących wiadomości na UART, posiada najwyższy priorytet ze wszystkich dla zagwarantowania responsywności wiersza poleceń.

Na koniec, zadanie `init` kolejkuje zadanie mające na celu odczytanie z pamięci FLASH istniejącej konfiguracji, a następnie zadania odpowiedzialne za cykliczne wykonywanie pomiarów (`sensor_handler_scheduler`) oraz cykliczną zmianę widoków (temperatura, ciśnienie, wilgotność) na ekranie (`display_handler_scheduler`) zakładając, że urządzenie nie jest skonfigurowane inaczej; stacja pogodowa może być bowiem uruchomiona jedynie częściowo. Przykładowo, ciągłe wykonywanie pomiarów może być wyłączone, a widok na ekranie cyklicznie się zmienia lub odwrotnie; pomiary są wykonywane, lecz ekran, na którym mogłyby one zostać wyświetlone, jest wyłączony.

Wspomniane wcześniej zadanie `serial_interrupt_handler`, ale także kolejkowany przez niego `serial_processor` (odpowiedzialny za przerysowywanie wiersza poleceń, reagowanie na klawisze specjalne, autouzupełnianie i wykonywanie właściwych komend użytkownika) jako jedyne nie są częścią regularnego przepływu kontroli programu i są kolejkowane tylko i wyłącznie w sytuacji, gdy użytkownik wprowadza dane na interfejsie UART; uwidacznia to brak strzałek wyrażających kolejkowanie pochodzących od pozostałych zdefiniowanych zadań. W wypadku komendy zmiany trybu działania ekranu bądź czujnika `serial_processor` kolejkuje właściwe zadanie wraz z komendą rekonfiguracji (`<<ConfigUpdate>>`) odpowiednio rzeczonego ekranu (`display_handler`) lub czujnika (`sensor_handler`). Ostatni z unikalnych scenariuszy przewiduje żądanie zapisania do pamięci nieulotnej bieżących ustawień stacji przez użytkownika. W takim wypadku `serial_processor` zakolejkuje zadania `display_handler` i `sensor_handler` z żądaniem zapisu na FLASH (`<<Config2Flash>>`), które następnie zakolejkują zadanie właściwego zapisu (`flash_handler`; `<<Config2Flash>>`), przekazując wraz z komendą instancję struktury zawierającej konfigurację.



Rysunek 31: Schemat blokowy zadań i używanych przez nie zasobów na poziomie systemu czasu rzeczywistego

4.3 Wnioski

Doświadczenie i wrażenia autora płynące z tworzenia aplikacji są zdecydowanie pozytywne i składa się na nie kilka elementów.

Po pierwsze, oszczędność czasu wynikająca z charakterystyki omawianego w pracy języka, który uniemożliwia wprowadzanie całej klasy powszechnych w języku C błędów. Pozwoliło to wykorzystać zaoszczędzony czas na rozwijanie właściwej logiki biznesowej tworzonej aplikacji. Na tempo pracy zauważalny wpływ miała również dostępność użytecznych konstruktów językowych takich jak np. iteratory, które zostały wykorzystane m.in. do budowy parsera komend; wycinek kodu źródłowego owego parsera znajduje się na listingu 58[25]. Iteratory pozwoliły na sprawne wyrażanie intencji względem przetwarzanych kolekcji; w tym przypadku kolekcji dostępnych komend.

```
6  pub type ArgName = &'static str;
7  pub type ArgDesc = &'static str;
8
9  pub struct Parser<'a, TCommand: Clone> {
10     config: &'a [(ArgName, ArgDesc, TCommand)],
11 }
12
13 impl<'a, TCommand: Clone> Parser<'a, TCommand> {
14     pub fn new(config: &'a [(ArgName, ArgDesc, TCommand)]) -> Self {
15         Self { config }
16     }
17
18     pub fn get_command(&self, args: &str) -> Option<TCommand> {
19         self.config
20             .iter()
21             .filter(|(arg_name, _, _)| arg_name.eq(&args))
22             .map(|(_, _, command)| command)
23             .last()
24             .map(|v| v.clone())
25     }
```

Listing 58: Rust – fragment kodu źródłowego parsera poleceń tekstowych

Po drugie, łatwa i szybka weryfikacja tworzonych rozwiązań. Zintegrowany w Cargo mechanizm do obsługi testów jednostkowych został wykorzystany do sprawdzenia poprawności logiki w parserze poleceń tekstowych. Warto wspomnieć, że system makr bardzo łatwo pozwala na sprawną definicję licznych scenariuszy testowych, co pokazuje listing 59[26].

[25] Listing zawiera konstrukcje nieomówione w poprzednich rozdziałach, takie jak funkcje anonimowe (w Rustcie noszą miano *closures*) oraz aliasy typów (słowo kluczowe `type`). Więcej w źródłach [23, rozdział 13][17, rozdział 14, s. 170].

[26] Rozdział 1.6.2 traktuje o makrach i podaje źródła rozszerzające temat; m.in w kwestii sposobu definicji makr i wyjaśnienia składni.

```
    macro_rules! test_get_autocompletion {
        ($($test_name:ident:
           $passed_args:expr,
           $expected_completion:expr,
           $expected_result:expr,)*) => {
        $(
            #[test]
            fn $test_name() {
                let mut buffer = ArrayString::<[u8; 384]>::new();
                let parser = Parser::new(&TEST_CONFIG);
                let result = parser.get_autocompletion($passed_args, &mut buffer);

                assert_eq!(buffer.as_str(), $expected_completion);
                assert_eq!(result, $expected_result);
            }
        )*
        }
    }

    test_get_autocompletion! {
        test_nothing: "", "", NoChange,
        test_d: "d", "dddd", Completed,
        test_dddd: "dddd", "", NoChange,
        test_a: "a", "aabb", Completed,
        test_aa: "aa", "aabb", Completed,
        test_aab: "aab", "aabb", Completed,
        test_aabb: "aabb", "", NoChange,
        test_aabbc: "aabbc", "aabbcc ddd", Completed,
        test_aabbcc_space: "aabbcc ", "aabbcc ddd", Completed,
        test_aabbcc_ddd: "aabbcc ddd", "", NoChange,
        test_aabbcc_dddf: "aabbcc dddf", "aabbcc dddff", Completed,
        test_aabbcc_dddhh: "aabbcc dddhh", "", NoChange,
        test_wrong_1: "b", "", NoMatch,
        test_wrong_2: "aaa", "", NoMatch,
    }
```

Listing 59: Rust – fragment testów jednostkowych parsera poleceń tekstowych

Co więcej, dzięki temu, że rzeczony parser nie posiada żadnych silnych zależności sprzętowych, autor umieścił go w osobnym *crate'cie*. Umożliwiło to napisanie korzystającego z niego programu testowego działającego na maszynie gospodarza i tym samym jego interaktywne testowanie bez konieczności dotykania mikrokontrolera. Podkreśla to łatwość, z jaką przychodzi tworzenie wieloplaformowego kodu w Rustcie.

Po trzecie, praktyczny brak całego szeregu błędów wynikających z niewłaściwej interakcji ze sprzętem. Przyczynił się do tego wprowadzany przez *Embedded Rust* paradygmat zarządzania peryferiami w duchu maszyny stanów. Stanowi on, zdaniem autora, jeden z największych atutów pracy z Rustem w środowisku wbudowanym.

Po czwarte, poziom zaangażowania społeczności języka Rust (a także *Embedded Rust* oraz RTIC) zarówno w kwestii oferowanej pomocy nowicjuszom poprzez różne kanały komunikacyjne, jak i w tempie naprawiania znalezionych błędów. W trakcie prac nad projektem autor manuskryptu znalazł błąd w implementacji wysokopoziomowego HALa I^2C w *crate'cie* `stm32f4xx-hal`. Polegał on na braku ustawiania bitu stopu po zakończeniu transmisji; co interesujące, błąd uszedł uwadze programistów, ponieważ w przypadku znacznej większości testowanego sprzętu brak rzeczonego bitu nie powodował żadnych problemów w komunikacji. Tym samym ekran korzystający z I^2C działał bez zarzutu, natomiast czujnik już nie. Po zgłoszeniu i konsultacji problemu z deweloperami opiekującymi się projektem korektę błędu wraz z wydaniem nowej wersji biblioteki [53] udało się zrealizować w ciągu godziny, co w porównaniu z innymi projektami stanowi imponujący rezultat.

Ostatnim i najważniejszym wg autora elementem stojącym za sukcesem tworzonej aplikacji jest specyfika wykorzystanego w projekcie systemu czasu rzeczywistego RTIC. Pozwolił on na jednocześnie szybkie i poprawne zaimplementowanie aplikacji korzystającej z przerwań sprzętowych o zróżnicowanych priorytetach, co bez niego byłoby praktycznie niemożliwe. Przykładowo, zasób dzielony między zadania o różnym priorytecie w zadaniu o priorytecie niższym jest zagnieżdżony w dodatkową strukturę synchronizującą wymagającą założenia blokady (ang. *lock*) podczas próby użycia. Dzięki temu nie ma możliwości naruszenia spójności pamięciowej takiego obiektu, gdyby doszło do wywłaszczenia w niefortunnym momencie przez zadanie o priorytecie wyższym. Narzucane przez RTIC reguły pozwalają na uniknięcie łatwych do popełnienia, a tym samym bardzo trudnych do analizy i reprodukcji błędów specyficznych dla systemów wbudowanych.

Największym problemem podczas pracy nad projektem było wspomniane na stronie 51. poprawne zaimplementowanie dzielenia dostępu do interfejsów zewnętrznych, które z natury są współdzielone (I^2C, SPI). Niefortunne podejście twórców sterowników (lub `embedded-hala` skoro na to pozwala; w zależności od punktu widzenia) zakłada przejęcie własności nad obiektami reprezentującymi rzeczone interfejsy. Nie stanowi to kłopotu w przypadku gdy mamy do czynienia z pojedynczym sterownikiem, ale zazwyczaj będzie ich kilka. Podczas pracy udało się ostatecznie znaleźć działające rozwiązanie, nie mniej nie jest ono idealne; przewiduje wywołanie makra `panic!` w sytuacji, gdy podczas używania obiektu interfejsu dojdzie do wywłaszczenia przez inne zadanie, które również z takowego korzysta. W projekcie stacji pogodowej nigdy do takiej sytuacji nie dojdzie, ponieważ oba korzystające z interfejsu I^2C zadania (`display_handler` oraz `sensor_handler`) posiadają ten sam priorytet i nie mogą się wzajemnie wywłaszczyć.

Podsumowanie

Branża systemów wbudowanych, szczególnie gdyby porównywać ją do innych segmentów branży IT, w zasadzie od wielu lat tkwi w stagnacji; co więcej, w odniesieniu do stosowanych języków programowania nie dochodzi praktycznie do żadnych zmian. Powodów można się doszukiwać m.in. w specyfice systemów wbudowanych czy też programowania systemowego w ogólności; język musi ściśle oddawać charakterystykę sprzętu, który oprogramowywuje. Niszę tą zapełnia język C oraz częściowo C++. Mimo rzeczonej stagnacji oczekiwania deweloperów wzrosły także wobec segmentu systemów wbudowanych i tym samym jego obecny inwentarz staje się dla nich niewystarczający.

Rust zdaje się idealnie trafiać w potrzebę rynku, wnosząc wiele niezwykłych, wartych uwagi nowinek, wciąż pozostając systemowym językiem programowania. Obie te charakterystyki są konieczne; z jednej strony język musi na tyle zainteresować programistów, by chcieli poświęcić swój czas na zgłębianie jego tajników, a z drugiej przepływ kontroli programu musi być możliwie spójny z zachowaniem obsługiwanego sprzętu. Rust dostarcza wiele użytecznych rozwiązań, takich jak typ sumy, dopasowanie do wzorca, mechanizm programowania uogólnionego czy choćby kompletny ekosystem narzędzi. Spośród wszystkich jego funkcjonalności, to jednak właśnie mechanizm weryfikacji poprawności kodu źródłowego w czasie kompilacji może w długofalowej perspektywie zaważyć o przyjęciu się Rusta i, być może, wyparciu przez niego języka C jako pierwszoplanowego gracza w tej branży. Stoi za tym przede wszystkim wyjątkowość i nowatorskość tej techniki; o ile wiele z mechanizmów dostarczanych przez Rusta jest relatywnie dostępnych (chociażby w C++), o tyle ten konkretny element stanowi ewenement. Z perspektywy biznesowej, może to potencjalnie wpłynąć na obniżenie wymaganej od programistów ekspertyzy w przypadku tworzenia komercyjnych systemów wbudowanych z gotowych, powszechnych komponentów, redukując tym samym koszt robocizny.

Należy pamiętać, że Rust, jak i *Embedded Rust* będą podlegać dalszemu rozwojowi. Stąd należy się spodziewać kolejnych innowacji i usprawnień; zarówno w zakresie języka, udostępnianych abstrakcji, narzędzi czy bibliotek, jak i stopniowego rozszerzania wsparcia dla nowych mikroarchitektur. Wykonany w ramach pracy przykładowy projekt udowadnia, że Rust już obecnie ma ogromny potencjał. Także w zastosowaniach gdzie krytyczne jest zapewnienie gwarancji dotyczących zależności czasowych dzięki istniejącym rozwiązaniom w zakresie systemów czasu rzeczywistego. Opis teoretyczny zamieszczony w manuskrypcie miał, w zamyśle autora, nakreślić czytelnikowi ogólny obraz języka i jego możliwości zachęcając do dalszej eksploracji tej technologii na własną rękę. I tu pozostaje wyrazić nadzieję, że ten dodatkowy, nieoficjalny cel udało się autorowi osiągnąć.

Bibliografia

[1] Microsoft Security Response Center. We need a safer systems programming language. https://web.archive.org/web/20200813104532/https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/.

[2] Microsoft Security Response Center. A proactive approach to more secure code. https://web.archive.org/web/20200930031715/https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/.

[3] Michael Barr. *Programming Embedded Systems in C and C++*. O'Reilly Media, 1 edition, 1999.

[4] Steve Heath. *Embedded Systems Design*. Newnes, 2 edition, 2003.

[5] ARM Keil. MDK Microcontroller Development Kit. https://web.archive.org/web/20200922193732/https://www2.keil.com/mdk5.

[6] ARM Developer. Pre-built GNU ARM Embedded toolchain for Cortex R/M family. https://web.archive.org/web/20201011003319/https://developer.arm.com/tools-and-software/open-source-software/developer-tools/gnu-toolchain/gnu-rm.

[7] Rust project. Początki języka. https://web.archive.org/web/20160609195720/https://www.rust-lang.org/faq.html#project.

[8] Rust project. Rust 1.0 Release. https://web.archive.org/web/20200915093943/https://blog.rust-lang.org/2015/05/15/Rust-1.0.html.

[9] Bjarne Stroustrup. A History of C++: 1979-1991. https://web.archive.org/web/20201017020741/http://www.stroustrup.com/hopl2.pdf.

[10] International Organization for Standardization. ISO/IEC 14882:1998. https://web.archive.org/web/20200911144554/https://www.iso.org/standard/25845.html.

[11] Dennis M. Ritchie. The Development of the C Language. https://web.archive.org/web/20201018044629/http://www.bell-labs.com/usr/dmr/www/chist.html.

[12] Stack Overflow. 2020 Developer Survey.
https://web.archive.org/web/20201012120939/https://insights.stackoverflow.com/survey/2020#technology-most-loved-dreaded-and-wanted-languages.

[13] Rust Project. Production users.
https://web.archive.org/web/20201021224842/https://www.rust-lang.org/production/users.

[14] LWN.net. Supporting Linux kernel development in Rust.
https://web.archive.org/web/20201013022251/https://lwn.net/Articles/829858/.

[15] The rustc book. Platform support.
https://web.archive.org/web/20200911151525/https://doc.rust-lang.org/nightly/rustc/platform-support.html.

[16] Rust project. Rust - inspiracje.
https://web.archive.org/web/20201021172836/https://doc.rust-lang.org/reference/influences.html.

[17] Jim Blandy, Jason Orendorff. *Programming Rust: fast, safe systems development*. O'Reilly Media, 1 edition, 2017.

[18] cppreference.com. Move constructors.
https://web.archive.org/web/20201021030838/https://en.cppreference.com/w/cpp/language/move_constructor.

[19] cppreference.com. std::vector.
https://web.archive.org/web/20201023234946/https://en.cppreference.com/w/cpp/container/vector/vector.

[20] Wikipedia. Expression-oriented programming language.
https://web.archive.org/web/20201016203211/https://en.wikipedia.org/wiki/Expression-oriented_programming_language.

[21] cppreference.com. Trivial destructor.
https://web.archive.org/web/20201020200842/https://en.cppreference.com/w/cpp/language/destructor.

[22] boost.org. boost::variant.
https://web.archive.org/web/20200812201859/https://www.boost.org/doc/libs/1_73_0/doc/html/variant.html.

[23] Steve Klabnik, Carol Nichols. *The Rust Programming Language*. No Starch Press, 1 edition, 2019.

[24] David Blaser. Simple Explanation of Complex Lifetime Errors in Rust, 2019.

[25] Rust Project. Strona domowa projektu Rust. https://web.archive.org/web/20201022202210/https://www.rust-lang.org/.

[26] GNU Autoconf manual. Specifying target triplets. https://web.archive.org/web/20200731202041/https://www.gnu.org/software/autoconf/manual/autoconf-2.65/html_node/Specifying-Target-Triplets.html.

[27] The Cargo book. The Cargo book. https://web.archive.org/web/20200914194038/https://doc.rust-lang.org/cargo/.

[28] GNU. Strona domowa projektu GNU Autoconf. https://web.archive.org/web/20201019201741/https://www.gnu.org/software/autoconf/.

[29] GNU. Strona domowa projektu GNU Make. https://web.archive.org/web/20201026024312/https://www.gnu.org/software/make/.

[30] CMake. Strona domowa projektu CMake. https://web.archive.org/web/20201026041140/https://cmake.org/.

[31] Bazel. Strona domowa projektu Bazel. https://web.archive.org/web/20201023085216/https://bazel.build/.

[32] Meson. Strona domowa projektu Meson. https://web.archive.org/web/20201021183956/https://mesonbuild.com/.

[33] Conan. Strona domowa projektu Conan. https://web.archive.org/web/20201012054642/https://conan.io/.

[34] ARM Keil. Strona domowa projektu Keil μVision IDE. https://web.archive.org/web/20201027000004/http://www2.keil.com/mdk5/uvision/.

[35] The Rust Embedded Book. Memory Mapped Registers. https://web.archive.org/web/20200912070225/https://rust-embedded.github.io/book/start/registers.html.

[36] Github. Repozytorium projektu embedded-hal.
https://web.archive.org/web/20201027000140/https://github.com/rust-embedded/embedded-hal.

[37] Github. Repozytorium projektu embedded-time.
https://web.archive.org/web/20201027000150/https://github.com/FluenTech/embedded-time.

[38] Github. Repozytorium projektu embedded-dma.
https://web.archive.org/web/20201027000212/https://github.com/rust-embedded/embedded-dma.

[39] Github. Repozytorium projektu embedded-nal.
https://web.archive.org/web/20201027000223/https://github.com/rust-embedded-community/embedded-nal.

[40] Github. Lista użytecznych zasobów *Rust Embedded*.
https://web.archive.org/web/20201027000320/https://github.com/rust-embedded/awesome-embedded-rust.

[41] The Rust Embedded Book. Peripherals as State Machines.
https://web.archive.org/web/20200912070224/https://rust-embedded.github.io/book/static-guarantees/state-machines.html.

[42] Docs.rs. Serial::usart1.
https://web.archive.org/web/20200920141502/https://docs.rs/stm32f4xx-hal/0.8.3/stm32f4xx_hal/serial/struct.Serial.html#method.usart1.

[43] Docs.rs. Serial::Pins.
https://web.archive.org/web/20200920141510/https://docs.rs/stm32f4xx-hal/0.8.3/stm32f4xx_hal/serial/trait.Pins.html.

[44] Docs.rs. Serial::PinTx.
https://web.archive.org/web/20200920141517/https://docs.rs/stm32f4xx-hal/0.8.3/stm32f4xx_hal/serial/trait.PinTx.html.

[45] Docs.rs. Serial::PinRx.
https://web.archive.org/web/20200920141523/https://docs.rs/stm32f4xx-hal/0.8.3/stm32f4xx_hal/serial/trait.PinRx.html.

[46] JetBrains. IntelliJ IDEA.
http://web.archive.org/web/20201027135136/https://www.jetbrains.com/idea/.

[47] Społeczność IntelliJ Rust. Strona domowa projektu IntelliJ Rust. http://web.archive.org/web/20201024171907/https://intellij-rust.github.io/.

[48] Społeczność Rust Analyzer. Strona domowa projektu Rust Analyzer. http://web.archive.org/web/20201003141803/https://rust-analyzer.github.io/.

[49] GNU Project. GNU Debugger. http://web.archive.org/web/20201022153202/https://www.gnu.org/software/gdb/.

[50] Społeczność OpenOCD. Strona domowa projektu OpenOCD. http://web.archive.org/web/20201019141201/http://openocd.org/.

[51] Rtic.rs. Real-Time Interrupt-driven Concurrency. https://web.archive.org/web/20201027000420/https://rtic.rs/0.5/book/en/.

[52] Johan Eriksson, Fredrik Haggstrom, Simon Aittamaa, Andrey Kruglyak, and Per Lindgren. Real-Time For the Masses, Step 1: Programming API and Static Priority SRP Kernel Primitives. In *Proceedings of the 8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013*, pages 110–113, 06 2013.

[53] Github. Poprawka do błędu w implementacji HALa I2C. https://web.archive.org/web/20201027000526/https://github.com/stm32-rs/stm32f4xx-hal/pull/169.