Dokumentacja końcowa

Tymoteusz Malec, Jakub Szweda

Politechnika Warszawska, Cyberbezpieczeństwo 24L

5 Czerwca 2024

Contents

1. Podział pracy

W ramach projektu podzieliliśmy się pracą następująco
— GFM Blocks - Tymoteusz Malec
— GFM Inline - Jakub Szweda
— Parser AST → Latex - Tymoteusz Malec
— Parser AST → Typst - Jakub Szweda
— Aplikacja CLI - Wspólnie
— Dokumentacja - każdy swoją część

2. Napotkane problemy

2.1. Pierwsze spotkanie z językiem

Z mojej perspektywy (Jakub Szweda) Rust okazał się językiem dużo cięższym niż podejrzewałem, myślałem że nauka jego podstawowych konstrukcji nie okaże się zbyt dużym problemem jednak ostatecznie zajęło to dużo więcej czasu niż przypuszczałem. Jednakże uważam że sporo się w tym projekcie nauczyłem.

2.2. Parsowanie linków i obrazków

Ze względu na niedocenienie trudności implementacji poszczególnych inline'ów nie wystarczyło nam czasu na implementacje całości mechanizmu związanego z linkami. Na ten moment parsujemy tylko najprostsze linki - te wskazujące na link zdefiniowany w innym miejscu i nie posiadające poprzedzającego tekstu.

2.3. Raw HTML

Zgodnie z założeniami opisanymi w sprawozdaniu wstępnym założyliśmy, że część bloków, możemy odrzucić - tak zrobiliśmy z blokami i inline'ami w postaci Raw HTML. Odrzuciliśmy je ponieważ parsowanie ich jest bardzo skomplikowane, a nie mają odpowiadającej reprezentacji w formacie LaTeX. Narzędzie Pandoc podczas konwersji zupełnie ignoruje istnienie tych bloków - uznaliśmy, że nieparsowanie ich nie będzie dużą stratą.

2.4. Trait objects

Ostatnim napotkanym problemem było przechowywanie wszystkich Readerów i Writerów w jednym obiekcie. Chcieliśmy dać możliwość wyboru zwracanego błędu Writerom i Readerom. Utrudnia to jednak przechowywanie tych typów jako jeden typ. Gdy nie wiemy jaki konkretny typ zwraca metoda, nie możemy na nim pracować. Dlatego utworzyliśmy specjalne kontenery przechowujące funkcje, które "wrappują" Readery i Writery. Zwracany błąd, boksują aby typ miał znaną w czasie kompilacji wielkość. Generyczna metoda, która to robi będzie musiała zostać wytworzona dla każdego Writera i Readera, który do niej przekażemy. Nie jesteśmy pewni czy jest to słuszne rozwiązanie. Wiemy że dla dwóch formatów wejściowych i trzech wyjściowych użycie zwykłego matcha nie byłoby problemem jednakże aplikacja miała być łatwo rozszerzalna. Gdyby nasza aplikacja obsługiwała tyle formatów co Pandoc, taki match nie byłby już dobrym rozwiązaniem. Nie chcieliśmy również zabierać możliwości określenia zwracanego błędu, albo wymusić aby on był od razu zboksowany, gdyż nie miało by to sensu w przypadku użycia konkretnego typu.

3. Architektura aplikacji

3.1. AST

Zgodnie z dodatkowym wymaganiem, nasza aplikacja jest kompatybilna z formatem Pandoc AST. W module przepisaliśmy jeden do jednego typy z Haskela do Rusta. Dzięki wygodzie typów algebraicznych w Ruscie nie było to zupełnie problemem. By móc przekazywać dokument w postaci AST między Pandoc'iem a naszą aplikacją używamy biblioteki `serde_json` do serializacji i deserializacji JSONa. Takie użycie opisane jest w dalszej części sprawozdania oraz w pliku `README.md`.

3.2. Traity

W aplikacji korzystamy z dwóch traitów - `AstReader` i `AstWriter`. Każdy z nich określa typ zwracanego błędu oraz ma jedną metodę. `AstReader` bierze slice tekstu i konwertuje go na format Pandoc AST, a `AstWriter` konwertuje format Pandoc AST na Stringa.

3.3. TempBlock

Parsowanie Pandoca składa się z dwóch etapów - określenie struktury blokowej dokumentu, a następnie dokończenie każdego bloku parsując elemennty typu Inline. Do tego pierwszego etapu wykorzystany jest typ `TempBlock`, który reprezentuje nieukończony blok. Ponieważ każdy blok ma minimalnie inne zasady na to jak się kończy i w jaki sposób po nim zaczynają się nowe bloki korzystamy z dwóch typów. `CheckResult` przechowuje wynik sprawdzenia, czy linijka zaczyna nowy blok. Zależnie od obecnego bloku jest od konwertowany do `LineResult`, który następnie określa jak obecnie badana linijka otwierała i / lub kończyła bloki. Dzięki temu, drastycznie zostało zredukowane powtarzanie się kodu, a metody przyjmujące kolejną linijkę w każdym z wariantów stały się krótkie, czytelne i zrozumiałe.

3.4. InlineParser

W wyniku uprzedniego parsowania bloków metody wewnątrz inline_parser mają do przeanalizowania zawartość tylko jednego bloku. Rozpoczynamy od analizy inline'ów typu code span, ponieważ wewnątrz nich nie panują żadne dodatkowe zasady więc łatwiej najpierw wyizolować część paragrafu która nie wymaga dalszego przetwarzania żeby następnie skupić się na tych które tego wymagają. Pozostała zawartość bloku (dalej slice'y) zostają przetwarzane wewnątrz metody parse_inline_slice wewnątrz której iterujemy przez dany fragment bloku i w zależności od napotkanego znaku funkcyjnego oddajemy dalsze parsowanie metodom typu handler (np. handle_open_bracket_temp()). Warto też wspomnieć o trudności z parsowaniem kursywy oraz pogrubienia. Zgodnie z definicją opisaną wewnątrz dokumentacji te elementy mogą być zagnieżdżane. Ze względu na to musimy przechowywać wszystkie ciągi znaków umożliwiające powstanie tego typu inline'ów oraz informacji dotyczących możliwego otwierania lub zamykania tych elementów.Po ukończeniu iteracji przechodzimy do parsowania kursyw i pogrubień wewnątrz metody parse_emph() która to zgodnie z algorytmem podanym w dokumentacji analizuje dostępne na stosie informacje o ciągach znaków i odpowiednio zagnieżdża istniejące inline'y wewnątrz tych elementów.

3.5. Pozostałe writery i readery

By aplikacja była kompatybilna z aplikacją Pandoc utworzyliśmy Writera i Readera, który format Pandoc AST zamienia na lub z JSONa. Writery do formatów LaTeX oraz Typst nie są bardzo rozbudowane, ponieważ postawiliśmy, że zrobimy konwersję tylko tych bloków i inline'ów, które można osiągnąć w formacie gfm. Uprościło to drastycznie problem.

3.6. Iteratory

Napisaliśmy swoje własne iteratory na slice'ach tekstu, które umożliwiają łatwe podglądanie oraz bezpieczne wycinanie tekstu (na podstawie wartości zwróconych przez `CharIndices`). Napisane są w taki sposób, że nie alokują żadnej pamięci, tylko pracują na referencjach z określonym lifetimem. Dzięki borrow checkerowi praca na referencjach jest bezpieczna, nie mamy wątpliwości, że w jakimś przypadku osiągniemy dangling pointer

3.7. CLI

Do utworzenia nowocześnie wyglądającej aplikacji CLI, użyliśmy bardzo rozbudowanej, wysokopoziomowej biblioteki `clap`. Umożliwia ona na bardzo łatwe zdefiniowane wszystkich argumentów, reguł dotyczących możliwych wartości oraz sposobu ich przetwarzania oraz bardzo łatwe uzyskanie argumentów podanych przez użytkownika i obsługę błędów. Biblioteka sama generuje funkcję help oraz wiadomości wyświetlane w przypadku nie podania wszystkich wymaganych argumentów lub podania błędnych wartości.

4. Dokumentacja

Do dokumentacji użyliśmy narzędzia `cargo doc`. Generuje on świetnie sformatowaną dokumentację, w ustandaryzowanym formacie, która zawiera odnośniki do innych elementów. Poza dokumentacją publicznych modułów, typów i funkcji opisaliśmy również te prywatne by lepiej wyjaśnić działanie programu.

4.1. Testy

Możemy stwierdzić, że iteratory, TempBlock i MdReader są praktycznie w 100% przetestowane. Zostawiliśmy również testy, z których korzystaliśmy w trakcie pisania, które wymagają zainstalowania Pandoca i porównują wyniki parsowania względem niego. Te testy nie przechodzą w całości, część przez niezaimplementowanie wszystkich funkcji, część przez drobne różnice, które nie mają znaczenia i są też pojedyncze przypadki, w których Pandoc zwraca błędny wynik (szczególnie przy tabelach).

5. Podsumowanie

Liczba linijek: 5271
Liczba testów: 30 zwykłych + 17 względem Pandoca (zawierających 488 case'ów)
Liczba godzin: 120h (sumarycznie)