

Tytuł

Patryk Wałach

January 2022

Contents

1	Ogólne wprowadzenie	2
1.1	Analiza leksykalna	2
1.2	Parser	3
1.3	System typów	4
1.4	System typów ML	4
1.5	Rekord z wariantami	4
2	Inferencja typów w teorii	5
2.1	Problem inferencji typów	5
2.2	Udawanie typów dla ML-the-calculus	6
2.3	Inferencja typów bazująca na ograniczeniach	6
2.4	Algorytm W	6
2.5	Algorytm J	7
3	Kompilacja dopasowania do wzorca (ang. <i>pattern matching</i>)	8
3.1	Motywacja	8
3.2	Kompilacja na przykładzie	9
3.3	Generalizacja algorytmu	10
4	ReScript — język realizujący podobne zadania	11
5	Założenia i priorytety opracowanej aplikacji	12
5.1	Opis formalny składni języka	17
6	Narzędzia	19
6.1	Język Python	19
6.2	Parsowanie i tokenizowanie przy użyciu biblioteki sly	20
6.2.1	Lekser	20
6.2.2	Parser	20
6.3	Środowisko nodejs do uruchomienia skompilowanego kodu	21

7	Implementacja	21
7.1	Lekser	21
7.2	Parser	23
7.3	Drzewo decyzyjne	24
7.4	Inferencja typów	26
7.4.1	Reprezentacja środowiska	26
7.4.2	Aplikowanie substytucji	26
7.4.3	Literały	28
7.4.4	Identyfikatory	28
7.4.5	Bloki wyrażeń	28
7.4.6	Operatory binarne	28
7.4.7	Wywołania funkcji	29
7.4.8	Wyrażenia warunkowe	29
7.4.9	Definicje funkcji	30
7.5	Dopasowanie do wzorca	30
7.6	Windowanie (ang. <i>Hoisting</i>)	33
7.7	Kompilacja	34
7.8	Literały	34
7.8.1	Wyrażenia warunkowe	35
7.8.2	Warianty	35
7.8.3	Dopasowanie do wzorca	35
8	Opis działania	36
8.1	Co działa	36
8.2	Uwagi do działania	36
8.2.1	Obsługa błędów	36
8.2.2	Dopasowanie do wzorca	36
9	[Podsumowanie]	37
10	[spisy — rysunków, tabel, listingów itp.]	37

[Wstęp]

1 Ogólne wprowadzenie

1.1 Analiza leksykalna

Analiza leksykalna w informatyce jest to proces rozbijania sekwencji znaków (np. takich jak kod źródłowy) na jednostki logiczne (zwane leksemami) złożone z jednego lub więcej znaków, które łącznie mają jakieś znaczenie [4]. Przykładami leksemów mogą być słowa kluczowe (np. `while`), identyfikator lub liczba składająca się z cyfr. Rozdzielaniem programu źródłowego na leksemy zajmuje się lekser.

Token jest strukturą reprezentującą leksem i wprost go kategoryzującą [1], co ułatwia późniejszą pracę parserowi. Tokeny kategoryzuje się na komputerowy odpowiednik tego, co lingwiści określiliby mianem części mowy. Biorąc jako przykład poniższy kod w języku C:

```
1 x = a + b * 2;
```

analiza leksykalna, zwraca tokeny:

```
1 [(identifier, x), (operator, =), (identifier, a), (operator, +),  
  (identifier, b), (operator, *), (literal, 2), (separator, ;)]
```

Dwoma ważnymi przypadkami znaków są znaki białe i komentarze. One również muszą być uwzględnione w gramatyce i przeanalizowane przez lekser, lecz mogą być odrzucone (nie produkować żadnych tokenów). Spełniają one wtedy zadanie, rozdzielenia dwóch tokenów (np. w `if x` zamiast `ifx`).

1.2 Parser

Analizator składniowy, parser – program dokonujący analizy składniowej danych wejściowych w celu określenia ich struktury gramatycznej w związku z określoną gramatyką formalną. Analizator składniowy umożliwia przetworzenie tekstu czytelnego dla człowieka w strukturę danych przydatną dla oprogramowania komputera. Wynikiem analizy składni, dokonywanej przez parser, najczęściej jest drzewo składniowe nazywane czasami drzewem wyprowadzenia [1].

Zadanie parsera sprowadza się do sprawdzenia czy i jak dane wejściowe mogą zostać wyprowadzone z symbolu startowego. To zadanie można zrealizować na dwa sposoby:

- Analiza zstępująca (ang. *top-down parsing*) to strategia znajdowania powiązań między danymi przez stawianie hipotez dotyczących drzewa rozbioru składniowego i sprawdzanie, czy zależności między danymi są zgodne z tymi hipotezami.
- Analiza wstępująca (ang. *bottom-up parsing*) – ogólna metoda analizy składniowej, w której zaczyna się od słowa wejściowego i próbuje się zredukować je do symbolu startowego. Drzewo wyprowadzenia jest konstruowane od liści do korzenia (stąd nazwa). W każdym momencie w trakcie tego procesu mamy formę zdaniową, która zawiera segment, powstały w ostatnim kroku wyprowadzenia. Segment ten nazywany uchwytem (ang. *handle*) jest prawą stroną produkcji i powinien zostać w tym kroku zredukowany do jej lewej strony, w wyniku czego powstanie poprzednia forma zdaniowa z wyprowadzenia. Główna trudność w analizie wstępującej polega właśnie na odpowiednim znajdowaniu uchwytów. Analiza wstępująca może przebiegać w określonym kierunku (np. od lewej do

prawej), lub w sposób bezkierunkowy, wtedy analizowane jest całe słowo naraz. Jednym z bardziej znanych przedstawicieli metody bezkierunkowej jest algorytm CYK. Do metod kierunkowych zalicza się między innymi parsery shift-reduce czyli LR, LALR, SLR, BC, pierwszeństwa.

1.3 System typów

System typów jest to system klasyfikacji wyrażeń w zależności od rodzajów wartości, jakie one generują [7]. Każdej obliczonej wartości przypisywany jest pewien typ, który jednoznacznie definiuje, jakie operacje można na niej wykonać. Śledząc przepływ wartości, system typów stara się udowodnić, że w programie występuje poprawne typowanie, tzn. nie dochodzi do sytuacji, w której na wartości określonego typu próbujemy wykonać niedozwoloną operację.

1.4 System typów ML

System typów ML jest to silny system typów stosowany w językach rodziny ML (Ocaml, Standard ML) oparty na inferencji.

Podstawowy system typów jest następujący: istnieją typy proste, takie jak `string`, `int`, `bool`, `unit` (typ pusty) itd. Z dowolnych typów można też generować typy złożone — przez krotki (`typ1 * typ2`, `typ1 * typ2 * typ3` itd.), konstruktory typów (typ `list`, typ `tree` itd.) i funkcje (`typ1 -> typ2`).

System próbuje nadać typy każdemu wyrażeniu języka i nie licząc kilku rzadkich przypadków, udaje mu się to całkiem dobrze.

Generalnie system taki wyklucza polimorfizm, jednak w Standard ML stworzono specjalne reguły umożliwiające polimorfizm dla wyrażeń arytmetycznych.

System typów w języku Standard ML jest interesujący z teoretycznego punktu widzenia — wiele problemów ma bardzo wysoką złożoność, jednak w praktyce inferencja zachodzi bardzo szybko — typy, które są rzeczywiście używane, są zwykle bardzo proste — rzadko używa się funkcji rzędów wyższych niż trzeci-czwarty oraz liczby argumentów większej niż kilkanaście.

W rzeczywistych implementacjach dochodzą do tego bardziej złożone problemy typizacji obiektów, modułów itd.

1.5 Rekord z wariantami

Rekord z wariantami jest to rodzaj rekordu, posiadającego tę właściwość, że zbiór rekordów posiada wspólny typ, lecz różną postać, określoną aktualną wartością specjalnego pola znacznikowego.

Na przykład, zakładając, że chcemy stworzyć drzewo binarne typu `int`. W języku Standard ML zrobilibyśmy to tworząc nowy typ danych w ten sposób:

```
1 datatype tree = Leaf
2   | Node of (int * tree * tree)
```

`Leaf` i `Node` są konstruktorami, które pozwalają nam na stworzenie konkretnego drzewa, np.

```
1 Node(5, Node(1, Leaf, Leaf), Node(3, Leaf, Node(4, Leaf, Leaf)))
```

2 Inferencja typów w teorii

2.1 Problem inferencji typów

Język ML przyjmuje wiele form, najpopularniejszymi wariantami są język Standard ML, język OCaml i język F#. Na potrzeby przykładu będziemy wzorować się na [3] i posługiwać się ML-the-calculus, drastycznie uproszczoną wersją języka ML co pozwoli na dojście do sedna w problemie rekonstrukcji typów.

Termy w języku ML-the-calculus są następujące:

$$\begin{aligned}
 e &::= x && \text{(identyfikatory)} \\
 &| c && \text{(stałe)} \\
 &| \lambda x. e \\
 &| e e \\
 &| \text{let } x = e \text{ in } e
 \end{aligned} \tag{1}$$

Typy które będziemy przypisywali do termów są następujące:

$$\begin{aligned}
 \tau &::= \alpha && \text{(typ zmienny)} \\
 &| B && \text{(typ podstawowy)} \\
 &| \tau \rightarrow \tau
 \end{aligned} \tag{2}$$

2.2 Udowadnianie typów dla ML-the-calculus

$$\overline{\Gamma \vdash c : B}$$

$$\frac{\Gamma, x : \tau' \vdash e : \tau}{\Gamma \vdash \lambda x. e : \tau' \rightarrow \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau_2 \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 e_2 : \tau}$$

$$\frac{\Gamma(x) = \bigwedge \alpha_1, \dots, \alpha_n. \tau' \quad \tau = [\beta_i / \alpha_i] \tau' \quad (\beta_1 \text{ fresh})}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau' \quad \Gamma, x : (\bigwedge \alpha_1, \dots, \alpha_n. \tau') \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau : B} (\{\alpha_1, \dots, \alpha_n\} = ftv(\tau') \setminus ftv(\Gamma))$$

Funkcja $ftv(\tau)$ oblicza zbiór zmiennych typów występujących w τ .

Powyższe dowody nie mogą jednak być w prosty sposób przedstawione jako algorytm. Fakt, że typ τ' jest dowolnie wybierany w dowodzie dla λ prowadzi do nieskończonej ilości dowodów, nawet dla prostego wyrażenia $\lambda x. x$.

2.3 Inferencja typów bazująca na ograniczeniach

Algorytmy inferencji typów bazujące na ograniczeniach generują dużą liczbę zmiennych typów oraz zbiór ograniczeń dla tych zmiennych. W drugim kroku algorytm dla każdej zmiennej, szuka typu, który spełnia wszystkie ograniczenia.

2.4 Algorytm W

Algorytm W inferencji typów przedstawiony w [6] oraz innych wcześniejszych publikacjach, opiera się na generowaniu ograniczeń i rozwiązywaniu ich w trakcie wykonywania rekursywnego przechodzenia przez termy.

Warto zauważyć, że w ten sposób: dostajemy, prostą strukturalnie rekursywną definicję rekonstrukcji typów, ale tracimy modularność algorytmu: rozszerzenie algorytmu W o dodatkowe funkcje jest o wiele trudniejsze niż algorytmu, który oddziela kroki generowania ograniczeń i ich rozwiązywania.

$$W : \Gamma \times e \rightarrow S \times \tau$$

$$\begin{aligned} W(\Gamma, x) &= ([\], [\beta_i/\alpha_i]\tau') \\ &\text{where } \Gamma(x) = \bigwedge \alpha_i, \dots, \alpha_n.\tau' \\ &\text{and } \beta_i \text{ are fresh} \end{aligned}$$

$$\begin{aligned} W(\Gamma, e_1 \ e_2) &= (V \circ S_2 \circ S_1, V\beta) \\ &\text{where } (S_1, \tau_1) = W(\Gamma, e_1) \\ &\text{and } (S_2, \tau_2) = J(S_1, \Gamma, e_2) \\ &\text{and } V = \text{unify}(\{S_2\tau_1 = \tau_2 \rightarrow \beta\}) \\ &\text{and } \beta \text{ is fresh} \end{aligned} \tag{3}$$

$$\begin{aligned} W(\Gamma, \lambda x.e) &= (S, S\beta \rightarrow \tau) \\ &\text{where } (S, \tau) = W((\Gamma, x : \beta), e) \\ &\text{and } \beta \text{ is fresh} \end{aligned}$$

$$\begin{aligned} W(\Gamma, \text{let } x = e_1 \text{ in } e_2) &= (S_2 \circ S_1, \tau_2) \\ &\text{where } (S_1, \tau_1) = W(\Gamma, e_1) \\ &\text{and } (S_2, \tau_2) = W((S_1\Gamma, x : (\bigwedge \alpha_1, \dots, \alpha_n.\tau_1)), e_2) \\ &\text{and } \{\alpha_1, \dots, \alpha_n\} = \text{ftv}(\tau_1) \setminus \text{ftv}(S_1\Gamma) \end{aligned}$$

2.5 Algorytm J

Algorytm W nie ma żadnych efektów ubocznych i świetnie zajmuje się aplikowaniem i komponowaniem substytucji w odpowiedniej kolejności. Częste aplikacje substytucji na wyrażeniach mogą znacznie zmniejszyć wydajność algorytmu rekonstruującego, dlatego też Milner zaprezentował bardziej efektywną imperatywną wariację algorytmu W nazwaną algorytmem J w [6].

Algorytm J jest funkcją, która dla dokonanych do tej pory substytucji S , środowiska Γ (ang. *context*) (zbioru przechowywującego pary identyfikator wraz

z typem) i wyrażenia e , zwraca kolejne substytucje S oraz typ wyrażenia τ .

$$J : S \times \Gamma \times e \rightarrow S \times \tau$$

$$\begin{aligned} J(S, \Gamma, x) &= (S, [\beta_i / \alpha_i] \tau') \\ \text{where } \Gamma(x) &= \bigwedge \alpha_i, \dots, \alpha_n. \tau' \\ \text{and } \beta_i &\text{ are fresh} \end{aligned}$$

$$\begin{aligned} J(S, \Gamma, e_1 \ e_2) &= (V, \beta) \\ \text{where } (S_1, \tau_1) &= J(S, \Gamma, e_1) \\ \text{and } (S_2, \tau_2) &= J(S_1, \Gamma, e_2) \\ \text{and } V &= \text{unify}'(\tau_1, \tau_2 \rightarrow \beta, S_2) \\ \text{and } \beta &\text{ is fresh} \end{aligned}$$

$$\begin{aligned} J(S, \Gamma, \lambda x. e) &= (S_1, \beta \rightarrow \tau) \\ \text{where } (S_1, \tau) &= J(S, (\Gamma, x : \beta), e) \\ \text{and } \beta &\text{ is fresh} \end{aligned}$$

$$\begin{aligned} J(S, \Gamma, \text{let } x = e_1 \text{ in } e_2) &= (S_2, \tau_2) \\ \text{where } (S_1, \tau_1) &= J(S, \Gamma, e_1) \\ \text{and } (S_2, \tau_2) &= J(S_1, (\Gamma, x : (\bigwedge \alpha_1, \dots, \alpha_n. \tau_1)), e_2) \\ \text{and } \{\alpha_1, \dots, \alpha_n\} &= \text{ftv}(S_2 \tau_1) \setminus \text{ftv}(S_2 \Gamma) \end{aligned} \tag{4}$$

Funkcja pomocnicza $\text{unify}'(\tau, \tau', S)$ rozszerza zbiór substytucji S o substytucje wynikające z unifikacji τ z τ' pod kontekstem S . Np. $\text{unify}'(\alpha_1 \rightarrow \alpha_2, B \rightarrow \alpha_3, S) = \{(\alpha_1, B), (\alpha_2, \alpha_3)\}$

3 Kompilacja dopasowania do wzorca (ang. *pattern matching*)

3.1 Motywacja

Wykorzystywany algorytm jest odrobinę inny od algorytmów opisanych w literaturze. Bierzemy pod uwagę następujące obserwacje:

- Część algorytmów w literaturze w celu uniknięcia wykładniczego wzrostu ilości wygenerowanego kodu, generuje zbędne testy [2].
- Wykładniczy wzrost jednak nie występuje w praktyce [8].

- Najlepszą praktyką więc wydało nam się by nigdy nie generować niepotrzebnych sprawdzeń i spróbować uniknąć duplikacji kodu używając heurystyki tak jak [5].
- Literatura udowadnia, że w przypadku kodu napisanego w praktyce, różne algorytmy generują prawie identyczny kod [8, 5].

Naszą metodą więc jest, aby: (1) zawsze skupiać się na przypasowaniu pierwszego przypadku, by uniknąć niepotrzebnych testów i (2) zachłannie spróbować zminimalizować duplikację przy użyciu heurystyki.

3.2 Kompilacja na przykładzie

Naszym celem jest kompilacja poniższego dopasowania do wzorca w języku ML na drzewo decyzyjne.

match α with		
Add(Zero, Zero)		$\rightarrow e_1$
Mul(Zero, x)		$\rightarrow e_2$
Add(Succ(x), y)		$\rightarrow e_3$
Mul(x , Zero)		$\rightarrow e_4$
Mul(Add(x , y), z)		$\rightarrow e_5$
Add(x , Zero)		$\rightarrow e_6$
x		$\rightarrow e_7$

Powyższe wyrażenie możemy przedstawić jako listę równań.

$\alpha = \text{Add}(\text{Zero}, \text{Zero})$	$\rightarrow e_1$
$\alpha = \text{Mul}(\text{Zero}, x)$	$\rightarrow e_2$
$\alpha = \text{Add}(\text{Succ}(x), y)$	$\rightarrow e_3$
$\alpha = \text{Mul}(x, \text{Zero})$	$\rightarrow e_4$
$\alpha = \text{Mul}(\text{Add}(x, y), z)$	$\rightarrow e_5$
$\alpha = \text{Add}(x, \text{Zero})$	$\rightarrow e_6$
$\alpha = x$	$\rightarrow e_7$

Nasz algorytm będzie otrzymywał taką listę równań i zwracał drzewo w którym każdy wierzchołek będzie dopasowaniem do jednego konstruktora.

match α with	
$C(a_1, \dots, a_n)$	$\rightarrow A$
$_$	$\rightarrow B$

Zaczniemy od pierwszego równania i porównania z konstruktorem `Add`.

$$\begin{array}{ll} \text{match } \alpha \text{ with} & \\ | \text{Add}(a_1, a_2) & \rightarrow A \\ | _ & \rightarrow B \end{array}$$

Porównanie to rozбивa cały problem na pod problemy:

Następujący pod problemem dla A:

$$\begin{array}{ll} | \alpha_1 = \text{Zero}, \alpha_2 = \text{Zero} & \rightarrow e_1 \\ | \alpha_1 = \text{Succ}(x), \alpha_2 = y & \rightarrow e_3 \\ | \alpha_1 = x, \alpha_2 = \text{Zero} & \rightarrow e_6 \\ | \alpha = x & \rightarrow e_7 \end{array}$$

Oraz pod problemem dla B:

$$\begin{array}{ll} | \alpha = \text{Mul}(\text{Zero}, x) & \rightarrow e_2 \\ | \alpha = \text{Mul}(x, \text{Zero}) & \rightarrow e_4 \\ | \alpha = \text{Mul}(\text{Add}(x, y), z) & \rightarrow e_5 \\ | \alpha = x & \rightarrow e_7 \end{array}$$

Wyrażenia takie jak $\alpha_2 = y$ to przypisanie do zmiennej, które odbywa się dopiero po dopasowaniu do wzorca, dlatego, możemy przenieść je na prawą stronę. Pod problem A po tym uproszczeniu wygląda następująco:

$$\begin{array}{ll} | \alpha_1 = \text{Zero}, \alpha_2 = \text{Zero} & \rightarrow e_1 \\ | \alpha_1 = \text{Succ}(x) & \rightarrow \text{let } y = \alpha_2 \text{ in } e_3 \\ | \alpha_2 = \text{Zero} & \rightarrow \text{let } x = \alpha_1 \text{ in } e_6 \\ | & \rightarrow \text{let } x = \alpha \text{ in } e_7 \end{array}$$

Następnie możemy kontynuować algorytm przez sprawdzenie $\alpha_1 = \text{Zero}$:

$$\begin{array}{ll} \text{match } \alpha_1 \text{ with} & \\ | \text{Zero} & \rightarrow C \\ | _ & \rightarrow D \end{array}$$

I kontynuować rekursywnie dla C i D.

3.3 Generalizacja algorytmu

Mając listę równań, wykonujemy następującą listę kroków:

- Równania $\alpha = y$ przenosimy na prawą stronę, aby pozostać z listą równań testujących konstruktory.
- Wybieramy któreś z równań $\alpha = C(\delta_1, \dots, \delta_n)$
- Generujemy następujące wyrażenie:

$$\begin{array}{ll} \text{match } \alpha \text{ with} & \\ | C(a_1, \dots, a_n) & \rightarrow A \\ | _ & \rightarrow B \end{array}$$

- Tworzymy dwa pod problemy, iterując przez równania w następujący sposób:
 - Jeżeli wyrażenie zawiera równanie $\alpha = C(\delta_1, \dots, \delta_n), \dots, \Gamma$.
Rozszerzamy wyrażenie tworząc nowe zmienne $\alpha_1 = \delta_1, \dots, \alpha_n = \delta_n, \dots, \Gamma$ i dodajemy je do A.
 - Jeżeli wyrażenie zawiera równanie $\alpha = D(a_1, \dots, a_n), \dots, \Gamma$ gdzie $D \neq C$.
Dodajemy równanie do B.
 - Jeżeli wyrażenie zawiera równanie α .
Dodajemy je do A i B.
- rekursywnie wywołujemy algorytm dla A i B.

Rekursja kończy się kiedy:

- lista jest pusta — generujemy błąd braku pełnego dopasowania,
- pierwsze z wyrażeń nie ma żadnych równań (udało nam się dokonać dopasowania) — po prostu zwracamy e_1 .

4 ReScript — język realizujący podobne zadania

OCaml jest funkcyjnym językiem programowania kompilowanym na kod maszynowy. ReScript jest to fork OCaml generujący kod w języku JavaScript, skupiający się na łatwej możliwości wdrożenia i adaptacji przez deweloperów korzystających z języka JavaScript. Język ten posiada wiele funkcji, które pojawiają się w naszym języku — między innymi generyczne typy danych, unie z dyskriminatorem, dopasowania do wzorca.

Przykład kodu w języku ReScript:

```
1 module Button = {  
2   @react.component  
3   let make = (~count: int) => {  
4     let times = switch count {  
5       | 1 => "once"  
6       | 2 => "twice"  
7       | n => Belt.Int.toString(n) ++ " times"  
8     }  
9     let msg = "Click me " ++ times  
10  
11     <button> {msg->React.string} </button>  
12   }  
13 }
```

5 Założenia i priorytety opracowanej aplikacji

Tworząc aplikację, chcieliśmy, by kompilator posiadał podstawowe typy danych (liczba, napis, wartość logiczna), kilka typów generycznych (funkcja, tablica), typ `option` oraz możliwość tworzenia własnych typów. Dodatkowo nie powinno być potrzeby podawania typów zmiennych w większości przypadków, kompilator sam powinien wykrywać typy zmiennych na podstawie ich użycia.

Język poza zmiennymi, potrzebuje możliwości wykonywania operacji na danych, dlatego ważne było dla nas, by zaimplementować operatory binarne oraz unarne. Operatory te miały też spełniać ważną rolę w trakcie inferencji typów. W języku JavaScript operator `+` może być wykorzystywany do dodawania liczb jak i konkatenacji napisów, ważne więc było by stworzyć dwa oddzielne operatory.

Porównanie prymitywnych typów danych i często wykonywanych operacji z ich odpowiednikami w języku JavaScript pokazane jest w tabeli 1.

Co	Przykład	wyjscie w języku JavaScript
Ciąg znaków	'Hello'	"Hello"
Konkatenacja	'Hello '++ 'World'	"Hello " + "World"
Liczby	23, -23.0	23, -23.0
Dodawanie/Odejmowanie	6 + 2.0 - 4	6 + 2.0 - 4
Dzielenie/Mnożenie	2 / 23 * 1	2 / 23 * 1
Modulo	12 % 3	12 % 3
Dzielenie całkowite	5 // 2	Math.floor(5 / 2)
Konkatenacja tablic	[1] [3, 4]	[1].concat([3, 4])
Porównywanie liczb	>, <, <=	>, <, !==
Równość/Nierówność	==, !=	===, !==
Zmienne logiczne	True, False	true, false

Table 1: Porównanie

Chcieliśmy również, by funkcje wieloargumentowe kompilowane były jako funkcje jednoargumentowe zwracające kolejne funkcje, co pozwala na wywołanie funkcji z niepełną liczbą argumentów w celu zwrócenia funkcji przyjmującej resztę argumentów (ang. *currying*).

```
1 def classy_greeting(first_name, last_name) do
2   "The name's " ++ last_name ++ ', ' ++ first_name ++ ' ' ++
    last_name
3 end
4
5 def compose1(f, g, a) do
6   f(g(a))
7 end
8
9 def compose2(f, g, a, b) do
10  f(g(a, b))
11 end
12
13 yell_greetings = compose2(to_upper, classy_greeting)
14 yell_greetings('James', 'Bond') # "THE NAME'S BOND, JAMES BOND"
15
16 compose1(compose1(abs, add(1)), multiply(2))(-4) # 7
```

examples/currying.uwu

```

1 const classy_greeting = (first_name) => (last_name) => {
2   return "The name's " + last_name + ", " + first_name + " " +
      last_name;
3 };
4 const compose1 = (f) => (g) => (a) => {
5   return f(g(a));
6 };
7 const compose2 = (f) => (g) => (a) => (b) => {
8   return f(g(a)(b));
9 };
10 const yell_greetings = compose2(to_upper)(classy_greeting);
11 yell_greetings("James")("Bond");
12 compose1(compose1(abs)(add(1.0)))(multiply(2.0))(-4.0);

```

examples/currying.uwu.js

Jednym z ważniejszych elementów każdego języka jest możliwość wykonywania różnego zbioru instrukcji warunkowo. W tym celu język powinien posiadać instrukcję `if` oraz `case`. Instrukcja `case` wykonywać ma odpowiedni zbiór instrukcji, zależnie do którego wzorca dopasowane zostaną wprowadzone dane. Kompilator, powinien ostrzegać, jeżeli istnieje możliwość niedopasowania danych do żadnego z wzorców.

- Przykład — funkcja łącząca dwie posortowane tablice

```

1 def merge<A>(a, b) do
2   merge2: Callable<Array<A>, Callable<Array<A>, Array<A>>> =
      'merge'
3
4   case Tuple(get_head(a), get_head(b)) of
5     Tuple(Empty(), _) do b end
6     Tuple(_, Empty()) do a end
7     Tuple(Head(head_a, rest_a), Head(head_b, rest_b)) do
8       if head_a < head_b then
9         [head_a] | merge2(rest_a, b)
10      else
11        [head_b] | merge2(a, rest_b)
12      end
13    end
14  end
15 end

```

examples/merge.uwu

```

1 const merge = (a) => (b) => {
2   const merge2 = merge;
3   return (() => {
4     const $ = { TAG: "Tuple", _0: get_head(a), _1: get_head(b)
5     };
6     if (typeof $ !== "string" && $.TAG === "Tuple") {
7       if ($._0 === "Empty") {
8         const _ = $_.1;
9         return b;
10      }
11      if ($._1 === "Empty") {
12        const _ = $_.0;
13        return a;
14      }
15      if (typeof $_.0 !== "string" && $_.0.TAG === "Head") {
16        if (typeof $_.1 !== "string" && $_.1.TAG === "Head") {
17          const head_b = $_.1._0;
18          const rest_b = $_.1._1;
19          const head_a = $_.0._0;
20          const rest_a = $_.0._1;
21          return (() => {
22            if (head_a < head_b) {
23              return [head_a].concat(merge2(rest_a)(b));
24            }
25            return [head_b].concat(merge2(a)(rest_b));
26          })();
27        }
28        throw new Error("Non-exhaustive pattern match");
29      }
30      throw new Error("Non-exhaustive pattern match");
31    })();
32  });
33 }

```

examples/merge.uwu.js

Kolejnym dość ważnym elementem języka jest brak wyrażenia ‘return’, które jest wykorzystywane do zwrócenia wartości z funkcji. Zamiast tego każdy blok instrukcji powinien zwracać ostatnie wyrażenie.

Przykład

```
1 message = if is_morning then
2   'Good morning!'
3 else
4   'Hello!'
5 end
6
7
8 result = do
9   arr1 = [1, 2, 3]
10  arr2 = map(arr1, add_one)
11
12  filter(arr2, def is_even(x) do
13    x % 2 == 0
14  end)
15 end
```

examples/return.uwu

```
1 const message = (() => {
2   if (is_morning) {
3     return "Good morning!";
4   }
5   return "Hello!";
6 })();
7 const result = (() => {
8   const arr1 = [1.0, 2.0, 3.0];
9   const arr2 = map(arr1)(add_one);
10  const is_even = (x) => {
11    return x % 2.0 === 0.0;
12  };
13  return filter(arr2)(is_even);
14 })();
```

examples/return.uwu.js

5.1 Opis formalny składni języka

```
 $S'$  ::= program
program ::= NEWLINE_optional do_exprs_optional
NEWLINE_optional ::= NEWLINE
| <empty>
do_exprs_optional ::= do_exprs
| <empty>
do_exprs ::= expr NEWLINE do_exprs
| expr NEWLINE_optional
expr ::= let
| ( expr ) [precedence = left, level = 7]
| enum
| variant_call
| case_of
| def_expr
| binary_expr
| call
| literal
| do
| array
| identifier
| - expr [precedence = right, level = 6]
| if_expr
| external
external ::= EXTERNAL
binary_expr ::= expr > expr [precedence = left, level = 3]
| expr INT_DIV expr [precedence = left, level = 5]
| expr / expr [precedence = left, level = 5]
| expr % expr [precedence = left, level = 5]
| expr - expr [precedence = left, level = 4]
| expr EQUAL expr [precedence = left, level = 2]
| expr CONCAT expr [precedence = left, level = 4]
| expr + expr [precedence = left, level = 4]
| expr NOT_EQUAL expr [precedence = left, level = 2]
| expr < expr [precedence = left, level = 3]
| expr | expr [precedence = left, level = 4]
```

```

        | expr * expr [precedence = left, level = 5]
do ::= DO type_optional NEWLINE_optional do_exprs_optional END
type_optional ::= type
        | <empty>
block_statement ::= NEWLINE_optional do_exprs_optional
def_expr ::= DEF identifier < type_identifier type_identifier_repeat > ( NEWLINE_opti
        | DEF identifier ( NEWLINE_optional params_optional ) type_optional do [pre
params_optional ::= <empty>
        | params
type_identifier_repeat ::= type_identifier_items
        | <empty>
type_identifier_items ::= type_identifier_item
        | type_identifier_items type_identifier_item
type_identifier_item ::= =, type_identifier
        params ::= params , NEWLINE_optional param NEWLINE_optional
        | param NEWLINE_optional
        type ::= type_identifier
        | type_identifier < type type_repeat > [precedence = left, level = 3]
type_repeat ::= type_items
        | <empty>
type_items ::= type_items type_item
        | type_item
type_item ::= =, type
        enum ::= ENUM type_identifier < type_identifier type_identifier_repeat > NEWLIN
        | ENUM type_identifier NEWLINE_optional variants_optional
variants_optional ::= variants
        | <empty>
variants ::= variant NEWLINE_optional
        | variants variant NEWLINE_optional
variant ::= type_identifier
        | type_identifier ( type type_repeat ) [precedence = left, level = 7]
param ::= identifier type_optional
if_expr ::= IF expr THEN type_optional block_statement or_else_optional END
or_else_optional ::= or_else
        | <empty>
or_else ::= ELSE block_statement
        | ELIF expr THEN block_statement or_else_optional

```

```

    case_of :: = CASE expr OF NEWLINE_optional cases_optional END
cases_optional :: = cases
    | <empty>
    cases :: = pattern do NEWLINE_optional
    | cases pattern do NEWLINE_optional
pattern :: = match_as
    | match_variant
match_as :: = identifier
match_variant :: = type_identifier
    | type_identifier ( NEWLINE_optional patterns_optional ) [precedence = left,
patterns_optional :: = <empty>
    | patterns
patterns :: = patterns , NEWLINE_optional pattern NEWLINE_optional
    | pattern NEWLINE_optional
array :: = [ NEWLINE_optional exprs_optional ]
exprs_optional :: = exprs
    | <empty>
    call :: = expr ( NEWLINE_optional exprs_optional ) [precedence = left, level = 7]
variant_call :: = type_identifier ( NEWLINE_optional exprs_optional ) [precedence = left, level = 7]
exprs :: = exprs , NEWLINE_optional expr NEWLINE_optional
    | expr NEWLINE_optional
identifier :: = IDENTIFIER
type_identifier :: = TYPE_IDENTIFIER
    let :: = identifier type_optional = expr [precedence = left, level = 1]
literal :: = NUMBER
    | STRING

```

6 Narzędzia

6.1 Język Python

Do implementacji naszego kompilatora postanowiliśmy wykorzystać język Python w wersji 3.10, ze względu na jego dynamiczność. W tej wersji języka pojawił się również pattern matching, który znacząco ułatwia pracę z drzewem syntaktycznym.

6.2 Parsowanie i tokenizowanie przy użyciu biblioteki sly

Biblioteka sly, jest implementacją narzędzi lex i yacc w języku Python, wykorzystywanych do tworzenia parserów i kompilatorów. Tworzenie leksera i parsera jest bardzo proste. W naszym języku korzystaliśmy z wersji 0.4.

6.2.1 Lekser

Tokeny są tworzone przy pomocy wyrażeń regularnych.

```
1 import sly
2 class Lex(sly.Lexer):
3     ID = r"\w+"
4     NUM = r"\d+"
```

Biblioteka udostępnia specjalną składnię do tworzenia tokenów, w przypadku gdy są już one opisane przez inny token.

```
1 ID["and"] = AND
```

Pojedyncze znaki można ignorować, poprzez ustawienie pola 'ignore'. Dodatkowo ignorowane są też tokeny zaczynające się od 'ignore'.

```
1 ignore = " \t"
2 ignore_comm = r"\#.*"
```

Możemy też stworzyć tokeny o typie równym ich wartości, o ile składają się tylko z jednego znaku.

```
1 literals = {"(", ")"} 
```

6.2.2 Parser

W przypadku parsera każda reguła jest implementowana jako metoda, pod której argumentem mamy dostęp do tokenów i wartości zwróconych z innych metod występujących w składni danej reguły.

```
1 import sly
2 class Parser(sly.Parser):
3     tokens = Lex.tokens
4
5     @_("NUM")
6     def expr(self, p):
7         return p[0] # Lub p.ID
8
9     @_("(' expr AND expr ')")
10    def expr(self, p):
```

```
11 | return p.expr0 and p.expr1
```

Powyższy parser pozwala na opisanie wyrażeń logicznych typu: `(1 and 0) and 4`.

6.3 Środowisko nodejs do uruchomienia skompilowanego kodu

JavaScript jest językiem programowania wykorzystywanym w przeglądarkach internetowych. Do uruchomienia skompilowanego kodu używali będziemy jednak środowiska nodejs. Pozwoli to na szybkie i wygodne testowanie wygenerowanego kodu. Po zainstalowaniu środowiska i skompilowaniu pliku `index.uwu` otrzymamy plik `index.uwu.js`, który możemy uruchomić w terminalu komendą `node index.uwu.js`.

7 Implementacja

7.1 Lekser

Nasz lekser implementować będzie poniższy zbiór tokenów, warto zauważyć, że identyfikatory zaczynające się z dużej litery są identyfikatorami typów.

```
1 | NOT_EQUAL = r"!="
2 | EQUAL = r"=="
3 | STRING = r"'[^']*'"
4 | NUMBER = r"\d+"
5 | CONCAT = r"\{2}"
6 | INT_DIV = r"/{2}"
7 | TYPE_IDENTIFIER = r"[A-Z\d][\w\d]*"
8 | IDENTIFIER = r"[a-z_][\w\d]*"
9 | IDENTIFIER["def"] = DEF
10 | IDENTIFIER["do"] = DO
11 | IDENTIFIER["end"] = END
12 | IDENTIFIER["if"] = IF
13 | IDENTIFIER["else"] = ELSE
14 | IDENTIFIER["elif"] = ELIF
15 | IDENTIFIER["case"] = CASE
16 | IDENTIFIER["enum"] = ENUM
17 | IDENTIFIER["then"] = THEN
18 |
19 | IDENTIFIER["of"] = OF
20 | EXTERNAL = r"'[^']*'"
```

parser.py

Nasz lekser zajmował się również będzie zwracaniem tokena dla znaku nowej linii

```
1 | @_(r"\n([\s\t\n]|#.*)"
2 | def NEWLINE(self, t):
3 |     self.lineno += t.value.count("\n")
4 |     return t
```

parser.py

Lekser ignorował będzie tokeny komentarza, które zaczynają się od znaku '#'.
Oraz znaki tabulacji i spacji.

```

1 ignore_comment = r"\#.*"
2 ignore = " \t"

```

parser.py

Poza tym nasz lekser zwracał będzie poniższy zbiór tokenów

```

1 literals = {
2     "=",
3     ".",
4     "[",
5     "]",
6     ",",
7     "{",
8     "}",
9     "(",
10    ")",
11    ":",
12    "+",
13    "-",
14    ">",
15    "<",
16    "*",
17    "/",
18    "|",
19    "%",
20 }

```

parser.py

7.2 Parser

Implementacja parsera jest bardzo prosta, każda z metod zajmuje jedynie liniijkę, gdzie zwracane jest odpowiednie wyrażenie drzewa syntaktycznego.

```

1 def binary_expr(self, p):
2     return terms.EBinaryExpr(p[1], p[0], p[2])
3
4 @_(
5     "DO [ ':' type ] [ NEWLINE ] [ do_exprs ] END",
6 )
7 def do(self, p):
8     return terms.EDo(p.do_exprs or [], hint=terms.EHint.
9         from_option(p.type))

```

parser.py

W przypadku zbiorów wyrażeń zwracane są listy.

```

1  @_("exprs ',' [ NEWLINE ] expr [ NEWLINE ]")
2  def exprs(self, p):
3      return p.exprs + [p.expr]
4
5  @_("expr [ NEWLINE ]")
6  def exprs(self, p):
7      return [p.expr]

```

parser.py

7.3 Drzewo decyzyjne

Dopasowanie do wzorca zamieniane jest w drzewo decyzyjne przy pomocy rekurencyjnej funkcji. Rekursja ma dwa podstawowe przypadki:

- lista przypadków jest pusta, więc generujemy pustą gałąź końcową
- pierwszy przypadek, nie ma już więcej wzorów, więc zwracamy gałąź końcową z blokiem.

```

1  def gen_match2(clauses1: typing.Sequence[Clause]) -> CaseTree:
2
3      clauses = [subst_var_eqs(clause) for clause in clauses1]
4
5      match clauses:
6          case []:
7              return MissingLeaf()
8          case [(patterns, body), *_] if not patterns:
9              return Leaf(body)

```

case_tree.py

W innym przypadku wybieramy wzorec przy pomocy heurystyki


```

1     case [(patterns, body), * _]:
2         branch_var = branching_heuristic(patterns, clauses)
3         branch_pattern = patterns[branch_var]
4         yes = list[Clause]()
5         no = list[Clause]()
6
7         #
8         vars = [f"{branch_var}._{i}" for i in range(len(
            branch_pattern.patterns))]

```

case_tree.py

Następnie tworzymy dwa pod problemy yes i no iterując przez wszystkie przypadki. Dla każdego z nich robimy jedną z trzech rzeczy:

- Przypadek nie zawiera wzorca, więc dodajemy go do yes i no

```

1         for patterns, body in clauses:
2             clause = Clause(dict[str, terms.Pattern](
3                 patterns), body)
4             match patterns.get(branch_var, None):
5                 case None:
6                     yes.append(clause)
7                     no.append(clause)

```

case_tree.py

- Przypadek zawiera szukany wzorec, więc dodajemy go do yes

```

1         case terms.EMatchVariant(id, patterns) if
2         id == branch_pattern.id:
3             yes.append(
4                 dataclasses.replace(
5                     clause,
6                     patterns={
7                         key: value
8                         for key, value in clause.
9                         patterns.items()
10                        if key != branch_var
11                    }
12                    | dict(zip(vars, patterns)),
13                )
14            )

```

case_tree.py

- Przypadek zawiera wzorec, więc dodajemy go do no

```

1         case terms.EMatchVariant():
2             no.append(clause)

```

case_tree.py

Rekursywnie generujemy kod dla yes i no i zwracamy gałąź decyzyjną

```

1         return Node(
2             branch_var, branch_pattern.id, vars, gen_match2(yes
3             ), gen_match2(no)
4         )

```

case_tree.py

7.4 Inferencja typów

7.4.1 Reprezentacja środowiska

Implementację inferencji typów zaczynamy przez stworzenie zmiennego typu, który będzie reprezentowany przez unikalny identyfikator.

```

1 counter = 0
2
3
4 def fresh_ty_var(kind=typed.KStar()) -> typed.TVar:
5     global counter
6     counter += 1
7     return typed.TVar(counter, kind)

```

algorithm_j.py

Substytucje to pary zmiennych i typów

```

1 Substitution: typing.TypeAlias = dict[int, typed.Type]

```

algorithm_j.py

Jednym z argumentów algorytmu J jest środowisko, w naszej implementacji reprezentowane jest one przez zmienną `context`, będący słownikiem identyfikatorów i schematów,

```

1 Context: typing.TypeAlias = dict[str, Scheme]

```

algorithm_j.py

gdzie schemat zawiera informację o typie oraz listę występujących w nim zmiennych typów

```

1 @dataclasses.dataclass
2 class Scheme:
3     vars: list[int]
4     ty: typed.Type

```

algorithm_j.py

7.4.2 Aplikowanie substytucji

Tworzymy funkcję aplikującą substytucje na typie.

Przykład: dla $subt = \{a : Num\}$ i $ty = a \rightarrow b$ funkcja zwraca $Num \rightarrow b$

```

1 def apply_subst(subst: Substitution, ty: typed.Type) -> typed.Type:
2     match ty:
3         case typed.TVar(var):
4             return subst.get(var, ty)
5         case typed.TAp(arg, ret):
6             return typed.TAp(apply_subst(subst, arg), apply_subst(
7 subst, ret))
8         case typed.TCon():
9             return ty
10        case _:
11            raise TypeError(f"Cannot apply substitution to {ty=}")

```

algorithm_j.py

Tworzymy funkcję, unifikującą dwa typy

```

1 def unify(a: typed.Type, b: typed.Type) -> Substitution:
2     match (a, b):
3         case (typed.TCon(), typed.TCon()) if a == b:
4             return {}
5         case (
6             typed.TAp(arg0, ret0),
7             typed.TAp(arg1, ret1),
8         ):
9             subst = unify_subst(arg0, arg1, {})
10            subst = unify_subst(ret0, ret1, subst)
11
12            return subst
13        case (typed.TVar(u), t) | (t, typed.TVar(u)) if typed.kind(
14 a) != typed.kind(b):
15            raise UnifyException(f"Kind for {a=} and {b=} does not
16 match")
17        case (typed.TVar(u), t) | (t, typed.TVar(u)):
18            return var_bind(u, t)
19        case _:
20            raise UnifyException(f"Cannot unify {a=} and {b=}")

```

algorithm_j.py

wraz z funkcją generującą substytucje w przypadku unifikacji zmiennego typu

```

1 def var_bind(u: int, t: typed.Type) -> Substitution:
2     match t:
3         case typed.TVar(tvar2) if u == tvar2:
4             return {}
5         case typed.TVar(_):
6             return {u: t}
7         case t if u in free_type_vars(t):
8             raise TypeError(f"circular use: {u} occurs in {t}")
9         case t:
10            return {u: t}

```

algorithm_j.py

Teraz zaczynamy implementację algorytmu J.

7.4.3 Literały

W przypadku literałów typ jest oczywisty.

```
1 def infer(  
2     subst: Substitution, ctx: Context, exp: terms.AstTree  
3 ) -> tuple[Substitution, typed.Type]:  
4     match exp:  
5         case terms.ELiteral(value=str()):  
6             return subst, typed.TStr()  
7         case terms.ELiteral(value=float()):  
8             return subst, typed.TNum()
```

algorithm_j.py

7.4.4 Identyfikatory

W przypadku identyfikatorów musimy zastąpić występujące w nich zmienne typy nowymi zmiennymi typami.

```
1         case terms.EIdentifier(var):  
2             return subst, instantiate(ctx[var])
```

algorithm_j.py

7.4.5 Bloki wyrażeń

W przypadku bloku wyrażeń inferujemy typ każdego z nich i zwracamy ostatni typ.

```
1         case terms.EProgram(body) | terms.EBlock(body):  
2             ty = typed.TUnit()  
3             ctx = ctx.copy()  
4  
5             for exp in body:  
6                 subst, ty = infer(subst, ctx, exp)  
7  
8             return subst, ty
```

algorithm_j.py

7.4.6 Operatory binarne

Implementacje operatorów binarnych są bardzo podobne.

Na przykład operator `|` służący do konkatencji dwóch list, oczekuje by wyrażenie z lewej i prawej strony były listami tego samego typu i zwraca nową listę tego właśnie typu.

```

1      case terms.EBinaryExpr("|", left, right):
2          ty = typed.TArray(fresh_ty_var())
3          subst, ty_left = infer(subst, ctx, left)
4          subst = unify_subst(ty_left, ty, subst)
5          subst, ty_right = infer(subst, ctx, right)
6          subst = unify_subst(ty_right, ty, subst)
7          return subst, ty

```

algorithm_j.py

7.4.7 Wywołania funkcji

W przypadku wywołania funkcji musimy zaimplementować currying.

Przykład: $J(fn) = Num \rightarrow Str \rightarrow Unit \rightarrow Arr < Str >$

$J(args) = [Num, Str]$

- typy argumentów zbieramy w odwrotnej kolejności $ty_args=[Str, Num]$
- Tworzymy nowy zmienny typ $ty = \alpha$
- Następnie przy użyciu funkcji `reduce_args` aplikujemy ty_args na typie α co tworzy typ $tmp = Num \rightarrow Str \rightarrow \alpha$
- Unifikujemy typy tmp i $J(fn)$ co tworzy substytucję $\{\alpha : Unit \rightarrow Arr < Str >\}$
- I zwracamy α

```

1      case terms.ECall(fn, args):
2          subst, ty_fn = infer(subst, ctx, fn)
3
4          ty_args = list[typed.Type]()
5
6          for arg in reversed(args):
7              subst, ty_arg = infer(subst, ctx, arg)
8              ty_args.append(ty_arg)
9
10         ty = fresh_ty_var()
11         subst = unify_subst(ty_fn, reduce_args(ty_args, ty),
12         subst)
13
14         return subst, ty

```

algorithm_j.py

7.4.8 Wyrażenia warunkowe

W przypadku wyrażeń warunkowych musimy, sprawdzić, czy warunek jest typu `Bool` i czy, typy zwracane z każdej gałęzi są takie same, po czym zwracamy ten typ.

```

1      case terms.Elif(test, then, or_else, hint=hint):
2
3          subst, hint = infer(subst, ctx, hint)
4          subst, ty_condition = infer(subst, ctx, test)
5          subst = unify_subst(ty_condition, typed.TBool(), subst)
6
7          subst, ty_then = infer(subst, ctx, then)
8          subst = unify_subst(ty_then, hint, subst)
9
10         subst, ty_or_else = infer(subst, ctx, or_else)
11         subst = unify_subst(ty_or_else, hint, subst)
12
13     return subst, hint

```

algorithm_j.py

7.4.9 Definicje funkcji

W przypadku definicji funkcji musimy upewnić się że typy generyczne, są przypisywane do środowiska. Musimy sprawdzić, czy typ zwracany z funkcji jest przypisywalny do zadeklarowanego typu oraz zapisać funkcję do środowiska.

```

1      case terms.EDef(id, params, body, hint, generics):
2          t_ctx = ctx.copy()
3
4          for generic in generics:
5              ty_generic = fresh_ty_var()
6              t_ctx[generic.name] = Scheme([], ty_generic)
7
8          subst, hint = infer(subst, t_ctx, hint)
9
10         ty_params = list[typed.Type]()
11
12         for param in reversed(params):
13             subst, ty_param = infer(subst, t_ctx, param)
14             ty_params.append(ty_param)
15
16         ty = reduce_args(ty_params, hint)
17
18         #
19         subst, ty_body = infer(subst, t_ctx, body)
20
21         subst = unify_subst(ty_body, hint, subst)
22
23         ctx[id] = Scheme.from_subst(subst, ctx, ty)
24
25     return subst, ty

```

algorithm_j.py

7.5 Dopasowanie do wzorca

W trakcie dopasowania do wzorca, musimy upewnić się, że konstruktory, posiadają odpowiednią liczbę argumentów. Dodatkowo przeprowadzać będziemy

sprawdzanie czy wyrażenie `case` jest wyczerpujące — czy przewiduje wszystkie przypadki. W tym celu funkcja przyjmuje argument `o_alts` będący bazą zmiennych i ich konstruktorów.

W przypadku pustej gałęzi końcowej, sprawdzamy, czy każda z `list` jest pusta i wyrzucamy błąd, jeżeli jest.

```

1 def infer_case_tree(
2     subst: Substitution,
3     ctx: Context,
4     tree: case_tree.CaseTree,
5     o_alts: dict[str, list[str]] | None = None,
6 ) -> tuple[Substitution, typed.Type]:
7     alts = o_alts or {}
8
9     match tree:
10         case case_tree.MissingLeaf():
11             if any(alts.values()):
12                 raise NonExhaustiveMatchException()
13             return subst, fresh_ty_var()

```

algorithm_j.py

W przypadku gałęzi decyzyjnej inferujemy typ zmiennej i sprawdzanego konstruktora

```

1     case case_tree.Node(var, pattern_name, vars, yes, no):
2
3         subst, ty_var = infer(subst, ctx, terms.EIdentifier(var
4     )) # x | x.0
5         subst, ty_pattern_name = infer(
6             subst, ctx, terms.EIdentifier(pattern_name)
7         ) # Some() | None()
8         t_ctx = ctx.copy()

```

algorithm_j.py

Następne dwa kroki dopełniają nasze sprawdzenie czy wyrażenie jest wyczerpujące. Wpierw uzupełniamy naszą bazę konstruktorów przy użyciu funkcji `alternatives` zwracającej listę wszystkich konstruktorów dla danego typu. Następnie usuwamy z bazy konstruktorów konstruktor do którego odnosi się nasza gałąź decyzyjna.

```

1         if var not in alts:
2             alts[var] = alternatives(ty_pattern_name)
3
4         if pattern_name in alts[var]:
5             alts[var].remove(pattern_name)

```

algorithm_j.py

Kolejna część inferuje typy zmiennych dla każdego z argumentów konstruktora. Oraz tworzy dla nich zmienne w środowisku.

```

1      ty_vars = list[typed.Type](fresh_ty_var() for _ in vars
2    )
3
4      pattern_name_con = typed.TCon(
5          pattern_name,
6          functools.reduce(
7              flip(typed.KFun),
8              map(typed.kind, ty_vars),
9              typed.KStar(),
10         ),
11     )
12
13     subst = unify_subst(
14         ty_pattern_name,
15         typed.TDef(
16             functools.reduce(
17                 typed.TAp,
18                 ty_vars,
19                 pattern_name_con,
20             ),
21             ty_var,
22         ),
23         subst,
24     )
25
26     for var2, ty_var in zip(vars, ty_vars):
27         t_ctx[var2] = Scheme.from_subst(subst, t_ctx,
28                                         ty_var)

```

algorithm_j.py

Reszta funkcji to sprawdzenie, czy typy zwracane z bloków są takie same.

```

1      ty = fresh_ty_var()
2
3      subst, ty_yes = infer_case_tree(
4          subst,
5          t_ctx,
6          yes,
7          {key: value for key, value in alts.items() if key
8            != var}
9          | {var2: alternatives(t_ctx[var2].ty) for var2 in
10             vars},
11      )
12      subst = unify_subst(ty_yes, ty, subst)
13
14      subst, ty_no = infer_case_tree(subst, ctx, no, alts)
15      subst = unify_subst(ty_no, ty, subst)
16
17      return subst, ty

```

algorithm_j.py

7.6 Windowanie (ang. *Hoisting*)

Ze względu, na to, że deklaracje zmiennych w języku JavaScript, np. `const x = 1` w przeciwieństwie do naszego języka nie zwracają przypisywanej do identyfikatora wartości, wyrażenia typu `f(a = b = c)` muszą być przekonwertowane do

```
1  b=c
2  a=b
3  f(a)
```

W tym celu zaimplementowaliśmy funkcję, która przyjmuje wyrażenie i zwraca, listę wyrażen do przeniesienia wyżej oraz zmodyfikowane wyrażenie.

W przypadku bloków wyrażen, listy z wyrażeniami do windowania są łączone z listą wyrażen danego bloku.

```
1 def hoist(
2     node: terms.EProgram | terms.EBlock,
3 ) -> terms.EProgram | terms.EBlock:
4     match node:
5         case terms.EBlock(body):
6             body2 = hoist_expr_list(body)
7             return terms.EBlock(filter_identifiers(body2[:-1:]) +
8                                 body2[-1::])
9
10        case terms.EProgram(body):
11            body2 = hoist_expr_list(body)
12            return terms.EProgram(filter_identifiers(body2))
```

compile.py

```
1 def hoist_expr_list(body: list[terms.Expr]) -> list[terms.Expr]:
2     body2 = list[terms.Expr]()
3
4     for expr in body:
5         let, expr2 = hoist_expr(expr)
6         body2.extend(let)
7         body2.append(expr2)
8
9     return body2
```

compile.py

W przypadku deklaracji zmiennych i funkcji, dodajemy elementy do listy wyrażen do windowania.

```

1 def hoist_expr(node: terms.Expr) -> tuple[list[terms.Expr], terms.
  Expr]:
2     match node:
3         case terms.ELet(id, init):
4             let, init2 = hoist_expr(init)
5
6             return let + [dataclasses.replace(node, init=init2)],
7             terms.EIdentifier(id)
8         case terms.EDef(id, params, body, hint):
9
10            return [
11                terms.EDef(id, params, body=hoist_do(body), hint=
  hint)
12            ], terms.EIdentifier(id)

```

compile.py

W każdym innym przypadku lista wyrażeń do windowania jest po prostu przepuszczana dalej.

```

1         case terms.EIf(test, then, terms.EIf() | terms.EIfNone() as
  or_else):
2             let_test, test2 = hoist_expr(test)
3             let_or_else, or_else2 = hoist_expr(or_else)
4             return let_test + let_or_else, dataclasses.replace(
5                 node,
6                 test=test2,
7                 then=hoist(then),
8                 or_else=or_else2,
9             )

```

compile.py

7.7 Kompilacja

Kompilacja opiera się na jednej funkcji zamieniającej drzewo syntaktyczne na kod języka JavaScript w postaci napisu.

7.8 Literały

Jak widać, dla literałów jest to dość trywialne.

```

1 def _compile(exp: terms.AstTree) -> str:
2     match exp:
3         case terms.EExternal(value=value):
4             return f"{value}"
5         case terms.ELiteral(value=str()):
6             return f'"{exp.value}"'
7         case terms.ELiteral(value=float()):
8             return f"{exp.value}"

```

compile.py

7.8.1 Wyrażenia warunkowe

Wyrażenia warunkowe wymagają kompilacji do postaci funkcji, gdyż w odróżnieniu od języka JavaScript, w naszym języku zwracają one wartość.

```
1     case terms.Elif(test, then, or_else):
2         return f"(()=>{{if({_compile(test)}}{{_compile(then)
    }}}{_compile(or_else)}})()"
```

compile.py

7.8.2 Warianty

Warianty przyjmują postać obiektów z dyskriminatorem

```
1     case terms.EVariantCall(id, args) if args:
2         args = [f"_{i}:{_compile(arg)}" for i, arg in enumerate
    (args)]
3
4         return f"{{TAG: '{id}', '{', '.join(args)}}'"
```

compile.py

7.8.3 Dopasowanie do wzorca

W przypadku pustej gałęzi z blokiem kompilujemy blok. W przypadku pustej gałęzi końcowej generujemy wyjątek.

```
1 def _compile_case_tree(tree: case_tree.CaseTree):
2     match tree:
3         case case_tree.Leaf(body):
4             return _compile(terms.EBlock(body.body))
5         case case_tree.MissingLeaf():
6             return "throw new Error('Non-exhaustive pattern match')"
```

compile.py

W przypadku gałęzi decyzyjnej generowany jest blok if sprawdzający zawartość zmiennej. Można też uniknąć generowania bloku else, w przypadku, gdy wygenerowany byłby wyjątek.

```

1      case case_tree.Node(var, pattern_name, vars, yes, no):
2          conditions = []
3
4          if pattern_name == "True":
5              conditions.append(f"{var}")
6          if pattern_name == "False":
7              conditions.append(f"!{var}")
8          elif vars:
9              conditions.insert(0, f"{var}.TAG=='{pattern_name}'")
10         ")
11         conditions.insert(0, f"typeof {var} != 'string'")
12     else:
13         conditions.insert(0, f"{var}=='{pattern_name}'")
14
15     # if isinstance(no, case_tree.MissingLeaf):
16     #     return _compile_case_tree(yes)
17
18     return f"if ({' && '.join(conditions)}) {{ {
19         _compile_case_tree(yes) }} {
20         _compile_case_tree(no) }}"

```

compile.py

8 Opis działania

Po uruchomieniu komendą `python3 main.py **/*.uwu` programu, program skompiluje wszystkie pliki kończące się rozszerzeniem `'uwu'` i wygeneruje dla każdego z nich odpowiedni plik z rozszerzeniem `'js'`.

8.1 Co działa

W programie udało się zaimplementować dużą część funkcjonalności potrzebnych do pisania faktycznych programów. Zabrakło jednak kilku początkowo planowanych funkcji: między innymi: rekursji, krotek, rekordów.

8.2 Uwagi do działania

8.2.1 Obsługa błędów

Błędy zwracane przez program nie są optymalne, nie zaimplementowaliśmy żadnej obsługi błędów w parserze. Błędy wynikające z inferencji typów, zwracają jedynie informacje o błędnych typach, nie o miejscu wystąpienia błędu. Wynika to z braku propagacji numerów linii i kolumn z parsera do drzewa syntaktycznego.

8.2.2 Dopasowanie do wzorca

W przypadku dopasowania do wzorca, kompilator powinien wyświetlić ostrzeżenie w momencie, gdy jedna ze ścieżek jest nieosiągalna.

9 [Podsumowanie]

10 [spisy — rysunków, tabel, listingów itp.]

References

- [1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers*. Addison Wesley, Boston, MA, Jan. 1985.
- [2] L. Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, Berlin , Heidelberg, Jan. 1985. Springer-Verlag.
- [3] L. Damas and R. Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of Programming Languages*, pages 207–212, 1982.
- [4] J. U. J.E. Hopcroft, R. Motwani. *Wprowadzenie do Teorii automatów, języków i obliczeń*. Wydawnictwo Naukowe PWN, 2005.
- [5] L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, USA, NY, Sept. 2008. Association for Computing Machinery.
- [6] R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
- [7] B. C. Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, London, England, Jan. 2002.
- [8] K. Scott and N. Ramsey. When do match-compilation heuristics matter? Technical report, University of Virginia, USA, 2000.