Tytuł

Patryk Wałach

January 2022

Contents

1 [Wstęp] 2

2 Ogólne wprowadzenie 2
2.1 Analiza leksykalna . . . 2
2.2 Parser . . . 3
2.3 System typów . . . 3
2.4 System typów ML . . . 4
2.5 Rekord z wariantami . . . 4

3 Inferencja typów w teori 5
3.1 Problem inferencji typów . . . 5
3.2 Udowadnianie typów dla ML-the-calculus . . . 5
3.3 Infernencja typów bazująca na ograniczeniach . . . 6
3.4 System typów Hindley–Milner . . . 6
3.5 Algorytm J . . . 6

4 Rescript/ReasonML jako języki realizujące podobne zadania 7

5 Założenia i priorytety opracowanej aplikacji 7
5.1 Opis formaly składni języka . . . 11

6 Narzędzia 11
6.1 Język python . . . 11
6.2 Parsowanie i tokenizowanie przy użyciu biblioteki sly . . . 11
6.2.1 Lexer . . . 12
6.2.2 Parser . . . 12
6.3 Środowisko nodejs do uruchomienia skompilowanego kodu . . . 13

7 Implementacja 13
7.1 lexer . . . 13
7.2 parser . . . 14
7.3 Drzewo decyzyjne . . . 15
7.4 inferencja typów . . . 17

7.4.1 Context . . . 17
7.4.2 Aplikowanie substytucji . . . 17
7.4.3 Literały . . . 19
7.4.4 Identyfikatory . . . 19
7.4.5 Bloki wyrażeń . . . 19
7.4.6 Operatory binarne . . . 19
7.4.7 Wywowałnia funkcji . . . 20
7.4.8 Wyrażenia warunkowe . . . 20
7.4.9 Definicje funkcji . . . 21
7.5 Pattern mathing . . . 21
7.6 Windowanie (Hoisting) . . . 24
7.7 Kompilacja . . . 25
7.8 Literały . . . 25
7.8.1 Wyrażenia warunkowe . . . 26
7.8.2 Warianty . . . 26
7.8.3 Pattern matching . . . 26

8 Opis działania **27**
8.1 Co działa . . . 27
8.2 Uwagi do działania . . . 27
8.2.1 Obsługa błędów . . . 27
8.2.2 Pattern matching . . . 27

9 [Podsumowanie] **28**

10 [spisy – rysunków, tabel, listingów ipt.] **28**

1 [Wstęp]

2 Ogólne wprowadzenie

2.1 Analiza leksykalna

Analiza leksykalna w informatyce jest to proces rozbijania program źródłowych na jednostki logiczne (zwane leksemami) złożone z jednego lub więcej znaków, które łącznie mają jakieś znaczenie[J.E. Hopcroft(2005)]. 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[Aho et al.(1985)Aho, Sethi, and Ullm 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:

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

Dwoma ważnymi przypadkami są znaki białe i komentarze. One również muszą być uwzględnione w gramatyce i przeanalizowane przez lexer, lecz mogą być odrzucone (nie produkować żadnych tokenów) i traktowane jako spełniające mało znaczące zadanie, rozdzielania dwóch tokenów (np. w `if x` zamiast `ifx`).

2.2 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[Aho et al.(1985)Aho, Sethi, and Ullman].

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 znajdywaniu 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.

2.3 System typów

System typów jest to system klasyfikacji wyrażeń w zależności od rodzajów wartości, jakie one generują[Pierce(2002)]. Każdej obliczonej wartości przypisy-

wany 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.

2.4 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 $\rightarrow$ 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 (nie licząc typów polimorficznych), jednak w Standard ML stworzono specjalne reguły umożliwiające polimorfizm dla wyrażeń arytmetycznych.

System typów 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.

2.5 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.

Przykład - Zakładając, że checmy stworzyć drzewo binarne intów. W języku ML, zrobilibyśmy to tworząc nowy typ danych w ten sposób:

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

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

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

3 Inferencja typów w teori

3.1 Problem inferencji typów

Język ML przyjmuje wiele form, najpopularniejszymi wariantami są Standard ML (SML), OCaml, i F#. Na potrzeby będziemy wzorować się na [Damas and Milner(1982)], i posługiwać się ML-the-calculus, drastycznie uproszczoną wersją języka co pozwoli na dojście do sedna w problemie rekonstrukcji typów.

Termy w języku ML-the-calculus są następujące:

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

Typy które będziemy przypisywali do termów są następujące:

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

3.2 Udowadnianie typów dla ML-the-calculus

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

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

$$\frac{\Gamma \vdash e_1 : \tau_2 \to \tau \qquad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1 \ e_2 : \tau}$$

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

$$\frac{\Gamma \vdash e_1 : \tau' \qquad \Gamma, x : (\bigwedge \alpha_1, ..., \alpha_n.\tau') \vdash e_2 : \tau}{\Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau : B} (\{\alpha_1, ..., \alpha_n\} = ftv(\tau') \backslash ftv(\Gamma)) \tag{3}$$

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 ilość dowodów, nawet dla prostego wyrażenia $\lambda x.x$.

3.3 Infernencja typów bazująca na ograniczeniach

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

3.4 System typów Hindley–Milner

Algorytm inferencji typów przedstwiony w [Milner(1978)], oraz innych wcześniejszych publikacjach, opiera się na generowaniu ograniczeń i rozwiązywaniu ich w trakcie wykonywania prostego rekursywnego przejścia przez termy.

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

3.5 Algorytm J

Algorytm W nie ma żadnych efektów ubocznych, i świetnie zajmuje się aplikowanie i komponowaniem substytucji w odpowiedniej kolejności. Częste apikacje substytucji na wyrażeniach mogą znacznie zmniejszyć wydajność algorytmu rekonstruującegom dlatego też, Milner zaprezentował bardziej efektywną impereatywną wariacją W nazwaną algorytmem J w [Milner(1978)].

Algorytm J jest funkcją, która dla dokonanych do tej pory substytucji S, środowiska Γ - zbioru przechowywującego pary identyfikator wraz z typem, i

wyrażnia e, zwraca kolejne substytucje S, oraz typ wyrażenia τ.

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

$$\begin{aligned}
J(S,\Gamma,x) =& (S, [\beta_i/\alpha_i]\tau') \\
& \text{where } \Gamma(x) = \bigwedge \alpha_i, ..., \alpha_n.\tau' \\
& \text{and } \beta_i \text{ are fresh} \\
\\
J(S,\Gamma,e_1\ e_2) =& (V, \beta) \\
& \text{where } \Gamma(S_1, \tau_1) = J(S, \Gamma, e_1) \\
& \text{and } (S_2, \tau_2) = J(S_1, \Gamma, e_2) \\
& \text{and } V = unify'(\tau_1, \tau_2 \to \beta, S_2) \\
& \text{and } \beta \text{ is fresh} \\
\\
J(S,\Gamma,\lambda x.e) =& (S_1, \beta \to \tau) \\
& \text{where } (S_1, \tau) = J(S, (\Gamma, x : \beta), e) \\
& \text{and } \beta \text{ is fresh} \\
\\
J(S,\Gamma,\text{let } x = e_1 \text{ in } e_2) =& (S_2, \tau_w) \\
& \text{where } (S_1, \tau_1) = J(S, \Gamma, e_1) \\
& \text{and } (S_2, \tau_2) = J(S_1, (\Gamma, x : (\bigwedge \alpha_1, ..., \alpha_n.\tau_1)), e_2) \\
& \text{and } \{\alpha_1, ..., \alpha_n\} = ftv(S_2\tau_1) \backslash ftv(S_2\Gamma)
\end{aligned} \tag{4}$$

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

4 Rescript/ReasonML jako języki realizujące podobne zadania

5 Założenia i priorytety opracowanej aplikacji

Tworząc aplikację, chcieliśmy, by język posiadał podstawowe typy danych (liczby, stringi, wartość logiczna), kilka typów generycznych (funkcje, tablice), 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 mnie, by zaimplementować operatory binarane, oraz

unarne. Operatory te miały też spełniać ważną rolą w trakcie inferencji typów. W języku javascript operator '+' może być wykorzystywany do dodawania liczb jak i konkatenacji stringów, ważne więc było by stworzyć dwa oddzielne operatory.

Prymitywne typy danych i często wykonywane operacje:

Co	Przykład	wyjście 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

Chcieliśmy również by funkcje wieloargumentowe kompilowane były jako funkcje jednoargumentowe zwracające kolejne funkcje, co pozwala na wywoływanie funkcji bez wszystkich argumentów w celu zwrócenia funkcji przyjmującej resztę argumentów tzw. currying.

```
def classy_greeting(first_name, last_name) do
    `"The name's "` ++ last_name ++ ', ' ++ first_name ++ ' ' ++
    last_name
end

def compose1(f, g, a) do
    f(g(a))
end

def compose2(f, g, a, b) do
    f(g(a, b))
end

yell_greetings = compose2(to_upper, classy_greeting)
yell_greetings('James', 'Bond') # "THE NAME'S BOND, JAMES BOND"

compose1(compose1(abs, add(1)), multiply(2))(-4) # 7
```

examples/currying.uwu

```
const classy_greeting = (first_name) => (last_name) => {
  return "The name's " + last_name + ", " + first_name + " " +
    last_name;
};
const compose1 = (f) => (g) => (a) => {
  return f(g(a));
};
const compose2 = (f) => (g) => (a) => (b) => {
  return f(g(a)(b));
};
const yell_greetings = compose2(to_upper)(classy_greeting);
yell_greetings("James")("Bond");
compose1(compose1(abs)(add(1.0)))(multiply(2.0))(-4.0);
```

examples/currying.uwu.js

Jednym z ważniejszych elemenów każdego języka jest możliwość wykonywania różnego zbioru instrukcji, warunkowo. W tym celu planowałem zaimplementowanie instrukcji 'if', oraz 'case'. Instrukcja 'case' wykonywać ma dopasowanie do wzorca (tzw. pattern matching), wykonywać, odpowiedni zbiór instrukcji zależnie od wprowadzonych danych. Kompilator, powinien ostrzegać, jeżeli ścieżka dla jednego z typów danych nie została zaimplementowana.

- Przykład - funkcja łącząca dwie posortowane tablice

```
def merge<A>(a, b) do
    merge2: Callable<Array<A>, Callable<Array<A>, Array<A>>> =
     `merge`

    case Tuple(get_head(a), get_head(b)) of
        Tuple(Empty(), _) do b end
        Tuple(_, Empty()) do a end
        Tuple(Head(head_a, rest_a), Head(head_b, rest_b)) do
            if head_a < head_b then
                [head_a] | merge2(rest_a, b)
            else
                [head_b] | merge2(a, rest_b)
            end
        end
    end
end
```

examples/merge.uwu

```
const merge = (a) => (b) => {
  const merge2 = merge;
  return (() => {
    const $ = { TAG: "Tuple", _0: get_head(a), _1: get_head(b)
     };
    if (typeof $ !== "string" && $.TAG === "Tuple") {
      if ($._0 === "Empty") {
        const _ = $._1;
        return b;
      }
      if ($._1 === "Empty") {
        const _ = $._0;
        return a;
      }
      if (typeof $._0 !== "string" && $._0.TAG === "Head") {
        if (typeof $._1 !== "string" && $._1.TAG === "Head") {
          const head_b = $._1._0;
          const rest_b = $._1._1;
          const head_a = $._0._0;
          const rest_a = $._0._1;
          return (() => {
            if (head_a < head_b) {
              return [head_a].concat(merge2(rest_a)(b));
            }
            return [head_b].concat(merge2(a)(rest_b));
          })();
        }
        throw new Error("Non-exhaustive pattern match");
      }
      throw new Error("Non-exhaustive pattern match");
    }
    throw new Error("Non-exhaustive pattern match");
  })();
};
```

examples/merge.uwu.js

Kolejnym dość ważnym elementem języka jest brak wyrażenia ‘return’, które jest wykorzystywane do zwrócenia wartości z funkcji. Zamiast tego każdy bloku instrukcji powinien zwracać ostatnie wyrażenie. Pozwoli to na łatwiejsze inicjowanie zmiennych, w przypadku gdy inicializacja wymaga więcej niż jedenej linii kodu.

Przykład

```
message = if is_morning then
  'Good morning!'
else
  'Hello!'
end


result = do
    arr1 = [1, 2, 3]
    arr2 = map(arr1, add_one)

    filter(arr2, def is_even(x) do
        x % 2 == 0
    end)
end
```

examples/return.uwu

```
const message = (() => {
  if (is_morning) {
    return "Good morning!";
  }
  return "Hello!";
})();
const result = (() => {
  const arr1 = [1.0, 2.0, 3.0];
  const arr2 = map(arr1)(add_one);
  const is_even = (x) => {
    return x % 2.0 === 0.0;
  };
  return filter(arr2)(is_even);
})();
```

examples/return.uwu.js

5.1 Opis formaly składni języka

6 Narzędzia

6.1 Język python

Do implementacji programu 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 ast.

6.2 Parsowanie i tokenizowanie przy użyciu biblioteki sly

Biblioteka sly, jest pythonową implementacją narzędzi lex i yacc, wykorzystywanych do tworzenia parserów i kompilatorów. Tworzenie leksera i parsera jest bardzo proste.

6.2.1 Lexer

Tokeny są tworzone przy pomocy wyrażeń regularnych.

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

Biblioteka udostępnia specjalny syntax do tworznia tokenów, które są już opisane jako inny token.

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

Pojedyńcze znaki mogą być ignorowane, poprzez ustawienie zmiennej 'ignore'. Dodatkowo ignorowane są też tokeny zaczynające się od 'ignore'.

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

Istnieje również możliwość tworzenia tokenów o typie równym ich wartości, tak długo jak składają się tylko z jedngo znaku.

```
  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.

```
  import sly
  class Parser(sly.Parser):
  tokens = Lex.tokens

  @_("NUM")
  def expr(self, p):
  return p[0] # Lub p.ID

  @_("'(' expr AND expr ')'")
  def expr(self, p):
  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 programowanie wykorzystywanym w przeglądarkach internetowych. W 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 lexer

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

```
    NOT_EQUAL = r"!="
    EQUAL = r"=="
    STRING = r"'[^']*'"
    NUMBER = r"\d+"
    CONCAT = r"\+{2}"
    INT_DIV = r"/{2}"
    TYPE_IDENTIFIER = r"[A-Z\d][\w\d]*"
    IDENTIFIER = r"[a-z_][\w\d]*"
    IDENTIFIER["def"] = DEF
    IDENTIFIER["do"] = DO
    IDENTIFIER["end"] = END
    IDENTIFIER["if"] = IF
    IDENTIFIER["else"] = ELSE
    IDENTIFIER["elif"] = ELIF
    IDENTIFIER["case"] = CASE
    IDENTIFIER["enum"] = ENUM
    IDENTIFIER["then"] = THEN

    IDENTIFIER["of"] = OF
    EXTERNAL = r"`[^`]*`"
```

parser.py

Nasz lexer zaimował się również będzie zwracaniem tokena dla znaku nowej linii

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

parser.py

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

```
    ignore_comment = r"\#.*"
    ignore = " \t"
```

parser.py

Pozatym Nasz lexer zwracał będzie poniższy zbiór tokenów

```
    literals = {
        "=",
        ".",
        "[",
        "]",
        ",",
        "{",
        "}",
        "(",
        ")",
        ":",
        "+",
        "-",
        ">",
        "<",
        "*",
        "/",
        "|",
        "%",
    }
```

parser.py

7.2 parser

Implementacja parsera jest bardzo prosta, każda z metod zajmuje jedynie linijkę, gdzie zwracane jest odpowiedne wyrażenie drzewa ast.

```
    def binary_expr(self, p):
        return terms.EBinaryExpr(p[1], p[0], p[2])

    @_(
        "DO [ ':' type ] [ NEWLINE ] [ do_exprs ] END",
    )
    def do(self, p):
        return terms.EDo(p.do_exprs or [], hint=terms.EHint.
    from_option(p.type))
```

parser.py

W przypadku zbiorów wyrażeń, zwracane są listy.

```
    @_("exprs ',' [ NEWLINE ] expr [ NEWLINE ]")
    def exprs(self, p):
        return p.exprs + [p.expr]

    @_("expr [ NEWLINE ]")
    def exprs(self, p):
        return [p.expr]
```

parser.py

7.3 Drzewo decyzyjne

Wykorzystywany algorytm jest odbrobinę inny od algorytmów opisanych w literaturze. Bierzemy pod uwagę następujące obserwacje. Część algorytmów w literaturze w celu uniknięcia wykładniczego wzrostu generowanego kodu, generuje zbędne testy [Augustsson(1985)]. Wykładniczy wzrostu generowanego jednak praktycznie nie wsystępuje w praktyce [Scott and Ramsey(2000)]. Najlepszą praktyką wieć wydało nam się by nigdy nie generować niepotrzebnych sprawdzeń, i spróbować uninkąć duplikacji kodu używając heurystyki tak jak [Maranget(2008)]. Literatura udowadnia, że w przypadku praktycznie napisanego kodu, różne algorytmy generują prawie identycny kod [Scott and Ramsey(2000), Maranget(2008)]. Naszą metodą więc jest, aby: (1) zawsze skupiać się na przypasowaniu pierszego przypadku, by uniknąć niepotrzebnych testów i (2) zachłannie spróbować zminimalizować duplikację przy użyciu heurystyki.

Pattern matching zamieniany jest w drzewo decyzyjne przy pomocy rekurencyjnej funkcji. Rekursja ma dwa podstawowe przypadki:

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

```
def gen_match2(clauses1: typing.Sequence[Clause]) -> CaseTree:

    clauses = [subst_var_eqs(clause) for clause in clauses1]

    match clauses:
        case []:
            return MissingLeaf()
        case [(patterns, body), *_] if not patterns:
            return Leaf(body)
```

case_tree.py

W innym przypadku wybieramy wzorzec przy pomocy heurystyki

```
        case [(patterns, body), *_]:
            branch_var = branching_heuristic(patterns, clauses)
            branch_pattern = patterns[branch_var]
            yes = list[Clause]()
            no = list[Clause]()

            #
            vars = [f"{branch_var}._{i}" for i in range(len(
    branch_pattern.patterns))]
```

case_tree.py

Następnie tworzymy dwa podproblemy yes i no iterując przez wszsytki przypadki. Dla każdego z nich robimy jedną z trzech rzeczy:

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

```
            for patterns, body in clauses:
                clause = Clause(dict[str, terms.Pattern](
    patterns), body)
                match patterns.get(branch_var, None):
                    case None:
                        yes.append(clause)
                        no.append(clause)
```

case_tree.py

- Przpadek zawiera szukany wzorzec, więc dodajemy go do yes

```
                    case terms.EMatchVariant(id, patterns) if
    id == branch_pattern.id:
                        yes.append(
                            dataclasses.replace(
                                clause,
                                patterns={
                                    key: value
                                    for key, value in clause.
    patterns.items()
                                    if key != branch_var
                                }
                                | dict(zip(vars, patterns)),
                            )
                        )
```

case_tree.py

- Przypadek zawiera wzorzec, więc dodajemy go do no

```
                    case terms.EMatchVariant():
                        no.append(clause)
```

case_tree.py

Rekursywnie generujemy kod dla yes i no i zwracamy gałąź decyzyjną

```
            return Node(
                branch_var, branch_pattern.id, vars, gen_match2(yes
    ), gen_match2(no)
            )
```

case_tree.py

7.4 inferencja typów

7.4.1 Context

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

```
counter = 0


def fresh_ty_var(kind=typed.KStar()) -> typed.TVar:
    global counter
    counter += 1
    return typed.TVar(counter, kind)
```

algorithm_j.py

Substytucje to pary zmiennych i typów

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

algorithm_j.py

Jednym z argumentów algorytmu J jest środowiko, w naszej implementacji reprezentowane jest one przez Context, będący mapą identyfikatorów i schem

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

algorithm_j.py

Gdzie Schema to zawieraja informację o typie, oraz listę występujących w nim zmiennych typów

```
@dataclasses.dataclass
class Scheme:
    vars: list[int]
    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->b funkcja zwraca Num -> b

```python
def apply_subst(subst: Substitution, ty: typed.Type) -> typed.Type:
    match ty:
        case typed.TVar(var):
            return subst.get(var, ty)
        case typed.TAp(arg, ret):
            return typed.TAp(apply_subst(subst, arg), apply_subst(
    subst, ret))
        case typed.TCon():
            return ty
        case _:
            raise TypeError(f"Cannot apply substitution to {ty=}")
```

algorithm_j.py

Tworzymy funkcję, unifikującą dwa typy

```python
def unify(a: typed.Type, b: typed.Type) -> Substitution:
    match (a, b):
        case (typed.TCon(), typed.TCon()) if a == b:
            return {}
        case (
            typed.TAp(arg0, ret0),
            typed.TAp(arg1, ret1),
        ):
            subst = unify_subst(arg0, arg1, {})
            subst = unify_subst(ret0, ret1, subst)

            return subst
        case (typed.TVar(u), t) | (t, typed.TVar(u)) if typed.kind(
    a) != typed.kind(b):
            raise UnifyException(f"Kind for {a=} and {b=} does not
    match")
        case (typed.TVar(u), t) | (t, typed.TVar(u)):
            return var_bind(u, t)
        case _:
            raise UnifyException(f"Cannot unify {a=} and {b=}")
```

algorithm_j.py

wraz z funkcją generującą substytucje w przypadku unifikacji zmiennego typu

```python
def var_bind(u: int, t: typed.Type) -> Substitution:
    match t:
        case typed.TVar(tvar2) if u == tvar2:
            return {}
        case typed.TVar(_):
            return {u: t}
        case t if u in free_type_vars(t):
            raise TypeError(f"circular use: {u} occurs in {t}")
        case t:
            return {u: t}
```

algorithm_j.py

Teraz zaczynamy implementację algorytmu J.

7.4.3 Literały

W przypadku literałów typ, jest oczywisty.

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

algorithm_j.py

7.4.4 Identyfikatory

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

```
        case terms.EIdentifier(var):
            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.

```
        case terms.EProgram(body) | terms.EBlock(body):
            ty = typed.TUnit()
            ctx = ctx.copy()

            for exp in body:
                subst, ty = infer(subst, ctx, exp)

            return subst, ty
```

algorithm_j.py

7.4.6 Operatory binarne

Implementacja operatorów binarnych są bardzo podobne.

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

```
        case terms.EBinaryExpr("|", left, right):
            ty = typed.TArray(fresh_ty_var())
            subst, ty_left = infer(subst, ctx, left)
            subst = unify_subst(ty_left, ty, subst)
            subst, ty_right = infer(subst, ctx, right)
            subst = unify_subst(ty_right, ty, subst)
            return subst, ty
```

algorithm_j.py

7.4.7 Wywowałnia funkcji

W przypadku wywołania funkcji musimy zaimplementować currying.
Przykład: $J(fn) = Num \to Str \to Unit \to Arr < Str >$
$J(args) = [Num, Str]$

- 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 $tmp = Num \to Str \to \alpha$
- Unifikujemy typy tmp i $J(fn)$ co tworzy substytucję $\{\alpha : Unit \to Arr < Str >\}$
- I zwracamy α

```
        case terms.ECall(fn, args):
            subst, ty_fn = infer(subst, ctx, fn)

            ty_args = list[typed.Type]()

            for arg in reversed(args):
                subst, ty_arg = infer(subst, ctx, arg)
                ty_args.append(ty_arg)

            ty = fresh_ty_var()
            subst = unify_subst(ty_fn, reduce_args(ty_args, ty),
    subst)

            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 tą typ.

```
        case terms.EIf(test, then, or_else, hint=hint):

            subst, hint = infer(subst, ctx, hint)
            subst, ty_condition = infer(subst, ctx, test)
            subst = unify_subst(ty_condition, typed.TBool(), subst)

            subst, ty_then = infer(subst, ctx, then)
            subst = unify_subst(ty_then, hint, subst)

            subst, ty_or_else = infer(subst, ctx, or_else)
            subst = unify_subst(ty_or_else, hint, subst)

            return subst, hint
```

algorithm_j.py

7.4.9 Definicje funkcji

W przypadku definicje funkcji musimy upenić się, że typy generyczne, są przypiswywane od contextu. Musimy sprawdzić, czy typ zwracany z funkcji jest przypisywalny do zadeklarowanego typu, oraz zapisać funkcję do contextu.

```
        case terms.EDef(id, params, body, hint, generics):
            t_ctx = ctx.copy()

            for generic in generics:
                ty_generic = fresh_ty_var()
                t_ctx[generic.name] = Scheme([], ty_generic)

            subst, hint = infer(subst, t_ctx, hint)

            ty_params = list[typed.Type]()

            for param in reversed(params):
                subst, ty_param = infer(subst, t_ctx, param)
                ty_params.append(ty_param)

            ty = reduce_args(ty_params, hint)

            #
            subst, ty_body = infer(subst, t_ctx, body)

            subst = unify_subst(ty_body, hint, subst)

            ctx[id] = Scheme.from_subst(subst, ctx, ty)

            return subst, ty
```

algorithm_j.py

7.5 Pattern mathing

W trakcie pattern matching, musimy upewnić się, że konstruktory, posiadają odpowiednią ilość argumentów. Dodatkowo przeprowadzać będziemy sprawdzanie

czy wyrażenie case jest wyczerpujące - przewiduje wszystkie przypadki. W tym celu funkcja przyjmuję argument `o_alts` będący bazą, zmiennych i ich kontruktorów.

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

```
def infer_case_tree(
    subst: Substitution,
    ctx: Context,
    tree: case_tree.CaseTree,
    o_alts: dict[str, list[str]] | None = None,
) -> tuple[Substitution, typed.Type]:
    alts = o_alts or {}

    match tree:
        case case_tree.MissingLeaf():
            if any(alts.values()):
                raise NonExhaustiveMatchException()
            return subst, fresh_ty_var()
```

algorithm_j.py

W przypadku gałęzi decyzyjnej inferujemy typ zmiennej i sprawdzanego konstruktora

```
        case case_tree.Node(var, pattern_name, vars, yes, no):

            subst, ty_var = infer(subst, ctx, terms.EIdentifier(var
    ))  # x | x.0
            subst, ty_pattern_name = infer(
                subst, ctx, terms.EIdentifier(pattern_name)
            )  # Some() | None()
            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 kontruktorów konstruktor do którego odnosi się nasza gałąź decyzjna.

```
            if var not in alts:
                alts[var] = alternatives(ty_pattern_name)

            if pattern_name in alts[var]:
                alts[var].remove(pattern_name)
```

algorithm_j.py

Kolejna część inferuje typy zmiennych dla każdego z argumentów konstruktora. Oraz, tworzy dla nich zmienne na kontekście.

```
            ty_vars = list[typed.Type](fresh_ty_var() for _ in vars
    )

            pattern_name_con = typed.TCon(
                pattern_name,
                functools.reduce(
                    flip(typed.KFun),
                    map(typed.kind, ty_vars),
                    typed.KStar(),
                ),
            )

            subst = unify_subst(
                ty_pattern_name,
                typed.TDef(
                    functools.reduce(
                        typed.TAp,
                        ty_vars,
                        pattern_name_con,
                    ),
                    ty_var,
                ),
                subst,
            )

            for var2, ty_var in zip(vars, ty_vars):
                t_ctx[var2] = Scheme.from_subst(subst, t_ctx,
    ty_var)
```

algorithm_j.py

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

```
            ty = fresh_ty_var()

            subst, ty_yes = infer_case_tree(
                subst,
                t_ctx,
                yes,
                {key: value for key, value in alts.items() if key
    != var}
                | {var2: alternatives(t_ctx[var2].ty) for var2 in
    vars},
            )
            subst = unify_subst(ty_yes, ty, subst)

            subst, ty_no = infer_case_tree(subst, ctx, no, alts)
            subst = unify_subst(ty_no, ty, subst)

            return subst, ty
```

algorithm_j.py

7.6 Windowanie (Hoisting)

Ze względu, na to, że deklaracje zmiennych w javascript `const x = 1` w przeciwieństwie do naszego języka nie zwracają przypisywanej do indentyfikatora wartości, wyrażenia typy `f(a = b = c)` muszą być przekonwertowane do

```
b=c
a=b
f(a)
```

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

W przypadku bloków wyrażeń, listy z wyrażeniami do przeniesienia wyżej są łączone z listą wyrażeń danego bloku.

```
def hoist(
    node: terms.EProgram | terms.EBlock,
) -> terms.EProgram | terms.EBlock:
    match node:
        case terms.EBlock(body):
            body2 = hoist_expr_list(body)
            return terms.EBlock(filter_identifiers(body2[:-1:]) +
    body2[-1::])

        case terms.EProgram(body):
            body2 = hoist_expr_list(body)
            return terms.EProgram(filter_identifiers(body2))
```

compile.py

```
def hoist_expr_list(body: list[terms.Expr]) -> list[terms.Expr]:
    body2 = list[terms.Expr]()

    for expr in body:
        let, expr2 = hoist_expr(expr)
        body2.extend(let)
        body2.append(expr2)

    return body2
```

compile.py

W przypadku deklaracji zmiennych i funkcji, dodajemy elementy do listy wyrażeń do przeniesienia wyżej.

```
def hoist_expr(node: terms.Expr) -> tuple[list[terms.Expr], terms.Expr]:
    match node:
        case terms.ELet(id, init):
            let, init2 = hoist_expr(init)

            return let + [dataclasses.replace(node, init=init2)], terms.EIdentifier(id)
        case terms.EDef(id, params, body, hint):

            return [
                terms.EDef(id, params, body=hoist_do(body), hint=hint)
            ], terms.EIdentifier(id)
```

compile.py

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

```
        case terms.EIf(test, then, terms.EIf() | terms.EIfNone() as or_else):
            let_test, test2 = hoist_expr(test)
            let_or_else, or_else2 = hoist_expr(or_else)
            return let_test + let_or_else, dataclasses.replace(
                node,
                test=test2,
                then=hoist(then),
                or_else=or_else2,
            )
```

compile.py

7.7 Kompilacja

Kompilacja opiera się na jednej funkcji zamieniającej, ast na kod javascript w postaci stringa.

7.8 Literały

Jak widać, dla literałów jest to dość trywialne.

```
def _compile(exp: terms.AstTree) -> str:
    match exp:
        case terms.EExternal(value=value):
            return f"{value}"
        case terms.ELiteral(value=str()):
            return f'"{exp.value}"'
        case terms.ELiteral(value=float()):
            return f"{exp.value}"
```

compile.py

7.8.1 Wyrażenia warunkowe

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

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

compile.py

7.8.2 Warianty

Warianty, przyjmują postać obiektów z dyskryminatorem

```
        case terms.EVariantCall(id, args) if args:
            args = [f"_{i}:{_compile(arg)}" for i, arg in enumerate
    (args)]

            return f"{{TAG:'{id}',{','.join(args)}}}"
```

compile.py

7.8.3 Pattern matching

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

```
def _compile_case_tree(tree: case_tree.CaseTree):
    match tree:
        case case_tree.Leaf(body):
            return _compile(terms.EBlock(body.body))
        case case_tree.MissingLeaf():
            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.

```
        case case_tree.Node(var, pattern_name, vars, yes, no):
            conditions = []

            if pattern_name == "True":
                conditions.append(f"{var}")
            if pattern_name == "False":
                conditions.append(f"!{var}")
            elif vars:
                conditions.insert(0, f"{var}.TAG==='{pattern_name}'
    ")
                conditions.insert(0, f"typeof {var} !== 'string'")
            else:
                conditions.insert(0, f"{var}==='{pattern_name}'")

            # if isinstance(no, case_tree.MissingLeaf):
            #     return _compile_case_tree(yes)

            return f"if({'&&'.join(conditions)}){{{
    _compile_case_tree(yes)}}}{_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 roszerzeniem '.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 ast.

8.2.2 Pattern matching

Pattern matching mógłby wyświetlać ostrzeżenie w momencie, gdy jedna ze ścieżek jest nieosiągalna.

9 [Podsumowanie]

10 [spisy – rysunków, tabel, listingów ipt.]

References

[Aho et al.(1985)Aho, Sethi, and Ullman] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. *Compilers*. Addison Wesley, Boston, MA, January 1985. ISBN 9780201101942.

[Augustsson(1985)] Lennart Augustsson. Compiling pattern matching. In *Proc. of a conference on Functional programming languages and computer architecture*, Berlin , Heidelberg, January 1985. Springer-Verlag.

[Damas and Milner(1982)] Luis Damas and Robin 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.

[J.E. Hopcroft(2005)] J.D. Ullman J.E. Hopcroft, R. Motwani. *Wprowadzenie do Teorii automatów, języków i obliczń*. Wydawnictwo Naukowe PWN, 2005. ISBN 8301145021.

[Maranget(2008)] Luc Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the 2008 ACM SIGPLAN workshop on ML*, pages 35–46, USA, NY, September 2008. Association for Computing Machinery.

[Milner(1978)] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Pierce(2002)] Benjamin C Pierce. *Types and Programming Languages*. The MIT Press. MIT Press, London, England, January 2002. ISBN 0262162091.

[Scott and Ramsey(2000)] Kevin Scott and Norman Ramsey. When do match-compilation heuristics matter? Technical report, University of Virginia, USA, 2000.