Tytuł

Patryk Wałach

January 2022

Contents

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:

```
1  [(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 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.

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:

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

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

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

3 Inferencja typów w teori

3.1 System typów Hindley–Milner

3.2 Algorytm J

Algorytm W nie ma żadnych side-effectów, i świetnie zajmuje się aplikowanie i komponowaniem substytucji w odpowiedniej kolejności. Te powtórne apikacje substytucji na wyrażeniach mogą znacznie zmniejszyć wydajność algorytmu rekomntrującegom 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 - zbioru przechowywującego pary identyfikator wraz z typem Γ i wyrażnia e, zwraca kolejne substytucje S, oraz typ wyrażenia τ.

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

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

$$\begin{aligned} J(S,\Gamma,e_1e_2) =& (V, \beta) \\ & \text{where } \Gamma(S_1,\tau_1) = J(S,\Gamma,e_1) \\ & \text{and } (S_2,\tau_2) = J(S_1,\Gamma,e_2) \\ & \text{and } V = unify'(\tau_1, \tau_2 \to \beta, S_2) \\ & \text{and } \beta \text{ is fresh} \end{aligned}$$

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

$$\begin{aligned} J(S,\Gamma,\text{let } x = e_1 \text{ in } e_2) =& (S_2, \tau_w) \\ & \text{where } (S_1,\tau_1) = J(S,\Gamma,e_1) \\ & \text{and } (S_2,\tau_2) = J(S_1,(\Gamma, x : (\bigwedge \alpha_1, ..., \alpha_n.\tau_1)), e_2) \\ & \text{and } \{\alpha_1, ..., \alpha_n\} = ftv(S_2\tau_1) \backslash ftv(S_2\Gamma) \end{aligned} \tag{1}$$

Funkcja pomocnicza $unify'(\tau, \tau', S)$ rozszerza zbiór substytucji S o substytucje wynikające z unifikacji τ z τ' pod kontekstem S.

4 Rescript/ReasonML jako języki realizujące podobne zadania

5 Założenia i priorytety opracowanej aplikacji

Tworząc aplikację, chciałem, 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:

- string

```
1 greeting = 'Hello world!'
```

examples/string.uwu

  Do konkatenacji stringów służy operator `++`

- Wartość logiczna ma typ `Bool` i wartość `True` lub `False`. Powiązane operacje:
  - `<>`, równość pomiędzy dwiema liczbami
  - `>`, `<`
  - `!=` równość
- liczby Powiązane operacje: `+`, `-`, `*`, `/`, `*`, `%`, `//`

Chciałem 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

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

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

```
```

examples/return.uwu.js

5.1 Opis formaly składni języka

6 Narzędzia

6.1 Język python

6.2 Parsowanie i tokenizowanie przy użyciu biblioteki sly

6.3 Środowisko nodejs do uruchomienia skompilowanego kodu

7 Implementacja

7.1 lexer

7.2 parser

7.3 inferencja typów

7.4 kompilacja

8 Opis działania

8.1 Co działa

8.2 Uwagi co do obsługi błędów

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.

[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.

[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.