Generowanie dobrze typowanych parserów w stylu kontynuacyjnym

(Generating well-typed parsers in continuation-passing style)

Adam Szeruda

Praca licencjacka

Promotor: Piotr Polesiuk

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

21 czerwca 2023

Streszczenie

Powszechnie stosowane generatory parserów, `ocamlyacc` i `menhir`, tworzą kod oparty o algorytm LR. Algorytm ten realizowany jest przez automat ze stosem, na którym przechowywane są wartości różnych typów — w językach silnie typowanych jest to problematyczne. W pracy tej przedstawiona zostaje metoda implementacji algorytmu LR wykorzystująca styl kontynuacyjny, która dobrze współgra z systemem typów. Omówionej metodzie towarzyszy przykładowa implementacja generatora parserów.

Two commonly used parser generators, `ocamlyacc` and `menhir`, produce code based on the LR algorithm. This algorithm is implemented as a pushdown automaton and requires storing values of different types on a stack, which is problematic in strongly typed languages. This thesis introduces a method of implementing the LR algorithm that utilizes a continuation-passing style and works well with the type system. The presented method is exemplified by an implementation of a parser generator.

Spis treści

Rozdział 1.

Wprowadzenie

Analiza składniowa (zwana też parsowaniem), mimo że niepozorna, jest jednym z najistotniejszych etapów działania niejednego programu. Rozwiązuje ona problem przeczytania pewnej struktury, którą program dostaje jako ciąg znaków. Jest ona obecna m. in. w pierwszym etapie kompilowania kodu, przy wczytywaniu pliku konfiguracyjnego wymaganego przez aplikację czy podczas odbierania danych przesyłanych przez internet. Napisanie parsera odręcznie jest jednak zadaniem bardzo żmudnym i uciążliwym, szczególnie dla skompilowanych języków — stąd też wynika powstanie generatorów parserów. W Ocaml-u do najpopularniejszych należą `ocamlyacc`[1] i `menhir`[2].

Kod generowany zarówno przez `ocamlyacca`, jak i `menhira` ma jednak pewną wadę. Tworzone przez nie parsery są automatem ze stosem, na którym przechowywane są wartości różnych typów. W silnie typowanym języku jakim jest Ocaml implementacja takiego stosu jest problematyczna, a wspomniane generatory stosują rozwiązania, które nie są zadowalające. Na przykład, `ocamlyacc` wymazuje typy przechowywanych wartości:

```
fun __caml_parser_env ->
    let _1 = (Parsing.peek_val __caml_parser_env 2 : int) in
    let _3 = (Parsing.peek_val __caml_parser_env 0 : int) in
    Obj.repr((action _1 _3) : int)
```

Przytoczony kod pochodzi bezpośrednio z generowanego przez `ocamlyacc` programu. Zmienna `__caml_parser_env` przechowuje stos typu `Obj.t array`[1]. Funkcja `Parsing.peek_val` zwraca typ `'a`, a wewnętrznie korzysta z `Obj.magic`[2] — oba miejsca oszukują system typów. Całkiem inne rozwiązanie problemu stosu wykorzystuje `menhir`. Użyte zostały uogólnione algebraiczne typy danych (*GADT*) [5], które pozwalają wyeliminować wszelkie użycia `Obj.repr`:

[1] https://github.com/ocaml/ocaml/blob/5.0/stdlib/parsing.ml#L24
[2] https://github.com/ocaml/ocaml/blob/5.0/stdlib/parsing.ml#L186

```
type ('s, 'r) _menhir_state =
    | MenhirState00 : ('s, _menhir_box_a) _menhir_state
    | MenhirState01 : (('s, _menhir_box_a) _menhir_cell1_A,
       _menhir_box_a) _menhir_state
    | MenhirState03 : ((('s, _menhir_box_a) _menhir_cell1_A,
       _menhir_box_a) _menhir_cell1_b) _menhir_state
    ...
```

Typ `_menhir_state` w powyższym kodzie to właśnie GADT. GADT należą do zaawansowanych cech systemów typów, a wiele innych języków w ogóle ich nie wspiera, stąd też mała przenośność tej metody. Kolejnym problemem jest inferencja typów — w programach korzystających z GADT jest ona nierozstrzygalna [7]. Z tego właśnie powodu `menhir` wymaga wcześniejszej znajomości wszystkich typów, które mogą pojawić się w programie.

Niniejsza praca przedstawia alternatywną metodę generowania parserów bazującą na stylu kontynuacyjnym. Parsery generowane w tym stylu w prosty sposób naśladują działanie algorytmu LR z wielotypowym stosem. Podejście takie umożliwia tworzenie miej skompilowanych programów, które korzystają z dobrze wspieranych cech języka, używają prostych typów i oferują lepszą ich inferencję. Generowany kod jest jednocześnie czytelniejszy i umożliwia zastosowanie dodatkowych optymalizacji przez kompilator. Omówionej metodzie towarzyszy przykładowa implementacja programu podobnego do `ocamlyacca` i `menhira`, który generuje kod w stylu kontynuacyjnym. Implementacja ta daje obiecujące wyniki testów wydajności.

Rozdział 2.

Algorytm parsowania LR(k)

Rozdział ten stanowi krótkie przypomnieniem pojęć dotyczących gramatyk bezkontekstowych, algorytmu LR(k) i budowy automatu LR(k). Dokładniejsze ich omówienie można znaleźć w 4 rozdziale książki *Compilers: Principles, Techniques, and Tools* (*2nd Edition*) [4].

Wygodnym i powszechnie przyjętym narzędziem do opisu parsowanych języków są gramatyki bezkontekstowe. Generatory parserów tworzą kod wczytując opis takiej właśnie gramatyki, na który składa się zbiór symboli terminalnych i nieterminalnych, a także zestaw produkcji i wyróżniony symbol startowy. Elementy gramatyki będą oznaczane w następujący sposób:

- A, B, S — symbole nieterminalne,
- a, b — symbole terminalne,
- s — symbol gramatyki, terminalny lub nieterminalny,
- u, v — słowa symboli terminalnych,
- α, β, γ — słowa symboli gramatyki.

Napis $|\alpha|$ oznaczać będzie długość słowa α. Dodatkowym założeniem o gramatykach będzie, że ich symbolem startowym jest S', a jedyną produkcją z tego symbolu jest $S' \to S$. Produkcja ta będzie pomijana.

Parsery generowane przez `ocamlyacca` i `menhira` działają w oparciu o algorytm LR. Przeprowadzają one analizę wstępująca, która polega na stopniowym redukowaniu słowa wejściowego do symbolu startowego, w każdym kroku zamieniając pewne słowo α na nieterminal A, zgodnie z produkcją $A \to \alpha$. Strategia taka realizowana jest poprzez stopniowe wczytywanie symboli z wejścia i dodawanie ich na stos, w międzyczasie redukując szczyt stosu. Działanie to śledzi prawostronne wyprowadzanie, stąd też nazwa algorytmu (***l**eft to right, **r**ightmost derivation*).

2.1. Działanie algorytmu LR(k)

Algorytm parsowania LR(k) oparty jest o deterministyczny automat ze stosem. Oprócz zbioru stanów $\mathcal{Q}$, korzysta on z dwóch funkcji, GOTO i ACTION. Funkcja GOTO określa przejścia między stanami etykietowane symbolami gramatyki, tzn. dla danego stanu i symbolu zwraca ona nowy stan. Funkcja ACTION określa decyzje podejmowane przez automat — dla danego stanu i dla słowa długości nie większej niż k (zwanego *lookahead*), zwraca jedno z *shift*, *reduce* $A \to \alpha$, *accept* i *error*, gdzie $A \to \alpha$ jest dowolną produkcją.

Automat LR(k) wczytuje słowo w, a na prawie całym stosie przechowuje pary ⟨stan, wartość semantyczna⟩, tylko na jego dnie trzymając stan początkowy bez wartości semantycznej. Automat rozpoczyna ze stosem zawierającym stan początkowy, a będąc w stanie I (który jest na szczycie stosu) podgląda słowo v długości k z początku wejścia (lub krótsze, gdy napotka jego koniec). W zależności od $\mathrm{ACTION}[I, v]$ podejmuje decyzje:

- *shift* a: wczytuje symbol a z początku wejścia, a na stos dodaje wartość semantyczną wczytanego symbolu i stan $\mathrm{GOTO}[I, a]$, do którego przechodzi;
- *reduce* $A \to \alpha$: zdejmuje $|\alpha|$ wartości ze stosu (są tam wartości semantyczne słowa α), co powoduje, że na szycie stosu znajduje się stan I'. Na podstawie zdjętych wartości automat wylicza wartość semantyczną symbolu A, po czym dodaje tę wartość i stan $\mathrm{GOTO}[I', A]$ na szczyt stosu. Na koniec automat przechodzi do stanu właśnie dodanego na stos.
- *accept*: akceptuje wejście i zwraca **jedyną** wartość semantyczną ze stosu;
- *error*: odrzuca wejście jako niepoprawne.

Konstrukcja funkcji ACTION gwarantuje poprawność powyższego algorytmu (w tym odpowiednią wielkość stosu przy akcji *reduce* i jedyność wartości przy akcji *accept*).

2.2. Budowa automatu LR(k)

W dalszej części pracy potrzebne będą niektóre ze szczegółów konstrukcji automatu LR(k), a mianowicie znajomość jego stanów i dokładnego działania funkcji GOTO. Stanem automatu LR(k) jest zbiór sytuacji LR(k), które przechowują informacje o wszystkich możliwych do wykonania redukcjach:

Definicja 2.1. *Sytuacja LR(k)* (ang. *LR(k) item*) to krotka $\langle A, \alpha, \beta, u\rangle$, gdzie $A \to \alpha\beta$ jest produkcją gramatyki, a u (*lookahead*) jest słowem symboli terminalnych nie dłuższym niż k. Sytuację taką zwyczajowo zapisuje się jako $A \to \alpha \cdot \beta\ /u$.

Gdy automat jest w stanie zawierającym sytuację $A \to \alpha \cdot \beta\ /u$, to na szczycie stosu jest słowo α, a gdy znajdzie się tam jeszcze β, to szczyt stosu $\alpha\beta$ będzie

można zredukować do A. W konstrukcji stanów LR(k) istotna jest funkcja domknięcia, CLOSURE, która uzupełnia dany zbiór o „rozpoczynające" się w nim sytuacje, oraz funkcja przejścia, GOTO:

Definicja 2.2. $\mathrm{CLOSURE}(I)$ to najmniejszy zbiór sytuacji taki, że $I \subseteq \mathrm{CLOSURE}(I)$ i dla każdej sytuacji $A \to \alpha \cdot B\beta\ /u$ znajdującej się w $\mathrm{CLOSURE}(I)$ oraz każdej produkcji $B \to \gamma$, w $\mathrm{CLOSURE}(I)$ jest też sytuacja $B \to \ \cdot \gamma\ /u'$. Słowo u' jest dokładnie określone przez metodę wybraną do konstrukcji automatu, np. LR(k) [4, s. 261] lub LALR(k) [4, s. 272].

Definicja 2.3. $\mathrm{GOTO}[I, s] = \mathrm{CLOSURE}\left(\left\{A \to \alpha s \cdot \beta\ /u \ \middle|\ (A \to \alpha \cdot s\beta\ /u) \in I\right\}\right)$

W definicji 2.2 zależność słowa u' od reszty zbioru została celowo pominięta — jest ona różna dla metod LR(k), SLR(k) i LALR(k). Dla wszystkich tych metod wspólne jednak są pozostałe, opisane części definicji — wystarczają też one do dobrego opisania rozważanej później metody. Funkcja CLOSURE ściśle związana jest z pojęciem jądra stanu LR(k):

Definicja 2.4. $\mathrm{KERNEL}(I)$ (*jądro*) to podzbiór I, który zawiera wszystkie jego sytuacje postaci $A \to \alpha \cdot \beta$, gdzie $|\alpha| > 0$, oraz $S' \to \ \cdot S$, gdzie S' jest symbolem startowym.

Wszystkie sytuacje spoza jądra są postaci $A \to \ \cdot \beta$, są to więc dokładnie sytuacje dodawane przez funkcję CLOSURE. Każdy stan I automatu spełnia zależność $I = \mathrm{CLOSURE}(\mathrm{KERNEL}(I))$ — zależność ta widoczna jest już w definicji 2.3, gdzie jądrem stanu $\mathrm{GOTO}[I, s]$ jest właśnie argument funkcji CLOSURE. Wprowadzone tak pojęcia pozwalają już określić zbiór stanów automatu LR(k):

Definicja 2.5. (*Kanoniczny*) *zbiór stanów LR*(k) to zbiór $\mathcal{Q}$ taki, że zawiera on stan początkowy $\mathrm{CLOSURE}(\{S' \to \ \cdot S\})$ i jest zamknięty na przejścia funkcją GOTO.

Rozdział 3.

Podejście kontynuacyjne

3.1. Styl kontynuacyjny

Styl kontynuacyjny jest metodą pisania programów, która daje większą kontrolę nad przepływem sterowania. W stylu tym każda funkcja, oprócz zwykłych argumentów, przyjmuje także jedną lub więcej *kontynuacji*. Po zakończeniu obliczeń sterowanie jawnie przekazywane jest właśnie jednej z nich, a nie funkcji wywołującej, jak ma to miejsce w stylu bezpośrednim. Wywołanie zarówno kontynuacji, jak i funkcji w stylu kontynuacyjnym zawsze jest ogonowe. Rozważmy poniższy program napisany w stylu kontynuacyjnym:

```
1 let rec factorial n ok err =
2     if n < 0 then err ()
3     else if n = 0 then ok 1
4     else factorial (n - 1) (fun x -> ok (x * n)) err
```

Przedstawiona funkcja wylicza silnię danej liczby `n`. Oprócz tej liczby przyjmuje ona dwie kontynuacje — `ok` i `err`, które wywoływane są po zakończeniu obliczeń odpowiednio sukcesem (np. w wierszu 3) i błędem (w wierszu 2). W ostatnim wierszu programu pojawia się rekurencyjne wywołanie funkcji. Definiowana jest tam nowa kontynuacja, której zadaniem jest zmodyfikowanie wyniku obliczeń wywołania rekurencyjnego i zakończenie działania z nową wartością `x * n`. Kontynuacja `err` jest dalej przekazana bez zmian.

Kontynuacje `ok` i `err` w omawianym przykładzie pozwalają jawnie wskazać właściwy koniec działania funkcji. Zaletą stylu kontynuacyjnego jest właśnie możliwość zakończenia obliczeń na wiele sposobów (każdy z nich to jednak kontynuacja), a każdy ze sposobów może wyprodukować dowolną liczbę wartości dowolnych typów (są to argumenty przyjmowane przez kontynuacje).

W stylu bezpośrednim wywoływane funkcje tworzą stos — funkcja zostaje dodana na szczyt stosu w momencie rozpoczęcia obliczeń, a zdjęta po zakończeniu.

W stylu kontynuacyjnym zakończenie obliczeń zawsze oznacza wywołanie kontynuacji — jest to więc sposób na „zdjęcie" funkcji ze stosu wywołań. Warto jednak zauważyć, że funkcja nie musi sama wywoływać przekazanych jej kontynuacji, ale może przekazać je dalej. Wywołanie takiej kontynuacji głębiej w drzewie obliczeń umożliwia zdjęcie ze stosu więcej niż jednego elementu, aż do miejsca definicji tej kontynuacji. Obserwacja ta okazuje się przydatna przy implementacji niektórych algorytmów, w tym algorytmu parsowania LR.

3.2. Przepływ sterowania w algorytmie LR

Stany automatu najprościej jest reprezentować jako funkcje, które jako argument przyjmują wejście pozostałe do przeczytania. W trakcie działania algorytmu zmiany stanów wykonywane są w ramach dwóch akcji: *shift* i *reduce*. Akcja *shift* dodaje jeden stan na szczyt stosu, może więc być zwykłym wywołaniem funkcji.

Nieco bardziej skompilowana jest akcja *reduce* $A \to \alpha$, która najpierw zdejmuje $|\alpha|$ stanów ze stosu, a następnie przechodzi do stanu $\mathrm{GOTO}[I, A]$, gdzie I jest nowym stanem na szczycie stosu. Operację taką można realizować jako wywołanie odpowiedniej kontynuacji, zdefiniowanej w stosie wywołań wyżej o $|\alpha|$ elementów. Za przejście do nowego stanu może być odpowiedzialna już sama kontynuacja — odbiera to konieczność poznania stanu I w miejscu wywołania kontynuacji, gdyż w miejscu jej definicji jest on w oczywisty sposób znany. Kontynuacja taka jako argument powinna przyjmować nowy stan wejścia, aby zachować ciągłość jego modyfikacji.

Gdy automat chce wykonać akcję *reduce* $A \to \alpha$, to jego stan zawiera sytuację $A \to \alpha \cdot$. Stan znajdujący się o $|\alpha|$ elementów wyżej w stosie wywołań musi zawierać $A \to \ \cdot \alpha$, czyli sytuację dodaną tam przez funkcję $\mathrm{CLOSURE}$. Funkcja generowana dla stanu I powinna więc definiować kontynuacje odpowiadające wszystkim redukcjom w $I \setminus \mathrm{KERNEL}(I)$, a jako argumenty powinna przyjmować kontynuacje odpowiadające pozostałym sytuacjom, czyli $\mathrm{KERNEL}(I)$. Definiowane kontynuacje służą jedynie do przejścia do odpowiedniego stanu, dlatego ciało każdej z nich zawiera jedynie wywołanie odpowiedniej funkcji, identyczne jak przy akcji *shift*.

Przy przechodzeniu do stanu $\mathrm{GOTO}[I, s]$ dalej należy przekazać wszystkie kontynuacje, które odpowiadają sytuacjom jądra $\mathrm{GOTO}[I, s]$. Są one postaci $A \to \alpha s \cdot \beta$, gdzie $A \to \alpha \cdot s\beta$ jest w I — wymaga to więc jedynie wybrania ze zbioru I tych sytuacji, które mają symbol s po kropce.

Jako przykład generowanej funkcji możemy rozważyć tę generowaną dla stanu $I = \{A \to a \cdot B, A \to a \cdot, B \to \ \cdot b\}$. Jej schemat wygląda następująco:

```
(* c0: A → a · B *)
(* c1: A → a ·   *)
let state_I input c0 c1 =
```

```
(* c2: B → · b *)
let c2 input = state_GOTO[I,B] input c1 in
podejrzyj słowo v z wejścia;
if ACTION[I, v] = shift b then
    wczytaj symbol b z wejścia;
    state_GOTO[I,b] input c2;
if ACTION[I, v] = reduce A → a then
    c1 input;
...
```

Funkcja ta dostaje dwie kontynuacje odpowiadające sytuacjom w jądrze I, kolejno $A \to a \cdot B$ i $A \to a \cdot$. Definiuje ona trzecią kontynuację pochodzącą od ostatniej sytuacji w I, dodanej tam przez CLOSURE. W ciele tej kontynuacji zawarte jest przejście do stanu $\mathrm{GOTO}[I, B]$. Widoczna jest obsługa dwóch akcji — *shift* b i *reduce* $A \to a$, których wybór dokonywany jest na podstawie słowa v. Akcja *shift* wiąże się z wczytaniem jednego symbolu z początku wejścia i przejściem do nowego stanu, $\mathrm{GOTO}[I, b]$. Kontynuacja `c2` jest przekazywana dalej, bo jądro stanu $\mathrm{GOTO}[I, b]$ zawiera $B \to b \cdot$. Akcja *reduce* jest zwykłym wywołaniem kontynuacji.

Kod generowany w ten sposób pozwala już w całości symulować stos stanów algorytmu LR(k), czyli tym samym potrafi odtworzyć przejście całego algorytmu.

3.3. Wartości semantyczne

Głównym celem podejścia kontynuacyjnego jest wyeliminowanie heterogenicznego stosu wymaganego przez algorytm LR. Jego problematyczność wynika z przechowywania w nim wartości semantycznych. Zauważmy, że dla każdego stanu I prawdziwe są następujące fakty:

Fakt 3.1. Dopóki stan I jest na stosie automatu, to jednorazową redukcją automat nie może zejść w stosie stanów niżej niż $|\alpha_I|$ stanów poniżej stanu I, gdzie α_I jest najdłuższym spośród $\{\alpha \mid (A \to \alpha \cdot \beta) \in I\}$.

Fakt 3.2. Dla każdego stanu I i każdego następnika $I' = \mathrm{GOTO}[I, s]$ tego stanu zachodzi $|\alpha_{I'}| \leq |\alpha_I| + 1$

Fakt 3.1 oznacza, że każda funkcja wygenerowana dla stanu I potrzebuje co najwyżej $|\alpha_I|$ wartości semantycznych ze szczytu stosu — wartości te może otrzymywać jako argumenty. Z faktu 3.2 wnioskujemy, że każdy z następników I potrzebuje co najwyżej o jedną wartość więcej, niż potrzebuje I. Ta dodatkowa wartość zawsze pochodzi z właśnie przeczytanego symbolu albo z dokonanej redukcji, więc pozostałych wartości nie jest więcej niż wymaga stan I.

Przechodząc do stanu $\mathrm{GOTO}[I, s]$ podać należy odpowiednie argumenty. Ponieważ prefiks α w sytuacjach spoza jego jądra jest pusty, zależą one tylko od sy-

tuacji w jądrze $\mathrm{GOTO}[I, s]$. Do określenia potrzebnych argumentów wystarczające jest więc znalezienie sytuacji w I, która ma symbol s po kropce i najdłuższy prefiks α.

Funkcja generowana dla rozpatrywanego wcześniej stanu $I = \{A \to a \cdot B, A \to a \cdot, B \to \cdot b\}$ wygląda teraz następująco:

```
(* c0: A → a · B *)
(* c1: A → b ·   *)
(* a0: a *)
let state_I input [a0] c0 c1 =
    (* c2: B → · b *)
    (* a1: B *)
    let c2 input [a1] = state_GOTO[I,B] input [a0 a1] c1 in
    podejrzyj słowo v z wejścia;
    if ACTION[I, v] = shift b then
        wczytaj symbol b z wejścia;
        [let a1 = wartość semantyczna b in]
        state_GOTO[I,b] input [a1] c2;
    if ACTION[I, v] = reduce A → a then
        [let a1 = wartość semantyczna A wyliczona z a0 in]
        c1 input [a1];
    ...
```

W kodzie zaznaczone zostały nowe elementy odpowiedzialne za obsługę wartości semantycznych. Funkcja przyjmuje argument `a0`, który zawiera wartość semantyczną symbolu a. Symbol ten pochodzi z prefiksu sytuacji $A \to a \cdot B$. Przy przejściu w kontynuacji `c2` podawane są wartości `a0` i `a1` odpowiadające symbolom a i B, bo stan $\mathrm{GOTO}[I, B]$ zawiera $A \to aB \cdot$. W akcji *shift* z właśnie przeczytanego symbolu otrzymywana jest nowa wartość semantyczna, którą program przekazuje dalej. W przejściu do nowego stanu nie jest podawana wartość `a0`, bo jądro $\mathrm{GOTO}[I, b]$ zawiera jedynie $B \to b \cdot$ i nie ma tam symbolu a. W akcji *reduce* nowa wartość jest wyliczana na podstawie `a0`, zgodnie z akcją semantyczną. Wartość ta przekazywana jest do kontynuacji.

3.4. Akcje *accept* i *error*

Akcję *error* w oczywisty sposób można zrealizować za pomocą dodatkowej kontynuacji. Akcję *accept* również można by zrealizować w ten sam sposób, istnieje jednak prostsze rozwiązanie. Automat wykonuje akcję *accept* tylko wtedy, gdy na stosie jest dokładnie jedna wartość semantyczna, odpowiadająca symbolowi S. Można więc o niej myśleć jak o redukcji $S' \to S$, co w omawianym podejściu jest wywołaniem odpowiedniej kontynuacji. Zauważmy, że działanie takie jest spójne

z opisaną wcześniej zależnością między kontynuacjami a sytuacjami — każdy stan prowadzący do akcji *accept* zawiera sytuację z symbolu S'. Dodatkowo, jądro stanu początkowego zawiera $S' \to \cdot\, S$, interesującą kontynuację funkcja tego stanu przyjmuje więc jako argument.

3.5. Kolejność kontynuacji

Kolejność kontynuacji w definicji funkcji i w jej wywołaniu oczywiście musi się zgadzać. Kontynuacje te pochodzą jednak jedynie od zbiorów sytuacji, które nie mają ustalonego porządku — podejście takie wymaga więc jego ustalenia.

Najprostszym rozwiązaniem jest porządkowanie sytuacji malejąco względem długości prefiksu α, a w przypadku ich równości w dowolny ustalony sposób, np. leksykograficznie. Przyjmując taki porządek łatwo jest zmodyfikować funkcje $\mathrm{CLOSURE}$ i GOTO, aby przyjmowały uporządkowaną listę sytuacji i zachowywały jej porządek. Jest tak, gdy $\mathrm{CLOSURE}$ dodaje nowe sytuacje tylko na koniec listy, a GOTO zwraca nowe sytuacje zgodnie z kolejnością sytuacji, od których pochodzą. Ustalenie takiego porządku ułatwia też znalezienie sytuacji potrzebnej do określenia przekazywanych argumentów — potrzebna sytuacja zawsze jest pierwszą z pasujących.

3.6. Dobre typowanie

Kod generowany opisanym sposobem dobrze się typuje. Argument, który odpowiada symbolowi s ma ten sam typ, co wartość semantyczna tego symbolu. Kontynuacja odpowiadająca redukcji $A \to \alpha$ ma typ `input -> t -> 'a`, gdzie typy `input`, `t` i `'a` to kolejno typ wejścia, wartości semantycznej A i typ wyniku kontynuacji wspólny dla całego programu. Ponieważ przy przejściu do nowego stanu wywoływane funkcje zawsze dostają argumenty i kontynuacje odpowiadające tym samym symbolom i redukcjom, co w stanie wywołującym, nie doprowadzi to do błędów typów.

Generowany kod nigdzie nie potrzebuje korzystać z `Obj.repr`, `Obj.magic` lub innych funkcji oszukujących system typów.

3.7. Przykład

Jako przykład rozważmy kod generowany dla języka dobrze rozstawionych nawiasów, danego gramatyką $S \to (S)S \mid \epsilon$. Niektóre ze stanów $\mathrm{LR}(0)$ dla tej gramatyki to:

$$I_0 = \{ \quad S' \to \cdot\, S, \quad S \to \cdot\, (S)S, \quad S \to \cdot \quad \}$$
$$I_4 = \{ \quad S \to (S)S \cdot \quad \}$$
$$I_5 = \{ \quad S' \to S \cdot \quad \}$$

Załóżmy, że funkcje ACTION i GOTO przyjmują następujące wartości:

	(	)	ϵ
I_0	*shift* (	*error*	*reduce* $S \to \epsilon$
I_4	*error*	*reduce* $S \to (S)S$	*reduce* $S \to (S)S$
I_5	*error*	*error*	*accept*

(a) ACTION

	(	)	S
I_0	I_1		I_5
I_4			
I_5			

(b) GOTO

Typ wartości semantycznej symbolu S to `value`, a nawiasów to `l` i `r`. Symbole terminalne wspólnie określone są typem `type token = L of l | R of r`, gdzie `L` i `R` to kolejno lewy i prawy nawias. Redukcjom $S \to (S)S$ i $S \to \epsilon$ towarzyszą akcje semantyczne:

```
action0: l -> value -> r -> value -> unit -> value (* S → (S)S *)
action1: unit -> value (* S → ϵ *)
```

Ostatni argument typu `unit` służy odroczeniu obliczeń do momentu wykonania akcji. Symbole wczytywane są z wejścia typu `input`, na którym program potrafi wykonywać następujące operacje:

- `shift: input -> input` — wczytuje i usuwa jeden symbol z początku wejścia;
- `lookahead: input -> token option` — podgląda początek wejścia. Na wartościach zwracanych przez tą funkcję można dokonać dopasowania `match`, którego ramiona zależą od wartości ACTION.

Fragment kodu generowanego przy takich założeniach dla stanu I_0 wygląda następująco:

```
(* c0: S' →  · S *)
let rec s0 input c0 err =
    (* c1: S →  · (S)S *)
    let c1 input x = s5 input x c0 err
    (* c2: S →  ·  *)
    and c2 input x = s5 input x c0 err in
    match lookahead input with
        (* shift *)
        | Some (L x) ->
            let input = shift input in
            s1 input x c1 err
        (* error *)
```

```
        | Some (R _) -> err ()
        (* reduce S → ϵ *)
        | None ->
            let x = action1 () in
            c2 input x
```

Powyższa funkcja przyjmuje dwie kontynuacje: `c0` odpowiadającą końcowej redukcji $S' \to S$ i kontynuację błędu. Definiuje ona dwie nowe kontynuacje pochodzące od pozostałych sytuacji w I_0. Używając `lookahead`, funkcja ta podgląda początek wejścia, na podstawie którego podejmuje zgodne z ACTION dalsze decyzje. Dla stanów I_4 i I_5 otrzymujemy poniższy kod:

```
(* c0: S → (S)S ·  *)
(* a0: (, a1: S, a2: ), a3: S *)
and s4 input a0 a1 a2 a3 c0 err =
    match lookahead input with
        (* error *)
        | Some (L _) -> err ()
        (* reduce S → (S)S *)
        | Some (R _) | None ->
            let x = action0 a0 a1 a2 a3 () in
            c0 input x

(* c0: S' → S ·  *)
(* a0: S *)
and s5 input a0 c0 err =
    match lookahead input with
        (* error *)
        | Some (L _) | Some (R _) -> err ()
        (* accept *)
        | None -> c0 input a0
;;
```

W stanie `s4` wykonywana jest redukcja — polega one na wywołaniu akcji semantycznej i przekazaniu jej wyniku odpowiedniej kontynuacji. Akcja *accept* w stanie `s5` również realizowana jest poprzez wywołanie kontynuacji, nie jest ona jednak związana z akcją semantyczną. W funkcjach tych nie są definiowane nowe kontynuacje, bo wszystkie sytuacje w stanach I_4 i I_5 są w ich jądrze. Funkcje dla pozostałych stanów generowane są bardzo podobnie.

Tak wygenerowany parser uruchomić można przechodząc do stanu startowego, a poniższa funkcja pozwala na zrobienie tego w wygodny sposób:

```
exception ParseError

let parse input =
    let ok input x = x
    and err () = raise ParseError in
    s0 input ok err
;;
```

Funkcja ta definiuje dwie kontynuacje wymagane przez `s0`. Kontynuacja `ok` ignoruje wejście, z którego po zakończeniu parsowania wczytane są już wszystkie symbole, i zwraca otrzymaną wartość. Druga kontynuacja ewentualne błędy parsowania zgłasza jako wyjątki.

Rozdział 4.

Implementacja

Przykładowa implementacja omówionej metody dostępna jest pod adresem `https://github.com/adampsz/cpspg`.

Generator jako opis gramatyki przyjmuje plik w formacie `.mly`, w większości zgodnym z formatem rozumianym przez `ocamlyacc`. Do obsługiwanych klas gramatyk należą LR(1), SLR(1) i LALR(1), dodatkowo rozumiane są deklaracje priorytetów i łączności operatorów. Generator w trakcie działania:

- Zbiera podstawowe informacje o gramatyce, tj. nazwy i typy symboli, przypisuje produkcjom akcje semantyczne (lib/GrammarGen.ml)
- Generuje automat LR(0)/LR(1) oraz ustala wartości funkcji ACTION i GOTO. Na tym etapie wyliczane są też odpowiednie wartości słów *lookahead* i rozwiązywane są konflikty (lib/AutomatonGen.ml).
- Na koniec generowany jest kod — każdemu stanowi odpowiada jedna funkcja, której sygnatura i ciało zależy jedynie od zawartości stanu i wartości ACTION i GOTO. Jest to właściwa część realizująca opisaną wcześniej metodę (lib/CodeGen.ml).

Konstrukcja programu umożliwia jego rozszerzenie w przyszłości, np. o obsługę produkcji parametrycznych. Zmiana taka nie wpłynęłaby na generowanie kodu.

Za parsowanie opisu gramatyki odpowiedzialny jest plik lib/Parser.ml, bootstrapowany z lib/Parser.mly za pomocą `cpspg`.

4.1. Inferencja typów

Implementacja pozwala na pominięcie typów niektórych symboli:

```
%token<int> X
%start a
%%
```

```
a: b { $1 };
b: X { $1 };
```

Powyższa gramatyka jest odrzucana przez narzędzia `menhir` i `ocamlyacc` z komunikatem o potrzebie podania typu symbolu startowego. Szczególnie uciążliwe jest to w `menhirze`, który wymaga też znajomości typów wszystkich symboli nieterminalnych (które może jednak sam poznać uruchamiając kompilator Ocaml-a, zob. `--infer`). Przytoczona gramatyka jest akceptowana przez `cpspg`, a kompilator Ocaml-a poprawnie dedukuje typ symbolu startowego jako `int` w wygenerowanym parserze.

4.2. Niezależność od wariantu algorytmu

Omówione podejście korzysta z niektórych szczegółów konstrukcji automatu LR(k), są one jednak wspólne dla metod LR(k), SLR(k) i LALR(k). Automaty generowane dla tych metod różnią się jedynie liczbą stanów, słowami u (*lookahead*) przy sytuacjach i decyzjami w ACTION. Wygenerowanie kontynuacyjnego parsera jest więc możliwe w każdej z nich.

Program `cpspg` generuje parsery LR(1), SLR(1) i LALR(1) w identyczny sposób, a wszystkie różnice między nimi są zawarte w pliku lib/AutomatonGen.ml. Ponieważ generowanie kodu oparte jest jedynie o gotowy automat i funkcje ACTION, różnice te nie są w nim widoczne.

4.3. Optymalizacja

W programie zastosowanych zostało kilka prostych optymalizacji, które polegają głównie na zaobserwowaniu pewnych powtórzeń i nieużytecznych części generowanego kodu. Części takie mogą być bezpiecznie pominięte, zmniejszając tym samym rozmiar parsera.

4.3.1. Grupy sytuacji

Najprostszą i najbardziej znaczącą optymalizacją jest łączenie sytuacji w grupy. *Grupą* sytuacji LR(k) nazwiemy w tej pracy taki zbiór, że każde dwie sytuacje $A \to \alpha \cdot \beta \ /u$ i $A' \to \alpha' \cdot \beta' \ /u'$ w nim zawarte spełniają $A = A'$ i $\alpha = \alpha'$. Wszystkie kontynuacje odpowiadające sytuacjom w jednej grupie są definiowane w jednym miejscu i zawierają przejście do tego samego stanu, $\mathrm{GOTO}[I, A]$. Ciała tych kontynuacji są identyczne, mogą więc zostać zastąpione jedną kontynuacją. Optymalizacja ta szczególnie ważna jest dla grup dodawanych przez CLOSURE, bo w każdej

takiej grupie jest tyle samo sytuacji, co produkcji z danego symbolu.

Łączenie sytuacji w grupy ma ciekawy efekt uboczny — okazuje się, że w metodach LR(k) i LALR(k) równości $A = A'$ i $\alpha = \alpha'$ implikują też $u = u'$. Wykorzystane zostało to w `cpspg`, umożliwiając trochę bardziej efektywne użycie tych metod. Obserwacja ta nie jest jednak istotna z punktu widzenia tej pracy, co jest powodem pominięcia dowodu jej poprawności.

4.3.2. Pomijanie nieużytecznych argumentów

W praktyce większość symboli terminalnych, takich jak interpunkcja, operatory czy słowa kluczowe nie ma wartości semantycznych. Przekazywanie ich jako argumenty do funkcji stanów nie ma więc większego sensu — można je pominąć. Nie powoduje to problemów w przejściach między stanami, wymaga jedynie pilnowania, które argumenty są dostępne i powinny zostać przekazane dalej.

4.3.3. Imperatywność

W implementacji wykorzystane zostały dwie zmiany korzystające z imperatywnych cech Ocaml-a. Użycie struktury `Lexing.lexbuf` pozwala na usunięcie argumentu `input` przyjmowanego przez kontynuacje, a nawet przez funkcje stanów, przy decyzji na przechowywanie go globalnie. Umożliwia ono też lepszą integrację z narzędziem `ocamllex`, które go wykorzystuje. Zamiast wywoływania kontynuacji `err` zgłaszany jest odpowiedni wyjątek, co eliminuje konieczność przyjmowania tej kontynuacji przez funkcje stanów.

4.4. Wydajność

Implementacja została porównana z generatorami `menhir` (wersja 20230415) i `ocamlyacc` (wersja 5.0). Zmierzony został czas parsowania wejść opisanych następującymi gramatykami:

- `math` — gramatyka prostych wyrażeń matematycznych z operatorami dodawania, odejmowania, mnożenia, dzielenia, modulo, potęgowania i nawiasami, z zachowaniem priorytetów i łączności operatorów. Sprawdzane wyrażenia miały głębokość 20 i były generowane losowo.
- `dyck` — język dobrze rozstawionych nawiasów danych gramatyką $S \to (S)S \mid \epsilon$, każde wyrażenie głębokości 15.
- `leftrec` — gramatyka z lewą rekursją $S \to S+ \mid \epsilon$, wejścia długości 2^{15},
- `rightrec` — analogiczna gramatyka z prawą rekursją $S \to +S \mid \epsilon$, wejścia długości 2^{15}.

- `lua` — gramatyka języka Lua 5.4[1]. Lua jest stosunkowo prostym językiem programowania o nieskomplikowanej gramatyce, okazała się więc dobrym kandydatem do testowania wydajności parserów. Jako wejście użyta została konkatenacja plików `*.lua` znalezionych w https://github.com/lua/lua/tree/v5.4.0/testes, o rozmiarze ok. 397KB.

Programy kompilowane były z użyciem kompilatora Ocaml-a w wersji 5.0. W pierwszych czterech przypadkach parsery uruchamiane były po 10 razy na 100 różnych wejściach, a w ostatnim po 100 razy na jednym wejściu. Wyniki testu:

Program	`math`	`dyck`	`leftrec`	`rightrec`	`lua`
`ocamlyacc`	3.729s	4.657s	1.885s	2.333s	2.527s
`menhir`	1.006s	0.660s	0.170s	0.522s	1.117s
`cpspg`	1.561s	2.523s	1.048s	2.006s	2.331s

Jak widać, najszybsze okazują się programy generowane przez `menhira`, które są szybsze ok. 4.5 i 2.5 raza od tych generowanych przez odpowiednio `ocamlyacc` i `cpspg`. Programy generowane z użyciem `cpspg` są jednak ok. 1.8 raza szybsze od tych tworzonych przez `ocamlyacc` dla prostych gramatyk i ok. 1.1 raza szybsze dla gramatyki `lua`. Wyniki te przemawiają za sukcesem metody, gdyż oba generatory działają niemal identycznie, generują taki sam automat nie wykonując na nim żadnych optymalizacji, a różnią się jedynie samym kodem odpowiadającym temu automatowi.

Na wydajność programów większego wpływu nie miały zastosowane flagi i wariant kompilatora. Optymalizacje dostarczone przez `Flambda`[2] (np. flaga `-inline`) zmieniają czas działania tylko nieznacznie, zachowując proporcje ok. 4.5 i 2.5 między szybkościami działania programów.

[1] https://www.lua.org/manual/5.4/
[2] https://ocaml.org/manual/flambda.html

Rozdział 5.

Przyszłe możliwości

5.1. Gramatyki niejednoznaczne i niedeterministyczne

Obsługiwaną klasę gramatyk stosunkowo prosto można rozszerzyć o gramatyki niejednoznaczne i niedeterministyczne, podobnie jak ma to miejsce w metodzie GLR [8]. W miejscach konfliktowych można bowiem sprawdzić każdą z możliwych ścieżek, zwyczajnie wywołując wszystkie odpowiadające im funkcje. Taki backtracking ma jednak pewne problemy — ścieżki parsowania są przeszukiwane w głąb, a nie wszerz (jak w metodzie GLR), co utrudnia zastosowanie znanych optymalizacji. Wymaga on również implementacji wejścia jako trwałej struktury danych albo możliwości umieszczenia wcześniej przeczytanych symboli z powrotem na wejściu. Z tego powodu nie jest możliwe użycie standardowej struktury `Lexing.lexbuf`, dobrze zintegrowanej z narzędziem `ocamllex`.

5.2. Kontynuacyjny back-end `menhira`

Omawiana metoda została zaimplementowana jako całkiem nowy generator parserów. Napisany program nie jest kompletny, a w porównaniu z `menhir`em brakuje mu wielu możliwości. Są to m. in. produkcje parametryczne, czytelniejsza składnia opisu gramatyki, lepsze zgłaszanie konfliktów czy optymalizacja automatu we wcześniejszym etapie. Ponieważ `menhir` wspiera więcej niż jeden back-end generatora kodu, możliwe powinno być rozszerzenie go o kolejny, kontynuacyjny. Rozwiązanie takie mogłoby przynieść pewne korzyści — generowane programy bogatsze byłyby o wszystkie pozytywne cechy niesione przez menhira, jednocześnie oferując lepszą inferencję typów i być może lepszą optymalizację przez kompilator.

5.3. Defunkcjonalizacja

Funkcje wyższego rzędu pojawiające się w generowanym kodzie można wyeliminować stosując defunkcjonalizację [6]. Kontynuacje przyjmowane przez funkcje stanów zastąpić można specjalnym typem danych, który ma po jednym konstruktorze dla każdej kontynuacji pojawiającej się w programie. Przechowuje on wszystkie zmienne wolne w kontynuacjach. Za wywołanie kontynuacji po transformacji odpowiedzialna jest specjalna funkcja `apply`.

Początek omawianego w podrozdziale 3.7 przykładu po ręcznej defunkcjonalizacji i usunięciu kontynuacji `err` wygląda następująco:

```
exception ParseError

type cont =
    | ContC0
    | ContS0C1 of cont
    | ContS1C1 of r * cont
    | ContS3C1 of l * value * r * cont

let rec apply input x = function
    | ContC0 -> x
    | ContS0C1 c0 -> s5 input x c0
    | ContS1C1 (a0, c0) -> s2 input a0 x c0
    | ContS3C1 (a0, a1, a2, c0) -> s4 input a0 a1 a2 x c0

and s0 input c0 =
    let c1 = ContS0C1 c0 in
    match lookahead input with
        | Some (L x) ->
            let input = shift input in
            s1 input x c1
        | Some (R _) -> raise ParseError
        | None ->
            let x = action1 () in
            apply input x c1

(* ... *)
```

Warto zauważyć, że typ `cont` w pewien sposób reprezentuje stos wartości semantycznych. Jest on rodzajem listy, a jego dno oznaczone jest przez `ContC0`. Każdy z pozostałych konstruktorów rekurencyjnie zawiera typ `cont` — tak samo jak w zwykłej liście. Inne wartości w tych konstruktorach to wartości semantyczne przechowywane na stosie. Nie powinno być to zaskakujące, gdy spojrzymy na to zjawisko

przez pryzmat odpowiedniości funkcyjnej [3] — parser w stylu kontynuacyjnym jest swojego rodzaju interpreterem, a otrzymywany kod to maszyna abstrakcyjna — automat ze stosem.

Oprócz pojawiającej się ciekawej struktury, transformacja taka zdaje się mieć pozytywny wpływ na czas działania generowanych programów. W przytoczonym przykładzie daje ona przyspieszenie ok. 20%, weryfikacja tych korzyści wymaga jednak dokładniejszych pomiarów.

Rozdział 6.

Podsumowanie

Metoda kontynuacyjna jest obiecującym nowym podejściem do generowania parserów. Pozwala ona uniknąć problemów `ocamlyacca` i `menhira` poprzez lepsze wykorzystanie cech Ocaml-a. Umożliwia ona też na lepszą inferencję typów w generowanych programach, pozwalając na pominięcie części deklaracji typowych.

Kod generowany w stylu kontynuacyjnym jest bardziej *idiomatyczny* od kodu manipulującego wielotypowym stosem danych. Przejście do stanu jest zwykłym wywołaniem funkcji i nie wymaga dodania jej numeru na jawny stos, co umożliwia zastosowanie dodatkowych optymalizacji już przez kompilator. Każda z funkcji jest kandydatem do rozwinięcia w miejscu wywołania, znajomość kontekstu tym miejscu pozwala na wykonanie dokładniejszej analizy programu. Wygenerowany tak parser jest też znacząco prostszy.

Parsery generowane z użyciem kontynuacji okazują się być szybsze o tych, które bezpośrednio manipulują stosem. Widoczne jest to na przykładzie programów `ocamlyacc` i `cpspg` — działają one w bardzo podobny sposób, różniąc się metodą generowania kodu. Mimo, że nie jest to najszybsza znana metoda i `menhir` generuje szybsze parsery, dalszy rozwój i zastosowanie dodatkowych optymalizacji może jeszcze poprawić ją w przyszłości.

Bibliografia

[1] Lexer and parser generators (ocamllex, ocamlyacc). https://v2.ocaml.org/manual/lexyacc.html. Dostęp: 21 czerwca 2023.

[2] Menhir reference manual (version 20230428). http://gallium.inria.fr/~fpottier/menhir/manual.html. Dostęp: 21 czerwca 2023.

[3] Mads Sig Ager, Dariusz Biernacki, Olivier Danvy, Jan Midtgaard. A functional correspondence between evaluators and abstract machines. *Proceedings of the 5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, 27-29 August 2003, Uppsala, Sweden*, strony 8–19. ACM, 2003.

[4] Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*, rozdział 4, strony 191–302. Addison-Wesley Longman Publishing Co., Inc., 2006.

[5] James Cheney, Ralf Hinze. First-class phantom types. Computer and Information Science Technical Report TR2003-1901, Cornell University, 2003.

[6] Olivier Danvy, Lasse R. Nielsen. Defunctionalization at work. *Proceedings of the 3rd international ACM SIGPLAN Conference on Principles and Practice of Declarative Programming, September 5-7, 2001, Florence, Italy*, strony 162–174. ACM, 2001.

[7] François Pottier, Yann Régis-Gianas. Stratified type inference for generalized algebraic data types. *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2006, Charleston, South Carolina, USA, January 11-13, 2006*, strony 232–244. ACM, 2006.

[8] Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1-2):31–46, 1987.