Implementacja języka funkcyjnego z rodziny ML z użyciem systemu kompilacji LLVM

(Implementation of ML-family functional language, using LLVM compiler infrastructure)

Mateusz Lewko

Praca licencjacka

Promotor: dr hab. Dariusz Biernacki

Uniwersytet Wrocławski
Wydział Matematyki i Informatyki
Instytut Informatyki

Streszczenie

TODO polish abstract

TODO english abstract

Spis treści

Rozdział 1.

Wprowadzenie

Pierwsze prace nad językiem ML zaczął Robin Milner na początku lat 70. W 1984, dzięki jego inicjatywie, powstał Standard ML - ustandaryzowana wersja języka ML. Już wtedy zawierał m. in. rozwijanie funkcji, dopasowanie do wzorca, inferencje typów oraz moduły parametryczne [1]. Są to elementy, które cechują większość dzisiejszych funkcyjnych języków programowania. Od tego czasu powstało wiele języków z rodziny ML. Jednymi z najpopularniejszych są: OCaml, F# oraz dialekty SMLa.

1.1. Klasy typów

Większość języków z rodziny ML w celu lepszego ustrukturyzowania programu stosuje system modułów. Pozwala on na podzielenie programu na niezależne od siebie funkcjonalności. Klasy typów, których głównym cele jest wprowadzenie ad-hoc polimorfizmu do języka, mogą po części także spełnić to zadanie [2]. Są obecne w językach takich jak Haskell, Scala czy Rust. Fakt, że pojawiają się w nowych językach ogólnego zastosowania, świadczy o ich atrakcyjności z punktu widzenia programisty. Mimo to nieznane są żadne popularne języki ML korzystającego z tego rozwiązania. Jedynym z celów tej pracy jest wprowadzenie klas typów do prostego języka funkcyjnego, bazującego na podstawowych cechach rodziny ML. W tym celu stworzyłem kompilator języka *Lang*, wymyślonego na potrzeby tej pracy.

1.2. Efektywna implementacja języka funkcyjnego

Drugim celem tej pracy jest implementacja głównych cech języków funkcyjnych w możliwie optymalny sposób. Skupię się na optymalizacji czasu wykonania programu, kosztem długości wygenerowanego kodu. Kompilacja będzie się odbywać do kodu maszynowego, gdyż daje to lepszą wydajność otrzymanego programu.

Stanowi to też większe wyzwanie przy kompilacji języka funkcyjnego, niż napisanie interpretera, ze względu na jego wysoką poziomowość. Oczywiście, trudnym będzie uzyskanie podobnej lub lepszej wydajności niż popularne kompilatory języków funkcyjnych, gdyż te stosują dużą liczbę skomplikowanych optymalizacji. Skupię się nad tym, aby moja implementacja prostego języka funkcyjnego, była porównywalna wydajnością z popularnymi rozwiązaniami. Omówię i porównam sposoby w jaki zdecydowałem się zaimplementować podstawowe cechy języków funkcyjnych, a w szczególności: częściową aplikację, zagnieżdżone funkcje, polimorfizm i klasy typów. Moje rozwiązania będą bazować na pomysłach z różnych języków programowania, w tym imperatywnych. Wspomniane cechy omówię dokładniej, ponieważ odbiegają od rozwiązań stosowanych w popularnych językach funkcyjnych.

1.3. Infrastruktura LLVM

W celu uproszczeniu konstrukcji nowego kompilatora i ułatwienia pracy z generowaniem niskopoziomowego kodu, zdecydowałem się skorzystać z infrastruktury LLVM. Jest to zbiór narzędzi i bibliotek wykorzystywanych przez wiele współczesnych kompilatorów. LLVM dostarcza kompilator LLVM IR, który jest niskopoziomowym językiem stworzonym na potrzeby pisania kompilatorów. Przykładowy program napisany w LLVM IR:

```
@.str = internal constant [14 x i8] c"hello, world\0A\00"

declare i32 @printf(i8*, ...)

define i32 @main(i32 %argc, i8** %argv) nounwind {
entry:
    %tmp1 = getelementptr [14 x i8], [14 x i8]* @.str, i32 0, i32 0
    %tmp2 = call i32 (i8*, ...) @printf( i8* %tmp1 ) nounwind
    ret i32 0
}
```

LLVM IR składa się przede wszystkim z: deklaracji i definicji funkcji, zmiennych globalnych, podstawowych bloków, przypisań oraz wywołań funkcji. Podstawowe bloki kodu jak i funkcje nie mogą być zagnieżdżone.

W moim kompilatorze nie generuję kodu LLVM'a, korzystam z oficjalnej biblioteki dla OCamla, udostępniającej interfejs potrzebny do tworzenia elementów wygenerowanego kodu. System LLVM jest odpowiedzialny za ostatni etap procesu kompilacji, zamianę kodu pośredniego (LLVM IR) na assembler. Cały kod jest w postaci Single Static Assignment, do jednej zmiennej (etykiety) można przypisać tylko jedno wyrażenie. Dzięki takiej formie kodu pośredniego, LLVM jest w stanie przeprowadzić na nim pewne optymalizacje, przed wygenerowaniem kodu maszynowego.

TODO: 2. Dlaczego LLVM i jakie są inne opcje (C, asembler)?

1.4. Klasy typów

Jako pierwsze pojawiły się w języku Haskell. Początkowo zostały użyte w celu umożliwienia przeładowania operatorów arytmetycznych i równości. Od tego czasu, znaleziono dla nich więcej zastosowań w różnych językach programowania. W języku Haskell, poza tym, że umożliwiają użycie przeładowanych funkcji, definiowania funkcjonalności wspólnej dla wielu typów (interfejsów), okazały się niezbędne do implementacji Monad. W języku systemowym Rust, odpowiednikiem klas typów są *cechy* (ang. trait). W podstawowych użyciach nie różnią się od klas typów, ale nie pozwalają na implementacje polimorfizmu wyższych rzędów [3] (ang. Higher-kinder polymorphism). Inną istotną różnicą jest fakt, że klasa typów z Haskella nie definiuje nowego typu, jedynie pozwala na ograniczenie typu do instancji klasy. *Cecha* z Rusta może być użyta jak zwykły typ, przykładowo można stworzyć listę zawierające obiekty, które są różnymi instancjami (implementacjami) *cechy*. W Haskellu istnieją także rozszerzenia, które pozwalają na definicje klas z wieloma parametrami.

Istnieje wiele wariantów klas typów oraz rozwiązań do nich podobnych, dlatego w swoim kompilatorze zdecydowałem się zaimplementować ich najprostszą wersje, umożliwiającą $ad - hoc$ polimorfizm.

Podstawowe użycie klas typów zaprezentuję na przykładzie Haskella. W celu stworzenia klasy typów C dla typu ogólnego a, należy zdefiniować zbiór funkcji, które musi zawierać instancja tej klasy. Dla danego typu i klasy może istnieć co najwyżej jedna instancja.

Listing 1..1: Przykładowa definicja klasy typów.

```
class Eq a where
  (==) :: a -> a -> Bool
  (/=) :: a -> a -> Bool
```

W powyższym przykładzie definiujemy klasę Eq zawierającą dwa operatory: $==$ oraz $/=$. Powiemy, że typ ukonkretniony z a jest instancją klasy Eq, jeśli zawiera deklaracje obu funkcji z odpowiednimi typami. Przykładowa instancja dla typu $Bool$, mogłaby wyglądać następująco:

Listing 1..2: Instancja klasy Eq dla typu $Bool$.

```
instance Eq Bool where
  True == True = True
  False == False = True
  _ == _ = False
  l /= r = not (l == r)
```

1.5. Let-polimorfizm

Istnieją funkcje, których implementacja jest taka sama, niezależenie od typu dla którego ją aplikujemy. Przykładowo, funkcja obliczają długość generycznej listy nie zależy od typu elementów, które się w niej znajdują. Funkcja $map :: (a \to b) \to [a] \to [b]$, transformująca zawartość listy z użyciem podanej funkcji mapującej, także nie zależy od zawartości listy. Nie oznacza to jednak, że podana funkcja mapująca i lista mogą mieć dowolny typ. Funkcja mapująca $(a \to b)$ musi przyjmować taki sam typ, jaki znajduje się w liście. W statycznie typowanym języku, kompilator, musi mieć pewność, że taki warunek zachodzi. Aby uniknąć powielania kodu, w większości języków funkcyjnych występuje *let-polimorfizm*.

Dzięki *let-polimorfizmowi*, przy definicji funkcji, dany argument może mieć ogólny typ, jeśli później w ciele tej funkcji, nie zostanie on ukonkretniony. Wprowadzenie *let-polimorfizmu* do języka, wymaga nie tylko jego obsługi w procesie generowania kodu (kompilacji), ale też przy etapie inferencji typów. Każdy inferowany typ musi być najbardziej ogólny. W swoim kompilatorze zaimplementowałem oba te elementy. Omówię i porównam swoje rozwiązanie z rozwiązaniami występującymi w innych językach.

1.6. Rozwijanie funkcji oraz częściowa aplikacja

Częściowa aplikacja występuje wtedy, gdy po zaaplikowaniu mniejszej liczby argumentów niż wynosi arność funkcji, otrzymujemy nową funkcje. Przykładowo dla funkcji $f : (A \times B) \to C$ po zaaplikowaniu pierwszego argumentu $a : A$, otrzymujemy funkcję $g : B \to C$. W szczególności, dla dowolnego $b : B$, zachodzi: $g(b) = f(a, b)$. Funkcja g, która jest częściowo zaaplikowaną funkcją f, musi zapamiętać zaaplikowane dotychczas argumenty.

Częściowa aplikacja jest spotykana nie tylko w językach funkcyjnych. Przykładowo, biblioteka standardowa języka `C++` dostarcza funkcję *bind* [4], która pozwala na zaaplikowanie części argumentów. Częściową aplikacje można osiągnąć poprzez rozwinięcie funkcji (ang. currying) do wielu funkcji jednoargumentowych. Na poniższym fragmencie kodu języka Javascript znajduje się przykład takiego rozwiązania.

Listing 1..3: Rozwinięcie funkcji w Javascriptcie.

```
var add = x => (y => x + y);
var add3 = add(3);

console.log(add3(12)); // 15
console.log(add(3)(12)); // 15
```

Javascript nie jest językiem funkcyjnym, a funkcje w nim zdefiniowane są w

zwiniętej formie. Z tego powodu konieczne jest zastosowanie rozwlekłej składni, takiej jak w ostatniej linii przytoczonego przykładu. Ta sama funkcja zdefiniowana w OCamlu wygląda następująco:

Listing 1..4: Rozwinięta funkcja w OCamlu.

```
let add x y = x + y
print_int (add 3 12)
```

Funkcja *add* w języku w OCaml jest już w postaci rozwiniętej, więc jej deklaracja i wywołanie mają bardziej atrakcyjną formę, niż w poprzednim przykładzie. Dlatego zdecydowałem się ją zaimplementować.

W praktyce taka metoda realizacji częściowej aplikacji, jak pokazałem na przykładzie Javascriptu, byłaby niepotrzebnie nieefektywna. Bardziej optymalny, ale też i złożony sposób obsługi aplikacji częściowej, który zastosowałem w tym kompilatorze, zaprezentuję w rozdziale poświęconym jego implementacji.

Rozdział 2.

Język *lang*

2.1. Inspiracja

Składnia języka *lang* jest w większości zapożyczona z języka *F#*, należącego do rodziny ML. Dzięki zastosowaniu składni czułej na wcięcia, która eliminuje konieczność użycia wielu słów kluczowych, jest jednym z prostszych języków z tej rodziny. Przy tworzeniu nowego języka funkcyjnego, kierowałem się głównie jego prostotą. Poza zapożyczeniem składni *F#* dla podstawowych wyrażeń, funkcji i typów rozszerzyłem ją o wyrażenia koniecznie do realizacji klas typów i ich instancji.

2.2. Podstawowe wyrażania

2.2.1. Wyrażenia warunkowe

Składnia wyrażeń warunkowych jest bardzo podobna do tej w *F#*. W języku *Lang* istnieją jednak pewne uproszczenia względem *F#*. Warunek musi być prostym wyrażeniem zawierającym operacje arytmetyczne i logiczne oraz wywołania funkcji. Nie może zawierać przykładowo: wielolinijkowych wyrażeń if i wyrażeń *let*. Ciało warunku może być złożonym wyrażeniem, takim jak ciało funkcji, o ile występuje w nowej lini i jest wcięte bardziej niż token if. TODO: Więcej o wcięciach Poniższa gramatyka, prezentują zbliżoną formę do rzeczywistej składni języka. Dokładny opis gramatyki znajduje się w pliku `lang-compiler/compiler/parsing/grammar.mly`. Jest bardziej skomplikowany ze względu na rozpoznawanie bloków kodu z takim samym poziomem wcięcia na poziomie parsera. Lepszym pod względem czytelności, jest wykonanie tej czynności na etapie lexera, tak jak to ma miejsce w *F#*. Dokładniejszy opis sposobu parsowania składni bazującej na wcięciach, w tym i innych językach, znajduje się w rozdziale TODO: rozdział.

Dla prostoty zapisu przyjąłem że:

1. * to wystąpienie poprzedzającego wyrażenia zero lub więcej razy,

2. + to wystąpienie poprzedzającego wyrażenia jeden lub więcej razy.

$$
\begin{aligned}
\langle\text{simple-if-exp}\rangle &\models \text{if } \langle\text{simple-exp}\rangle \text{ then } \langle\text{simple-exp}\rangle\langle\text{simple-elif-exp}\rangle\langle\text{simple-else-exp}\rangle \\
\langle\text{if-exp}\rangle &\models \text{if } \langle\text{simple-exp}\rangle\langle\text{newline}\rangle \text{ then } \langle\text{body-exp}\rangle + \langle\text{elif-exp}\rangle * \langle\text{else-exp}\rangle \\
\langle\text{simple-else-exp}\rangle &\models \text{else } \langle\text{simple-exp}\rangle \mid \epsilon \\
\langle\text{simple-elif-exp}\rangle &\models \text{elif } \langle\text{simple-exp}\rangle \text{ then } \langle\text{simple-exp}\rangle \mid \epsilon \\
\langle\text{elif-exp}\rangle &\models \text{elif } \langle\text{body-exp}\rangle + \mid \langle\text{simple-elif-exp}\rangle \mid \epsilon \\
\langle\text{else-exp}\rangle &\models \text{else } \langle\text{body-exp}\rangle + \mid \langle\text{simple-else-exp}\rangle \mid \epsilon \\
\langle\text{newline}\rangle &\models \textit{nowa linia}
\end{aligned}
$$

2.2.2. Wyrażenia arytmetyczne i logiczne

Wyrażenia arytmetyczne i logiczne mają taką samą składnię jak w pozostałych językach z rodziny ML.

$$
\begin{aligned}
\langle\text{bool-exp}\rangle &\models \langle\text{simple-exp}\rangle\langle\text{bool-op}\rangle\langle\text{simple-exp}\rangle \\
\langle\text{arith-exp}\rangle &\models \langle\text{simple-exp}\rangle\langle\text{arith-op}\rangle\langle\text{arith-exp}\rangle \\
\langle\text{arith-op}\rangle &\models + \mid - \mid * \mid / \\
\langle\text{bool-op}\rangle &\models \&\& \mid \|
\end{aligned}
$$

2.3. Deklaracja funkcji (wyrażenie let)

Argumenty funkcji muszą być w tym samym wierszu co słowo *let*. Po znaku =, ciało może być złożonym wyrażeniem o ile zaczyna się w następnym wierszu i jest w późniejszej kolumnie niż słowo *let*. Wyrażenie *let* może być zdefiniowane w jednej linii, o ile jego ciało jest pojedyńczym wyrażeniem prostym.

$$
\langle\text{let-exp}\rangle \models \text{let}\langle\text{identifier}\rangle + =\langle\text{simple-exp}\rangle \mid \text{let}\langle\text{identifier}\rangle + =\langle \text{ newline } \rangle\langle\text{body-exp}\rangle
$$

2.4. Rekordy

2.5. Klasy typów

2.6. Moduły

2.7. Tablice

2.8. Wołanie funkcji z C

1. Opis, szczegóły składni, (przykłady: każda cecha języka i krótki przykład)

1. Proste wyrażenia, rekurencja, let-polymorphism, rekordy, wzajemnie rekurencyjne funkcje na top levelu, klasy typów, proste moduły, wyrażanie na top levelu, efekty uboczne, inferencja typów, anotacje.

1. Wprowadzenie czym są

2. Dlaczego? Jakie są alternatywy

3. Opis tego co zostało zaimplementowane, porównanie do innych języków, (Haskell, Rust, Scala)

Rozdział 3.

Kompilator

3.1. Etapy kompilacji

1. Jakie są etapy (lexer → parser → untyped ast → typed ast bez zagnieżdżonych funkcji → generowanie kodu (ast high-ollvm) → wywoływanie funkcji z api llvma → llc → gcc i external)

2. Krótko o każdym etapie

3.2. Analiza leksykalna

1. Czego użyłem.

2. Analiza wcięć

3.3. Parsowanie

1. Czego użyłem, coś o Menhirze, dlaczego Menhir

2. Wyzwania (składnia bazująca na wcięciach)

3. Gramatyka

3.4. Inferencja typów

1. Po co? Jak działa u mnie

3.5. Generowanie kodu

3.5.1. Częściowa aplikacja

3.5.2. Opis działania

1. Dlaczego jest to nietrywialne
2. Jakie miałem cele
3. Jak to działa u mnie
4. Przykład (wygenerowanego pseudo-kodu)

3.5.3. Porównanie z innymi implementacjami

1. Push/enter vs eval/apply

Porównanie z pracą "Making a fast curry: ..."

3.6. Zagnieżdżone funkcje

1. Co to są zagnieżdżone funkcje
2. Na czym polega trudność w ich implementacji
3. Jak zostały zaimplementowane: lambda lifting + closure conversion + wykorzystanie aplikacji częściowej

3.7. Rekordy

Implementacja, porównanie do rekordów w F#.

3.8. Let polimorfizm

1. Krótki opis, czym jest let-polimorfizm
2. Sposoby implementacji w różnych językach, zalety i wady
3. Sposób implementacji u mnie

3.9. Klasy typów

1. Czym są? Po co?
2. Sposoby implementowania, porównanie do pracy TODO
3. Jak zostały zaimplementowane, dlaczego tak

Rozdział 4.

Instrukcja obsługi

4.1. Instalacja

4.2. Sposób użycia z przykładami

4.3. Użyte narzędzia i biblioteki

4.4. Struktura projektu

Bibliografia

[1] The Standard ML Core Language, by Robin Milner, July 1984.
http://sml-family.org/history/SML-proposal-7-84.pdf

[2] ML Modules and Haskell Type Classes: A Constructive Comparison Stefan Wehr and Manuel M. T. Chakravarty
https://www.cse.unsw.edu.au/ chak/papers/modules-classes.pdf

[3] Higher kinded polymorphism - Rust Github issues.
https://github.com/rust-lang/rfcs/issues/324

[4] std::bind - C++ Reference
https://en.cppreference.com/w/cpp/utility/functional/bind

[5] TODO: FS lang, user voice - type classes
https://fslang.uservoice.com/forums/245727-f-language/filters/top