Implementacja języka funkcyjnego z rodziny ML z użyciem systemu kompilacji LLVM

(English title)

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.

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.

Sposób ich użycia zaprezentuję na przykładzie języka Haskell. W celu stworzenia klasy typów C dla typu ogólnego a, należy zdefiniować zbiór funkcji, które musi zawierać instancja tej klasy. Instancje klasy definiuje się dla typu konkretnego

// Co zrobiłem, po co, dlaczego // co to let polymorphism, type class

1.5. Język ML

1. Dlaczego ML, jakie są inne języki ML 2. Bazowanie na $F\#$

Rozdział 2.

Cechy języka *lang*

// Cechy z przykładami

2.0.1. Składnia

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

2.1. Cechy języka

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)

2.2. Infrastruktura LLVM

1. Co to jest?

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

3. Jak działa kompilowanie do LLVM?

4. Krótki opis high-ollvm

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

3.6. Częściowa aplikacja

3.6.1. Opis działania

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

3.6.2. Porównanie z innymi implementacjami

1. Push/enter vs eval/apply

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

3.7. 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

3.8. Rekordy

Implementacja, porównanie do rekordów w F#.

3.9. Let polimorfizm

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

3.10. Klasy typów

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

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