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.

TODO: Let z argumentami z adnotacjami. TODO: Rekurencja

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

2.3.1. Wzajemnie rekurencyjne wyrażenia let

TODO:

2.4. Rekordy

2.4.1. Deklaracja rekordu

⟨record-decl⟩ ⊨ type⟨identifier⟩='{'⟨field-decl⟩ + '}'
⟨field-decl⟩ ⊨ ⟨identifier⟩:⟨identifier⟩ + ⟨newline⟩ | ⟨identifier⟩:⟨identifier⟩ + ;

2.4.2. Literał rekordu

Literał może być zdefiniowany w jednym lub wielu wierszach. W przypadku definicji w jednym wierszu, kolejne pola muszą być oddzielone średnikami. Średnik może być pominięty jeśli kolejne pola są oddzielone nową linią. Dle definicji wielowierszowej, klamra otwierająca i zamykająca muszą być w tej samej kolumnie.

⟨record-lit⟩ ⊨ type ⟨identifier⟩='{'⟨field-lit⟩ + '}'
⟨field-lit⟩ ⊨ ⟨identifier⟩ = ⟨simple-exp⟩⟨newline⟩ | ⟨identifier⟩ = ⟨simple-exp⟩ ;

2.4.3. Uaktualnianie rekordu

Rekordy w *Langu*, podobnie jak rekordy w *F#* i OCamlu, są trwałe. Uaktualnienie jednego z pól skutkuje stworzeniem nowego rekordu. Dlatego to wyrażenie ma inną składnię niż ta znana z języków imperatywnych.

⟨record-update⟩ ⊨ { ⟨simple-exp⟩ with ⟨field-update⟩ + }
⟨field-update⟩ ⊨ ⟨identifier⟩=⟨simple-exp⟩⟨newline⟩ | ⟨identifier⟩=⟨simple-exp⟩;*TODO* : *samopo*

2.5. Klasy typów

Jako, że w językach z rodziny ML nie występują klasy typów, ich składnie zdecydowałem się zapożyczyć z Haskella.

2.5.1. Deklaracja klasy

2.5.2. Deklaracja instancji

2.6. Moduły

Moduły w *Langu* spełniają takie zadanie jak te w *F#* – służą jako przestrzeń nazw dla związanych ze sobą definicji. Nie są odpowiednikiem systemu dużo bardziej zaawansowanych modułów SMLa czy OCamla. Moduł zawiera: wyrażenia let, zagnieżdżone moduły, import innych modułów oraz deklaracje funkcji zewnętrznych. Nazwa modułu musi się zaczynać z wielkiej litery.

TODO: Gramatyka modułu

2.7. Tablice

Zaimplementowane zostały jedynie tablice zawierające typ *int*. Tablice są ulotną strukturą danych. Na poziomie języka można stworzyć literał tablicy. Zmiana i odczytanie komórki tablicy, bądź utworzenie niezainicjalizowanej tablicy odbywa się poprzez zewnętrzne funkcje zaimplementowane *C*. Tablice w langu są reprezentowane tak samo jak języku w *C*, jako spójny ciąg w pamięci.

Elementami literału tablicy, mogą być proste wyrażenia, oddzielone średnikiem. Podobnie jak w
OCamlu i *F#* tablica zaczyna się od symbolu [|, a kończy symbolem |].

Uwaga. Dla wielowierszowego literału tablicy, symbol rozpoczynający [| i kończący |], muszą być w tej samej kolumnie.

2.8. Wołanie funkcji z *C*

Mała część funkcjonalności języka została zaimplementowana z użyciem zewnętrznych funkcji w *C* (wypisywanie oraz operacje na tablicy). Dlatego koniecznym było dołożenie wyrażeń pozwalających zadeklarować zewnętrzny symbol wraz z jego typem. Ich składnia jest prawie taka sama jak w OCamlu. Typy *Langa* są dosłownie tłumaczone na typy w *C*, z wyjątkiem typu *unit*, który jest zamieniany na *bool* (o rozmiarze jeden bajt).

TODO: Gramatyka external

Rozdział 3.

Kompilator

3.1. Etapy kompilacji

Cały proces kompilacji, od momentu wczytania pliku z kodem źródłowym, do wyprodukowania pliku wykonywalnego, składa się z następujących etapów:

1. Analiza leksykalna, w efekcie której otrzymujemy ciąg tokenów. TODO: w jakim pliku.

2. Otrzymany ciąg jest następnie poddany analizie składniowej (ang. parsing), która zgodnie z podaną gramatyką generuje *drzewo składni abstrakcyjnej (ang. abstract syntax tree)*. Węzły tego drzewa zawierają jedno z wyrażeń, lecz nie posiadają informacji o typie tego wyrażenia. TODO: W jakim pliku.

3. Następnie, wykonywana jest transformacja drzewa składni, która:

   (a) Dzięki przeprowadzeniu inferencji typów, nadaje każdemu wyrażeniu jego typ z języka *Lang* (na późniejszym etapie, wyrażenia będą miał typ z *LLVM IR*).

   (b) Eliminuje zagnieżdżone wyrażenia *let*.

   (c) Eliminuje moduły oraz otwarcia modułów poprzez translacje symboli do ich w pełni kwalifikowanych nazw (ang. fully qualified name).

4. Generowanie drzewa wyrażeń z LLVM IR. Jest to największy etap z całego procesu kompilacji. Zamienia skomplikowane wyrażenia wysokopoziomowego języka na proste wyrażenia LLVM IR, które już łatwo mogą być przetłumaczone na niskopoziomowe instrukcje.

5. Konwersja drzewa wyrażeń LLVM IR na kod LLVM IR. Odbywa się to dzięki interfejsowi programistycznemu (ang. api), udostępnionym przez oficjalną bibliotekę LLVM dla *OCamla*.

3.2. Analiza leksykalna

Do przeprowadzania analizy leksykalnej skorzystałem z biblioteki *sedlex*. Jest to generator lekserów dla języka OCaml.

3.2.1. Analiza wcięć

Istnieje wiele języków programowania, realizujących ideę składni czułej na wcięcia. Sposób w jaki działa to w *F#* jest jednym z bardziej zaawansowanych, bo pozwala na zdefiniowanie wielowierszowych aplikacji funkcji, warunków itp., bez użycia znaków przełamania wiersza bądź słów kluczowych znanych z języka OCaml (*begin*, *end*, *;*). W *F#* istnieje także możliwość mieszania tych słów kluczowych z wcięciami.

Analiza wcięć w *Langu* jest zbliżona do tej w *Pythonie* [6]. Dla każdego wiersza, na bieżąco jest obliczany numer kolumny pierwszego znaku (wcięcie). Długości wcięć z poprzednich wierszy są trzymane na stosie. Na początku na stosie znajduję się wcięcie długości 0. Gdy wcięcie w obecnym wierszu jest większe od ostatniego na stosie, generowany jest token *INDENT*, oznaczający początek wciętego bloku. Gdy wcięcie jest mniejsze od ostatniego na stosie, wszystkie większe są zdejmowane ze stosu i dla każdego zdjętego, generowany jest token *DEDENT*. Oznacza on koniec wciętego bloku. Po zdjęciu wszystkich większych wcięć, ostatnie wcięcie, które zostanie na stosie musi być równe obecnemu wcięciu, w szczególności może być równe 0. W przeciwnym przypadku, kod źródłowy jest źle wcięty i kompilator zwróci błąd.

3.3. Parsowanie

Popularnym narzędziem do generowania parserów jest *Menhir*. Na
podstawie podanej gramatyki *LR(1)*, generuje kod *OCamla*, który ją parsuje. Częściowo wspiera składnię *EBNF*, m. in. operatory: +, ?, ∗. Zdecydowałem się skorzystać z tego narzędzia ze względu na łatwość użycia, możliwość interaktywnego debugowania gramatyki oraz ekspresywność składni w porównaniu do podobnych narzędzi takich jak *ocamlyacc*. Całość gramatyki znajduje się w pliku `lang-compiler/compiler/parsing/grammar.mly`. TODO: Może coś o kontekstach w F#

3.4. Częściowa aplikacja i funkcje

Jak wspomniałem we wprowadzeniu, generowanie wszystkich funkcji w rozwiniętej formie (każda funkcja przyjmuje tylko jeden argument) jest nieoptymalne pod względem długości kodu jak i szybkości jego wykonania. Pomimo że, aplikacja częściowa jest bardzo przydatną cechą języków funkcyjnych, to często funkcje wywoływane są ze wszystkimi argumentami. W takich przypadkach chcielibyśmy korzystać

z wywołania funkcji, które jest tak szybkie jak w *C*. W kompilatorze *Langa* pracowałem nad rozwiązaniem, które w pozostałych przypadkach korzystałoby z przekazywania argumentów funkcji przez rejestry i pozwalałoby na przekazywanie typów o różnych rozmiarach przez wartość.

3.4.1. Opis działania

Podzielmy wszystkie wywołania funkcji na dwie grupy. Wywołania do znanych (ang. known call) i nie znanych funkcji (ang. unknown call). Znane funkcje to takie, których definicję można łatwo wskazać na etapie kompilacji. Na poniższym przykładzie, wywołana funkcja TODO: jest statycznie znana.

Listing 3..1: Wywołanie statycznie znanej funkcji.

```
TODO: Known call example
```

Przykładem nieznanych funkcji są funkcje, które:

- zostały podane jako argument,
- są wynikiem wywołania funkcji,
- są wynikiem częściowej aplikacji funkcji.

W tym przykładzie wywołane funkcje a, b i c są nieznane.

Listing 3..2: Przykłady statycznie nieznanej funkcji.

```
TODO: Unknown call example
```

Wywołanie funkcji znanej

Gdy funkcja, którą chcemy wywołać jest znana, możemy wyróżnić trzy przypadki ze względu na liczbę zaaplikowanych argumentów względem liczby argumentów w definicji funkcji.

1. Liczba zaaplikowanych argumentów jest mniejsza od liczby zdefiniowanych argumentów. W tym przypadku utworzony zostaje obiekt reprezentujący częściowo zaaplikowaną funkcję. Zostaną w nim zapisane zaaplikowane argumenty oraz wskaźnik na odpowiednią funkcję. Skopiowane argumenty i sam obiekt zostaną utworzone na stercie.
2. Zaaplikowanych argumentów jest tyle co zdefiniowanych. Funkcja zostanie wywołana w stylu z *C*. Jest to najbardziej optymalny przypadek wywołania funkcji i nie powoduje on zaalokowania żadnej dodatkowej pamięci. Jeśli początkowe argumenty mieszczą się w rejestrach to mogą zostać przez nie przekazane.

3. Zaaplikowanych argumentów może być więcej niż zdefiniowanych jeśli wynikiem wywoływanej funkcji jest funkcja. Niech n to będzie liczba zdefiniowanych argumentów. Najpierw nastąpi wywołanie znanej funkcji z pierwszymi n argumentami. Do wyniku pierwszego wywołania, który teraz jest nieznaną funkcją, zostaną zaaplikowane pozostałe argumenty. W tym momencie zastosowany zostanie jeden z przypadków dla wywołań nieznanych funkcji.

   TODO: Coś o tym że funkcje / symbole trzymane są w środowisku z informacją known / unknown.

Wywołanie funkcji nieznanej

Wywołania funkcji nieznanych podzielimy na takie, których wynikiem jest dowolna funkcja $a \to b$ i takie których wynikiem jest wartość. Podczas fazy inferencji typów, obliczany jest typ każdego wyrażenia, więc kompilator jest w stanie określić do której grupy należy dana aplikacja funkcji. Każda nieznana lub częściowo zaaplikowana funkcja jest reprezentowana przez strukturę (taką jak w C), zawierającą następujące pola:

- wskaźnik na funkcje,
- wskaźnik na środowisko (zapamiętane argumenty), które jest pamiętane jako spójny ciąg bajów. TODO: Można też tablice wskaźników na argumenty i dlaczego tak nie zrobiłem
- liczba bajtów w środowisku,
- liczba obecnie zaaplikowanych argumentów
- pozostała liczba argumentów koniecznych do zaaplikowana, aby należało wywołać wskazywaną funkcje.

Definicja takiej struktury w C wyglądałaby następująco:

Listing 3..3: Rozwinięta funkcja w OCamlu.

```
struct function {
    void (*fn)();
    unsinged char *args;
    unsinged char left_args;
    unsinged char arity;
    int used_bytes;
};
```

Wskaźnik na funkcje *fn*, przed wywołaniem musi zostać zrzutowany na prawidłowy typ. Dla aplikacji funkcji, których wynikiem jest funkcja, typ wynikowy

funkcji *fn* to struktura *function*. Jako argumenty funkcji *fn*, poza argumentami podanymi w aplikacji funkcji, przekazane zostaną dodatkowo: wskaźnik na środowisko i liczba aplikowanych argumentów. Funkcja wołana jest odpowiedzialna za nadmiarowe argumenty i przekazanie ich dalej.

Aplikacja funkcji nie zawsze musi się wiązać z faktycznym wywołaniem funkcji. Na poniższym przykładzie w pierwszym przypadku funkcja *TODO:* zostanie wywołana, a w drugim, pomimo aplikacji tych samych argumentów, nic nie zostanie wywołane.

Listing 3..4: To czy funkcja zostanie wywołana, nie jest wiadome w czasie kompilacj.

```
TODO: Przykład z wywołaniem i nie wywołaniem częściowo
zaaplikowanej funkcji
```

Dla każdego wywołania nieznanej funkcji, generowany jest dodatkowy kod, który jest odpowiedzialny za sprawdzenie, czy aplikowaną funkcje faktycznie trzeba wywołać, czy jedynie zapisać dodatkowe argumenty do środowiska. Aby to sprawdzić, porównywana jest liczba argumentów pozostałych do wywołania funkcji (*left_args*) z liczbą zaaplikowanych argumentów. Jeśli liczba pozostałych argumentów jest większa, to wszystkie argumenty zostaną skopiowane do pola *args* w strukturze *function*, a odpowiednie jej pola uaktualnione.

TODO: Przykład w pseudokodzie? Może algorytm

TODO: generowanie każdej funkcji

Generowanie funkcji

Jednym z założeń implementacji tego języka, była możliwość przekazywania typów o różnym rozmiarze, przez ich wartość a nie przez wskaźnik. W obecnej implementacji istnieje tylko kilka typów o różnej długości: *bool*, *int* i *rekord*. Rekordy mają taki sam rozmiar, ponieważ są przekazywane przez wskaźnik do ich zawartości zapamiętanej na stercie, ale łatwo rozszerzyć język o typy o dowolnej długości.

Takie założenie komplikuje implementację częściowej aplikacji funkcji. Aby zrozumieć dlaczego, weźmy dwie instancje struktury *function*, dla funkcji o typie *int* $\rightarrow$ *bool* $\rightarrow$ *int* $\rightarrow$ *bool*. Niech pierwsza zostanie częściowo zaaplikowana dwoma argumentami o typach *int* i *bool*, a druga pierwszym argumentem o typie *int*. W kolejnym kroku, chcąc wywołać obie funkcje, do pierwszej struktury aplikujemy pozostały argument o typie *int*, a do drugiej pozostałe dwa o typach *bool* i *int*. Wskaźnik *fn* z pierwszej struktury zostałby zrzutowany na wskaźnik na funkcję przyjmującą jako pierwszy argument zmienną typu *int*, a funkcja wskazywana przez *fn* z drugiej struktury przyjmowałaby jako pierwszy argument typ *bool*. Nie można dopuścić do takiej sytuacji. Nasuwa się możliwe rozwiązanie, w którym w momencie gdy dochodzi do wywołania funkcji (co może być sprawdzone w czasie działania programu dzięki

polu *left_args*) można przekazać wszystkie argumenty znajdujące sie w środowisku. Wtedy typ zrzutowanych funkcji wskazywanych przez *fn* byłby taki sam, niezależnie od tego ile dotychczas argumentów zostało zaaplikowanych. Jednak, argumenty w środowisku są zapamiętane przez wartość w tablicy, *args* a ich reprezentacja nie jest jednorodna, co uniemożliwia ich przekazanie. Jednorodną reprezentacje wszystkich argumentów można uzyskać poprzez reprezentowanie ich przez wskaźnik, ale takiego rozwiązania chciałem uniknąć. Innym sposobem byłoby zapamiętanie wszystkich argumentów, także tych aplikowanych jako ostatnie, w środowisku. Wywoływana funkcja, wie już w czasie kompilacji jakich argumentów (i o jakim rozmiarze), spodziewać się w środowisku, więc jest w stanie je z niego odzyskać. To rozwiązanie rozwiązuje wspomniany problem, lecz wykonuje niepotrzebne zapisywanie i ładowanie argumentów zaaplikowanych jako ostatnie. Moje rozwiązanie unika tej operacji.

W tym celu, poza generowaniem właściwej funkcji, generowane są także funkcje wejściowe (ang. entry point), które będą używane w przypadku wywoływania nieznanej funkcji. Funkcja wejściowa przyjmuje:

- wskaźnik na środowisko *unsigned char*$*$,
- liczbę przekazywanych argumentów *unsigned char* (obsługiwane jest maksymalnie 255 argumentów),
- część argumentów oryginalnej funkcji.

Załóżmy, że oryginalna funkcja ma typ: $a_1 \to a_2 \to ... \to a_n \to t$. Wtedy, dla takiej funkcji zostanie wygenerowanych n funkcji wejściowych, gdzie $i - ta$ funkcja będzie przyjmowała sufiks ciągu oryginalnych argumentów, od $i - tego$ argumentu. Jeśli funkcja oryginalna zwraca funkcję to jej argumenty także będą uwzględnione w funkcji wejściowej.

Dla oryginalnej funkcji, tworzona jest globalna tablica wskaźników na wszystkie jej funkcje wejściowe. Funkcje są zapamiętane po kolei, tj. wskaźnik na pierwszą funkcję wejściową jest pierwszym elementem tablicy, na drugą, drugim itd. Gdy tworzona jest instancja struktury *function*, jak wskaźnik na funkcję do wywołania ustawiany jest wskaźnik na początek tablicy funkcji wejściowych. Oznacza to, że typ pola *fn* w języku *C* to *void (**fn)()*, oraz że przed wywołaniem funkcji ze struktury, należy zdereferować (ang. dereference) wskaźnik. Wskaźnik na wołaną funkcją musi być w każdym momencie programu aktualny, tzn musi odpowiadać liczbie początkowych argumentów zapamiętanych w środowisku. Dlatego gdy kolejne argumenty są aplikowane, wskaźnik jest zwiększany.

Funkcje wejściowe są odpowiedzialne za odczytanie argumentów ze środowiska i przekazanie ich do wywołania oryginalnej funkcji. Jeśli wynikiem funkcji oryginalnej jest funkcja, następuje jeden z dwóch przypadków sprawdzanych w czasie działania programu.

1. Nie ma pozostało więcej argumentów do zaaplikowania, funkcja wyjściowa jako swój wynik może zwrócić wynik funkcji oryginalnej.

2. Pozostałych argumentów jest mniej lub tyle samo niż wynosi wartość pola *left_args* ze struktury otrzymanej jako wynik pierwszego wywołania. Należy zapisać pozostałe argumenty do środowiska i uaktualnić pola struktury *function*.

Po wywołaniu funkcji należy jeszcze sprawdzić czy nie została zwrócona struktura, którą od razu można wywołać (taka która ma pole *left_args* równe 0). Taki wynik mógł powstać w funkcji wołanej w drugim przypadku.

3.4.2. Porównanie z innymi implementacjami

1. Push/enter vs eval/apply

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

3.5. Zagnieżdżone funkcje

Zagnieżdżone funkcje są nieodłączną częścią języków funkcyjnych. Ich implementacja wykorzystuje *closure conversion*, które zostało wykorzystane przy częściowej aplikacji funkcji. Ideą *closure conversion* jest pamiętanie funkcji wraz z jej domknięciem. *Closure conversion* dodaje duży narzut pamięciowy i czasowy na wygenerowany program, dlatego należy ustalić dlaczego taka operacja jest potrzebna.

Zdefiniujmy funkcje *make_adder* w *OCamlu*, która będzie zwracać zagnieżdżona funkcje. Ciało zagnieżdżonej funkcji *add* odwołuje się do zmiennej z zewnętrznego zakresu.

Listing 3..5: Zagnieżdżona funkcja w *OCamlu*

```
let make_adder x =
    let add y = x + y in
    add

let _ =
    let add1 = make_adder 1 in
    let add5 = make_adder 5 in

    print_int (add1 1);
    print_int (add5 5)
```

Powyższy kod wypisze wynik działań $1 + 1$ oraz $5 + 5$.

Niskopoziomowy język, taki jaki *LLVM IR* nie obsługuje zagnieżdżonych funkcji, można w nim zadeklarować jedynie procedury na takim samym, najwyższym poziomie (ang. top level). Konieczna jest transformacja wyrażeń *let*, polegająca na przeniesieniu ich na najwyższy poziom. Jeśli wykonamy taką transformacje na funkcji *add*, bez dodatkowych zmian, to zmienna x, przesanie być dostępna z ciała funkcji.

Listing 3..6: Po przeniesieniu funkcji *add* na najwyższy poziom

```
let add y = x + y
let make_adder x = add
```

Powyższy kod nie jest poprawnym programem w języku *OCaml*, oraz nie mógbły zostać poprawnie przetłumaczony na kod *LLVM IR*. Skoro x jest poza zasięgiem ciała *add*, można zaproponować rozwiązanie w którym wprowadzona zostaje globalna zmienna, odpowiadająca zmiennej wolnej x. Na poniższym przykładzie zostało przedstawione jak mogłaby wyglądać taka transformacja.

Listing 3..7: Wprowadzenie globalnej zmiennej.

```
let global_x = ref 0

let add y = !x + y
let make_adder x =
    global_x := x
    add
```

Powyższe przekształcenia nie wprowadzają dużego narzutu na wynikowy program i rozwiązują problem z zasięgiem symbolu x. Ten kod jednak nie zwróci poprawnego wyniku, przy założeniu o statycznym zasięgu widoczności (ang. static scoping). Drugie wywołanie *make_adder* dla argumentu 5, nadpisze jego pierwszą wartość z której korzysta pierwsze wywołanie funkcji *add1*. Koniecznym jest zapamiętanie x w środowisku funkcji *add*, w momencie w którym jest zwracana.

W *Langu*, do implementacji *closure conversion* postanowiłem wykorzystać, już zaimplementowaną częściową aplikację. W czasie analizy programu dla każdego zagnieżdżonego wyrażenia *let* wyznaczam jego zmienne wolne. Zmienne wolne zostaną dodane jako dodatkowe argumenty, przed tymi podanymi pierwotnie. Następnie, symbol pod którym wyrażenie *let* było zapamiętane w środowisku, zostaje związany z częściową aplikacją zmienny wolnych do oryginalnej funkcji (rozszerzonej o dodatkowe argumenty – zmienne wolne). Poniżej została na funkcji *add* została wykonana ta transformacja.

Listing 3..8: Wprowadzenie globalnej zmiennej.

```
let make_adder x =
    let add_extended_with_free_vars x y = x + y
    let add = add_extended_with_free_vars x
```

```
add
```

Po tym etapie, funkcję *add_extended_with_free_vars* można przenieść na najwyższy poziom (*lambda lifting*). *add* jest teraz zwykłym przypisaniem wyrażenia do zmiennej, więc może być łatwo przetłumaczone na niskopoziomowy kod.

3.6. Rekordy

Rekordy są podstawowym sposobem na tworzenie własnych typów danych w wielu językach programowania. Do *Langa* zostały wprowadzone głównie po ty, aby urozmaici przykłady zastosowania klas typów.

Jako że LLVM IR wspiera struktury, które są odpowiednikiem implementowanych rekordów, dodanie ich do języka nie stanowiło problemu. W obecnej implementacji wszystkie struktury alokowane są na stercie i przekazywane przez wskaźnik. Pola struktury są pamiętane przez ich wartość, chyba że polem jest inna struktura. Są trwałym typem danych, więc aktualizacja pól struktury z wyrażeniem *with*, powoduje skopiowane jej zawartości do nowej instancji.

3.7. Let polimorfizm

Bez let polimorfizmu, którego odpowiednikiem w językach imperatywnych jest polimorfizm parametryczny, ciężko wyobrazić sobie nowoczesny język. Mimo wygody jaką dostarcza programiście, często wiąże sie z dodatkowym obciążeniem czasowym i pamięciowym. Polimorficzna funkcja `let identity x = x` może być użyta niezależenie od typu podanego argumentu. Jednak, jeśli funkcja *identity* jest aplikowana do argumentów typu *int* i textitfloat, to nie jest jasne jak powinien wyglądać jej wygenerowany kod. Argument typu *float* zostałby przekazany przez specjalny rejestr dla liczb zmiennoprzecinkowych, inny od tego dla argumentu typu *int*. W wygenerowanym kodzie jasno należy określić jaki wariant będzie wspierać dana funkcja. Dlatego wiele języków rozwiązuje ten problem poprzez jednorodną reprezentację wszystkich typów, które mogą być użyte w funkcjach polimorficznych. Jednorodna reprezentacja sprowadza się do alokowania wartości obiektu na stercie, a następnie przekazywanie wskaźnika na ten obiekt. Wszystkie wskaźniki niezależnie od typu i rozmiaru obiektu na który wskazują, mają ten sam rozmiar i są przekazywane w ten sam sposób. Niesie to ze sobą kilka wad. Przykładowo, dla każdego *inta* który sam zajmuje 4 bajty, dodatkowo alokowane jest 8 bajtów na wskaźnik do niego. Jako, że jest zaalokowany na stercie, będzie musiał być ręcznie zwolniony przez programistę lub przez automatyczne odśmiecanie pamięci (które często występuje w językach funkcyjnych).

Do języków które stosują powyższą metodę należą m. in. Java i Haskell. W

Haskellu konieczna jest taka reprezentacja danych także ze względu na jego leniwość. Dostępne w nim są także prymitywne typy reprezentowane przez ich wartość (ang. unboxed), takie jak *#Int* i *#Double*. Jednak nie mogą być one użyte w funkcjach polimorficznych.

3. Sposób implementacji u mnie

3.8. Inferencja typów

???? 1. Po co? Jak działa 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.

Podsumowanie

4.1. Wnioski

4.2. Dalsze prace

Rozdział 5.

Instrukcja obsługi

5.1. Instalacja

5.2. Sposób użycia z przykładami

5.3. Użyte narzędzia i biblioteki

5.4. 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

[6] Indentation. Python Reference Manual.
https://docs.python.org/2.5/ref/indentation.html

[7] Levity Polymorphism.
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/11/levity-pldi17.pdf