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

Popularne kompilatory języków funkcyjnych z rodziny ML często charakteryzują się podobnymi kompromisami przy implementacji częściowej aplikacji i polimorfizmu parametrycznego. Zazwyczaj nie pozwalają na przekazywanie dowolnych typów przez ich wartość do funkcji polimorficznych. W swojej pracy przedstawię implementację kompilatora języka funkcyjnego, w której unikam jednorodnej reprezentacji danych przez wskaźnik dzięki monomorfizacji. Zmniejszając narzut pamięciowy przy użyciu funkcji polimorficznych decyduję się zwiększenie rozmiaru wygenerowanego programu. Moja implementacja częściowej aplikacji, choć bazowana na popularnym modelu *push/enter*, radzi sobie z problemem manualnego zarządzania stosem.

Drugim celem pracy jest implementacja prostych klas typów w języku z rodziny ML, wprowadzając tym samym *ad–hoc* polimorfizm. Będą one głównym elementem języka pozwalającym na modularyzację programu. Odbiega to od standardowo stosowanego systemu modułów. Monomorfizacja funkcji pozwoli na statyczne wybranie odpowiednich instancji klasy.

TODO english abstract, pozostawiam na koniec.

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 języków programowania wywodzących się z *SMLa* takich jak *OCaml* i *F#*. W swojej pracy zaimplementowałem kompilator języka funkcyjnego *Lang*, bazującego na języku F#. Zaimplementowałem w nim najważniejsze cechy języków z rodziny ML, starając się uniknąć wad rozwiązań z popularnych kompilatorów. Ponadto wprowadziłem do niego *klasy typów* zamiast skomplikowanego systemu modułów. Porównam swoje rozwiązania z innymi rozwiązaniami i omówię kompromisy na które się zdecydowałem.

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 celem jest wprowadzenie ad–hoc polimorfizmu do języka, mogą po części także spełnić to zadanie [2]. Są obecne w językach takich jak Haskell, Scala[15] czy Rust [16]. Fakt, że pojawiają się w nowych językach ogólnego zastosowania, świadczy o ich atrakcyjności z punktu widzenia programisty.

Jako pierwsze pojawiły się w języku Haskell[17]. 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 i 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, a 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* [4]. 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 z jednym parametrem, wprowadzając ad–hoc polimorfizm do języka.

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)
```

1.2. Efektywna implementacja języka funkcyjnego

Kolejnym 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 jednak duże wyzwanie przy kompilacji języka funkcyjnego ze względu na konieczność translacji abstrakcyjnych i wysokopoziomowych konstrukcji do języka niskiego poziomu. Uzyskanie podobnej lub lepszej wydajności niż popularne kompilatory języków funkcyjnych jest trudne, gdyż te stosują dużą liczbę skom-

plikowanych optymalizacji. Skupię się nad tym, aby moja implementacja prostego języka funkcyjnego, była porównywalna wydajnością z popularnymi rozwiązaniami bez dodatkowych optymalizacji. Omówię i porównam sposoby w jaki zdecydowałem się zaimplementować: częściową aplikację, zagnieżdżone funkcje, polimorfizm i klasy typów. Moje rozwiązania bazują na pomysłach z różnych języków programowania, w tym imperatywnych.

1.3. Infrastruktura LLVM

W celu uproszczeniu konstrukcji nowego kompilatora i ułatwienia pracy z generowaniem niskopoziomowego kodu, zdecydowałem się skorzystać z infrastruktury LLVM [5]. 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. Oznacza to że, 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. 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* w definicji funkcji dany argument może mieć ogólny typ, jeśli nie został on ukonkretniony w jej ciele. 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.

1.5. 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* [7], 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. Ich składnia została zaczerpnięta z Haskella.

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 lub 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*. 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 wciętych bloków kodu na poziomie parsera. Lepszym pod względem czytelności gramatyki, jest wykonanie tej czynności na etapie lexera, tak jak to ma miejsce w `F#`. Dokładniejszy opis sposobu parsowania składni bazującej na wcięciach znajduje się w rozdziale 3.2.1..

Poniższa gramatyka jest w notacji *EBNF*.

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

Listing 2..1: Przykłady wyrażeń warunkowych.

```
let isZero x = if x = 0 then true else false

let rec factorial x =
    if x = 0
    then 1
    else x * factorial (x - 1)

let rec pow x n =
    if (n = 0) && (x = 0)
    then
      0
    elif n = 0
    then
      one ()
    elif n = 1
    then x
    else
      mult x (pow x (n - 1))
```

2.2.2. Wyrażenia arytmetyczne i logiczne

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

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

Listing 2..2: Przykłady wyrażeń arytmetycznych i logicznych.

```
let isZero x = x = 0
let add5 x = x + 5
let mod2 x = x - ((x / 2) * 2)
let alwaysTrue x = (mod2 x = 0) || (mod2 (x + 1) = 0)
```

2.3. Deklaracja funkcji (wyrażenie let)

Argumenty funkcji muszą być w tym samym wierszu co słowo *let*. Po znaku = ciało funkcji może być złożonym wyrażeniem o ile zaczyna się w następnym wierszu i jest w późniejszej kolumnie niż *let*. Wyrażenie *let* może być zdefiniowane w jednej linii, o ile jego ciało jest pojedynczym wyrażeniem prostym. Standardowo dla języków z rodziny ML należy explicite zaznaczyć słowem kluczowym *rec* jeśli funkcja jest rekurencyjna. Dodatkowo do argumentów funkcji jak i jej wyniku można dodać adnotację typu.

Listing 2..3: Deklaracja funkcji z anotacjami.

```
let compose (f : 'b -> 'c) (g : 'a -> b) (x : 'a) : 'c = f (g x)

let printInt (x : int) : () = ll_putint x (* funkcja wewnętrzna *)

let _ =
    printnInt (120 * 120)
    printnInt (compose square factorial 5)
```

2.3.1. Wzajemnie rekurencyjne wyrażenia let

Możliwe jest także zdefiniowanie globalnych (ang. top level) funkcji wzajemnie rekurencyjnych z użyciem słowa *and*. Deklaracja pierwszej funkcji musi zaczynać się od *let rec*, a kolejne od *and*.

Listing 2..4: Funkcje wzajemnie rekurencyjne.

```
let rec even x =
    if x = 0
    then 0
    else odd (x - 1)

and odd x = 1 + (even (x - 1))
```

2.4. Rekordy

2.4.1. Deklaracja rekordu

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

Listing 2..5: Definicja rekordu.

```
type simple = { a : int; b : int }
type complex = { n : int; s : simple }

type point3d =
  { x : int
    y : int; z : int
  }
```

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. W 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⟩ ;

Listing 2..6: Literał rekordu.

```
let s : simple = { a = 0; b = 1}

let p3 : point3 =
  { x = 10
    y = 1000000
    z = -10
  }
```

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.

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

Listing 2..7: Uaktualnianie rekordu.

```
let s : simple = { a = 0; b = 1}

let add (curr : complex) : simple =
    {  curr.s with a = curr.s.b
                   b = curr.s.a + curr.s.b
    }
```

2.5. Klasy typów

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

2.5.1. Deklaracja klasy

Deklaracja klasy została zaadoptowana z Haskella.

Listing 2..8: Deklaracja klasy.

```
class Print 'a where
    print : 'a -> ()

class Pow 'a where
    one  : () -> 'a
    mult : 'a -> 'a -> 'a
```

2.5.2. Deklaracja instancji

Listing 2..9: Instancja klasy.

```
instance Pow int where
    let one () = 1
```

```
    let mult x y = x * y

instance Print int where
    let print x = printInt x

instance Print Vec where
    let print (v : Vec) =
        printInt v.x
        printSpace ()
        printInt v.y
```

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.

Listing 2..10: Tworzenie i importowanie modułów.

```
module Prelude =
    module Internal =
        external ll_putint    : int -> ()
        external ll_print_bool : bool -> ()
        external ll_print_line : () -> ()
        external ll_print_space : () -> ()

    let printInt x = Internal.ll_putint x

    open Internal

    let printNl = ll_print_line
    let printSpace = ll_print_space
    let printBool b = ll_print_bool b

open Prelude
open Prelude.Internal
```

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 w 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 symbole rozpoczynający [| i kończący |] muszą być w tej samej kolumnie.

Listing 2..11: Tablice.

```
let arr : int array = [| 1; 2; 0 + (6 / 2); add 1 3|]

let multiLineArr : int array =
  [|1; 2;
    0 + (6 / 2)
    add 1 3
  |]
```

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 jednego bajtu).

Listing 2..12: Interfejs zewnętrzny.

```
external set_array_elem : int array -> int -> int -> ()
```

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. Implementacja w pliku: `lang-compiler/compiler/parsing/lexer.cppo.sedlex.ml`

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. Implementacja w pliku: `lang-compiler/compiler/parsing/grammar.mly`

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).

   Implementacja w pliku: `lang-compiler/compiler/typed_ast.ml`

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. Implementacja w pliku: `lang-compiler/compiler/codegen.ml`

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 [6].

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* [9]. 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`.

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 *double* jest statycznie znana.

Listing 3..1: Wywołanie statycznie znanej funkcji.

```
let double x = x + x
let main =
  printInt (double 4)
```

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.

```
let getIdentity () =
  let id x = x
  id

let apply f x = f x

let main b =
  let c = apply b
  let a = getIdentity ()
  (a 1 = b 1) && (b 1 = c 1)
```

```
let main (getIdentity ())
```

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.

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 (dynamicznie zaalokowana tablica z C),
- pozostała liczba argumentów koniecznych do zaaplikowana, aby należało wywołać wskazywaną funkcje,

- arność funkcji,
- liczba bajtów w środowisku.

Definicja takiej struktury w *C* wyglądałaby następująco:

Listing 3..3: Rozwinięta funkcja w OCamlu.

```
struct function {
    void (*fn)();
    unsigned char *args;
    unsigned char left_args;
    unsigned 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 ciele funkcji *apply*, w pierwszym przypadku funkcja *f* zostanie wywołana, a w drugim, pomimo aplikacji tych samych argumentów i tego samego typu funkcji, nic nie zostanie wywołane.

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

```
let called a b =
  let inner c d = a + b + c + d in
  print_string "called\n";
  inner

let not_called a b c d = a + b + c + d

let apply f a b c : int -> int =
    f a b c

let _ =
    apply called 1 2 3;
    apply not_called 1 2 3
```

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.

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 (mogą mieć różny rozmiar), 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 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 w kolejności malejących prefiksów. Gdy tworzona jest instancja struktury *function*, jako 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 — 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 pozostały żadne argumenty 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

Istnieję dwa modele realizacji częściowej aplikacji: push/enter i eval/apply. Zostały dokładnie opisane przez Marlow i Peyton Jones[14]. Ich zasadnicza różnica polega na sposobie działania przy aplikacji funkcji nieznanej. W push/enter przed właściwym wywołaniem funkcji jej argumenty są ładowane na stos. Następnie to funkcja wołana jest odpowiedzialna rozważenie wszystkie przypadków związanych

z liczbą zaaplikowanych argumentów. Podejmuje decyzje o wywołaniu właściwiej funkcji i ewentualnym przekazaniu pozostałych argumentów dalej bądź zwróceniu obiektu reprezentującego częściowo zaaplikowana funkcje. W modelu eval/apply, to po stronie funkcji wołającej leży zadanie rozważania tych przypadków. Strona wołająca (ang. call site) zna liczbę zaaplikowanych argumentów, a w czasie działania sprawdza faktyczną arność funkcji.

Moje rozwiązanie istotnie czerpie z modelu push/enter. Większość decyzji jest podejmowana po stronie funkcji wołanej. Strona wołająca przekazuje wszystkie argumenty do funkcji wskazywanej w strukturze *function*. Jednak zanim to się stanie, to strona wołająca dynamicznie sprawdza arność funkcji. W przypadku gdy jest ona większa od liczby zaaplikowanych argumentów, to w tym miejscu uaktualniany jest obiekt częściowej aplikacji, oszczędzając na zbędnym wywołaniu funkcji. Jest to podobne do rozwiązania z enter/apply. Strona wołająca nie musi także dynamicznie przeszukiwać stosu w poszukiwaniu przekazanych argumentów. Dzięki generacji wielu funkcji wejściowych, funkcja wołana wie dokładnie ile argumentów i o jakim typie jest w środowisku, a jakie zostały właśnie przekazane.

Marlow i Peyton Jones w pracy *Making a Fast Curry: Push/Enter vs. Eval/Apply* jasno zalecają korzystanie z modelu eval/apply ze względu na prostszą implementację i trudności w kompilacji push/enter do przenośnego języka takiego jak C `C--` czy LLVM. Wspomniana trudność wynika z konieczności manualnego zarządzania stosem, co nie jest łatwe (o ile możliwe) w takich językach. Jednak moje rozwiązanie unika tego problemu, a ponadto umożliwia przekazywanie argumentów przez rejestry, co zostało wykluczone dla modelu push/enter. Te cechy zostały osiągnięte dzięki generacji wielu funkcji wejściowych, dla każdego sufiksu argumentów z typu funkcji. Niesie to ze sobą jednak istotną wadę, nieobecną w żadnym z dwóch wspomnianych modeli — generowanie dużej ilości dodatkowego kodu. Nie jest to problemem dla małych przykładowych programów, jednak mogłoby znacznie wydłużyć czas kompilacji i rozmiar wynikowego programu przy dużych, profesjonalnych projektach.

3.5. Zagnieżdżone funkcje

Zagnieżdżone funkcje są nieodłączną częścią języków funkcyjnych. Ich implementacja wykorzystuje *closure conversion*, które zostało już użyte 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 transformacja 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, globalnym poziomie (ang. top level). Konieczna jest transformacja wyrażeń *let*, polegająca na przeniesieniu ich na globalny poziom. Jeśli wykonamy taką transformacje na funkcji *add*, bez dodatkowych zmian, to zmienna x przestanie być dostępna z ciała funkcji.

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

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

Powyższy kod nie jest poprawnym programem w języku *OCaml*, oraz nie mógłby 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 zmiennych wolnych do oryginalnej funkcji (rozszerzonej o dodatkowe argumenty — zmienne wolne). Poniżej 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 globalny 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 to, aby urozmaicić 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 *float*, 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 [18] i Haskell [19](sekcja *Haskell implementation*). 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 types), takie jak `Int#` i `Double#`. Jednak nie mogą być one użyte w funkcjach polimorficznych. W fazie optymalizacji, kompilator *GHC* może zamienić typ boxed na unboxed, ale nie jest to gwarantowane. Jako że kierowałem się wydajnością przy implementacji kompilatora *Langa*, to rozwiązanie nie jest satysfakcjonujące.

OCaml reprezentuje *inty* i wskaźniki na jednym słowie maszynowym. To czy do funkcji został przekazany wskaźnik, czy *int* przez wartość jest rozpoznawane na podstawie najniższego bitu, który jest traktowany jak znacznik [20]. Jeśli wynosi 1 to dana wartość jest *intem*. To rozwiązania pozwala na uniknięcie większości narzutu przy wykonywaniu operacji na typach prymitywnych, ale wiąże się z niestandardowym podejście do arytmetyki liczb i wciąż nie pozwala na przekazywanie przez wartość obiektów dłuższych niż słowo maszynowe. W związku ze wspominanymi wadami tego rozwiązania, nie zdecydowałem sie go zaimplementować.

Języki takie jak `C++` i *Rust* oraz kompilator *SMLa* MLton [21], wyspecjalizowują każdą polimorficzną funkcję przed fazą generowania kodu. Ten proces nazywany jest monomorfizacją. Dokładniej, polega on na utworzenia osobnej implementacji polimorficznej funkcji dla każdego konkretnego typu, który został podstawiony pod typ ogólny. W `C++` użytkownik może explicite podać dla jakich typów ukonkretnia daną funkcje, bądź może być to wykryte przez kompilator.

Listing 3..9: Polimorficzna funkcja w `C++` z użyciem szablonów (ang. template)

```
template<class Type1, class Type2>
void foo(Type1 t1, Type2 t2) {
    // ...
}

int main() {
    foo<std::string, int>("hello world", 42);
```

```
    foo("hello world2", 24);
    foo(1.0, 4.0);
}
```

Dla funkcji *foo* z powyższego przykładu zostaną wygenerowane dwie implementacje, a oryginalne wywołania funkcji zostaną już w fazie kompilacje zamienione na wywołania odpowiednich, wyspecjalizowanych wersji tej funkcji. Takie rozwiązanie pozwala na przekazanie dowolnego typu przez jego wartość, bez zmian jego reprezentacji. Jednak nie jest ono pozbawione kompromisów. Statyczna monomorfizacja funkcji, nie pozwala na zastosowanie polimorficznej rekurencji. Dodatkowo, zwiększa długość wygenerowanego kodu, tym samym wydłużając czas kompilacji i zwiększać rozmiar wynikowego programu. Kolejną wadą w praktycznych zastosowaniach, jest fakt, że wysokopoziomowy kod implementacji funkcji polimorficznej musi być opublikowany wraz z wygenerowaną bibliotekę, aby istniała możliwość użycia wywołania tej funkcji dla typów dla których nie byłą wcześniej ukonkretniona. Mimo to, język `C++` cieszy się ogromną popularnością, a system szablonów jest szeroko używany, także w zastosowaniach produkcyjnych.

W *Langu* postanowiłem zaimplementować rozwiązanie bazujące na monomorfizacji. Jest ono zgodne z założeniami projektu oraz dobrze współgra z pozostałymi rozwiązaniami. W trakcie inferencji typów każda polimorficzna funkcja z *Langa* jest reprezentowana w kompilatorze jako funkcja z ukonkretnień typów ogólnych w implementację. Gdy w fazie generacji kodu dochodzi do wywołania funkcji polimorficznej, muszą być znane wszystkie ukonkretnienia. Wtedy wywoływana jest funkcja generująca implementację na podstawie typów konkretnych. Implementacje funkcji dla tych samych typów konkretnych są zapamiętywane.

3.8. Inferencja typów

Inferencja typów zwalnia programistę z obowiązku wyspecyfikowania typów każdej deklaracji, pozostawiając wszystkie zalety statycznego systemu typów. W *Langu* działa ona następująco. Na początku każdemu argumentowi, który nie został oznaczony przez użytkownika typem konkretny, zostaje przypisany różny typ ogólny. Podczas inferencji w ciele funkcji, dla każdego wyrażenia, dla którego może być wywnioskowane jakieś ograniczenie (np. warunek w *if* powinien mieć typ *bool*), odpowiednia równość między typami zostaje zapisana w drzewie ast tej funkcji. Po przetworzeniu funkcji, zostaje zbudowany graf równości między typami. Sprawdzane jest czy któreś argumenty w ciele funkcji zostały ukonkretnione, jeśli tak to definicja funkcji jest aktualizowana, jeśli nie to typ pozostaje ogólny. Graf równości typów jest także używany w celu znalezienia konkretnych typów przy wywoływaniu polimorficznej funkcji (dzieję się to w fazie generowania kodu).

3.9. Klasy typów

Najpopularniejszą implementacją klas typów, jest ta zastosowana w Haskellu. Jej idea polega na przekazywaniu słownika (rekordu) z implementacjami funkcji z danej instancji klasy (ang. dictionary passing) [23]. Przedstawię w jaki sposób działa ta implementacja, porównując kod używający klas typów w Haskellu i odpowiadający mu kod w OCamlu (w którym nie ma klas typów).

Listing 3..10: Deklaracja klasy typów w Haskellu

```
class Show a where
    show :: a -> String
```

Listing 3..11: Deklaracja odpowiednika klasy typów w OCamlu z użyciem metody przekazywania słownika

```
type 'a show = {
    show : 'a -> string
}
```

W przypadku Haskella zdefiniowaliśmy klasę typów *Show* z metodą *show* o typie $a \rightarrow String$. W przypadku *OCamla* musieliśmy stworzyć polimorficzny rekord z jednym polem *show*. Instancja tego rekordu będzie instancją tej klasy dla danego typu.

Na poniższym przykładzie w OCamlu, obie instancje *Show*, zwracają rekord z odpowiednimi implementacjami zapisanymi w polach rekordu.

Listing 3..12: Instancja klasy w Haskellu

```
instance Show String where
    show s = "'" ++ s ++ "'"

instance Show Bool where
    show True = "true"
    show False = "false"
```

Listing 3..13: Instancja klasy w OCamlu

```
let show_string = {
    show = fun s -> "'" ^ s ^ "'"
}

let show_bool = {
    show = function
           | false -> "false"
           | true -> "true"
}
```

Funkcja, która korzysta z typu będącego instancją klasy, musi przyjmować dodatkowy argument — słownik z implementacją metod klasy.

Listing 3..14: Instancja klasy w Haskellu

```
printArg :: Show a => a -> IO ()
printArg arg =
    putStrLn ("arg: " ++ show arg)

main1 = printArg True
main2 = printArg "Hello World"
```

Listing 3..15: Instancja klasy w OCamlu

```
let printArg show_instance arg =
    show_instance.show arg
    |> printf "arg: %s"

let main1 =
    printArg show_bool true

let main2 =
    printArg show_string "Hello"
```

Zasadnicza różnica między symulowaniem klas typów w języku ich nie wspierającym, a użyciem ich w Haskellu, jest to, że to do użytkownika należy udowodnienie istnienia instancji klasy dla danego typu.

Metoda przekazywania słownika nie niesie ze sobą dużego narzutu na czas kompilacji i wykonywania programu. Wymaga dodania co najmniej jednego argumentu do funkcji polimorficznych korzystających z klas typów. Jednak w implementacji klas typów w *Langu*, ze względu na decyzje podjęte w poprzednich cechach języka (monomorfizacja dla let polimorfizmu), nie skorzystałem z tej metody. Po przeprowadzeniu monomorfizacji w kodzie nie występują żadne funkcje polimorficzne. Jedyne co należy zrobić w kolejnej fazie, to sprawdzić czy dla danego typu i klasy istnieje instancja. Jeśli tak to użycie każdej metody z tej klasy dla konkretnego typu, należy zamienić na wywołanie odpowiedniej implementacji. Po tym przekształceniu, kod jest pozbawiony wszystkich konstrukcji klas typów, zawiera jedynie ich implementacje i wywołania odpowiednich metod. Oczywistą zaletą tego rozwiązania jest fakt, że te wysoko poziomowe abstrakcje nie wiążą ze sobą żadnego kosztu. Jest to istotna cecha w praktycznych zastosowania, bo pozwala na swobodne korzystanie z wysokopoziomowych konstrukcji języka, bez rozważania nad ich wpływem na wydajność. Jednak wady nad tym rozwiązaniem ciążą wszystkie wady monomorfizacji: brak rekurencji polimorficznej, brak polimorfizmu wyższych rzędów (ang. higher-rank polymorphism) oraz konieczność kompilacji całego kodu programu na raz. Z podobnego rozwiązania przy implementacji klas typów korzysta *MLton* [24].

Rozdział 4.

Podsumowanie

4.1. Wnioski

Najtrudniejszym element przy tworzeniu kompilatora dla języka funkcyjnego okazała się efektywna implementacja częściowej aplikacji. Główną trudnością było spełnienie założeń o optymalnym przekazywaniu zmiennych i unikaniu jednorodnej reprezentacji różnych typów danych poprzez boxowanie. Pomimo że jest to standardowa cecha wielu języków i istnieją opisy jej implementacji w części z nich, to cele, które postawiłem zaowocowały odmiennym rozwiązaniem.

Drugim celem projektu, było wprowadzenie klas typów do języka z rodziny ML. Zastosowanie monomorfizacji znacznie ułatwiło ich implementację. Te proste, rozszerzenie języka znacznie zwiększyło jego możliwości. Pozwoliło na pisanie modularnych funkcji oraz abstrakcję funkcjonalności od reprezentacji danych. Ponadto nie wprowadziły one żadnego narzutu pamięciowego i czasowego.

4.2. Dalsze prace

Tworzenie kompilatorów jest obszernym i czasochłonnym zadaniem. Zdecydowałem się pominąć pewne standardowe cechy języków funkcyjnych, aby lepiej skupić się na celach projektu. W przypadku kontynuacji pracy na językiem *Lang* i kompilatorem, naturalnym rozwinięciem byłby następujące elementy:

- Wyrażenia lambda. Mogłyby być zrealizowane poprzez ich zamianę na globalne wyrażenia let o unikalnych nazwach.
- Klasy typów z wieloma parametrami. Nie są obecne w standardzie Haskell 98, jednak istnieje możliwość ich włączenia poprzez odpowiednią dyrektywę. Ich implementacja byłaby łatwym rozszerzeniem obecnej (z jednym parametrem), jednak dołożenie ich do języka wiązałoby się z pewnymi niejednoznacz-

nościami [12] przy wyborze odpowiedniej instancji klasy. Koniecznym mogłoby by sie okazać dodanie zależności funkcyjnych [11].

- Kompilacja wielu plików. Obecna implementacja pozwala na skompilowanie jednego pliku. Umożliwienie kompilacji wielu plików w jednym projekcie nie jest trudnością, gdyż wystarczy połączyć pliki w odpowiedniej kolejności (lub też pozwolić użytkownikowi na zdefiniowanie ich kolejności). Tworzenie przenośnych bibliotek może okazać się wyzwaniem ze względu na zastosowanie monomorfizacji.

- Składnia pozwalająca na opcjonalne użycie słów kluczowych, które tworzą kontekst (w OCamlu to `begin`, `end`, `in` oraz nawiasy). Jest to cecha znana z języka *F#*. Należałoby przenieść wykrywanie całych bloków wciętego kodu do etapu analizy leksykalnej. Wcięcia generowały by różne tokeny (nie tylko *INDENT* i *DEDENT*), w zależności od kontekstu w których się znajdują. Dokładny opis tego rozwiązania znajduje sie w Specyfikacji języka *F#* 4.0[13].

- Reprezentowanie małych rekordów poprzez ich wartość. Rekord zawierający dwa pola będącymi intami może być zapamiętany w jednym słowie maszynowym. Obecnie wszystkie rekordy są pamiętane przez wskaźnik do ich zawartości, jednak dzięki monomorfizacji, nic nie stoi na przeszkodzi aby zastosować tę optymalizację.

Rozdział 5.

Instrukcja obsługi

5.1. Instalacja

Projekt można skompilować w systemie *Linux*. Uprzednio należy wykonać następujące kroki.

1. Instalacja kompilatora OCamla. Instrukcja dostępna na stronie: `https://ocaml.org/docs/install.html`.

2. Instalacja LLVMa. Instrukcja dostępna na stronie: `https://llvm.org/`.

3. Wymagany jest kompilator LLVM w wersji co najmniej 5. Zainstalowaną wersję można sprawdzić poleceniem:

```
$ llc --version
```

4. Ustawienie wersji kompilatora:

```
$ opam switch 4.05.0
```

5. Konfiguracja menadżera pakietów:

```
$ eval ‘opam config env‘
```

6. Instalacja bibliotek. Należy uruchomić skrypt `install_deps.sh` instalujący odpowiednie pakiety z użyciem *opama*. Plik znajduje się w głównym folderze `lang-compiler`.

```
$ ./install_deps.sh
```

5.2. Sposób użycia z przykładami

5.3. Użyte narzędzia i biblioteki

- `OCaml`
- `Menhir`
- `Sedlex`
- `OCaml-parsing`
- `Core`
- `JBuilder`

5.4. Struktura projektu

- `compiler/` – compiler library which contains: lexer, parser and codegen
- `compiler/grammar.mly` – gramatyka w składni Menhira.
- `compiler/lexer.cppo.sedlex.ml` – lexer
- `compiler/codegen.ml` – main *LLVM IR* code generation
- `compiler/codegenUtils.ml` – some helper functions for code generation
- `compiler/ast.ml` – abstract syntax tree
- `langc/` – Plik wykonywalny Interfejs
- `test/` – Testy kompilatora.
- `test/compiler-test-srcs` – input files for compiler tests

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] Using Trait Objects that Allow for Values of Different Types. Rust.
https://doc.rust-lang.org/book/second-edition/ch17-02-trait-objects.html

[5] LLVM.
https://llvm.org/

[6] "Implementing a language with LLVM" tutorial.
https://llvm.org/docs/tutorial/OCamlLangImpl1.html

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

[8] F# Language - User Voice. Suggestions about Future evolution of the F# Language and Core Library.
https://fslang.uservoice.com/forums/245727-f-language/filters/top

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

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

[11] Functional dependencies.
https://wiki.haskell.org/Functional_dependencies

[12] Multi-parameter type class
https://wiki.haskell.org/Multi-parameter_type_class

[13] Specyfikacja języka F# 4.0
https://fsharp.org/specs/language-spec/4.0/FSharpSpec-4.0-final.pdf

[14] Making a Fast Curry: Push/Enter vs. Eval/Apply for Higher-order Languages.
Simon Marlow, Simon Peyton Jones
https://www.microsoft.com/en-us/research/wp-content/uploads/2016/07/eval-apply.pdf

[15] Scala docs: Traits.
https://docs.scala-lang.org/tour/traits.html

[16] Rust by example. Traits.
https://doc.rust-lang.org/rust-by-example/trait.html

[17] Type class. Wikipedia.
https://en.wikipedia.org/wiki/Type_class?oldformat=true

[18] Type Erasure. The Java Tutorials.
https://docs.oracle.com/javase/tutorial/java/generics/erasure.html

[19] Type Systems, Type Inference, and Polymorphism.
http://www.cs.tau.ac.il/ msagiv/courses/pl14/chap6-1.pdf

[20] Chapter 20. Memory Representation of Values. Real World OCaml.
https://v1.realworldocaml.org/v1/en/html/memory-representation-of-values.html

[21] Monomorphise. MLton.
http://mlton.org/Monomorphise

[22] Template (C++). Wikipedia.
https://en.wikipedia.org/wiki/Template_(C%2B%2B)?oldformat=true

[23] How to make *ad–hoc* polymorphism less *ad–hoc*. P. Wadler, S. Blott
https://people.csail.mit.edu/dnj/teaching/6898/papers/wadler88.pdf

[24] Implementing, and Understanding Type Classes.
http://okmij.org/ftp/Computation/typeclass.html