Implementacja języka funkcyjnego z rodziny ML z wykorzystaniem infrastruktury LLVM

(An implementation of a ML-family functional language using the 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 wymagają jednorodnej reprezentacji danych, więc opakowują wszystkie polimorficzne argumenty we wskaźnik. Częścią tej pracy jest implementacja kompilatora języka funkcyjnego *MonoML*, w której dzięki monomorfizacji unikam jednorodnej reprezentacji danych. Zmniejszając narzut pamięciowy związany z implementacją funkcji polimorficznych, decyduję się na 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, co prowadzi do wzbogacenia języka o *ad–hoc* polimorfizm. Są one głównym elementem języka pozwalającym na modularyzację programu. Odbiega to od standardowo stosowanego systemu modułów w większości języków z rodziny ML. Monomorfizacja funkcji pozwala na statyczne wybranie odpowiednich instancji klasy.

Popular compilers for ML-family functional programming languages, often made similiar trade-offs when implementing partial aplication and parametric polimorphism. They usually require uniform type representation which leads to boxing of all polymorphic arguments. In this thesis I present an implementation of a compiler for *MonoML* — a functional programming language, in which I choose to avoid uniform data representation by boxing. Monomorphization allows to decrease memory overhead for polymorphic functions, however it increases overall code size. My implementation of partial application is based on the *push/enter* model. However, I manage to overcome the issue of manual stack management, related to *push/enter* model.

Another goal of this thesis is to introduce *ad–hoc* polymorphism in a ML-family programming language, by implementing *type classes* in *MonoML*. They are the main feature which allows to modularize programs. This approach is different from modules which are usually present in ML-family languages. Monomorphization allows us to statically choose an appropriate instance for each class.

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 — standaryzowana wersja języka ML. Już wtedy zawierał m.in. rozwijanie funkcji, dopasowanie do wzorca, inferencje typów oraz moduły parametryczne [22]. Są to elementy, które cechują większość dzisiejszych języków programowania wywodzących się z SMLa takich jak OCaml i F#.

Wysokopoziomowe funkcjonalności języków funkcyjnych, takie jak częściowa aplikacja i polimorfizm parametryczny, wiążą się z dodatkowym narzutem pamięciowym i czasowym. W wielu kompilatorach implementacja polimorfizmu parametrycznego wiążę się z koniecznością zastosowania jednorodnej reprezentacji danych poprzez opakowywanie argumentów we wskaźnik (ang. boxing). W tej pracy prezentuję, odmienne od standardowego, podejście do implementacji częściowej aplikacji i polimorfizmu parametrycznego, które unika niepotrzebnego narzutu wydajnościowego. Zaimplementowałem kompilator języka funkcyjnego *MonoML*, bazującego na języku F#. Skupiając się na możliwe optymalnej implementacji najważniejszych cech języków z rodziny ML, zdecydowałem się na inne kompromisy niż popularne kompilatory. Zastosowałem monomorfizację, która pozwala na zmniejszenie narzutu pamięciowego związanego z implementacją funkcji polimorficznych, kosztem zwiększonego rozmiaru wynikowego programu i trudności przy tworzeniu przenośnych bibliotek.

Języki z rodziny ML zazwyczaj zawierają skomplikowany system modułów, jako główną cechę pozwalającą na modularyzację pisanych w nich programów. *Klasy typów*, choć spopularyzowane przez Haskella, pojawiają się w nowy językach ogólnego zastosowania. Są prostsze w implementacji i użyciu niż system modułów oraz umożliwiają pisanie skomplikowanych i modularnych programów, bez duplikowania kodu. Mimo to nieznany jest żaden popularny język z rodziny ML zawierający klasy typów, dlatego zdecydowałem się je zaimplementować w MonoMLu.

1.1. Efektywna implementacja języka funkcyjnego

Istotnym celem tej pracy jest implementacja głównych cech języków funkcyjnych w możliwie optymalny sposób. Skupię się na optymalizacji czasu wykonania programu kosztem rozmiaru 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 skomplikowanych 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.

1.2. Infrastruktura LLVM

W celu uproszczeniu konstrukcji nowego kompilatora i ułatwienia pracy z generowaniem niskopoziomowego kodu, zdecydowałem się skorzystać z infrastruktury LLVM [6]. 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 (procedur), zmiennych globalnych, podstawowych bloków, przypisań oraz wywołań funkcji. Podstawowe bloki kodu jak i funkcje nie mogą być zagnieżdżone.

W moim kompilatorze nie generuję kodu LLVMa bezpośrednio, 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.

1.3. 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.

1.4. Rozwijanie funkcji oraz częściowa aplikacja

Częściowa aplikacja występuje wtedy, gdy po zaaplikowaniu mniejszej liczby argumentów niż wynosi arność funkcji, otrzymujemy nową funkcję. Przykładowo dla funkcji $f : (A \times B \times C) \to D$ o arności 3, po częściowym zaaplikowaniu jej do pierwszego argumentu $a : A$ otrzymujemy funkcję $g : (B \times C) \to D$ o arność 2. Formalnie, w trakcie procesu częściowej aplikacji, funkcja f musiała zostać zamieniona na funkcję $A \to (B \times C) \to D$, aby następnie móc zaaplikować do niej argument $a : A$. Wartość a musi być zapamiętana w środowisku nowo powstałej funkcji g. W szczególności dla dowolnego $b : B$ i $c : C$ musi zachodzić: $g(b, c) = f(a, b, c)$.

Częściowa aplikacja jest spotykana nie tylko w językach funkcyjnych. Przykładowo, biblioteka standardowa języka C++ dostarcza funkcję *bind* [15], która pozwala na zaaplikowanie części argumentów, w dowolnej kolejności.

Częściową aplikacje można osiągnąć poprzez rozwinięcie funkcji (ang. currying) do wielu funkcji jednoargumentowych. Funkcja f z powyższego przykładu w rozwiniętej formie ma typ $f : A \to B \to C \to D$. Funkcje w takiej postaci, wspierają częściową aplikację bez użycia dodatkowych funkcjonalności języka (na przykład funkcji

bind w języku C++). Na poniższym fragmencie kodu języka Javascript znajduje się przykład wprowadzenia częściowo aplikowalnej funkcji poprzez jej rozwinięcie.

```
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 powyższego przykładu. Ta sama funkcja zdefiniowana w OCamlu wygląda następująco:

```
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 sekcji 3.4.

1.5. 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 [26]. Są obecne w językach takich jak Haskell, Scala[13] czy Rust [12]. Fakt, że pojawiają się w nowych językach ogólnego zastosowania, świadczy o ich atrakcyjności z punktu widzenia programisty.

Jako pierwsze pojawiły się w języku Haskell[16]. 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 [2] (ang. Higher-

rank polymorphism). Inną istotną różnicą jest fakt, że klasa typów z Haskella nie definiuje nowego typu, a jedynie pozwala na ograniczenie typu polimorficznego do instancji klasy. *Cecha* z Rusta może być użyta jak zwykły typ, przykładowo można stworzyć listę zawierające obiekty, które są różnymi instancjami (implementacjami) *cechy* [19]. 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. Definiujemy klasę `Eq` zawierającą dwa operatory: $==$ oraz $/=$.

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

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:

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

Rozdział 2.

Język *MonoML*

2.1. Inspiracja

Składnia języka *MonoML* 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, pod względem składni, 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.

Gramatyka w notacji *EBNF* przedstawiona w poniższych sekcjach jest zbliżona do rzeczywistej składni języka. Dokładny opis gramatyki znajduje się w pliku `monoml-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..

2.2. Podstawowe wyrażania

2.2.1. Wyrażenia warunkowe

Składnia i semantyka wyrażeń warunkowych jest bardzo podobna do tej w F#. W języku MonoML 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*. Listingi 2.2.1 i 2.2.2 prezentują składnię wyrażeń warunkowych i przykład ich użycia w języku MonoML.

Listing 2.2.1: Wyrażenia warunkowe.

```
<simple-if-exp> ::=
    if <simple-exp>
    then <simple-exp>
    <simple-elif-exp>? <simple-else-exp>?

<if-exp> ::=
    if <simple-exp> <newline>
    then
    <body-exp>+ <elif-exp>* <else-exp>

<simple-else-exp> ::=
    else <simple-exp>

<simple-elif-exp> ::=
    elif <simple-exp>
    then <simple-exp>

<elif-exp> ::=
    elif <body-exp>+
  | <simple-elif-exp>

<else-exp> ::=
    else <body-exp>+
  | <simple-else-exp>
```

Listing 2.2.2: 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))
```

Sematyka wyrażeń warunkowych

Wyrażenie w warunku musi mieć typ logiczny bool. Wszystkie gałęzie wynikowe wyrażenia warunkowego muszą mieć taki sam typ. Jeśli typ wyrażenia po `then` to `unit`, przypadek `else` może zostać pominięty. Jest to podejście znane z pozostałych języków z rodziny ML.

2.2.2. Wyrażenia arytmetyczne i logiczne

Wyrażenia arytmetyczne i logiczne mają taką samą składnię i semantykę jak w pozostałych językach z rodziny ML. Obsługiwane są operatory $$, ||, =, +, −, ∗, /, oraz pozostałe operatory porównywania. Operatory logiczne wymagają operandów o typie bool lub int i zwracają typ bool, operatory arytmetyczne typów int. Operatory porównywania sprawdzają równość fizyczną zmiennych o tym samym typie. Poniższy listing prezentuje gramatykę wyrażeń arytmetycznych i logicznych.

```
<bool-exp> ::=
    <simple-exp> <bool-op> <simple-exp>

<arith-exp> ::=
    <simple-exp> <arith-op> <arith-exp>

<arith-op> ::=
    + | - | * | /

<bool-op> ::=
    && | || | = | < | > | <= | >=
```

Na kolejnym listingu znajdują się proste funkcje zawierające wyrażenia arytmetyczne i logiczne.

```
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.

Semantyka wyrażeń let

Typ wynikowy całego wyrażenia jest taki jak typ ostatniego wyrażenie w ciele let. Wyrażenie let wprowadza nowy symbol do środowiska, dostępny od momentu definicji. Typ wprowadzonego symbolu to $a_1 \to ... \to a_n \to r$, gdzie $a_1, ..., a_n$ to typy kolejnych argumentów, a r to typ ostatniego wyrażenia w ciele. Jeśli nie zostały zdefiniowane żadne argumenty to typ symbolu to r. W przypadku braku argumentów, ciało wyrażenia let jest ewaluowane gorliwie, przy uruchomieniu programu.

Poniższy listing zawiera przykłady wyrażeń let bez argumentów, z wielo- i jedno wierszowym ciałem.

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

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

let adder a b =
  printInt a
  printInt b

  a + b

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*. Wzajemnie rekurencyjne funkcje mają dostęp do zdefiniowanych symboli ze swojej grupy, niezależnie ich kolejności. Nie można zdefiniować wzajemnie rekurencyjnych wartości, każde wyrażenie musi definiować niezerową liczbę argumentów.

Poniższy fragment kodu zawiera dwie 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

Składnia rekordów jest podobna do tej z pozostałych języków z rodziny ML. Rekord zawiera wiele pól o różnych nazwach. Każde pole musi mieć zdefiniowany typ. Typem pola może być inny, wcześniej zadeklarowany, rekord. Gramatyka deklaracji rekordów:

```
  <record-decl> ::=
      type <identifier> =
        { <field-decl}+
        }

  <field-decl> ::=
      <identifier> : <identifier>+ ( <newline> | ; )
```

Przykładowe deklaracje rekordów:

```
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. Typ wyrażeń podstawianych pod każde pole musi zgadzać się ze zdefiniowanym typem dla danego pola. Gramatyka literału rekordu:

```
<record-lit> ::=
     { <field-lit}+
     }

<field-lit> ::=
     <identifier} = <simple-exp> (<newline> | ; )
```

Przykładowe literały rekordu:

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

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

2.4.3. Uaktualnianie rekordu

Rekordy w MonoMLu, podobnie jak rekordy w F# i OCamlu, są trwałe. Uaktualnienie jednego z pól skutkuje stworzeniem nowego rekordu i nie zmienia wartości któregokolwiek z pól z pierwotnej instancji. Typ wyrażenia, które uaktualniana dane pole musi być taki sam jak typ pola w deklaracji rekordu. Gramatyka uaktualniana rekordu:

```
<record-update> ::=
      { <simple-exp> with
        <field-update}+
      }

<field-update> ::=
      <identifier} = <simple-exp> (<newline> | ; )
```

Poniższy fragment kodu zawiera deklarację, literał i uaktualnienie rekordu.

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

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

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

Uwaga. Przy przekazywaniu rekordu jako argument, należy dodać adnotację typu do argumentu. W obecnej implementacji typy rekordów nie są inferowane.

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. Deklaracja definiuje typ ogólny, który może być użyty w definicji metod klasy. Metody klasy muszą być funkcjami, nie mogą być wartościami. Obecna implementacja nie wspiera metod klasy, które muszą być ewaluowane przy uruchomieniu programu, ale nie ma przeszkód utrudniających ich dodanie. Poniższy fragment kodu prezentuje przykładowe deklaracje klas `Print` i `Pow`.

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

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

2.5.2. Deklaracja instancji

Instancja klasy musi zawierać nazwę klasy, którą implementuje oraz typ konkretny, dla którego tworzona jest instancja. Typ zadeklarowanych metod w instancji klasy musi być zgodny z typami w definicji danej metody z klasy, przy założeniu, że typ generyczny został podmieniony z typem konkretnym.

Instancje klas dla różnych typów konkretnych mogą być zaimplementowane w sposób następujący:

```
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 MonoMLu 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.

Otwarcie modułu powoduje wprowadzenie zawartych w nim symboli do lokalnej przestrzeni nazw. Przykład modułów z definicjami i otwieranie 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 MonoMLu są reprezentowane tak samo jak języku w C — jako spójny ciąg w pamięci.

Elementami literału tablicy mogą być literały liczbowe 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.

Przykład jedno i wielo wierszowego literału:

```
let arr : int array = [| 1; 1; 2; 3; 5 |]

let multiLineArr : int array =
  [|1; 2;
    3
    4
  |]
```

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 MonoMLa są dosłownie tłumaczone na typy z C z wyjątkiem typu `unit`, który jest zamieniany na `bool` (o rozmiarze jednego bajtu).

Przykładowo definicja zewnętrznego symbolu, o sygnaturze w C `void set_array_elem(int* arr, int i, int val)`, wygląda następująco:

```
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: `monoml-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 języka, lecz nie posiadają informacji o jego typie. Implementacja w pliku: `monoml-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 *MonoML* (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: `monoml-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: `monoml-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 [3].

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`, `;`, `in`). W F# istnieje także możliwość mieszania tych słów kluczowych z wcięciami.

Analiza wcięć w MonoMLu jest zbliżona do tej w Pythonie [5]. 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 wcięcia są zdejmowane ze stosu i dla każdego zdjętego wcięcia generowany jest token *DEDENT*. Oznacza on koniec wciętego bloku. Po zdjęciu wszystkich większych wcięć, ostatnie wcięcie, które zostanie na stosie musi być równe wcięciu z obecnie przetwarzanego wiersza, a 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* [7]. 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* [7]. Całość gramatyki znajduje się w pliku `monoml-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 rozmiaru 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 Caml. W kompilatorze MonoMLa pracowałem nad rozwiązaniem, które w pozostałych przypadkach korzystałoby z przekazywania argumentów funkcji przez rejestry i pozwalałoby, na przekazywanie typów danych o różnych rozmiarach przez ich wartość (bez opakowywania we wskaźnik).

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.

W następującym fragmencie kodu wywołana funkcja `double` jest statycznie znana.

```
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.

Na poniższym fragmencie wywołane funkcje `a`, `b` i `c` są nieznane.

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

let apply f x = f x

let main (b : 'a -> 'a) =
  let c = apply b
  let a = getIdentity ()
  (a 1 = b 1) && (b 1 = c 1)
```

Wywołanie funkcji znanej

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

1. Liczba argumentów z aplikacji jest mniejsza od liczby zdefiniowanych argumentów. W tym przypadku utworzony zostaje obiekt reprezentujący częściowo zaaplikowaną funkcję. Zostaną w nim zapisane argumenty z aplikacji oraz wskaźnik na odpowiednią funkcję. Skopiowane argumenty i sam obiekt zostaną utworzone na stercie.

2. Argumentów z aplikacji jest tyle samo co zdefiniowanych. Funkcja zostanie wywołana w stylu z C. Jest to 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. Argumentów z aplikacji może być więcej niż zdefiniowanych, jeśli wynikiem wywoływanej funkcji jest funkcja. Niech k będzie liczbą argumentów w definicji funkcji, a n liczbą argumentów z aplikacji, gdzie $n > k$. Najpierw nastąpi wywołanie znanej funkcji z pierwszymi k argumentami. Wynik pierwszego wywołania, który teraz jest nieznaną funkcją, zostanie zaaplikowany do pozostałych $n-k$ argumentów. 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 funkcję,
- wskaźnik na środowisko — zapamiętane argumenty, które są pamiętane jako spójny ciąg bajów (dynamicznie zaalokowana tablica z C),
- pozostała liczba argumentów do wywołania wskazywanej funkcji,
- arność funkcji,
- liczba bajtów w środowisku.

Definicja takiej struktury w C wyglądałaby następująco.

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

Wskaźnik na funkcję `fn` przed wywołaniem musi zostać zrzutowany na prawidłowy typ. Dla aplikacji funkcji, których wynikiem jest funkcja, typ wynikowy wygenerowanej funkcji `fn` to struktura `function`. Jako argumenty funkcji `fn`, poza argumentami podanymi w aplikacji funkcji, przekazane zostaną dodatkowo: wskaźnik na środowisko i liczba argumentów w aplikacji. 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 funkcji o tym samym typie do tych samych argumentów, nic nie zostanie wywołane.

```
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 argumentów, do których funkcja jest aplikowana. 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 MonoMLa, była możliwość przekazywania typów o różnym rozmiarze przez ich wartość, a nie przez wskaźnik. W obecnej implementacji istnieje tylko kilka typów o różnym rozmiarze: 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 wielkości.

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 do dwóch argumentów o typach int i bool, a druga do jednego argumentu o typie int. W kolejnym kroku, chcąc wywołać obie funkcje, pierwszą strukturę aplikujemy do pozostałego argumentu o typie int, a drugą do pozostałych dwóch 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 się w środowisku. Wtedy typ funkcji wskazywanych przez `fn` po zrzutowaniu byłby taki sam, niezależnie od tego do ilu dotychczas argumentów zostały zaaplikowane. 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 reprezentację wszystkich argumentów można uzyskać poprzez reprezentowanie ich przez wskaźnik, ale takiego rozwiązania chciałem uniknąć. Innym sposobem byłoby zapamiętanie wszystkich argumentów, także tych, do których funkcja jest aplikowana na końcu, 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 przy ostatniej aplikacji. 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: $a_1 \rightarrow a_2 \rightarrow ... \rightarrow a_n \rightarrow t$. Wtedy, dla takiej funkcji zostanie wygenerowanych n funkcji wejściowych, gdzie $i - ta$ funkcja będzie

przyjmowała sufiks ciągu oryginalnych argumentów, od $i - tego$ argumentu. Jeśli wynikiem oryginalnej funkcji jest funkcja, to argumenty funkcji wynikowej 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 funkcja jest aplikowana do kolejnych argumentów, 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 aplikacji — funkcja wejściowa 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[21]. 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 za rozważenie wszystkich przypadków związanych z liczbą argumentów w aplikacji. Podejmuje decyzje o wywołaniu właściwiej funkcji i ewentualnym przekazaniu pozostałych argumentów dalej bądź zwróceniu obiektu reprezentującego częściowo zaaplikowaną funkcję. W modelu eval/apply, to po stronie funkcji wołającej leży zadanie rozważania tych przypadków. Strona wołająca (ang. call site) zna liczbę argumentów z aplikacji, a w czasie działania sprawdza właściwą 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 argumentów z aplikacji, 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 generowaniu 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 [21] 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 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 generowaniu 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 zwiększyć 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 funkcję `make_adder` w OCamlu, która będzie zwracać zagnieżdżoną funkcję. Ciało zagnieżdżonej funkcji `add` odwołuje się do zmiennej z zewnętrznego zakresu.

```
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 transformację na funkcji `add`, bez dodatkowych zmian, to zmienna x przestanie być dostępna z ciała funkcji.

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

Powyższy kody 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`, tak jak na poniższym fragmencie kodu.

```
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 MonoMLu, 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 oryginalnej funkcji (rozszerzonej o dodatkowe argumenty — zmienne wolne) do zmiennych wolnych. Na funkcji `add` została wykonana ta transformacja:

```
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*). Funkcja `add` jest teraz zwykłym przypisaniem wyrażenia do symbolu, 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 MonoMLa 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 zawartości całej struktury do nowej instancji.

3.7. Let polimorfizm

Bez let polimorfizmu (polimorfizm parametryczny) ciężko wyobrazić sobie nowoczesny język. Mimo wygody, jaką dostarcza programiście, często wiąże się 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 [17] i Haskell

[18, 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 MonoMLa, to rozwiązanie nie jest satysfakcjonujące.

OCaml reprezentuje inty i wskaźniki na jednym słowie maszynowym [20]. To czy do funkcji został przekazany wskaźnik, czy int przez wartość jest rozpoznawane na podstawie najniższego bitu, który jest traktowany jak znacznik [23, Chapter 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 podejściem do arytmetyki liczb i wciąż nie pozwala na przekazywanie przez wartość typów danych większych niż słowo maszynowe. W związku ze wspominanymi wadami tego rozwiązania nie zdecydowałem się go zaimplementować.

Języki takie jak C++ i Rust [20] oraz kompilator SMLa MLton [8], wyspecjalizowują każdą polimorficzną funkcję przed fazą generowania kodu. Ten proces nazywany jest *monomorfizacją*. Dokładniej, polega on na utworzeniu 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 funkcję, bądź może być to wykryte przez kompilator. Przykład polimorficznej funkcji 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 zmiany jego reprezentacji. Jednak nie jest ono pozbawione wad. Zwiększa rozmiar wygenerowanego kodu, tym samym wydłużając czas kompilacji i powiększając rozmiar wynikowego programu. Kolejną wadą w praktycznych zastosowaniach, jest fakt, że wysokopoziomowy kod implementacji funkcji polimorficznej musi być opublikowany wraz z wy-

generowaną bibliotekę, aby istniała możliwość wywołania tej funkcji dla typów, dla których nie była wcześniej ukonkretniona. Mimo to, język C++ cieszy się ogromną popularnością, a system szablonów jest szeroko używany, także w zastosowaniach produkcyjnych. Dodatkowo, statyczna monomorfizacja funkcji nie pozwala na zastosowanie polimorficznej rekursji, chociaż nie jest ona popularną cechą języków programowania.

W MonoMLu 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 MonoMLa 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 w każdej deklaracji, pozostawiając wszystkie zalety statycznego systemu typów. W MonoMLu działa ona następująco. Na początku każdemu argumentowi, który nie został adnotowany przez użytkownika typem konkretnym, zostaje przypisany inny 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).

Uwaga. Przy wywoływaniu polimorficznej funkcji, kompilator musi być w stanie ukonkretnić wszystkie typy ogólne. Typ argumentu może pozostać nieukonkretniony, jeśli argument nie będzie używany w ciele funkcji. W takim przypadku kompilator zwróci błąd, bo nie będzie w stanie wygenerować kodu dla funkcji z polimorficznym argumentem (już po przeprowadzeniu monomorfizacji).

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) [24, 25]. 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).

Deklaracja klasy typów w Haskellu:

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

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. Następnie definiujemy instancje klasy w Haskellu:

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

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

Instancje klasy `Show` w OCamlu zwracają rekord z odpowiednimi implementacjami zapisanymi w polach rekordu.

```
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.

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

let main1 =
```

```
    printArg show_bool true

let main2 =
    printArg show_string "Hello"
```

Korzystanie z instancji klasy w Haskellu jest dużo bardziej przejrzyste.

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

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

Zasadnicza różnica między symulowaniem klas typów w języku ich niewspierającym 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 MonoMLu, 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 z tym rozwiązaniem związane są wszystkie wady monomorfizacji: konieczność kompilacji całego kodu programu na raz, brak rekursji polimorficznej oraz brak polimorfizmu wyższych rzędów (ang. higher-rank polymorphism). Z podobnego rozwiązania przy implementacji klas typów korzysta MLton [4].

Rozdział 4.

Podsumowanie

4.1. Wnioski

Najtrudniejszym elementem 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 *MonoML* i kompilatorem, naturalnym rozwinięciem byłby następujące elementy:

- Polimorficzne struktury danych, w tym polimorficzna tablica. Obecnie wspierane są jedynie rekordy z polami, które muszą mieć konkretny typ. Przy zastosowaniu monomorfizacji, ich implementacja jest łatwa. Komplikuje się jedynie inferencja typów. Może się okazać koniecznym obsłużenie słabych polimorficznych typów [11].
- Wyrażenia lambda. Mogłyby być zrealizowane poprzez ich zamianę na globalne

wyrażenia let o unikalnych nazwach.

- Warianty [23, Chapter 6]. Są bardzo popularną i użyteczną cechą języków z rodziny ML. Dzięki nim można łatwo zaimplementować typy danych takie jak *Optional* lub trwała lista.

- 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 niejednoznacznościami [9] przy wyborze odpowiedniej instancji klasy. Koniecznym mogłoby by się okazać dodanie zależności funkcyjnych (ang. functional dependencies) [1].

- 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 generowałyby różne tokeny (nie tylko *INDENT* i *DEDENT*), w zależności od kontekstu, w którym się znajdują. Dokładny opis tego rozwiązania znajduje się w specyfikacji języka F# 4.0[14].

- 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 zastosowanie tej optymalizacji nie stanowi problemu.

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 menadżera pakietów *OPAM*. Instrukcja dostępna na stronie: `https://opam.ocaml.org/doc/Install.html#Binary-distribution`
3. Instalacja LLVMa. Instrukcja dostępna na stronie: `https://llvm.org/`.
4. Wymagany jest kompilator LLVM w wersji 6. Zainstalowaną wersję można sprawdzić poleceniem:

```
$ llc --version
```

5. Ustawienie wersji kompilatora:

```
$ opam switch 4.05.0
```

6. Konfiguracja menadżera pakietów:

```
$ eval ‘opam config env‘
```

7. Instalacja bibliotek. Należy uruchomić skrypt `install_deps.sh` instalujący odpowiednie pakiety z użyciem OPAMa. Plik znajduje się w głównym folderze `monoml-compiler`.

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

8. Kompilacja projektu:

```
$ make
```

5.2. Sposób użycia

Skompilowany plik wykonywalny kompilatora znajduje się w folderze `monoml-compiler/_build/default/langc/langc.exe`. Pierwszym argumentem musi być ścieżka do kodu źródłowego w Langu. Program posiada także dodatkowe argumenty.

- `--help` — Opis użycia oraz dostępnych argumentów.
- `-o` — Opcjonalny argument dzięki któremu można określić ścieżkę i nazwę skompilowanego programu. Domyślnie jest to `a.out`.
- `--ll-only` — Dodanie tego argumentu powoduje wygenerowanie kodu w języku pośrednim LLVM IR do pliku `out.ll`, bez tworzenia pliku wykonywalnego.

Przykładowy program z folderu `test/input` można skompilować następującym poleceniem:

```
$ _build/default/langc/langc.exe test/input/hello_world.la # kompilacja
$ ./a.out # uruchomienie wygenerowanego programu
```

Uwaga. Podczas kompilacji program szuka pliku `external.c` w obecnym folderze (tym, w którym znajduje się użytkownik). Zawiera on implementacje kilku funkcji bibliotecznych w C. Znajduje on się w głównym folderze projektu (`monoml-compiler`), dlatego zalecanym jest kompilowanie z tego folderu.

5.2.1. Przykładowe programy i testowanie

W folderze `test/input` znajduje się dwadzieścia przykładowych programów napisanych w MonoMLu (pliki z rozszerzeniem `.la`). Testują różne funkcjonalności języka m. in częściową aplikację, polimorfizm, klasy typów oraz inferencję. Wszystkie testy można uruchomić następującym poleceniem:

```
$ make test
```

Testy należy uruchomić z głównego folderu projektu. Testowanie polega na skompilowaniu każdego programu z rozszerzeniem `.la` z folderu `test/input`, uruchomieniu go i porównaniu standardowego wyjścia z odpowiadającym plikiem o sufiksie `-expected-out.txt`.

5.3. Użyte narzędzia i biblioteki

- `OCaml: https://ocaml.org/`
- `Menhir: http://gallium.inria.fr/f̃pottier/menhir/`
- `Sedlex: https://github.com/alainfrisch/sedlex`
- `High OLLVM`: Biblioteka opakowująca oficjalną bibliotekę do generowania kodu LLVM IR w funkcyjny interfejs. Bazująca w przeważającej większości na bibliotece `ollvm`: `https://github.com/OCamlPro/ollvm` Ta zależność została dołączona wraz z kodem źródłowym kompilatora.
- `OCaml-parsing: https://github.com/smolkaj/ocaml-parsing`
- `Core: https://github.com/janestreet/core`
- `Dune (JBuilder): https://github.com/ocaml/dune`
- `OUnit: http://ounit.forge.ocamlcore.org/`

5.4. Struktura projektu

Projekt znajduje się w folderze `monoml-compiler`.

- `compiler/` — Biblioteka kompilatora. Zawiera wszystkie najważniejsze elementy procesu kompilacji. Głównym jej zadaniem jest wygenerowanie kodu LLVM IR dla podanego kodu MonoMLa.
- `compiler/parsing/grammar.mly` — Gramatyka w składni Menhira.
- `compiler/parsing/lexer.cppo.sedlex.ml` — Analiza leksykalna.
- `compiler/parsing/ast.ml` — Typy reprezentujące drzewo składni po parsowaniu.
- `compiler/codegen.ml` — Zbiór funkcji odpowiedzialnych za generowanie kodu.
- `compiler/lang_types_def` — Definicja typów reprezentujących typy z MonoMLa.
- `compiler/lang_types` — Funkcje służące znalezieniu typu wyrażeń. Inferencja typów.
- `compiler/letexp.ml` — Generowaniu kodu funkcji (wyrażeń *let*). Główna część implementacji częściowej aplikacji.
- `compiler/envn.ml` — Środowisko używane przy generacji kodu.

- `compiler/typed_ast_def` — Typy reprezentujące drzewo z otypowanymi wyrażeniami.
- `compiler/typed_ast` — Zamiana drzewa otrzymanego po sparsowaniu na drzewo z otypowanymi wyrażeniami. Wykonywane są także transformacje: modułów, funkcji zagnieżdżonych, closure conversion oraz klasy typów.
- `langc/` — Projekt odpowiedzialny za plik wykonywalny kompilatora. Kompilacja kodu LLVM IR, obsługa parametrów wywołania programu oraz odczytywanie kodu z pliku wejściowego.
- `langc/compilation/compilation.ml` — Kompilacja wygenerowanego kodu LLVM z użyciem programu `llc` (kompilator LLVM IR) i `gcc`.
- `test/` — Projekt zawierający testy kompilatora.
- `test/input/` — Kody źródłowe w MonoMLu do przetestowania. W czasie testowania każdy z nich zostanie skompilowany, uruchomiony, a standardowe wyjście porównane z plikami `-expected-out.txt` z tego samego folderu.
- `test/compiler_tests.ml` — Moduł generujący testy na podstawie plików znajdujących się w folderze `test/input/`.
- `high-ollvm/` — Biblioteka udostępniająca funkcyjne api do generowaniu kodu LLVM IR, bazowana na bibliotece ollvm[10].

Bibliografia

[1] Functional dependencies. https://wiki.haskell.org/Functional_dependencies. Dostęp: 2018-08-01.

[2] Higher kinded polymorphism in Rust (Github issues). https://github.com/rust-lang/rfcs/issues/324. Dostęp: 2018-08-01.

[3] Implementing a language with LLVM tutorial. https://llvm.org/docs/tutorial/OCamlLangImpl1.html. Dostęp: 2018-08-01.

[4] Implementing, and Understanding Type Classes. http://okmij.org/ftp/Computation/typeclass.html. Dostęp: 2018-08-01.

[5] Indentation. Python Reference Manual. https://docs.python.org/2.5/ref/indentation.html. Dostęp: 2018-08-01.

[6] LLVM. https://llvm.org/. Dostęp: 2018-08-01.

[7] Menhir. http://gallium.inria.fr/~fpottier/menhir/. Dostęp: 2018-08-01.

[8] Monomorphise. MLton. http://mlton.org/Monomorphise. Dostęp: 2018-08-01.

[9] Multi-parameter type class. https://wiki.haskell.org/Multi-parameter_type_class. Dostęp: 2018-08-01.

[10] Ollvm. https://github.com/OCamlPro/ollvm. Dostęp: 2018-08-01.

[11] Polymorphism and its limitations. OCaml Manual. https://caml.inria.fr/pub/docs/manual-ocaml/polymorphism.html. Dostęp: 2018-08-01.

[12] Rust by example. Traits. https://doc.rust-lang.org/rust-by-example/trait.html. Dostęp: 2018-08-01.

[13] Scala docs: Traits. https://docs.scala-lang.org/tour/traits.html. Dostęp: 2018-08-01.

[14] Specyfikacja języka F# 4.0. https://fsharp.org/specs/language-spec/4.0/FSharpSpec-4.0-final.pdf. Dostęp: 2018-08-01.

[15] std::bind C++ Reference. https://en.cppreference.com/w/cpp/utility/functional/bind. Dostęp: 2018-08-01.

[16] Type class. Wikipedia. https://en.wikipedia.org/wiki/Type_class?oldformat=true. Dostęp: 2018-08-01.

[17] Type Erasure. The Java Tutorials. https://docs.oracle.com/javase/tutorial/java/generics/erasure.html. Dostęp: 2018-08-01.

[18] Type Systems, Type Inference, and Polymorphism. http://www.cs.tau.ac.il/~msagiv/courses/pl14/chap6-1.pdf. Dostęp: 2018-08-01.

[19] Using trait objects that allow for values of different types. Rust. https://doc.rust-lang.org/book/second-edition/ch17-02-trait-objects.html. Dostęp: 2018-08-01.

[20] Richard A. Eisenberg and Simon Peyton Jones. Levity polymorphism. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 525–539. ACM, 2017.

[21] Simon Marlow and Simon L. Peyton Jones. Making a fast curry: push/enter vs. eval/apply for higher-order languages. *J. Funct. Program.*, 16(4-5):415–449, 2006.

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

[23] Yaron Minsky, Anil Madhavapeddy, and Jason Hickey. *Real World OCaml - Functional Programming for the Masses*. O'Reilly, 2013.

[24] John Peterson and Mark P. Jones. Implementing type classes. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 227–236. ACM, 1993.

[25] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989.

[26] Stefan Wehr and Manuel M. T. Chakravarty. ML modules and haskell type classes: A constructive comparison. In G. Ramalingam, editor, *Programming Languages and Systems, 6th Asian Symposium, APLAS 2008, Bangalore, India, December 9-11, 2008. Proceedings*, volume 5356 of *Lecture Notes in Computer Science*, pages 188–204. Springer, 2008.