ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ
ВЫСШЕГО ПРОФЕССИОНАЛЬНОГО ОБРАЗОВАНИЯ
«НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ УНИВЕРСИТЕТ
«ВЫСШАЯ ШКОЛА ЭКОНОМИКИ»

**Факультет Санкт-Петербургская школа**
**физико-математических и компьютерных наук**

Трилис Алексей Андреевич

# ПЕРЕОСНАЩЕНИЕ НЕЯВНЫХ МОДУЛЕЙ ДЛЯ ЯЗЫКА 1ML

Выпускная квалификационная работа - БАКАЛАВРСКАЯ РАБОТА
по направлению подготовки *01.03.02 Прикладная математика и информатика*
образовательная программа «Прикладная математика и информатика»

Рецензент
В.Н. Брагилевский

Руководитель
д-р физ.-мат. наук, проф.
Б.А. Новиков

Консультант
канд.физ.-мат.наук
Д.А. Березун

Санкт-Петербург 2021

# Оглавление

# Аннотация

*АБСТРАКТ*

Ключевые слова: неявные параметры, неявные модули, имплициты, классы типов, язык модулей ML, OCaml, 1ML.

*ABSTRACT*

Keywords: implicit parameters, implicit modules, type classes, ML modules, OCaml, 1ML.

# Введение

*Ad hoc полиморфизм* — это свойство языка программирования, позволяющее создать функцию, семантика которой будет зависеть от типов входных параметров. В качестве примеров таких функций можно назвать функцию `print`, позволяющую выводить значения различных типов, и оператор `+`, позволяющий складывать как целые числа, так и числа с плавающей точкой, а в некоторых языках также и строки. Ad hoc полиморфизм позволяет программисту писать более лаконичный код, улучшает читаемость кода, а также способствует парадигме обобщённого программирования.

В процедурных и объектно-ориентированных языках классическим методом реализации ad hoc полиморфизма является *перегрузка* функций и операторов. К сожалению, этот простой метод не подходит для языков с мощным выводом типов и строгой системой типов, так как в них не всегда можно однозначно выбрать перегрузку в месте вызова функции. Для таких языков были разработаны более сложные методы, а некоторые языки до сих пор не поддерживают ad hoc полиморфизм.

Эта работа посвящена семейству функциональных языков ML. Эти языки относительно часто используются в некоторых узких областях, таких как создание компиляторов, автоматическое доказательство теорем, анализ кода, финансовая сфера и веб-разработка. ML-подобные языки ведут свою историю с 1973 года и оказали влияние на развитие таких популярных языков, как C++, Scala, Rust, Haskell и другие.

Одной из существенных проблем языков из семейства ML является отсутствие ad hoc полиморфизма. Так, в стандартную библиотеку языка OCaml, одного из самых популярных диалектов ML, включено множество функций для вывода значений различных типов: `print_int`, `print_float`, `print_string` и так далее, а также отдельные операторы `+` и `+.`, для сложения целых чисел и чисел с плавающей точкой, соответственно. Так как ML-подобные языки знамениты мощным выводом типов, который позволяет ставить меньше явных типовых аннотаций и писать обобщённый код, такая избыточность идёт вразрез с философией языка.

Исследователями было предложено несколько работ, вносящих ad hoc полиморфизм в языки семейства ML. Работа Вайта, вводящая *неявные модули* для языка OCaml, предлагает наиболее мощную функциональность по сравнению с аналогами, и именно этот метод послужит основанием для данной работы. Тем не менее, полноценное введение этого метода в OCaml требует значительной практической и теоретической работы. Так, тайп-чекер OCaml в данный момент не поддерживает в общем случае унификацию на уровне модулей, которая необходима для работы этого метода.

Помимо мощного вывода типов, ML-подобные языки также отличаются продвинутой *модульной системой*, основанной на теории зависимых типов [1]. Тогда как обе эти технологии успешно используются программистами, они существуют независимо

друг от друга. Так, язык ML по историческим причинам фактически поделён на два слоя. Во внутреннем, основном слое действует вывод типов, тогда как во внешнем, модульном слое поддерживаются более выразительные типы, в обмен на их явное объявление и общую избыточность. Интеграция между этими слоями затруднена, так как модули не являются объектами первого класса. Было предложено несколько решений этой проблемы, но они порождают избыточный код и не поддерживают некоторые случаи.

Экспериментальный язык 1ML [2] был создан для решения проблем модульной системы. Он предлагает минималистичный и единообразный подход к данной проблеме, стирая синтаксическое разделение между основным и модульным слоями, взамен предлагая более тонкое семантическое разделение. Предположительно, такой подход позволит добавить в язык многие важные модели, реализация которых при традиционном подходе к модулям считались затруднительными, включая, например, поддержку ad hoc полиморфизма.

## Цель и задачи

Целью данной работы является поддержка ad hoc полиморфизма в языке 1ML, подобная неявным модулям из [3], но при этом предлагающая более полное решение в аспектах, касающихся порядка разрешения неявных модулей и поддержки неявных функторов. Для этого ставятся следующие задачи:

- Реализация неявных модулей, повторяющих функциональность решения [3] для OCaml.
- Разработка алгоритма, позволяющего полно и эффективно осуществлять вставку неявных модулей.
- Дополнение поддержки неявных функторов.
- Сравнение с существующим решением [3] для OCaml.

## Обзор последующих глав

Глава 1 посвящена обзору существующих статей и результатов, относящихся к области исследования.

В главе 2 описываются предложенный синтаксис, семантика и общая схема алгоритма.

Глава 3 содержит описание важных деталей решения и даёт обоснование выборов при дизайне алгоритма.

В главе 4 обсуждаются проблемы, возникающие при предопределённом порядке разрешения неявных аргументов, и предлагается алгоритм, позволяющий эффективно решить эти проблемы.

# 1. Обзор литературы

## 1.1. Модули первого класса и 1ML

Отсутствие первоклассных модулей порождает практические проблемы при попытке интеграции между основным и модульными слоями ML. Например, долгое время был невозможен динамический выбор модуля:

```
module Map = if maxElems < 100 then BinTreeMap else HashTableMap
```

С целью сделать модули первоклассными Руссо предложил дополнить основной язык явной упаковкой и распаковкой модулей (packaged modules) [4]. Это предложение было расширено последующими работами и было реализовано во многих диалектах ML, включая OCaml [5]. Упаковка и распаковка модулей решает самые остро стоящие практические проблемы, например, позволяя во многих случаях динамически выбирать модуль. Вместе с тем, решение Руссо предполагает наличие множества избыточных типовых аннотаций и недостаточно выразительные типы для некоторых случаев. Рассмотрим, например, динамический выбор модуля в OCaml:

```
module Map = (val (if maxElems < 100
                   then (module BinTreeMap : MAP)
                   else (module HashTableMap : MAP))) : MAP)
```

Несмотря на то, что задача решена, этот код выглядит намного менее лаконично и естественно, чем код выше. Что же касается выразительности, приведём пример, ограничивающий использование полиформизма. Типизация полиморфной функции, использующая модули, вынуждена использовать полиморфизм основного языка, например, так:

```
(module S with type t = 'a) -> (module S with type t = 'a) -> 'a
```

Тогда как с действительно первоклассными модулями можно было бы выразить этот тип следующим образом:

```
(X : S) -> (module S with type t = X.t) -> X.t
```

Используя полиморфизм основного языка, попробуем типизировать стандартную функцию `return`, принимающую монаду как модуль:

```
return : (module MONAD with type 'a t = ?) -> 'a -> ?
```

Так как основной язык не поддерживает полиморфизм высших кайндов, на место вопросов в сигнатуре выше ничего нельзя подставить.

Таким образом, упаковка и распаковка модулей частично решает проблемы, стоящие перед языком модулей ML, но имеет существенные недостатки. С другой стороны, сложно предложить решение, полностью решающее эти проблемы без существенной переработки основ языка.

Ряд более поздних работ показал, что историческое разделение языка на основной и модульный слои не является обязательным. Так, Россберг и другие показали, что все конструкции ML, включая модули, выражаются в рамках Системы $F_\omega$ [6]. Эти исследования привели к созданию Россбергом экспериментального диалекта 1ML [2], в котором стёрты различия между слоями и поддерживаются модули первого класса в истинном смысле этого выражения. Так, практические проблемы, описанные выше, в 1ML решены. Для обеспечения разрешимости проверки и вывода типов, автор использует идею Харпера — Митчела [7] разделения типов на *малые типы* и *большие типы*, накладывая на последние ряд ограничений, например, вывод типов распространяется только на малые типы.

## 1.2. Ad hoc полиморфизм

Термин "ad hoc полиморфизм" был предложен Стрэчи в 60-х годах XX века [8]. Стрэчи разделяет два вида полиморфизма: *параметрический полиморфизм*, то есть определение функции для разных типов аргументов, на каждых из которых функция работает одинаково, и *ad hoc полиморфизм*, в котором функция работает разным образом в зависимости от типов аргументов. Тогда как общепринятым методом для параметрического полиморфизма в функциональных языках стала система типов Хиндли — Милнера, долгое время не существовало подходящего метода для ad hoc полиморфизма.

Одной из классических работ, решающих эту проблему, стало добавление в язык Haskell *классов типов* [9], вдохновлённое идеями из объектно-ориентированной парадигмы. Авторы предложили расширение системы типов Хиндли — Милнера ad hoc полиморфизмом и алгоритм, преобразующий код с классами типов в код без них, в стандартную систему типов Хиндли — Милнера. Таким образом, два вида полиморфизма были объединены в единую систему. Рассмотрим классы типов на простом примере.

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

instance Show Int where
  show = showSignedInt

show_twice x = show x ++ show x
```

В примере выше определён класс типов `Show`. Описание этого класса типов состоит из набора функций, в данном случае — единственной функции `show`. Чтобы тип входил в класс типов, для него должны быть определены все эти функции. Эти функции для конкретного типа определяются с помощью создания *экземпляра* (instance) и в дальнейшем могут использоваться как обычные функции. Рассмотрим тип функции `show_twice`:

```
show_twice :: Show a => a -> string
```

Эту типизацию следует трактовать следующим образом: функция `show_twice` имеет тип `a -> string` для всех `a`, которые входят в класс типов `Show`. Важно отметить, что в Haskell используемые классы типов не нуждаются в явном указании, а выводятся из тела функции. Так, в приведённом примере компилятор самостоятельно вывел ограничение `Show a`, исходя из того, что в теле функции использована функция `show`.

Похожие на классы типов решения были предложены в ряде других языков, например, в Rust (типажи, traits) [10]. Решения из этой группы объединяет необходимость *каноничности* — для каждого типа и класса типов может быть определено не более одного экземпляра. (TODO: orphan instances)

Представителем другой группы является Scala и реализованные в ней *неявные аргументы*, также называемые *имплициты* [11]. В Scala не требуется каноничность и присутствует возможность явно передать экземпляр, если система типов не способна вывести его самостоятельно. С другой стороны, для каждой функции требуется явно указывать, какие имплициты в ней используются, тогда как в решениях, основанных на классах типов, эта информация была бы выведена автоматически.

```
trait Showable [T] {
  def show (x: T): String
}

implicit object IntShowable extends Showable [Int] {
  def show (x: Int) = x.toString
}

def show[T](x : T)(implicit s: Showable [T]): String = {
  s.show(x)
}

show(7)(IntShowable)
show(7)
```

Неявные параметры в разных формах часто встречаются в языках с зависимыми типами. Рассмотрим подробнее Coq, язык, по историческим причинам имеющий много общего с ML. (TODO)

Вайтом и другими было предложено ввести в OCaml *неявные модули* [3], решение, вдохновлённое имплицитами из Scala. Именно на этом решении основывается данная работа. Так же как и в Scala и Coq, предложение Вайта требует явного указания используемых неявных параметров и не требует каноничности, позволяя в случае неоднозначности явно указывать необходимый параметр.

```
module type Show = sig
  type t
  val show : t -> string
end

implicit module Show_int = struct
  type t = int
  let show x = string_of_int x
end

implicit module Show_list {S : Show} = struct
  type t = S.t list
  let show x = string_of_list S.show x
end

let show {S : Show} x = S.show x

show 5 (* show {Show_int} 5 *)
show [1;2;3] (* show {Show_list (Show_int)} [1;2;3] *)
```

Как уже было упомянуто, проверка каноничности возможна не во всех языках. Модульная система ML позволяет скрывать экземпляры за абстракцией. Рассмотрим, например, следующий код на OCaml:

```
module type Show = sig
  type t
  val show : t -> string
end

module F (X : Show) = struct
  implicit module S = X
```

```
end

implicit module Show_int = struct
  type t = int
  let show = string_of_int
end

module M = struct
  type t = int
  let show _ = "An int"
end

module N = F(M)
```

(TODO: пояснить пример)

Дрейером и другими были предложены модулярные классы типов [12], попытка применить классы типов из Haskell в ML. Поскольку в классическом подходе к классам типов требуется каноничность, а в ML невозможно проверить, соблюдается ли она, авторы вводят ряд нежелательных ограничений:

- неявные модули могут быть объявлены только на верхнем уровне;
- все модули на верхнем уровне должны быть явно типизированы;
- на верхнем уровне могут находиться только модули.

Эти ограничения только усиливают синтаксическое разделение между модульным и основным слоем, о котором шла речь выше, что особенно нежелательно в 1ML, языке, построенного с целью ослабить это разделение. Помимо этого, модулярные классы типов, в отличие от решения Вайта и других, не поддерживают неявные параметры высшего ранга и кайнда, а также классы типов с несколькими параметрами.

Также в нескольких работах были предложены неявные аргументы [13] и классы типов [14] для ML как независимая от модулей функциональность. Такие решения привносят значительную дубликацию конструкций (так как модульные типы естественно подходят под роль описания классов типов) и не поддерживают продвинутую функциональность, например, параметры высшего кайнда.

## 1.3. Выводы и результаты по главе

Решение Вайта и

# 2. Синтаксис, семантика, описание алгоритма

## 2.1. Описание синтаксиса и семантики

Для начала опишем, какие новые конструкции языка вносит в 1ML данная работа. Чтобы задать функцию с неявным параметром, неявный параметр нужно обернуть в квадратные скобки и явно протипизировать:

```
type SHOW = {
    type t
    show : t -> text
}

show [S: SHOW] x = S.show x
show_fun = fun [S : SHOW] x => S.show x
```

Неявный модуль будет строиться, используя доступные в области видимости переменные, объявленные неявными с помощью ключевого слова `implicit`. Множество таких переменных будем в дальнейшем называть *неявной областью видимости*.

```
implicit Show_int = {
    type t = int
    show = primitive "Int.toText"
}
```

Заметим, что при объявлении модуля `Show_int` не требуется явно указывать, что он удовлетворяет интерфейсу `SHOW`, как бы это, например, понадобилось в Scala или Haskell. Такое возможно из-за того, что в языках семейства ML используется структурная подтипизация.

Ключевым словом `implicit` можно аннотировать любое объявление переменной. В том числе, можно уже объявленный модуль внести в неявную область видимости следующим кодом:

```
Bool_to_string = {
    type t = bool
    show x = if x then "true" else "false"
}

implicit Show_bool = Bool_to_string
```

Тем же ключевым словом `implicit` помечаются *неявные функторы*:

```
implicit Show_list [S: SHOW] = {
    type t = list S.t
    show x = "[" ++ (toText S.show x) ++ "]"
}
```

Применение функции `show` возможно двумя способами: с явным указанием модуля и с неявным. В первом случае желаемый модуль оборачивается в квадратные скобки, во втором не указывается вовсе. Ниже будем называть вызов функции с пропущенными неявными аргументами *неявной аппликацией*. Следующие два вызова будут семантически эквивалентны:

```
show [Show_int] 5
show 5
```

А если вызвать функцию `show`, передав ей список чисел, то подставится `Show_list [Show_int]`, то есть следующие два вызова эквивалентны:

```
show [Show_list [Show_int]] (1 :: (2 :: (3 :: nil)))
show (1 :: (2 :: (3 :: nil)))
```

Локальные переменные могут быть также объявлены в качестве неявных:

```
let implicit Show_int = {
    type t = int
    show = primitive "Int.toText"
} in let implicit Show_list [S: SHOW] = {
    type t = list S.t
    show x = "[" ++ (toText S.show x) ++ "]"
} in show (1 :: (2 :: (3 :: nil)))
```

Синтаксис в данной работе довольно близко следует за синтаксисом из [3], это сделанно намеренно — с целью упростить сравнение мощности решений. Единственное существенное синтаксическое различие заключается в добавлении в этой работе синтаксической формы `[_]`. Её можно использовать, чтобы явно указать место, где должен быть вставлен неявный модуль:

```
show [_] 5
```

Мотивация для добавления этой конструкции будет описана в параграфе ???.

## 2.2. Общая схема алгоритма

Вывод неявных модулей производится на основе типов, которые эти модули должны иметь. То есть, чтобы определить, какой модуль необходимо неявно вывести, требуется вначале определить тип этого модуля.

Рассмотрим, как выглядят такие типы, для примера снова возьмём функцию `show`. Напомним её определение:

```
show [S: SHOW] x = S.show x
```

И протипизируем её:

```
show : [S : SHOW] -> S.t -> text
```

Применение такой функции, например, к переменной типа `int` инстанциирует `S.t = int`, то есть модуль будет искаться по типу `SHOW with t = int`, что уже (при условии, что есть только одно строковое представление для `int`) однозначно определяет модуль.

Не всегда тип неявного модуля известен заранее, так как он может зависеть от ещё невыведенных типов. Рассмотрим, например, такую функцию:

```
f x = Text.print (show x); x + 5
```

Изначально `x` типизируется как некоторая типовая переменная `α`. Если вывод неявного модуля будет запущен до обработки подвыражения `x + 5`, то поиск модуля будет производиться по типу `SHOW with t = α` и потерпит неудачу, ведь каждый модуль типа `SHOW` удовлетворяет такому типу. Если же подвыражение `x + 5` уже будет типизировано, то полученное уравнение `α = int` позволит однозначно вывести корректный модуль.

Данный пример демонстрирует, что вывод неявных модулей зависит от вывода типов. Верно и обратное. Например, пусть есть только одна реализация типа модулей `SINGLE`, имеющая тип `SINGLE with t = int`. Тогда поиск неявного модуля по типу `SINGLE with t = α` приведёт к унификации `α = int`.

Можно предположить, что, раз вывод неявных модулей зависит от вывода типов, который в свою очередь, зависит от вывода неявных модулей, то вывод одних неявных модулей может зависеть от вывода других неявных модулей. Это действительно так, и обсуждению проблем, связанных с этим, посвящена глава 4.

Так как вывод неявных модулей и вывод типов взаимно зависимы, нельзя завершить одно раньше начала другого, необходимо выполнять эти процессы в одной фазе компилятора. В существующем компиляторе 1ML проверка типов, вывод типов и преобразование в Систему $F_\omega$ осуществляются одним проходом по синтаксическому дереву. Соответственно, вывод неявных типов будет добавлен к этому проходу.

Простым решением, которое и было реализовано в первую очередь, было бы выводить неявный модуль в момент обработки неявной аппликации. Но, как было проиллюстрированно рассуждениями выше, в этот момент некоторые типы могут быть ещё не выведены, и для вывода модуля не хватит информации. На практике такое решение оказалось почти бесполезным.

Чтобы выводить модули в момент, в который про их типы будет известно больше информации, будем откладывать вывод модулей и выводить несколько модулей за раз. Обрабатывая неявную аппликацию, вставим вместо модуля специальную *неявную переменную* с уникальным именем. Значение этой переменной будет определено позднее. А вот её тип уже известен, правда, он может зависеть от нескольких типовых переменных. Поэтому проверка типов останется корректной после добавления такой фиктивной переменной.

После добавления неявной переменной продолжается типизация кода и его преобразование в Систему $F_\omega$, в процессе чего некоторые из типовых переменных, связанных с типами неявных переменных, могут быть выведены. Затем в некоторый момент (точное место этого момента будет определёно ниже, в параграфе ???) запускается поиск модулей для накопившихся неявных переменных. В случае успеха неявные переменные заменяются на термы Системы $F_\omega$, соответствующие найденным модулям, а их типы окончательно выводятся, если не были выведены до этого.

Перейдём теперь к описанию непосредственно поиска модулей по типу. Заметим, что при наличии в неявной области функторов, которые принимают в качестве аргументов другие модули, число модулей, которые могут быть подставлены, бесконечно. Поэтому решение, перебирающее всех возможных кандидатов и сравнивающее их тип с нужным, не будет завершаться.

Поэтому будем выводить модуль рекурсивно, поддерживая множество ограничений на тип. TODO

В результате поиск модулей либо корректно завершается, возвращая нужный модуль, либо возвращает одну из трёх ошибок: ”нужный модуль не найден”, ”неоднозначность” (то есть нашлось несколько подходящих кандидатов) или ”поиск не завершается” (чему будет посвящён параграф ???).

## 2.3. Выводы и результаты по главе

TODO

# 3. Детали решения и примеры использования

## 3.1. Проверка завершаемости

Легко заметить, что описанный алгоритм поиска модуля может не завершаться. Рассмотрим следующий модуль:

```
implicit Show_id [Delegate: SHOW] = {
    type t = Delegate.t
    show = Delegate.show
}
```

В присутствии такого модуля любая попытка вывести модуль будет бесконечно строить модуль вида `Show_id [Show_id [Show_id ...]]`. Чтобы гарантировать завершаемость алгоритма, применим классический метод. Будем проверять, что с момента последнего применения функтора, который алгоритм применяет в данный момент, была получена новая информация, то есть в данный момент решается более простая задача, чем решалась в предыдущий раз.

Понятие "задача стала проще" определим следующим образом: хотя бы одно ограничение на тип стало строго меньше, а остальные не увеличились. Под порядком "меньше" будем понимать простое структурное включение одного ограничения в другой. Ясно, что с таким ограничением поиск всегда будет завершаться.

Поясним эту идею на примере. (TODO)

Если было определено, что поиск может не завершаться, то следует вернуть ошибку, даже если в другой ветке поиска найден подходящий модуль, так как в таком случае нельзя гарантировать уникальность найденного модуля.

На самом деле, в некоторых случаях поиск может не завершаться, из-за того, что некоторые типы в ограничениях ещё не полностью выведены. При этом

## 3.2. Генерализация типов (???) переименовать

TODO

## 3.3. Локальные неявные модули

TODO

## 3.4. Синтаксис `[_]`

Объясним мотивацию, стоящую за введением синтаксической формы `[_]`, означающей "в этом месте необходимо вывести модуль". Дело в том, что алгоритм определяет, что модуль необходимо вывести, если обнаруживает неявную аппликацию. Тем

не менее, существуют неявные выражения, в которых аппликация вовсе отсутствует. Рассмотрим, например, следующий интерфейс, определяющий нейтральный элемент, и модуль, реализующий этот интерфейс для сложения чисел:

```
type NEUTRAL = {
    type t
    neutral: t
}

implicit NeutralAdd = {
    type t = int
    neutral = 0
}
```

И определим неявную функцию, возвращающую нейтральный элемент:

```
neutral [N : NEUTRAL] = N.neutral
```

Но как использовать эту функцию? Действительно, так как тип этой функции — `[N : NEUTRAL] -> N.t`, то нет явных аргументов, к которым можно применить эту функцию, чтобы появилась неявная аппликация. Тем не менее, в некоторых контекстах этот модуль можно вывести, например, в выражении `5 + neutral`. Такое выражение не будет компилироваться в решении для OCaml, но аналогичные конструкции поддержаны в других языках, например, в Scala. Единственный способ применить такую функцию в решении для OCaml — передать ей явный аргумент, `5 + neutral [NeutralAdd]`.

Проблема заключается в том, что с такими функциями не всегда ясно, что имел в виду программист: обращение к функции как есть, или вызов её с выводом неявного параметра. Особенно эта двусмысленность видна в выражениях вида `let x = neutral in ...` или в случае передачи функции `neutral` в качестве аргумента функции высшего порядка.

В этой работе предлагается явное синтаксическое разделение двусмысленности семантики. В случае, если за неявной функцией, у которой нет явных аргументов, будет стоять токен `[_]`, выражение будет трактоваться как неявная аппликация, и модуль будет выведен, в противном же случае функция будет трактоваться как есть. Это разумный компромисс: он позволяет не выписывать явно неявные модули для таких функций, ограничиваясь только явным указанием, что вывести нужно, при этом не усложняет ни тайп-чекер, ни читаемость кода.

## 3.5. Неявные аргументы для функторов

В решении для OCaml, тогда как модули могут использоваться в качестве неявных аргументов для *функций*, использование модулей в качестве неявных аргументов для *функторов* не поддерживается. Напомним, что в OCaml функции и функторы находятся на разных слоях языка, поэтому поддержка неявных аргументов для функторов в OCaml требует дополнительных теоретических исследований, и, вероятно, потребует дублирования функциональности. Так как в языке 1ML функции и функторы являются частными случаями одной и той же конструкции, можно минимальными усилиями переиспользовать наше решение, чтобы поддержать неявные аргументы для функторов.

Опишем несколько случаев, в которых эта функциональность может быть полезна. Допустим, были определены два модуля, отвечаюшие за строковое представление списков: `Show_list1` и `Show_list2`. Их сигнатура одинакова, поэтому при применении неявной функции `show` требуется выбрать модуль явно. Без поддержки неявных аргументов это делается следующим кодом:

```
show [Show_list1 [Show_pair [Show_string] [Show_int]]]
     (("hello" , 1) :: (("world", 2) :: nil))
```

Заметим, что неоднозначность проявляется только при выборе `Show_list1`, а вот часть `Show_pair [Show_string] [Show_int]` может быть выведена автоматически. Поддержка неявных аргументов для функторов позволяет так переписать этот вызов:

```
show [Show_list1 [_]] (("hello" , 1) :: (("world", 2) :: nil))
```

Отметим, что в этом примере была использована синтаксическая форма `[_]`, введённая в предыдущем параграфе. Тогда как необходимость в ней для неявных функций появляется достаточно редко, для функторов она требуется достаточно часто, так как у неявных функторов все аргументы неявные.

В примере выше неявные аргументы для функторов применены в комбинации с неявными аргументами для функций. Неявные аргументы для функторов могут быть использованы и сами по себе. Рассмотрим, например (TODO придумать пример (MAP?))

## 3.6. Логическое программирование

Фреймворк неявных модулей можно воспринимать как (достаточно неудобный и урезанный) способ создавать программы в логической парадигме.

## 3.7. Выводы и результаты по главе

TODO

# 4. Порядок разрешения

## 4.1. Анализ проблемы

Как уже было упомянуто в главе 2, вывод одних неявных модулей может зависеть от вывода других неявных модулей. То есть, на некоторых примерах успешность вывода модулей зависит от порядка, в котором они будут обрабатываться. Эта работа уделяет особое внимание исследованию таких примеров.

Проиллюстрируем важность этой проблемы на достаточно простом коде, реализующего возможность смешения чисел типа `int` и `float` при сложении: например, `1 + 1.1`. В рамках этой работы было обнаружено, что, несмотря на свою простоту, данный код не будет работать в решении для OCaml:

```
module type PLUS = sig
  type t and u and res
  val ( + ) : t -> u -> res
end;;

let ( + ) {P : PLUS} = P.( + );;

implicit module Int_Int = struct
  type t = int and u = int and res = int
  let ( + ) = ( + )
end;;

implicit module Float_Float = struct
  type t = float and u = float and res = float
  let ( + ) = ( +. )
end;;

implicit module Int_Float = struct
  type t = int and u = float and res = float
  let ( + ) l r = (float_of_int l) +. r
end;;

implicit module Float_Int = struct
  type t = float and u = int and res = float
  let ( + ) l r = l +. (float_of_int r)
end;;
```

```
print_float ((1 + 1.1) + 2.5);;
```

В этом коде опередеяется перегрузка оператора + для всех возможных комбинаций из `int` и `float`. Прототип неявных модулей для OCaml выдаст ошибку "неоднозначность" на последней строчке. Дело в том, что в решении для OCaml модули обрабатываются в следующем порядке: неявный аргумент в функции будет выведен до того, как будут выведены неявные аргументы в других аргументах этой функции. Так, в выражении `(1 + 1.1) + 2.5` первым будет выведен неявный аргумент для второго плюса. Так как первый плюс ещё не обработан, тип выражения `1 + 1.1` ещё не выведен, поэтому второй плюс имеет тип `'a -> float -> float`. Такому типу соответствуют и `Float_Float`, и `Int_Float`. Если бы первый плюс обрабатывался первым, то оба неявных аргумента были бы корректно выведены.

Заметим, что проблема проявляется, даже если отказаться от требования уникальности и подставить любой из кандидатов: если в примере выше не угадать и подставить `Int_Float`, а не `Float_Float`, то после этого тайп-чекер выдаст ошибку. Можно было бы после такой неудачной вставки откатиться к неоднозначному выводу и выбрать другой модуль, то есть осуществить поиск с возвратом (backtracking), но легко сконструировать код, на котором такое решение будет иметь экспоненциальную сложность.

Более того, возможны случаи, когда неправильный порядок обработки приводит к бесконечному числу кандидатов. Рассмотрим логическое сложение из параграфа ???. (TODO)

Во многих языках с неявными аргументами и выводом типов эта проблема проявляется. Из рассмотренных аналогов решение этой проблемы было обнаружено только в классах типов для Coq [15]. Для этого авторы переиспользуют специфическую тактику `eauto`, то есть их решение трудно применимо вне контекста автоматических систем доказательств.

## 4.2. Алгоритм

TODO

## 4.3. Выводы и результаты по главе

TODO

# Заключение

В рамках данной работы были достигнуты следующие результаты:
TODO

# Список литературы


[1] MacQueen David B. Using Dependent Types to Express Modular Structure // Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '86. — New York, NY, USA : Association for Computing Machinery, 1986. — P. 277–286. — URL: https://doi.org/10.1145/512644.512670.

[2] Rossberg Andreas. 1ML – Core and Modules United (F-ing First-Class Modules) // Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. — ICFP 2015. — New York, NY, USA : Association for Computing Machinery, 2015. — P. 35–47. — URL: https://doi.org/10.1145/2784731.2784738.

[3] White Leo, Bour Frédéric, Yallop Jeremy. Modular implicits // Electronic Proceedings in Theoretical Computer Science. — 2015. — Dec. — Vol. 198. — P. 22–63. — URL: http://dx.doi.org/10.4204/EPTCS.198.2.

[4] Russo Claudio V. First-Class Structures for Standard ML // Programming Languages and Systems / Ed. by Gert Smolka. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2000. — P. 336–350. — URL: https://doi.org/10.1007/3-540-46425-5_22.

[5] Frisch Alain, Garrigue Jacques. First-class modules and composable signatures in Objective Caml 3.12 // ACM SIGPLAN Workshop on ML, Baltimore, MD. — 2010.

[6] Rossberg Andreas, Russo Claudio V., Dreyer Derek. F-Ing Modules // Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation. — TLDI '10. — New York, NY, USA : Association for Computing Machinery, 2010. — P. 89–102. — URL: https://doi.org/10.1145/1708016.1708028.

[7] Harper Robert, Mitchell John C. On the Type Structure of Standard ML // ACM Trans. Program. Lang. Syst. — 1993. — apr. — Vol. 15, no. 2. — P. 211–252. — URL: https://doi.org/10.1145/169701.169696.

[8] Strachey Christopher. Fundamental concepts in programming languages. Lecture notes for International Summer School in Computer Programming. — 1967. — August.

[9] Wadler P., Blott S. How to Make Ad-Hoc Polymorphism Less Ad Hoc // Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '89. — New York, NY, USA : Association for Computing Machinery, 1989. — P. 60–76. — URL: https://doi.org/10.1145/75277.75283.

[10] Matsakis Nicholas D., Klock Felix S. The Rust Language // Proceedings of the 2014 ACM SIGAda Annual Conference on High Integrity Language Technology. — HILT '14. — New York, NY, USA : Association for Computing Machinery, 2014. — P. 103–104. — URL: https://doi.org/10.1145/2663171.2663188.

[11] Oliveira Bruno C.d.S., Moors Adriaan, Odersky Martin. Type Classes as Objects and Implicits // Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. — OOPSLA '10. — New York, NY, USA : Association for Computing Machinery, 2010. — P. 341–360. — URL: https://doi.org/10.1145/1869459.1869489.

[12] Dreyer Derek, Harper Robert, Chakravarty Manuel M. T., Keller Gabriele. Modular Type Classes // Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — POPL '07. — New York, NY, USA : Association for Computing Machinery, 2007. — P. 63–70. — URL: https://doi.org/10.1145/1190216.1190229.

[13] Chambard Pierre, Henry Grégoire. Experiments in generic programming: runtime type representation and implicit values // Presentation at the OCaml Users and Developers meeting, Copenhagen, Denmark. — 2012.

[14] Schneider Gerhard. ML mit Typklassen : Master's thesis / Gerhard Schneider ; Saarland University. — 2000. — jun.

[15] Sozeau Matthieu, Oury Nicolas. First-class type classes // International Conference on Theorem Proving in Higher Order Logics / Springer. — 2008. — P. 278–293.