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 **3**

Введение **5**

1. Обзор литературы **8**

1.1. Модули первого класса и 1ML 8

1.2. Ad-hoc-полиморфизм 9

1.3. Выводы и результаты по главе 14

2. Синтаксис, семантика, описание алгоритма **15**

2.1. Описание синтаксиса и семантики 15

2.2. Общая схема алгоритма 17

2.3. Выводы и результаты по главе 19

3. Детали решения и примеры использования **21**

3.1. Проверка завершаемости 21

3.2. Время вывода модулей 22

3.3. Локальные неявные модули 23

3.4. Синтаксис `[_]` 23

3.5. Неявные аргументы для функторов 24

3.6. Выводы и результаты по главе 26

4. Порядок разрешения **27**

4.1. Анализ проблемы 27

4.2. Алгоритм 29

4.3. Выводы и результаты по главе 32

Заключение **33**

Аннотация

Отсутствие ad-hoc-полиморфизма в языках семейства ML может считаться существенным недостатком. Цель данной работы — расширить экспериментальный язык 1ML ad-hoc-полиморфизмом с помощью неявных модулей. Язык 1ML отличается минимальным и однородным подходом к системе модулей ML и отсутствием зависимых типов, что значительно упрощает некоторые задачи, возникающие при разработке ad-hoc-полиморфизма. Предлагаемое решение основывается на прототипе Вайта и других для языка OCaml, при этом является более полным. В решении поддержаны неявные аргументы для функторов, которые до этого не поддерживались ни в одной работе о ML, а также разработан алгоритм, эффективно находящий корректный порядок вывода неявных модулей.

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

The lack of ad hoc polymorphism in languages of the ML family can be considered a significant problem. The goal of this work is to extend the experimental language 1ML with ad hoc polymorphism using implicit modules. 1ML is distinguished by a minimal and uniform approach to the ML module system and the absence of dependent types. These features greatly simplify some of the problems that arise when developing ad hoc polymorphism. Our solution is based on the prototype of White et al. for the OCaml language and is more complete compared to it. This work supports implicit arguments for functors, which were not supported in previous literature on ML. Also, we present an algorithm that efficiently finds the correct order to resolve implicit modules.

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 исследованию способов поддержки ad-hoc-полиморфизма в языках семейства ML. Эти языки широко применяются в таких областях, 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.

Одним из существенных недостатков 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, система проверки типов (type checker) 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 *системой модулей*, основанной на теории зависимых типов [**?**]. 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 [?] был создан для решения проблем системы модулей. 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 [?], 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 для языка 1ML, повторяющих функциональность решения для OCaml;
- Разработка алгоритма, позволяющего полно и эффективно осуществлять вставку неявных модулей;
- Поддержка неявных аргументов для функторов;
- Сравнение с существующим решением [?] 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

В данной главе будут описаны имеющиеся результаты по теме работы. Сначала будут рассмотрены причины появления и особенности языка 1ML. Затем будут описаны методы, с помощью которых ad-hoc-полиморфизм достигается в некоторых языках. Наконец, будут представлены работы, связанные с добавлением ad-hoc-полиморфизма в языки семейства ML.

1.1. Модули первого класса и 1ML

Отсутствие модулей первого класса порождает практические проблемы при попытке интеграции между основным и модульными слоями ML [?]. Например, долгое время был невозможен динамический выбор модуля. Классическим примером, представленным в листинге 1, является невозможность выбрать реализацию ассоциативного массива в зависимости от числа элементов. Такая функциональность доступна почти в любом объектно-ориентированном языке, поэтому невозможность добиться этого в ML считалась крупным недостатком.

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

Листинг 1: Естественный код для динамического выбора модуля

С целью сделать модули объектами первого класса Руссо предложил дополнить основной язык явной упаковкой и распаковкой модулей (packaged modules) [?]. Это предложение было расширено последующими работами [?] [?] и было реализовано во многих диалектах ML, включая OCaml [?]. 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.

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

Листинг 2: Динамический выбор модуля в OCaml

В листинге 2 представлено, как в данный момент можно динамически выбрать модуль. Несмотря на то, что задача решена, этот код выглядит намного менее лаконично и естественно, чем код в листинге 1. Что же касается выразительности, приведём пример, ограничивающий использование полиформизма. Типизация полиморфной функ-

ции, 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 модулями, которые действительно являются объектами первого класса, 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, а также что теория зависимых типов не является обязательной для выражения системы модулей ML. Так, Россберг и другие показали, что все конструкции ML, включая модули, выражаются в рамках Системы F_ω [**?**]. Эти исследования привели к созданию Россбергом экспериментального диалекта 1ML [**?**], 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 [**?**] 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 [**?**]. 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* [?], 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
```

Листинг 3: Классы типов в Haskell

В листинге 3 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) [?]. Важным свойством, выполняющимся для этих решений, является *согласованность* (coherence) — каждая корректная типизация программы должна приводить к одной и той же динамической семантике. Решения из этой группы достигают согласованности с помощью *каноничности* — для каждого типа и класса типов в текущей области видимости может быть определено не более

одного экземпляра. Существуют расширения для Haskell, ослабляющие требования к каноничности, однако эти расширения могут приводить к нарушению согласованности.

Представителем этой группы является Scala и реализованные в ней *неявные аргументы*, также называемые *имплициты* [?]. В Scala не требуется каноничность и присутствует возможность явно передать экземпляр, если система типов не способна вывести его самостоятельно. Для неявных аргументов также выполняется согласованность. Чтобы гарантировать согласованность, 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. Пример использования неявных аргументов в Scala приведён в листинге 4. Помимо списка обычных параметров, функция может объявить второй список неявных параметров, которые при условии однозначности можно опустить.

```
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)
```

Листинг 4: Неявные аргументы в Scala

Неявные параметры в разных формах часто встречаются в языках с зависимыми типами. Рассмотрим подробнее Coq, язык, по историческим причинам имеющий много общего с ML. Функциональность, которая в Coq называется неявными параметрами, достаточно примитивна и является скорее аналогом параметрического полиморфизма для языка с зависимыми типами. Ad-hoc-полиморфизм же достигается с помощью двух механизмов: более простых и предсказуемых *канонических структур* [?] и более мощных *классов типов* [?]. Несмотря на название, и классы типов, и канонические структуры в Coq имеют больше общего с неявными аргументами в Scala, чем с классами типов в Haskell. Так, все используемые в функции классы ти-

пов следует указывать явно, а каноничность не требуется. Согласованность в Coq не выполняется, но для Coq это не является таким крупным недостатком, как для неинтерактивных языков. Так как Coq в основном используется в контексте автоматических доказательств, в котором достаточно найти любое подходящее доказательство, канонические структуры и классы типов не считают ошибкой неоднозначный вывод неявного параметра — вместо этого подставляется любой подходящий.

```
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] *)
```

Листинг 5: Неявные модули в OCaml

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

В листинге 5 приведён пример кода, использующего неявные модули для OCaml. Модуль можно аннотировать ключевым словом `implicit`, что делает его доступным для поиска неявных аргументов. Функции, принимающей неявный модуль, можно либо передать этот модуль в качестве аргумента, либо опустить его, и модуль будет выведен автоматически.

Вайтом и другими был реализован прототип, который добавляет в OCaml неявные

модули, но в основном языке данная функциональность до сих пор не поддерживается. Реализация полного решения сложна как практически (добавление такой нетривиальной функциональности в столь сложный язык, как OCaml, требует значительной работы), так и теоретически. Основной преградой стала унификация на модульном уровне [?]. На данный момент система проверки типов OCaml не поддерживает такую унификацию.

Данная работа основывается именно на работе Вайта и других для реализации неявных модулей в 1ML. Отказ языка 1ML от зависимых типов снимает упомянутые выше проблемы с унификацией. Сравнение также будет осуществлено с работой для OCaml. Существуют также альтернативные предложения для поддержания ad-hoc-полиморфизма в языках семейства ML, но по сравнению с работой Вайта и других они имеют существенные недостатки, которые будут рассмотрены ниже.

```
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)
```

Листинг 6: Пример, демонстрирующий невозможность каноничности в OCaml

Как уже было упомянуто, проверка каноничности возможна не во всех языках, включая ML. Система модулей ML позволяет скрывать экземпляры за абстракцией. Рассмотрим, например, следующий код на OCaml, представленный в листинге 6. В этом примере уже определён экземпляр класса типов `SHOW` для типа `int`. Тем не менее,

посредством модулей скрыт ещё один экземпляр того же класса типов для того же типа, а именно `N.S`. Тогда как в данном примере можно определить наличие такого экземпляра, анализируя внутреннюю структуру модулей, это невозможно в общем случае и противоречит самой идее модульности. Аналогичный приведённому пример можно построить и для других языков семейства ML, включая 1ML.

Дрейером и другими были предложены модульные классы типов [?], 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, модульные 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 [?] и классы типов [?] 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

Рассмотренны особенности языка 1ML. Изложены с примерами два подхода к поддержанию согласованности: каноничность и указание неявных аргументов. Описано решение Вайта и других, на котором основывается данная работа. Приведены недостатки альтернативных решение для семейства ML: решение Дрейера и других накладывает ограничения в связи с невозможностью проверки каноничности в ML, а решения, не зависящие от системы модулей приводят к дубликации конструкций.

2. Синтаксис, семантика, описание алгоритма

В данной главе будет описаны синтаксис и семантика для новой функциональности на примере модулей, отвечающих за строковое представление данных различных типов, а затем дано высокоуровневое описание алгоритма, осуществляющего вывод неявных модулей.

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
```

Функтором в ML называется модуль, параметризуемый другим модулем. Функтор, который помечен ключевым словом `implicit` и имеет только неявные аргументы называется *неявным функтором*:

```
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*. Так же как в решении для OCaml, в данной работе неявную аппликацию можно применить только в случае, если существует единственный модуль, который можно подставить, в противном случае компиляция завершится с ошибкой "неоднозначность". В случае, если есть несколько подходящих модулей, выбор между ними следует осуществить явно. Таким образом, в данной работе представлено решение, для которого верна согласованность, но не обязательна каноничность. 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 [?], это сделано 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 3.4.

2.2. Общая схема алгоритма

Вывод неявных модулей производится на основе типов, которые должны иметь эти модули. 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 переменная вывода типов `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 компиляции. В существующем компиляторе 1ML проверка типов, вывод типов и преобразование в Систему F_ω 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 быть не полностью выведен, то есть зависеть от нескольких переменных вывода типов. Поэтому проверка типов останется корректной после добавления такой фиктивной переменной.

После добавления неявной переменной продолжается типизация кода и его преобразование в Систему F_ω, в процессе чего некоторые из переменных вывода типов, связанных с типами неявных переменных, могут быть выведены. Затем в некоторый момент (точное место этого момента будет определёно ниже, в параграфе 3.2) запускается поиск модулей для накопившихся неявных переменных. В случае успеха неявные переменные заменяются на термы Системы F_ω, соответствующие найденным модулям, а их типы окончательно выводятся, если не были выведены до этого.

Перейдём теперь к описанию непосредственно поиска модулей по типу[1]. 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.

В связи с этим будем выводить модуль рекурсивно, поддерживая множество ограничений на тип, и проверять модуль на соответствие ограничениям с помощью отно-

[1]Эта функциональность реализована в файле implicitsearch.ml, в функции implicit_search. Здесь и далее ссылки на реализацию даются на репозиторий github.com/trilis/1ml

шения подтипизации. Псевдокод для поиска модулей по типу приведён в алгоритме 1. Для примера рассмотрим вызов `show (1 :: (2 :: (3 :: nil)))`. Из контекста будет определено, что необходимо найти модуль с типом `SHOW with t = list int`. Функтор `Show_list` имеет тип `[S : SHOW] -> SHOW with t = list S.t`. Чтобы проверить, подходит ли он под требуемые ограничения, проверим, является ли `list S.t` подтипом `list int`. Это верно, если `S.t = int`, поэтому выполним такую унификацию и запустимся рекурсивно. Чтобы вывести аргумент функтора, нужно найти модуль с типом `SHOW with t = int`, чему удовлетворяет модуль `Show_int`. Таким образом, ответ — `Show_list [Show_int]`.

В результате поиск модулей либо корректно завершается, возвращая нужный модуль, либо возвращает одну из трёх ошибок: "нет кандидатов", "неоднозначность" (то есть нашлось несколько подходящих кандидатов) или "поиск не завершается" (чему будет посвящён параграф 3.1).

2.3. Выводы и результаты по главе

Синтаксис решения повторяет предложенный для OCaml, за исключением добавления синтаксической формы `[_]`. Вывод модулей и вывод типов тесно связаны между собой. Дойдя до неявной аппликации, подставим неявную переменную с неизвестным типом. Будем откладывать вывод модулей, чтобы получить больше информации из вывода типов. Чтобы найти модуль по типу, перебираются все модули и сравниваются с помощью подтипизации, аргументы функторов выводятся рекурсивно.

Алгоритм 1
Поиск подходящего модуля

```
function SEARCH_HELPER(constraints)
    candidates ← ∅
    for module in неявная область видимости do
        if module — функтор then
            argTypes ← типы аргументов функторов
            resType ← тип возвращаемого значения функтора
            if resType удовлетворяет constraints then
                type ← тип, построенный по constraints
                unify(type, resType)
                candidates ← candidates ∪ {search_args(argTypes)}
        else
            if type(module) удовлетворяет constraints then
                candidates ← candidates ∪ {module}
    return candidates
function SEARCH_ARGS(argTypes, head)
    if argTypes = ∅ then
        return head
    result ← ∅
    argType, otherArgTypes ← argTypes
    candidates ← search_helper(generate_constraints(argType))
    for candidate in candidates do
        argType′, otherArgTypes′ ← copy(argType, otherArgTypes)
        unify(argType′, type(candidate))
        result ← result ∪ search_args(otherArgTypes′, head @ candidate)
    return results
function SEARCH(type)
    constraints ← generate_constraints(type)
    candidates ← search_helper(constraints)
    if |candidates| = 0 then
        return ошибка "нет кандидатов"
    else if |candidates| > 1 then
        return ошибка "неоднозначность"
    else
        candidate ← единственный элемент в candidates
        unify(type, type(candidate))
        return candidate
```

3. Детали решения и примеры использования

В данной главе будут описаны важные детали решения: метод, которыми достигается проверка завершаемости, выбор времени, в который выводить накопившиеся модули, особенности реализации локальных неявных модулей. Далее, будет дана мотивация для добавления в язык синтаксической формы `[_]`. Наконец, будут приведены примеры использования новой функциональности, предложенной в данной работе — неявных аргументов для функторов.

3.1. Проверка завершаемости[2]

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 классическую эвристику. 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. При выводе неявного модуля для строкового представления списка списка чисел сначала будет искаться модуль с типом `SHOW with t = list list int`. После применения функтора `Show_list` в качестве аргумента будет искаться модуль с типом `SHOW with t = list int`, что структурно меньше, чем предыдущее ограничение, так что эта ветка поиска корректна. Если же вместо функтора `Show_list` был подставлен функтор `Show_id`, то снова бы искался модуль с типом `SHOW with t = list list int`, и алгоритм возвращает ошибку "поиск не завершается".

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.

[2]Эта функциональность реализована в файле implicitsearch.ml, в функции termination_check.

На самом деле, в некоторых случаях поиск может не завершаться, из-за того, что некоторые типы в ограничениях ещё не полностью выведены. При этом эти типы могут быть выведены после обработок некоторых других, завершающихся веток поиска. Поэтому, обнаружив ветку поиска, которая может не завершиться, следует не сразу заканчивать поиск с ошибкой, а обозначить эту ветку *заблокированной* и попробовать продвинуться в других ветках. Только если все ветки либо полностью просмотрены, либо заблокированы, следует вернуть ошибку.

3.2. Время вывода модулей

Напомним, что в некоторый момент накопившиеся неявные переменные будут выведены все вместе. Определим, в какой именно момент это будет происходить.

Главным соображением на этот счёт является следующее: чем дальше откладывается вывод модулей, тем больше типов будет выведено, следовательно, тем больше шансов на однозначность вывода модулей. Поэтому было бы разумно откладывать вывод модулей как можно дольше.

Если переменная вывода типов, связанная с неявным модулем, будет обобщена (generalized), то она превратится в типовую переменную, которую ни с чем нельзя унифицировать, что приведёт к ошибке. Руководствуясь этим, в решении для OCaml вывод модулей происходит непосредственно до обобщения типов, то есть, фактически, в момент обработки ближайшего к неявной аппликации let-определения.

В данной работе было реализовано следующее решение. Переменные вывода типов, связанные с неявными модулями, должны в результате унификации стать конкретными типами, если поиск модулей будет успешным. После обработки всей программы такие переменные не могут оставаться невыведенными — это бы значило, что поиск одного из неявных модуей был неудачен. Следовательно, обобщать эти переменные вообще не нужно. Будем помечать переменные, связанные с неявными модулями, и пропускать их при обобщении.

Такое изменение делает выбор времени, в которое будут выводиться модули, более свободным. В частности, ничто не мешает выводить модули в самом конце типизации, получив максимально выведенные типы. В реализации этой работы было принято решение обрабатывать модули в момент достижения верхнеуровнего let-определения. Зависимости между неявными аппликациями из разных верхнеуровневых конструкций крайне редки, а предлагаемое решение позволяет независимо обрабатывать верхнеуровневые конструкции, что, на мой взгляд, является более естественным поведением.

3.3. Локальные неявные модули

Решение выводить модули в момент достижения верхнеуровнего let-определения поднимает небольшую проблему, связанную с неявными модулями, объявленными локально.

К примеру, рассмотрим следующую функцию, сортирующую (с помощью функции `sort`, неявно принимающей компаратор) строки сначала лексикографически, а затем по длине:

```
sort_strings strs =
    let by_lex = let implicit Sort_Lex = ... in sort strs in
    let by_len = let implicit Sort_Len = ... in sort strs in
    (by_lex, by_len)
```

В этом примере оба неявных аргумента будут выведены при обработке верхнеуровнего let-определения функции `sort_strings`. Но к этому моменту модули `Sort_Lex` и `Sort_Len` уже покинули область видимости и недоступны для поиска.

Получается, чтобы поддерживать неявную область видимости, требуется иметь доступ к контексту, который был доступен в момент неявной аппликации. Но хранить отдельную копию контекста для каждого контекста было бы расточительно по памяти.

Будем поддерживать неявную область видимости в виде дерева, повторяющего структуру абстрактного синтаксического дерева, но хранящего только релевантные вершины. В этом дереве будут храниться неявные модули вместе со справочной информацией, такой как, например, доступные в контексте и используемые в неявных модулях типовые переменные.

Для каждой неявной аппликации будем хранить вершину на таком сжатом дереве, которая соответствует её месту в синтаксическом дереве. Тогда, чтобы получить неявную область видимости для конкретной неявной аппликации, достаточно рассмотреть все модули на пути от корня до сохранённой для этой аппликации вершины. Заметим, что такая обработка на пути столь же эффективна, как обычный перебор всех модулей из контекста.

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 систему проверки типов, 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 решение, представленное в данной работе, чтобы поддержать неявные аргументы для функторов. Можно сказать, что в данной работе поддержан ad-hoc-полиморфизм на модульный уровнем.

Опишем несколько случаев, в которых эта функциональность может быть полезна. Допустим, были определены два модуля, отвечающие 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, чтобы сделать короче код, применяющий функторы. Например, определим `Set` как функтор с неявным аргументом:

```
Set [Elem : ORD] :> SET with (elem = Elem.t) = {
    (* ... *)
}
```

Тогда не будет нужно явно передавать модуль, отвечающий за элементы `Set`:

```
let S = Set [_] in S.add 3 (S.add 5 S.empty)
```

3.6. Выводы и результаты по главе

Проверка завершаемости достигается стандартной эвристикой — проверяется, что задача с каждым вызовом становится проще. Из-за игнорирования переменных, связанных с неявными модулями, модули в этой работе выводятся позднее, чем в решении для OCaml, что приводит к получению большего числа информации о выводе типов. Такое решение поднимает новую проблему: к моменту вывода модули уже покинули область видимости. Эта задача решается с помощью поддерживания сжатого дерева из неявных модулей, симулирующего нужную часть контекста. Введена синтаксическая категория `[_]` для поддержки неявных функций и функторов, у которых все аргументы неявные. Наконец, продемонстрированы неявные аргументы для функторов как совмещённые с неявными аргументами для функций, так и сами по себе.

4. Порядок разрешения

Данная глава посвящена исследованию влияния, которое выбор порядка вывода модулей оказывает на корректность алгоритма. Вначале будет представлены на примерах недостатки решения для OCaml, а затем будет представлен алгоритм, лишённый этих недостатков, а также описаны оптимизации и эвристики к этому алгоритму и анализ его сложности.

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.

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

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

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;;

(* Аналогично объявленные Int_Int и Float_Int опущены для краткости *)

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

Листинг 7: Пример кода на OCaml, который не работает из-за некорректного порядка вывода

Проиллюстрируем важность этой проблемы на достаточно простом коде, приведённый в листинге 7. Этот код реализует 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.

В листинге 7 определяется 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 `α -> 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.

Этот конкретный пример был бы обработан корректно, если предложить обратный принятому в OCaml порядку вывода: выводить сначала неявные аргументы в аргументах неявной функции, и лишь потом неявные аргументы этой функции. Но и этот порядок может быть неоптимальным. Чтобы показать это, дополним пример следующим кодом:

```
module type SQRTABLE = sig
  type t
  val sqrt : t -> t
end;;

sqrt [S: SQRTABLE] = S.sqrt

implicit Squartable_float = {
    type t = float
    sqrt = Float.sqrt
}
```

И попробуем обработать следующую функцию:

```
test x = (1.1 + x) + (sqrt x)
```

Вывод неявного модуля для `sqrt` даёт типизацию `x : float`, и остальные неявные модули выводятся однозначно. Если же первый плюс будет обработан до обработки `sqrt`, то результат будет неоднозначным.

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 система проверки типов 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 построить пример, на котором такое решение будет иметь экспоненциальную сложность.

Эта проблема проявляется во многих языках с неявными аргументами и выводом типов, например, в канонических структурах для Coq и в языке Arend. Из рассмотренных аналогов решение этой проблемы было обнаружено только в классах типов для Coq [**?**]. Как уже было описано, это решение опирается на возможность несогласованности, что нежелательно в неинтерактивном языке, поэтому решение из Coq неприменимо для решения задач данной работы.

4.2. Алгоритм[3]

Имеется набор неявных переменных, каждой из которых соответствует тип, возможно, зависящий от переменных вывода типов, то есть не полностью выведенный. В разных типах могут быть использованы одни и те же переменные. Если поиск выдаёт ошибку "нужный модуль не найден", следует сразу же завершиться с этой ошибкой. Но если поиск завершился с ошибкой "неоднозначность" или "поиск не завершается", имеется возможность, что с более конкретным типом (который будет получен после обработки других неявных переменных) поиск может корректно завершиться.

Псевдокод для полной версии алгоритма приведён в алгоритме 2. Опишем сначала упрощённую версию алгоритма. Будем запускать поиск для различных неявных переменных, пока не найдётся переменная, для которой поиск успешно завершится. Успешно завершённый поиск может дать новую информацию посредством конкретизации некоторых переменных вывода типа, поэтому снова будем запускать поиск для оставшихся неявных переменных. В конце либо все неявные переменные разрешены, либо ни одна из оставшихся неявных переменных не может быть однозначно выведена. Если существует некоторый порядок, позволяющий вывести все неявные переменные, то такой алгоритм найдёт его. Таким образом, мы решили задачу, однако крайне неэффективно, поэтому опишем несколько оптимизаций.

Ниже будут определены три приоритета. На каждом шаге алгоритма будем пробовать обработать неявную переменную с высшим приоритетом из всех оставшихся.

1. Высший приоритет присваивается тем неявным переменным, которые зависят только от переменных вывода типов, не встречающихся в типах других неявных переменных (в частности, высшим приоритетом обладают неявные переменные с полностью выведенным типом). Если такая "уникальная" неявная переменная

[3]Эта функциональность реализована в файле implicitsearch.ml, в функциях resolve_step и init_state.

появилась, нужно обработать её в первую очередь, ведь если их обработка завершилась с ошибкой, можно не надеяться на новую информацию от других неявных переменных, а следует завершить весь алгоритм;

2. Второй приоритет присваивается неявным переменным, которые не были обработаны ни разу;

3. Третий приоритет — неявным переменным, которые уже были безуспешно обработаны до этого, но с момента последней обработки которых некоторые переменные вывода типов, содержащиеся в типе этих неявных переменных, были выведены.

В отличие от наивного квадратичного алгоритма, алгоритм с такими оптимизациями будет сначала обрабатывать (за оптимальное, линейное время) те неявные переменные, между которыми нет сложных зависимостей. То есть применение такого алгоритма строго лучше имеющегося в OCaml: тогда как решение, предложенное в этой работе, не замедлится на случаях, которые можно обрабатывать в любом порядке, оно позволяет дополнительно поддержать все случаи, работающие только при определённом порядке обработки.

В то же время, сложность вывода модулей со сложными зависимостями не сильно хуже. Пусть N — число модулей, которые должен вывести алгоритм, а K — сумма количеств переменных вывода типов по типам всех неявных модулей. Так как каждый запуск поиска происходит либо в первый раз для данной неявной переменной, либо после конкретизации переменных вывода типов, число запусков поиска имеет сложность $\mathcal{O}(N + K)$. Нужно заметить, что для большинства практических случаев эта сложность близка к линейной, так как чаще всего тип модуля зависит от небольшого числа переменных вывода типов.

Также были применены следующие эвристики, которые эффективно показали себя на имеющихся тестах:

- Среди неявных переменных, которые не были обработаны ни разу, в первую очередь будем обрабатывать ту, которая зависит от меньшего числа переменных вывода типов. Мотивация следующая: если неявные переменным соответствуют модулям одного и того же типа (как это часто бывает), то больше шансов корректно обработать неявную переменную с более конкретным типом;

- Среди неявных переменных, которые уже были обработаны, в первую очередь будем обрабатывать ту, у которой с момента последней обработки больше переменных вывода типа было выведено.

Алгоритм можно ещё больше оптимизировать с помощью мемоизации, сохраняя некоторые результаты неудачного поиска. Например, можно сохранять информацию

о ветках, в которых не найдено ни одного кандидата (в последующих запусках в эти ветки можно не заходить) или частичную информацию о переменных вывода типов (если стало известно, что какое-то подмножество переменных вывода типов принимает одинаковое значения для всех возможных кандидатов, то можно унифицировать эти переменные даже в случае безуспешного поиска). В данной работе были реализованы только самые примитивные из подобных техник, но подробное исследование мемоизации для данной задачи является интересным направлением дальнейшей работы.

Алгоритм 2 Перебор неявных переменных

```
V ← множество неявных переменных
V' ← ∅
while |V ∪ V'| ≠ 0 do
    V_uniq ← {v | v ∈ V ∪ V' и ∀v' ≠ v ∈ V типы v и v' независимы }
    if |V_uniq| ≠ 0 then
        v ← любая из V_uniq
        обработать v
        if обработка завершилась с ошибкой then
            завершиться с ошибкой
        else
            V ← V \ {v}
            V' ← V' \ {v}
    else if |V| ≠ 0 then
        v ← min V по числу переменных в типе
        обработать v
        if успешно обработана then
            V ← V \ {v}
        else if обработка завершилась с ошибкой "нет кандидатов" then
            завершиться с ошибкой
        else
            V ← V \ {v}
            V' ← V' ∪ {v}
    else
        V'_new ← {v | v ∈ V' и с последней обработки v получили о v информацию}
        if |V'_new| = 0 then
            завершиться с ошибкой
        else
            v ← max V'_new по выведенным с последней обработки переменным в типе v
            обработать v
            if успешно обработана then
                V' ← V' \ {v}
            else if обработка завершилась с ошибкой "нет кандидатов" then
                завершиться с ошибкой
```

4.3. Выводы и результаты по главе

Исследованы недостатки решений, обрабатывающих вывод модулей в фиксированном порядке. Был найден пример достаточно простого кода, который не работает в прототипе OCaml из-за этого недостатка. Предложен алгоритм, позволяющий корректно типизировать программу, если при каком-то порядке вывода эта программа корректно типизируется. Описанные оптимизации и эвристики делают предложенный алгоритм столь же эффективным на поддержанных в прототипе OCaml случаях, как и в прототипе для OCaml. При этом предложенный алгоритм способен корректно обрабатывать программы, не поддержанные в прототипе для OCaml, то есть предложенное решение является более полным.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- Разработано решение, добавляющее в язык 1ML ad-hoc-полиморфизм с помощью неявных модулей. Это решение было реализовано на языке OCaml в качестве расширения компилятора 1ML. Получившийся расширенный компилятор можно найти по адресу github.com/trilis/1ml;

- Поддержаны неявные аргументы для функторов. Эта функциональность не была поддержана в прототипе для OCaml, но авторы решения для OCaml упомянули добавление данной функциональности как интересное направление для дальнейших исследований. Этот результат достигнут за счёт использования языка 1ML, а именно за счёт уникального подхода к системе модулей ML в этом языке, и может восприниматься как дополнительный аргумент в пользу диалектов с однородным подходом к модулям;

- Показано с приведением явных примеров, что решение для OCaml может не работать на достаточно простых программах. Исследована причина, по которым это происходит, а именно неоптимальный порядок вывода модулей. Предложено и реализовано решение, которое является более полным по сравнению с решением для OCaml и при этом почти не проигрывает ему в эффективности. Это решение может с некоторой доработкой быть применено и в прототипе для OCaml;

- Осуществлена апробация решения с помощью запуска модифицированного компилятора на программах, содержащих неявные модули. Проверялось, что программы корректно типизируются. В основу тестового набора легли примеры из статей по теме и тестовый набор прототипа для OCaml. Апробация показала, что решение работает на тестах, на которых работает решение для OCaml. Также предлагаемое решение работает на тестах, которые в OCaml не поддерживаются, связанных с порядком разрешения и неявными аргументами для функторов, то есть является более полным.