

# Переоснащение неявных модулей для языка 1ML

Трилис Алексей Андреевич

научный руководитель: к.ф.-м.н. Д.А. Березун

НИУ ВШЭ — Санкт-Петербург

30 апреля 2021 г.

# Введение. Ad hoc полиморфизм

- Ad hoc полиморфизм — свойство языка, позволяющее функциям иметь различную семантику в зависимости от типов аргументов
- Например, `print` и `+`
- В процедурных и объектно-ориентированных языках обычно достигается перегрузкой
- Но в языках с мощным выводом типов нужны более сложные методы

# Введение. Семейство языков ML

- ML, OCaml, SML, F#, ...
- Функциональные языки с мощным выводом типов
- Продвинутая система модулей, основанная на теории зависимых типов<sup>1</sup>
- Активно используется в разработке и исследовании языков программирования
- А также в верификации, финансах, веб-разработке и других областях
- Отсутствует ad hoc полиформизм:
  - + для int, +. для float
  - print\_int, print\_float, print\_string, ...

---

<sup>1</sup>MacQueen, “Using Dependent Types to Express Modular Structure”, 1986.

# Введение. Модули в ML

- Язык разделён на два слоя: основной и модульный
- Модульный язык более мощный, но требует избыточности и излишней явности деклараций
- Слои плохо интегрируются между собой: нельзя динамически выбрать модуль
- Нельзя:

```
1  module Table = if size > threshold then HashMap  
    else TreeMap
```

- Можно:

```
1  module Table = (val (if size > threshold then (  
    module HashMap : MAP) else (module TreeMap :  
    MAP))) : MAP)
```

- В некоторых случаях такая интеграция невозможна

- Было показано<sup>2</sup>, что модули могут быть выражены и без использования теории зависимых типов
- А именно, что модули можно полностью выразить в System  $F_{\omega}$
- В результате этих исследований был создан экспериментальный диалект 1ML<sup>3</sup>, где основной и модульный слои объединены
- В нём действительно есть модули первого класса

---

<sup>2</sup>Rossberg, Russo и Dreyer, “F-Ing Modules”, 2010.

<sup>3</sup>Rossberg, “1ML – Core and Modules United (F-ing First-Class Modules)”, 2015.

# Обзор литературы. Классы типов

```
1 class Show a where
2   show :: a -> String
3
4 instance Show Int where
5   show = showSignedInt
6
7 show_twice x = show x ++ show x
8
9 show_twice : Show a => a -> string
```

- Впервые в Haskell<sup>4</sup>, затем в Agda, Rust, ...
- Требуется каноничность — не более одного экземпляра для каждого типа

---

<sup>4</sup>Wadler и Blott, “How to Make Ad-Hoc Polymorphism Less Ad Hoc”, 1989.

- Впервые в Scala<sup>5</sup>
- Не требуется каноничность, только однозначность
- Требуется явно описать неявные параметры, которые используются в функции

---

<sup>5</sup>Oliveira, Moors и Odersky, “Type Classes as Objects and Implicits”, 2010.

# Обзор литературы. Неявные параметры

```
1 trait Showable [T] { def show (x: T): String }
2
3 implicit object IntShowable extends Showable [Int] {
4   def show (x: Int) = x.toString
5 }
6
7 def show[T](x : T)(implicit s: Showable [T]): String = {
8   s.show(x)
9 }
10
11 show(7)(IntShowable)
12 show(7)
```



- Попытка применить классы типов в ML<sup>6</sup>
- Каноничность невозможна в модульном языке
- Поэтому решение вводит ряд серьёзных 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
  - Все неявные модули должны определять тип  $t$ , по которому будет проходить унификация

---

<sup>6</sup>Dreyer и др., “Modular Type Classes”, 2007. 

- Попытка применить неявные параметры в OCaml<sup>7</sup>, вместо обычных параметров используются модули
- Не нужна каноничность, нет ограничений как в модульных классах типов
- Нет неявных преобразований
- Нет приоритета неявных модулей, несколько подходящих кандидатов приводят к ошибке
- Есть прототип, но в основной язык интегрировать не получилось
- Требуется унификация на модульном уровне, в прототипе недостаточно сильная

---

<sup>7</sup>White, Bour и Yallop, “Modular implicits”, 2015. 

# Обзор литературы. Неявные модули

```
1 module type Show = sig
2   type t
3   val show : t -> string
4 end
5
6 implicit module Show_int = struct
7   type t = int
8   let show x = string_of_int x
9 end
10
11 implicit module Show_list {S : Show} = struct
12   type t = S.t list
13   let show x = string_of_list S.show x
14 end
15
16 let show {S : Show} x = S.show x
17
18 show 5 (* show {Show_int} 5 *)
19 show [1;2;3] (* show {Show_list (Show_int)} [1;2;3] *)
```

- Расширение OCaml неявными модулями — сложная задача, требующая огромной и практической, и теоретической работы
- Попробуем реализовать неявные модули на более простом диалекте OCaml и попытаемся улучшить слабые стороны предыдущего решения
- Выбран 1ML из-за уникального подхода к модульной системе, который может помочь в получении новых результатов

## Цель:

- Дополнить язык 1ML поддержкой неявных модулей

## Задачи:

- Реализация неявных модулей, повторяющих функциональность решения для OCaml
- Расширение решения новой функциональностью
- Тестирование и сравнение с неявными модулями для OCaml

- Общая схема
- Интересные места реализации
  - Проверка завершаемости
  - Генерализация типов
  - Локальные неявные модули
- Новая функциональность
  - Порядок разрешения
  - Неявные функторы
- Тестирование и сравнение

# Общая схема. Фаза неявных модулей

- Обработка неявных модулей тесно связана с выводом типов, одно нельзя сделать до второго. Поэтому их нужно делать в одной фазе компиляции
- Обработывая неявную аппликацию, подставим неявную переменную
- Зная, какой нужен тип, нужно найти модуль с таким типом
- Отложим определение этого модуля, будем обрабатывать несколько неявных переменных за раз

- Бесконечное число модулей
- Представим текущее состояние поиска как набор ограничений на тип
- Переберём все доступные модули, если подходит под ограничения — подставим
- Если это функтор, то запустимся рекурсивно с новыми ограничениями
- Результат — "успех", "нет кандидатов", "неоднозначность", "не завершается"



# Проверка завершаемости

```
1 implicit module Show_it {S : Show} = struct
2   type t = S.t
3   let show = S.show
4 end
```

- Поиск может не завершаться:  
Show\_it(Show\_it(Show\_it...))
- Нужно определять ситуации, когда алгоритм может не завершиться
- Будем определять незавершаемость, если между двумя применениями одного и того же модуля входные данные не стали меньше
- То есть хотя бы одно ограничение на тип стало меньше, остальные стали не больше
- Нужно подождать, не уточнится ли вход из других веток, и если нет, то завершиться

- До какого момента откладывать разрешение неявных переменных?
- Чем дольше, тем больше информации
- В OCaml — до ближайшей генерализации, то есть до ближайшего `let`-связывания
- Можно пропускать типовые переменные, связанные с неявными переменными, при генерализации
- Тогда в теории можно откладывать разрешение до самого конца обработки программы
- В реализации откладываем до ближайшего объявления на верхнем уровне

# Локальные неявные модули

```
1 let f = show 5 ^ " " ^  
2   (let implicit module Show_float = struct  
3     type t = float  
4     let show x = string_of_float x  
5   end in show 3.14)
```

- Так как разрешение неявных переменных отложено, к моменту разрешения некоторые модули могут выйти из контекста
- Храним дерево из неявных модулей и побочной информации
- Каждая неявная переменная сопоставляется с вершиной в дереве, может использовать модули на пути от этой вершины до корня

- В решении для OCaml неявные переменные разрешаются в некотором фиксированном порядке
- Это уменьшает полноту решения, например, нельзя реализовать сложение `int` с `float`
- Будем запускать разрешение неявных переменных несколько раз, с появлением новой информации
- Запоминаем и используем предыдущие результаты

# Порядок разрешения. Пример

```
1 module type Num = sig
2   type t and u and res
3   val ( + ) : t -> u -> res
4 end;;
5
6 let ( + ) {N : Num} = N.( + );;
7
8 implicit module Float_Float = struct
9   type t = float and u = float and res = float
10  let ( + ) = ( +. )
11 end;;
12 implicit module Int_Float = struct
13   type t = int and u = float and res = float
14   let ( + ) l r = (float_of_int l) +. r
15 end;;
16
17 (* Int_Int и Float_Int пропущены для краткости *)
18
19 print_float (1 + 1.1 + 2.5);; (* неоднозначность! *)
```

# Порядок разрешения. Постановка задачи

- Каждая неявная переменная характеризуется своим типом, зависящим от нуля или нескольких типовых переменных:  $T_i(x_{a_{i,1}}, \dots, x_{a_{i,n_i}})$
- Типовые переменные  $x_k$  могут повторяться для разных  $T_i$
- После поиска модуля по  $T_i$  могут быть определены все или некоторые из  $x_{a_{i,1}}, \dots, x_{a_{i,n_i}}$
- Если поиск модуля вернул "нет кандидатов", то нужно завершить алгоритм. В случаях же "неоднозначность" или "не завершается" нужно подождать новой информации
- Для  $N$  неявных переменных в худшем случае потребуется  $\mathcal{O}(N^2)$  запусков поиска

# Порядок разрешения. Алгоритм

- 1 На каждом шаге, если нашлось такое  $T_i$ , что все  $x_{a_{i,j}}$ , от которых оно зависит, уникальны — обработать его
- 2 Если таких не нашлось, обработать любой  $T_i$ , который ни разу не был обработан
- 3 Если и таких не нашлось, обработать  $T_i$ , с момента последней неудачной обработки которого хотя бы одна из  $x_{a_{i,j}}$  была определена

Сложность:  $\mathcal{O}(N + K)$  запусков поиска, где  $K = \sum_i n_i$

Эвристики:

- В (2) обработать сначала  $T_i$  с меньшим числом переменных
- В (3) обработать сначала  $T_i$ , про который стало известно больше новой информации

# Неявные функторы

Допустим, есть `Show_list1` и `Show_list2` с одинаковой сигнатурой, нужно выбрать из них явно







```
1 show {Show_list1 (Show_pair (Show_int Show_bool))}
2   [(1, true); (2, false)]
3
4 (* Слишком длинно. В OCaml можно только так *)
5
6 show {Show_list1} [(1, true); (2, false)]
7
8 (* В IML можно поддержать такое *)
```

Это достигнуто за счёт того, что в IML различие между функциями и функторами существенно меньше



- Нет существующей кодовой базы ни на OCaml, ни на 1ML
- Нужно собрать собственный набор тестов
- Источники:
  - Набор тестов для решения на OCaml и примеры из статей
  - Тесты на порядок разрешения и на неявные функторы
  - Тесты, аналогичные коду на других языках с неявными параметрами (Scala)
  - Реализация стандартных функциональных структур (например, монады)
  - Тесты, написанные в логической парадигме

- Реализованы неявные модули как расширение компилятора языка 1ML
- Решение работает на тестах, на которых работает решение для OCaml
- Также работает на тестах, которые в OCaml не поддерживаются
  - Порядок разрешения
  - Неявные функторы
- Тестовый набор будет дополняться

-  Dreyer, Derek и др. “Modular Type Classes”. B: 2007. ISBN: 1595935754.
-  MacQueen, David B. “Using Dependent Types to Express Modular Structure”. B: 1986. ISBN: 9781450373470.
-  Oliveira, Bruno C.d.S., Adriaan Moors и Martin Odersky. “Type Classes as Objects and Implicits”. B: 2010. ISBN: 9781450302036.
-  Rossberg, Andreas. “1ML – Core and Modules United (F-ing First-Class Modules)”. B: 2015. ISBN: 9781450336697.
-  Rossberg, Andreas, Claudio V. Russo и Derek Dreyer. “F-Ing Modules”. B: 2010. ISBN: 9781605588919.
-  Wadler, P. и S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. B: 1989. ISBN: 0897912942.



White, Leo, Frédéric Bour и Jeremy Yallop. “Modular  
implicits”. B: (2015). ISSN: 2075-2180.