Переоснащение неявных модулей для языка 1ML

Трилис Алексей Андреевич

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

НИУ ВШЭ — Санкт-Петербург

13 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[1]
- 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`, ...

[1] 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

Введение. 1ML

- Было показано[2], что модули могут быть выражены и без использования теории зависимых типов
- А именно, что модули можно полностью выразить в System F_ω
- В результате этих исследований был создан экспериментальный диалект 1ML[3], где основной и модульный слои объединены
- В нём действительно есть модули первого класса

[2] Rossberg, Russo и Dreyer, “F-Ing Modules”, 2010.

[3] Rossberg, “1ML – Core and Modules United (F-ing First-Class Modules)”, 2015.

Обзор литературы. Классы типов

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

instance Show Int where
  show = showSignedInt

show_twice x = show x ++ show x

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

- Впервые в Haskell[4], затем в Agda, Rust, ...
- Требуется каноничность — не более одного экземпляра для каждого типа

[4] Wadler и Blott, “How to Make Ad-Hoc Polymorphism Less Ad Hoc”, 1989.

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)
```

Впервые в Scala[5]

[5] Oliveira, Moors и Odersky, “Type Classes as Objects and Implicits”, 2010.

Обзор литературы. Модульные классы типов

- Попытка применить классы типов в ML[6]
- 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

[6] Dreyer и др., “Modular Type Classes”, 2007.

Обзор литературы. Неявные модули

- Попытка применить неявные параметры в OCaml[7], 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

[7] 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] *)
```

Мотивация

- 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 и задачи

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

- Обработка неявных модулей тесно связана с выводом типов, одно нельзя сделать до второго. Поэтому их нужно делать в одной фазе компиляции
- Обрабатывая 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

Поиск модулей

- 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

Генерализация типов

- 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

Порядок разрешения. Мотивация

- 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: $\mathrm{T_i}(\mathrm{x_{a_{i,1}}}, \ldots, \mathrm{x_{a_{i,n_i}}})$
- Типовые переменные $\mathrm{x_k}$ могут повторяться для разных $\mathrm{T_i}$
- После поиска модуля по $\mathrm{T_i}$ могут быть определены все или некоторые из $\mathrm{x_{a_{i,1}}}, \ldots, \mathrm{x_{a_{i,n_i}}}$
- 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 $\mathcal{O}(\mathrm{N}^2)$ запусков поиска

Порядок разрешения. Алгоритм

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

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

Эвристики:

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

Неявные функторы

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

```
show {Show_list1 (Show_pair (Show_int Show_bool)}
  [(1, true); (2, false)]

(* Слишком длинно. В OCaml можно только так *)

show {Show_list1} [(1, true); (2, false)]

(* В 1ML можно поддержать такое *)
```

Это достигнуто за счёт того, что в 1ML различие между функциями и функторами существенно меньше

Тестирование и сравнение

- 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

Результаты

- 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

Ссылки I

Dreyer, Derek и др. “Modular Type Classes”. В: Proceedings of the 34th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’07. Nice, France: Association for Computing Machinery, 2007, с. 63—70. ISBN: 1595935754.

MacQueen, David B. “Using Dependent Types to Express Modular Structure”. В: Proceedings of the 13th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. POPL ’86. St. Petersburg Beach, Florida: Association for Computing Machinery, 1986, с. 277—286. ISBN: 9781450373470.

Ссылки II

Oliveira, Bruno C.d.S., Adriaan Moors и Martin Odersky. “Type Classes as Objects and Implicits”. B: Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications. OOPSLA ’10. Reno/Tahoe, Nevada, USA: Association for Computing Machinery, 2010, c. 341—360. ISBN: 9781450302036.

Rossberg, Andreas. “1ML – Core and Modules United (F-ing First-Class Modules)”. B: Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming. ICFP 2015. Vancouver, BC, Canada: Association for Computing Machinery, 2015, c. 35—47. ISBN: 9781450336697.

Ссылки III

Rossberg, Andreas, Claudio V. Russo и Derek Dreyer. “F-Ing Modules”. B: Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation. TLDI ’10. Madrid, Spain: Association for Computing Machinery, 2010, с. 89—102. ISBN: 9781605588919.

Wadler, P. и S. Blott. “How to Make Ad-Hoc Polymorphism Less Ad Hoc”. B: Proceedings of the 16th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. POPL ’89. Austin, Texas, USA: Association for Computing Machinery, 1989, с. 60—76. ISBN: 0897912942.

Ссылки IV

White, Leo, Frédéric Bour и Jeremy Yallop. “Modular implicits”. B: Electronic Proceedings in Theoretical Computer Science 198 (дек. 2015), с. 22—63. ISSN: 2075-2180.