Переоснащение неявных модулей для языка 1ML

Трилис Алексей Андреевич

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

НИУ ВШЭ — Санкт-Петербург

11 июня 2021 г.

Введение

- Ad-hoc-полиморфизм — свойство языка, позволяющее функциям иметь различную семантику в зависимости от типов аргументов
- Часто достигается перегрузкой, но в языках с мощным выводом типов нужны более сложные методы
- В семействе языков ML ad-hoc-полиморфизм до сих пор не поддерживается
- `+` для `int`, `+.` для `float`
  `print_int`, `print_float`, `print_string`, ...

Обзор литературы

- Согласованность — каждая типизация программы должна приводить к одной и той же семантике
- Каноничность — в области видимости не более одного экземпляра для каждого типа
- Решения с каноничностью
  - Haskell, Rust
  - В ML невозможно проверить каноничность
  - Есть решение для ML[1], но оно вводит серьёзные ограничения
- Решения без каноничности
  - Scala
  - Решение для OCaml[2] в основной язык интегрировать пока не получилось
  - Система проверки типов недостаточно сильна

[1] Dreyer и др., «Modular Type Classes», 2007.

[2] White, Bour и Yallop, «Modular implicits», 2015.

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] *)
```

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] *)
```

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] *)
```

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] *)
```

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] *)
```

1ML

- В ML исторически язык разделён на основной язык и более мощный и избыточный модульный язык
- Интеграция этих слоёв затруднена
- Экспериментальный диалект 1ML[3] решает эту проблему: в нём нет существенного различия между основным языком и модулями
- Предположительно, такой подход позволит добавить новую функциональность, например, ad-hoc-полиморфизм

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

Цель и задачи

Цель: Разработать поддержку неявных модулей для языка 1ML, которое будет более полным, чем аналогичное решение для OCaml, в том числе включать поддержку неявных аргументов для функторов

Задачи:

- Реализация неявных модулей для 1ML, повторяющих функциональность решения для OCaml
- Разработка алгоритма, позволяющего полно и эффективно осуществлять вставку неявных модулей
- Поддержка неявных аргументов для функторов
- Сравнение с решением для OCaml

Общая схема

- На месте, где должен стоять модуль, подставим неявную переменную
- Пока не знаем значение, но знаем тип
- Отложим определение этого модуля, будем обрабатывать несколько неявных переменных за раз
- 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

Детали решения

- Проверка завершаемости
  - В рекурсии проверяем, что хотя бы одно ограничение уменьшилось, а остальные не стали больше
- В какой момент разрешаем накопившиеся переменные?
  - Дойдя до объявления верхнего уровня
  - Но в теории можно в любой момент
- Локальные неявные модули
  - В момент разрешения модули могут выйти из контекста
  - Храним дерево из неявных модулей и побочной информации

Порядок разрешения. Мотивация

- В решении для OCaml неявные переменные разрешаются в некотором фиксированном порядке
- Это уменьшает полноту решения
- В этой работе найдено, что решение для OCaml не работает в простых случаях: невозможно реализовать сложение переменных типа `int` и `float`
- Будем запускать разрешение неявных переменных несколько раз, с появлением новой информации

Порядок разрешения. Алгоритм

- Обрабатываем неявные переменные в следующем порядке:
  1. Независимые от других ещё не разрешённых
  2. Не обработанные раннее
  3. Те, с последней обработки которых была получена новая информация
- Несколько эвристик
- В случаях, которые были поддержаны раньше, работает столь же эффективно

Неявные аргументы для функторов

Есть Show_list1 и Show_list2, 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 1ML можно поддержать такое *)
```

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

Сравнение

- Тест — две аналогичные программы на OCaml и на 1ML
- Проверяем, корректна ли проверка типов для теста в каждом из языков
- Сейчас около 30 тестов

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**
    Идея алгоритма может быть использована в OCaml
  - **Неявные аргументы для функторов**
    Результат достигнут из-за особенностей 1ML
- Работа представляет дополнительный аргумент в пользу диалектов с однородным подходом к модулям

Репозиторий: `github.com/trilis/1ml`

Детали алгоритма. Примеры

Пример модуля, в присутствии которого поиск не будет завершаться:

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

Пример локального неявного модуля:

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

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. Эвристики

1. Независимые от других ещё не разрешённых
2. Не обработанные раннее
3. Те, с последней обработки которых была получена новая информация

- В (2) обработать сначала неявные переменные с меньшим числом типовых переменных
- В (3) обработать сначала те, про которых стало известно больше новой информации

Невозможность каноничности в OCaml

```
1  module F (X : Show) = struct
2    implicit module S = X
3  end
4
5  implicit module Show_int = struct
6    type t = int
7    let show = string_of_int
8  end
9
10 module M = struct
11   type t = int
12   let show _ = "An int"
13 end
14
15 module N = F(M)
```

Ограничения на модульные классы типов

- 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