Санкт-Петербургский государственный университет

***Косарев Дмитрий Сергеевич***

**Выпускная квалификационная работа**

# Обобщённое программирование с комбинаторами и объектами

Уровень образования:

Направление *09.06.01*

Основная образовательная программа

*“Информатика и вычислительная техника”*

Научный руководитель:
доктор физико-математических наук, профессор Терехов Андрей Николаевич

Рецензент:
доктор технических наук Новиков Федор Александрович

Санкт-Петербург
2020

# Оглавление

# 1. Введение

Фредерк Брукс (Frederic Brooks) в своей известной книге по инженерии программ "Мифический человеко-месяц" ("The Mythical Man-Month") [5] охарактеризовал сущность программирования следующим образом:

> "Программист, подобно поэту, работает почти непосредственно с чистой мыслью. Он строит свои замки в воздухе и из воздуха, творя силой воображения. Трудно найти другой материал, используемый в творчестве, который столь же гибок, прост для шлифовки или переработки и доступен для воплощения грандиозных замыслов. (Как мы позднее увидим, такая податливость таит свои проблемы.)"

Действительно, нематериальность программ и гибкость их представления призывает к структурированию; отсутствие подходящей структуры легко может привести к катастрофическим последствиям (как это случалось с некоторыми промышленными проектами в прошлом). Одним из наиболее распространенных способов структурировать программы является использование *типов данных*. Они позволяют описывать свойства данных; что можно с ними сделать, а что нельзя; а также в некоторой степени описывают семантику структур данных. Если информация о типах данных присутствует во время работы программы, то становится возможным реализовать мета-преобразования путём анализа типов (*интроспекция*) или путём создания новых типов данных на лету (*рефлексия*).

Однако, в статически типизированных языках, как правило, типы полностью стираются после фазы компиляции и отсутствуют во время исполнения. Статическая типизация обладает серьёзным преимуществом по сравнению с динамической, потому что программам не нужно инспектировать типы во время выполнения и больше количество плохих поведений программ – ошибок типизации – не случается. С другой стороны, некоторые преобразования, которые в динамических языках могли быть реализованы "раз и навсегда" не проходят проверку типов

и должны быть перереализованы для каждого конкретного типа по отдельности. Одни из подходов к преодолению этого недостатка является разработка более выразительной системы типов, где большее количество функций может быть протипизировано. Примером этому подходу будет поддержка перегрузки (*ad hoc* полиморфизма) в языке Haskell в виде классов типов [28] и семейств типов [14]. Однако, по причине требования тотальности к алгоритму проверки типов и фундаментальной неразрешимости проблемы проверки типов, всегда будут существовать "хорошие" программы, которые не могут быть протипизированы. Другим подходом является *обобщенное программирование* [11] (*datatype-generic programming*), целью которого является разработка методов для реализации практически важных семейств функций индексированных типами, использую имеющиеся возможности языка. Например, типы могут быть закодированы на внутреннем языке [1, 6, 12], либо часть информации о типах может быть сделана доступной во время исполнения, или обобщенные функции для конкретного типа данных могут быть сгенерированы во время компиляции автоматически [30, 33]. Два похода, описанные нами, дополняют друг друга: чем более мощной является система типов, тем больше возможностей для обобщенного программирования язык может предложить. Например, параметрический полиморфизм позволяет естественно выразить функцию для вычисления длины списка произвольных элементов и т.п.

Мы представляем библиотеку для обобщенного программирования GT[1] (*Generic Transformers*), которая находится в активной разработке с 2014 года. Одним из важных наблюдений, которые спровоцировали разработку, является то, что многие обобщенные функции можно рассматривать как модификации некоторых других обобщенных функций. Наш подход, являясь генеративным (мы создаем функции на основе объявлений типов), также позволяет конечным пользователям легко получать новые преобразования видоизменяя некоторые части уже имеющихся. Это достигается путём кодирования конструкторов один к одному в методы классов, что несколько напоминает подход, называе-

[1] `https://github.com/kakadu/GT/tree/ppx`

мый алгебрами объектов [21].

Отличительным особенностями нашего подхода являются:

- каждое преобразование выражается с помощью *функции преобразования* и *объекта преобразования*, которые содержат в себе "интересные" части преобразования;
- функция преобразования уникально для данного типа и всех объектов преобразования, которые являются образцами уникального класса;
- и функция преобразования, и класс генерируются из объявления типа; мы поддерживаем регулярные алгебраические типы данных, структуры, полиморфные вариантные типы и базовые заранее описанные типы;
- мы предоставляем несколько плагинов, для того, чтобы генерировать практически полезные преобразования в виде конкретных классов;
- система плагинов расширяема – программист может реализовать свои собственные плагины.

Представленная в данной работе библиотека является логическим продолжением более ранней работы [4] на тему реализации подхода "Scrap Your Boilerplate" [17, 18, 19]. Однако, опыт показал, что выразительность и расширяемость SYB недостаточна, к тому же преобразования, которые зависят только от типа, не очень удобно использовать. Изначальной идеей данной работы было совмещение комбинаторов и объектно-ориентированного подхода: первый позволит делать реализовать параметризацию удобным образом, а второй предоставит расширяемость за счет позднего связывания (late binding). Идея в виде конкретного шаблона проектирования была успешно апробирована [2], а затем реализована в виде библиотеки и синтаксического расширения [3]. Последующий опыт, связанный с разработкой библиотеки [16], снова указал на некоторые недостатки в реализации. В данной работе

представляется почти полностью переписанная реализация, где найденные недостатки были исправлены.

Оставшаяся часть работы организована следующим образом. В следующем разделе 2 мы неформально опишем наш подход с помощью примеров. Затем 3 опишем реализацию в деталях, подчеркнув аспекты, которые считаем важными или интересными. Далее представим несколько примеров, реализованных 4 с помощью нашей библиотеки. В разделе 5 обсудим аналогичные подходы и библиотеки и сравнимся с ними. В последнем разделе 6 укажем направления для дальнейшего развития.

# 2. Неформальное описание

В этом разделе мы постепенно представим наш подход используя несколько примеров. Хотя изложение не предоставляет конкретных деталей и не может использоваться как точная спецификация, мы здесь предоставляем основные составляющие решения и мотивацию, которая привела к ним. Далее мы будет использовать следующее соглашение: будем обозначать $\langle \ldots \rangle$ представление некоторого понятия в конкретном синтаксисе языка OCaml. Например, "$\langle f_t \rangle$" является обозначением конкретной функции индексированной типом "$f$" для типа "$t$". В конкретном синтаксисе оно может быть выражено как "`f_t`", но мы пока воздержимся от указания конкретной формы.

Начнем с простого примера. Рассмотрим такое объявление типа арифметических выражений:

```
type expr =
| Const of int
| Var   of string
| Binop of string * expr * expr
```

Рекурсивная функция "$\langle show_{expr} \rangle$" (наиболее естественный кандидат на реализацию) преобразует выражение в строку:

```
let rec ⟨show_expr⟩ = function
| Const  n          → "Const " ^ string_of_int n
| Var    x          → "Var " ^ x
| Binop (op, l, r)  →
  Printf.sprintf "Binop (%S, %s, %s)"
                 op (⟨show_expr⟩ l) (⟨show_expr⟩ r)
```

Представление, возвращаемое "$\langle show_{expr} \rangle$", сохраняет имена конструкторов. Оно может быть полезно при отладке или сериализации. Однако, как правило, также требуется иное, "красивое"(*pretty-printed*) представление. В этом представлении выражение представляется в "естественном синтаксисе" с использованием инфиксных операций и без имён конструкторов, где скобки расставлены только там, где они действительно

нужны. Мы можем реализовать это преобразование очень просто:

```
let ⟨pretty_expr⟩ e =
  let rec pretty_prio p = function
  | Const  n          → string_of_int n
  | Var    x          → x
  | Binop (o, l, r)   →
    let po = prio o in
    (if po ≤ p then br else id) @@
    pretty_prio po l ^ " " ^ o ^ " " ^ pretty_prio po r
  in
  pretty_prio min_int e
```

Здесь мы пользуемся функциями "`prio`", "`br`" и "`id`", доступными из вне. Функция "`prio`" возвращает приоритет бинарной операции, "`br`" окружает свой аргумент скобками, а "`id`" — тождественная функция. Дополнительная функция "`pretty_prio`" принимает числовой параметр, который обозначает приоритет окружающей операции (если такая имеется). Если приоритет текущей операции меньше или равен переданному, тогда выражение окружается скобками. Для простоты мы считаем, что все операции неассоциативны, но такой же шаблон кода может быть использован для поддержки ассоциативных операций. На верхнем уровне мы передаем наименьшее возможное число как приоритет, чтобы убедиться, что мы не получим скобок, окружающих выражение целиком

Реализации этих двух функций имеют очень мало общего. Обе возращают строки, но вторая принимает дополнительный аргумент, и правые части сопоставления с образцом для соответствующих конструкторов различаются. Единственной общей частью является сопоставление с образцом само по себе. Мы может извлечь его в отдельную функцию и параметризовать эту функцию множеством трансформаций, соответствующих конструкторам:

```
let ⟨gcata_expr⟩ ω ι = function
| Const n           → ω#⟨Const⟩ ι n
```

```
| Var   x          → ω#⟨Var⟩ ι x
| Binop (o, l, r)  → ω#⟨Binop⟩ ι o l r
```

Здесь мы представляем множество семантически связанных функций объектом. "$\omega$" – это объект, где методы соответствуют конструктором один к одному. "$\iota$" представляет дополнительный параметр, который может использоваться функциями как, например, "$\langle pretty_{expr} \rangle$" (и игнорироваться функциями на подобие "$\langle show_{expr} \rangle$").

Упомянутая в начале функция "$\langle show_{expr} \rangle$" может быть выражена следующим образом[2]:

```
let rec ⟨show_expr⟩ e = ⟨gcata_expr⟩
  object
    method ⟨Const⟩ _ n = "Const " ^ string_of_int n
    method ⟨Var⟩   _ x = "Var " ^ x
    method ⟨Binop⟩ _ o l r =
      Printf.sprintf "Binop (%S, %s, %s)"
                     o (⟨show_expr⟩ l) (⟨show_expr⟩ r)
  end
  ()
  e
```

И, разумеется, всё то же самое применимо к функции $\langle pretty_{expr} \rangle$.

Вы могли заметить, что оба объекта, необходимые для реализации этих функций, могут быть созданы с помощью общего виртуального класса:

```
class virtual [ι, σ] ⟨expr⟩ =
object
  method virtual ⟨Const⟩ : ι → int → σ
  method virtual ⟨Var⟩   : ι → string → σ
  method virtual ⟨Binop⟩ : ι → string → expr → expr → σ
end
```

[2]Для ясности понимания мы опустили некоторые аннотации типов, которые помогают этому листингу кода пройти проверку типов.

Конкретный класс, представляющий преобразование будет наследоваться от этого общего предка. Чтобы иметь возможность вызывать рекурсивно данное преобразование, мы параметризуем класс функцией самотрансформации "`fself`" (*открытая рекурсия*). Написание в стиле открытой рекурсии необходимо для возможности поддержки полиморфных вариантных типов и рекурсивных определений. Теперь мы сможем реализовать логику распечатки в формат, удобный человеку, в изоляции, отдельно от функции "красивой" распечатки (обратите внимание на использование "`fself`"):

```
class ⟨pretty_expr⟩ (fself : ι → expr → σ) = object
  inherit [int, string] ⟨expr⟩
  method ⟨Const⟩ p n = string_of_int n
  method ⟨Var⟩ p x = x
  method ⟨Binop⟩ p o l r =
    let po = prio o in
    (if po ≤ p then fun s → "(" ^ s ^ ")" else fun s → s) @@
    fself po l ^ " " ^ o ^ " " ^ fself po r
end
```

Функция распечатки в удобный человеку формат может быть легко описана с использованием класса выше и функции обобщенной трансформации[3]:

```
let ⟨pretty_expr⟩ e =
  let rec pretty_prio p e =
    ⟨gcata_expr⟩ (new ⟨pretty_expr⟩ pretty_prio) p e
  in
  pretty_prio min_int e
```

Также мы может избежать объявления вложенной функции с помощью комбинатора неподвижной точки "`fix`":

```
let ⟨pretty_expr⟩ e =
  fix (fun fself p e → ⟨gcata_expr⟩ (new ⟨pretty_expr⟩ fself) p e)
```

[3]Так как имена функции и классов находятся в разных пространствах имен в OCaml, мы может использовать одно и то же имя для класса и функции трансформации.

```
    min_int e
```

Выше мы смогли выделить две общие части для двух существенно различных преобразований: функцию обобщенного обхода ("$\langle gcata_{expr} \rangle$") и такой виртуальный класс ("$\langle expr \rangle$"), что все трансформации можно представить как его экземпляры. Но стоило ли это того? В действительности, в этом примере мы добились не очень большого переиспользования кода путём добавления большого количества абстракций. Итоговый код получился по размеру даже больше исходного.

Мы утверждаем, что преобразования в данном конкретном случае были недостаточно обобщенные. Чтобы оправдать описанный подход, давайте рассмотрим более оптимистичный сценарий. Широко известно, что многие трансформации могут быть представлены (по понятным причинам) как *катаморфизмы*, т.е. как "свёртки" [10, 13, 20]. Формально, чтобы определить канонический катаморфизм нам нужно абстрагировать тип "`expr`" от самого себя и описать функтор, но здесь мы воспользуемся более легковесным решением:

```
class [ι] ⟨fold_expr⟩ (fself : ι → expr → ι) = object
  inherit [ι, ι] ⟨expr⟩
  method ⟨Const⟩ i n = i
  method ⟨Var⟩ i x = i
  method ⟨Binop⟩ i o l r = fself (fself i l) r
end
```

Эта реализация просто передает "`i`" сквозь все узлы трансформируемого значения, что выглядит довольно бесполезно на первый взгляд. Однако, слегка изменив поведение, можно получить кое-что полезное:

```
let fv e =
  fix (fun fself i e →
         ⟨gcata_expr⟩ (object inherit [string list] ⟨fold_expr⟩ fself
                         method ⟨Var⟩ i x = x :: i
                       end) i e
      ) [] e
```

Эта функция создает список всех свободных переменных в выражении, а так как в языке выражений нет способа связывать переменные, то это просто список всех переменных в терме. Объект, который мы сконструировали, наследуется от "бесполезного" класса "$\langle fold_{expr} \rangle$" и переопределяет только один метод – метод для обработки переменных. Весь остальной код уже работает так, как нам нужно — "$\langle gcata_{expr} \rangle$" обходит выражение, а остальные метода объекта аккуратно передают построенный список дальше. Таким образом, мы смогли реализовать интересное преобразование с помощью очень малой модификации существующего кода, предоставленного уже имеющимся классом "$\langle fold_{expr} \rangle$". Чтобы избежать впечатления, что мы аккуратно подготавливались к представлению именно этого примера, мы покажем ещё один:

```
let height e =
  fix (fun fself i e →
         ⟨gcata_expr⟩
           (object
             inherit [int] ⟨fold_expr⟩ fself
             method ⟨Binop⟩ i _ l r = 1 + max (fself i l) (fself i r)
           end)
           i
           e
      ) 0 e
```

Здесь мы вычисляем высоту дерева выражения, используя тот же самый класс "$\langle fold_{expr} \rangle$" как базовый для другого самостоятельно объекта, переопределяем метод для бинарного оператора, который теперь будет вычислять высоты поддеревьев, выбирать из них максимальную высоту и прибавлять единицу.

Одной из других всеми известных обобщенных функций является "map":

```
class ⟨map_expr⟩ fself = object
  inherit [unit, expr] ⟨expr⟩
  method ⟨Var⟩ _ x = Var x
```

```
  method ⟨Const⟩ _ n = Const n
  method ⟨Binop⟩ _ o l r = Binop (o, fself () l, fself () r)
end
```

Опять, так как нам не известно, что "expr" – это функтор, то всё, что мы можем сделать в функции "$\langle map_{expr} \rangle$" — это копированиею. Однако, отнаследовавшись от этого класса, мы может получить новый вид преобразований:

```
class simplify fself = object
  inherit ⟨map_expr⟩ fself
  method ⟨Binop⟩ _ o l r =
    match fself () l, fself () r with
    | Const l, Const r → Const ((op o) l r)
    | l      , r       → Binop (o, l, r)
end
```

Данный класс проводит упрощение выражения: если оба аргумента бинарной операции вычисляются в константу той же самой трансформацией, тогда мы может произвести операцию сразу. Функция "op" объявлена где-то ещё, она возвращает функцию, которая будет производить вычисление данного бинарного оператора.

Вот ещё один пример:

```
class substitute fself state = object
  inherit ⟨map_expr⟩ fself
  method ⟨Var⟩ _ x = Const (state x)
end
```

Здесь мы выполняем подстановку переменных в выражении на значения, определенные в некотором состоянии, представленном функций "state". Два класса, объявленных выше могут быть скомбинированы для получения интерпретатора выражений:

```
class eval fself state =  object
  inherit substitute fself state
  inherit simplify   fself
end
```

```
let eval state e =
  fix (fun fself i e → ⟨gcata_expr⟩ (new eval fself state) i e) () e
```

Во всех примерах мы выбрали достаточно стандартные преобразования и можно сказать, что реализовали всё достаточно малыми усилиями, если закрыть глаза на несколько многословный синтаксис классов и объектов в OCaml. В каждом случае было необходимо переопределить только один метод и воспользоваться функцией, однозначно получаемой по типу. С другой стороны, мы работали с очень просто устроенным типом, он даже не был полиморфным, а поддержка полиморфизма может привести к специфически проблемам. В оставшейся части статьи мы покажем, что идеи, представленные выше, может быть расширены до подхода к обобщенному программированию, где все компоненты могут быть синтезированы из объявления типа. В частности, наш подход предоставляет полную поддержку:

- полиморфизма;
- применения типовых операторов (type operators);
- взаимной рекурсии, где поддержка воистину *расширяемых* преобразований потребует некоторых усилий;
- полиморфных вариантных типов, с которыми будет необходимо позаботиться о гладкой интеграции возможностей полиморфных вариантом и наследования классов;
- раздельной компиляции — мы можем сгенерировать код по определениям типов не заглядывая внутрь модулей, от которых зависит обрабатываемый тип;
- инкапсуляции, а именно поддержки сигнатур модулей, включая абстрактные типы и приватные определения. Обобщенные функции для абстрактных типов могут использоваться из вне модуля, но не позволят инспектировать или изменять содержимое абстрактного типа.

Что касается вопросов производительности, то как Вы могли заметить, во всех примерах мы создавали большое количество *идентичных* объектов во время преобразования (под одному на каждый узел структуры данных). Далее мы увидим, что с этим можно побороться с помощью мемоизации. Наконец, наш подход предоставляет система плагинов, которые могут быть использованы для генерации большого количества преобразований, как, например, "`show`", "`fold`" или "`map`". Система плагинов расширяема, т.е. пользователи могут реализовать их собственные плагины с помощью небольших усилий, так большая часть функциональности по обходу структуры данных предоставляется библиотекой.

# 3. Реализация

Основными компонентами нашего решения являются синтаксические расширения (и для `camlp5` [32], и для `ppxlib` [33]), библиотека времени исполнения и система плагинов. Синтаксическое расширение касается объявлений типов, аннотированных пользователем, и генерирует следующие сущности:

- обобщенная функция трансформации (одна на каждый тип);
- виртуальный класс, который используется как общий предок для всех трансформаций (один на каждый тип);
- некоторое количество конкретных классов (по одному на каждый вид плагина);
- структуру данных *typeinfo*, которая содержит в себе информацию, специфичную для данного типа, а именно обобщенную функцию трансформации и набор функций трансформации, которые порождаются плагинам. Всё представлено как самостоятельный объект.

Мы поддерживаем большинство вариантов в правой части объявлений типов со следующими ограничениями:

- поддерживаются только регулярные алгебраические типы данных; GADT'ы обрабатываются как обычные алгебраические типы;
- ограничения на типы (constraints) не учитываются;
- объекты, модули и типы с ключевым словом "**`nonrec`**" не поддерживаются;
- расширяемые типы данных (" ... "/"+=") не поддерживаются.

| С использованием `camlp5` | |
|---|---|
| **@type** ... = ...<br>**and** ... = ...<br>[ **with** $p_1, p_2, \ldots$ ]<br>@$typ$<br>@$plugin$[$typ$] | синтаксическая конструкция для обработки типа с плагинами $p_1, p_2, \ldots$; взаимно рекурсивные типы также поддерживаются;<br>название виртуального класса для типа $typ$;<br>имя класса плагина для типа $typ$ и плагина $plugin$ |
| С использованием `ppxlib` | |
| **type** ... = ...<br>**and** ... = ...<br>[@@deriving gt<br>~options:{ $p_1, p_2, \ldots$}] | синтаксическая конструкция для обработки типа с плагинами $p_1, p_2, \ldots$ |

Рис. 1: Конструкции расширенного синтаксиса

К примеру, если к типу "`t`" применить плагин "`show`", то в файле реализации сгенерируются следующие сущности (с помощью "..." мы обозначаем части, объяснение которых пока опускаем):

**let** $\langle gcata_t \rangle$ ... = ...

**class virtual** [...] $\langle t \rangle$ = **object**
  ...
**end**

**class** [...] $\langle show_t \rangle$ ... = **object**
  **inherit** [...] $\langle t \rangle$ ...
  ...
**end**

**let** t = {
  gcata = $\langle gcata_t \rangle$;
  ...
  plugins = **object**
    **method** show = ...
  **end**
}

С помощью структуры "`t`" с информацией о типе мы можем симитировать функции трансформаций, индексированные типами :

```
let transform typeinfo = typeinfo.gcata
let show      typeinfo = typeinfo.plugins#show
```

Функция "`transform`(`t`)" – это функция верхнего уровня из библиотеки, которая может быть инстанциирована для любого поддерживаемого типа "`t`". На рисунке 1 мы описываем конкретные синтаксические конструкции, реализованные как синтаксическое расширение. Обратите внимание, что конкретное представление имен для классов и функций трансформации (представленных выше как$\langle ... \rangle$) является несущественными пока используется `camlp5`, так как предоставляется соответствующее синтаксическое расширение.

## 3.1. Типы трансформаций

Дизайн библиотеки основан на идеи описания катаморфизмов [20] с помощью атрибутных грамматик [15, 23, 26]. Вкратце, мы рассматриваем только трансформации следующего типа

$$\iota \to t \to \sigma$$

где $t$ – это тип, значения которого мы преобразуем, $\iota$ и $\sigma$ — типы *наследуемых* и *синтезируемых* атрибутов. Мы не будем использовать атрибутные грамматики, чтобы описывать алгоритмическую часть трансформаций, мы только переиспользуем терминологию для описания типов типов.

Если рассматриваемый тип является параметрическим, то преобразование тоже будет параметрическим. Далее мы будет обозначать с помощью $\{...\}$ множественное вхождение сущности в скобках. С помощью такой нотации мы сможем описать обобщенную форму преобразований, представимых с помощью нашей библиотеки, как

$$\{\iota_i \to \alpha_i \to \sigma_i\} \to \iota \to \{\alpha_i\}\ t \to \sigma$$

Здесь $\iota_i \to \alpha_i \to \sigma_i$ является функцией-преобразованием для типового параметра $\alpha_i$. В общем, функции-трансформации структуры данных действуют на наследуемые атрибуты и конкретные значения и возвращают синтезируемые атрибуты для различных типов. Общий для всех преобразований класс-предок для $n$-параметрического типа имеет $3(n+1)$ типовых параметров:

- тройка $\iota_i$, $\alpha_i$, $\sigma_i$ для каждого типового параметра $\alpha_i$, где $\iota_i$ и $\sigma_i$ — это типовые переменные для наследуемого и синтезированного атрибутов для преобразования $\alpha_i$;
- пара дополнительных типовых переменных $\iota$ и $\sigma$ для представления наследуемого и синтезированного атрибутом трансформируемого типа;
- дополнительная типовая переменная $\varepsilon$, которая приравнивается к "$\{\alpha_i\}$ `t`" для типов отличных от полиморфных вариантные, и приравнивается к *открытому* типу "$[> \{\alpha_i\}$ `t`$]$" для полиморфных вариантных типов (подробнее в разделе 3.5).

Например, если нам дан двупараметрический тип $(\alpha, \beta)$ `t`, то заголовком общего класса-предка будет

**`class virtual`** $[\iota_\alpha, \alpha, \sigma_\alpha, \iota_\beta, \beta, \sigma_\beta, \iota, \varepsilon, \sigma]$ $\langle t \rangle$

Конкретные преобразования будут наследоваться от этого класса и, возможно, конкретизировать некоторые из типовых параметров. Дополнительно конкретные классы получают несколько аргументов-функций:

- $n$ функций, преобразующих типовые параметры: $\mathtt{f}_{\alpha_i} : \iota_i \to \alpha_i \to \sigma_i$;
- функция для реализации открытой рекурсии: `fself` $: \iota \to \varepsilon \to \sigma$.

Например, для типа, упомянутого выше и преобразования "`show`" заголовок конкретного класс будет выглядеть как

**`class`** $[\alpha, \beta, \varepsilon]$ $\langle show_t \rangle$
  $(\mathtt{f}_\alpha$ `: unit` $\to \alpha \to$ `string`$)$

```
  (f_β : unit → β → string)
  (fself : unit → ε → string) =
object
  inherit [unit, α, string, unit, β, string, unit, ε, string] ⟨t⟩
  ...
end
```

Обратите внимание, что мы поддерживаем это соглашения для всех типов, хотя для некоторых типов некоторые компоненты могут быть излишни, например, "`fself`" нужен только для рекурсивных типов. Объяснение этому простое: если мы *используем* некоторый тип то мы в общем случае не знаем его определения. Следовательно, для поддержки раздельной компиляции интерфейсы всех сущностей должны иметь общую структуру.

Эта схема типизации выглядит очень многословной и неочевидной. Присутствует большое количество типовых параметров в которых легко запутаться. Однако, пользователям понадобится разбираться с ними только если они будут реализовывать преобразование *вручную* с нуля путём наследования от общего класса-предка. В большинстве случаем преобразование реализуется путём небольшой специализации конкретного плагина или используя систему плагинов. В первом случае многие типовые параметры будут уже специализированная (например, для "`show`" большинство типовых параметров конкретизируется в базовые типы), во втором система плагинов упрощает процесс правильной конкретизации типовых параметром (подробнее в разделе 3.3).

Нам также необходимо описать типы аргументов у методов общего класса. Метод для конструктора "`C of` $\mathtt{a}_1 * \mathtt{a}_2 * \ldots * \mathtt{a}_k$" имеет следующую сигнатуру:

```
method virtual ⟨C⟩ : ι → ε → a_1 → a_2 → ... → a_k → σ
```

Обратите внимание, метод принимает не только наследуемый атрибут и аргументы, соответствующие конструктору, но и значение, которое сейчас преобразуется.

Наконец, мы опишем тип обобщенных функций преобразования. Тип

слегка изменяется для случая полиморфных вариантных типов.

Для не типа, не являющегося полиморфным вариантным типом, с именем "$\{\alpha_i\}$ `t`" обобщенная функция трансформации имеет следующий тип:

$$\texttt{val}\ \langle gcata_t \rangle\ :\ [\{\iota_{\alpha_i},\ \alpha_i,\ \sigma_{\alpha_i}\},\ \iota,\ \{\alpha_i\}\ \texttt{t},\ \sigma]\#\langle t \rangle\ \ \to \iota\ \ \to \{\alpha_i\}\ \texttt{t}\ \ \to \sigma$$

Она принимает объект, представляющий преобразование, у которого типовые параметры, полученные путём наследования от базового класса, соответствующим образом конкретизированы , наследуемый атрибут, значение, которое будет преобразовано и возвращает синтезируемый атрибут. Дополнительный параметр "$\varepsilon$" конкретизируется в обрабатываемый тип. Для полиморфных вариантных типов дополнительный параметр конкретизируется в *открытую* версию типа ("$[>\ \{\alpha_i\}\ \texttt{t}]$"). Это позволяет применять функцию преобразования к объекту, представляющего преобразование расширенного типа с большим количество конструкторов.

## 3.2. Комбинатор неподвижной точки и мемоизация

В нашем подходе мы полагаемся на открытую рекурсию: класс, реализующий конкретное преобразование принимает функцию преобразования самого себя как параметр. Чтобы создать такую функцию необходим комбинатор неподвижной точки. В этом разделе мы рассмотрим только простой такой комбинатор, а именно для одиночного объявления типа. Во взаимно рекурсивном случае понадобится более сложная реализация (подробнее в разделе 3.4).

Мы напоминаем вам пример из раздела 2:

```
let ⟨pretty_expr⟩ e =
  fix (fun fself p e → ⟨gcata_expr⟩ (new ⟨pretty_expr⟩ fself) p e)
    min_int e
```

Здесь присутствует лямбда абстракции, тело которой вычисляется всякий раз, когда вызывается `fself`" в классе преобразования (по сути, для каждого узла в дереве трансформируемого значения). Так как все

объекты одинаковы, то их создание можно соптимизировать.

Мы мемоизируем создания объекта, представляющего преобразование, с помощью ленивых вычислений. Для этого мы абстрагируем создание объекта в функцию, которая принимает аргумент "`fself`". Реализация комбинатора неподвижной точки выглядит следующим образом:

```
let fix gcata make_obj ι x =
  let rec obj = lazy (make_obj fself)
  and fself ι x = gcata (Lazy.force obj) ι x in
  fself ι x
```

Этот комбинатор может использоваться для всех типов и не является генерируемым по типу данных. Теперь мы может немного исправить объявление функции "`transform`":

```
let transform typeinfo = fix typeinfo.gcata
```

С помощью этого определения пользователю не нужно использовать комбинатор неподвижной точки явно:

```
let ⟨show_expr⟩ e =
  transform(expr) (fun fself → new ⟨show_expr⟩ fself) () e
```

## 3.3. Система плагинов

Поведением по умолчанию для нашей библиотеки является создание обобщенной функции трансформации, обобщенного класса и структуры с информацией о типе. Они не создает никаких конкретных встроенных преобразований. Все преобразования создаются *плагинами*, а система плагинов позволяет пользователям реализовывать их собственные. Присутствует некоторое количество плагинов, поставляемых вместе с библиотекой (таблица 2), но ни один из них не обрабатывается каким-то особым образом остальной частью библиотеки.

Каждый плагин реализован как динамически загружаемый объект, и чтобы создать плагин, разработчик должен правильно воспользоваться интерфейсом, предоставляемым библиотекой. Аналогичный под-

| Название | Тип функции трансформации | Комментарий |
|---|---|---|
| `show` | $\{$ `unit` $\to \alpha_i \to$ `string` $\} \to$ `unit` $\to \{\alpha_i\}$ `t` $\to$`string` | преобразование в строку |
| `fmt` | $\{$ `formatter` $\to \alpha_i \to$ `unit` $\} \to$ `formatter` $\to \{\alpha_i\}$ `t` $\to$`unit` | форматированный вывод с помощью модуля “`Format`” |
| `html` | $\{$ `unit` $\to \alpha_i \to$ `HTML.t` $\} \to$ `unit` $\to \{\alpha_i\}$ `t` $\to$`HTML.t` | преобразование в HTML представление |
| `compare` | $\{ \alpha_i \to \alpha_i \to$ `comparison` $\} \to \{\alpha_i\}$ `t` $\to \{\alpha_i\}$ `t` $\to$`comparison` | сравнение |
| `eq` | $\{ \alpha_i \to \alpha_i \to$ `bool` $\} \to \{\alpha_i\}$ `t` $\to \{\alpha_i\}$ `t` $\to$`bool` | проверка на равенство |
| `foldl` | $\{ \alpha \to \alpha_i \to \alpha \} \to \alpha \to \{\alpha_i\}$ `t` $\to \alpha$ | протаскивание наследуемого атрибута через все узлы сверху вниз |
| `foldr` | $\{ \alpha \to \alpha_i \to \alpha \} \to \alpha \to \{\alpha_i\}$ `t` $\to \alpha$ | протаскивание наследуемого атрибута через все узлы снизу вверх |
| `gmap` | $\{$ `unit` $\to \alpha_i \to \beta_i \} \to$`unit` $\to \{\alpha_i\}$ `t` $\to \{\beta_i\}$ `t` | функтор |

Рис. 2: Список предоставляемых по умолчанию плагинов

ход используется в нескольких уже существующих библиотеках [30, 33], но, мы заявляем, что в нашем случае реализация плагинов выглядит несколько проще. Причиной этому является то, что конкретная и обобщенная части трансформаций разделены. Следовательно, создание плагина выливается только в правильное создание класса трансформации, что требует минимального вмешательства разработчика. В общем случае, только следующая информация о новом плагине должна быть указана:

- Типы наследуемого и синтезируемого атрибутов для каждого параметра типа.
- Типы наследуемого и синтезируемого атрибутов для самого преобразуемого типа.
- Тело метода для преобразования конструкторов.

Итого, количество мест, где плагин генерирует код для обработки типов довольно мало, а генерируемый код относительно прост. Ин-

терфейс построения синтаксического дерева напоминает интерфейс в `ppxlib` (а именно, подмодуль `Ast_builder`), который должен быть знаком всем, кто когда-то разрабатывал синтаксические расширения для OCaml В разделе 4.3 мы представим полный пример создания свежей реализации плагина.

## 3.4. Взаимная рекурсия

Полная поддержка взаимно рекурсивных определений типов требует дополнительных усилий. Формально, создание всех необходимых сущностей может быть произведена также, как и для одиночного случая, но это может нарушить расширяемость получаемых преобразований. Мы продемонстрируем это феномен в примере ниже. Рассмотрим определение типа

```
type expr = ... | LocalDef of def * expr
and  def  = Def of string * expr
```

где мы опустили неважные части (переменные, бинарные операции и т.д.) в объявлении типа выражений. Довольно очевидно, что обобщённые функции преобразований для обоих типов могут быть оставлены как они есть, так как они по сути просто перекладывают работы про выполнению преобразования на плечи методов объекта и не зависят от наличия рекурсии в определениях типов.

```
let ⟨gcata_expr⟩ ω ι = function
...
| LocalDef (d, e) as x → ω#⟨LocalDef⟩ ι x d e

let ⟨gcata_def⟩ ω ι = function
| Def (s, e) as x → ω#⟨Def⟩ ι x s e
```

То же самое верно и для общего класса-предка. Однако, если мы начнем реализовывать конкретные преобразования, то нам понадобится преобразование значений типа “`expr`” внутри класса для “`def`”, и наоборот. Это может быть сделано с помощью взаимно рекурсивных

определений классов (мы опять же опускаем неважные части кода):

```
class ⟨show_expr⟩ fself = object
  inherit [unit, _, string] ⟨expr⟩ fself
  ...
  method ⟨LocalDef⟩ ι x d e =
    ... (fix ⟨gcata_def⟩ (fun fself → new ⟨show_def⟩ fself) ...) ...
end
and ⟨show_def⟩ fself = object
  inherit [unit, _, string] ⟨def⟩ fself
  method ⟨Def⟩ ι x s e =
    ... (fix ⟨gcata_expr⟩ (fun fself → new ⟨show_expr⟩ fself) ...) ...
end
```

Заметьте, что в обоих аргументах "`fix`" мы создаем *конкретные* классы ("$\langle show_{def} \rangle$" и "$\langle show_{expr} \rangle$"). На первый взгляд, это должно работать как полагается. Строго говоря, это *конкретное* преобразование действительно работает. Но что случится, если нам понадобится переопределить поведение в классе "$\langle show_{expr} \rangle$"? Согласно подходу, определенному выше, на необходимо отнаследоваться от "$\langle show_{expr} \rangle$", переопределить некоторые метода и сконструировать функцию с помощью комбинатора неподвижной точки:

```
class custom_show fself = object
  inherit ⟨show_expr⟩ fself
  method ⟨Const⟩ ι x n = "a constant"
end

let custom_show e =
  fix ⟨gcata_expr⟩ (fun fself → new custom_show fself) () e
```

А это не будет работать так, как мы ожидаем, потому мы не определили метод "$\langle LocalDef \rangle$", который использует класс по умолчанию для типа "`def`", который в свою очередь пользуется классом по умолчанию для типа "`expr`". Получается, что мы переопределили поведение только одной компоненты взаимно рекурсивного преобразования типов, а

именно для типа "`expr`". Все вхождения типа "`expr`" в других типах всё ещё преобразуются стандартным образом. Чтобы исправить это поведение, нам придется повторить реализацию взаимно рекурсивных классов *целиком*, что обесценивает всю идею расширяемости.

Наше решение проблемы снова полагается на идею открытой рекурсии. Вкратце, мы параметризируем конкретный класс преобразования трансформациями *всех* типов, участвующих во взаимно рекурсивном определении типов. так как эта параметризация нарушает соглашение об интерфейсах классов, нам придется объявить эти классы как дополнительные. Для нашего примера они будут выглядит вот так:

```
class ⟨show_stub_expr⟩ f_expr f_def = object
  inherit [unit, _, string] ⟨expr⟩ f_expr
  ...
  method ⟨LocalDef⟩ ι x d e = ... (f_def ...) ...
end

class ⟨show_stub_def⟩ f_expr f_def = object
  inherit [unit, _, string] ⟨def⟩ f_def
  method ⟨Def⟩ ι x s e = ... (f_expr ...) ...
end
```

Обратите внимание на отсутствие рекурсивных классов.

Затем мы сгенерируем комбинатор неподвижной точки для этого взаимно рекурсивного определения:

```
let ⟨fix_expr,def⟩ (c_expr, c_def) =
  let rec t_expr ι x = ⟨gcata_expr⟩ (c_expr t_expr t_def) ι x
  and t_def ι x = ⟨gcata_def⟩ (c_def t_expr t_def) ι x in
  (t_expr, t_def)
```

Здесь $c_{expr}$ и $c_{def}$ являются генераторами объектов, которые принимают как параметры функции преобразования всех типов, которые встречаются во взаимно рекурсивном определении. Обратите внимание, что тот же самый комбинатор неподвижной точки может использоваться для того, чтобы сконструировать любое конкретное преобразование

для данного взаимно рекурсивного определения типов.

С этими дополнительными классами мы может сконструировать реализации по умолчанию для любого конкретного преобразования:

```
let ⟨show_expr⟩ , ⟨show_def⟩ =
  ⟨fix_expr,def⟩ (new ⟨show_stub_expr⟩ , new ⟨show_stub_def⟩)
```

Эти преобразования по умолчанию, во-первых, должны сохраниться во всех структурах с информацией о типах для соответствующих типов, и во-вторых, используются для создания классов трансформацией, с ожидаемым интерфейсом:

```
class ⟨show_expr⟩ fself = object
  inherit ⟨show_stub_expr⟩ fself ⟨show_def⟩
end
class ⟨show_def⟩ fself = object
  inherit ⟨show_stub_def⟩ ⟨show_expr⟩ fself
end
```

Здесь мы снова сделали взаимно рекурсивные типы неотличимыми от простых (в терминах интерфейсов классов), что позволяет единообразным способом конструировать преобразования этих типов в файлах, где эти типы используются, но не объявлены.

С другой стороны, чтобы расширить имеющееся преобразование, теперь необходимо наследоваться от *дополнительных* классов и использовать специальный комбинатор неподвижной точки. Для нашего предыдущего неудачного случая преобразование выглядит почти также просто, как и для одиночного объявления типа:

```
let custom_show, _ =
  ⟨fix_expr,def⟩ ((fun f_expr f_def →
                     object inherit ⟨show_stub_expr⟩ f_expr f_def
                       method ⟨Const⟩ ι x n = "a constant"
                     end),
                  new ⟨show_stub_def⟩)
```

В конкретной реализации библиотеки мы генерируем мемоизирующий комбинатор неподвижной точки, который следует тому же шабло-

ну, который был описан в разделе 3.2. К тому же, мы сохраняем данный комбинатор в структуре с информацией о типе, чтобы для типа "`t`" этот комбинатор мог быть использован с помощью выражение "`fix(t)`". Пользователям, однако, придется держать в уме, что тип является взаимно рекурсивным, чтобы воспользоваться комбинатором правильно.

Однако присутствует одна сложность с поддержкой взаимной рекурсии: мы полагаемся на то свойства, что добавление одной функции преобразования для типа достаточно, чтобы реализовать открытую рекурсию. Однако, строго говоря, это не так. Например, рассмотрим следующее объявление типа:

```
type (α, β) a = A of α b * β b
and  α b = X of (α, α) a
```

В аргументах конструктора "`A`" мы имеем *различные* параметризации типа "`b`", и поэтому нам понадобятся *две* функции — для"$\alpha$ `b`" и для "$\beta$ `b`". Однако, тип "`a`" не является регулярным — начав преобразование типа "($\alpha$, $\beta$) `a`" мы придём к необходимости преобразования значений типов "($\alpha$, $\alpha$) `a`" и "($\beta$, $\beta$) `a`".

Следовательно, мы уже отсеяли такие объявления типов. Получается, что взаимно рекурсивные объявления типов являются *существенными* в том смысле, что они не всегда могут быть разделены на два не взаимно рекурсивных определения, а именно, когда каждая пара типов взаимно достижима. Если мы заменим второе объявление типа, скажем, на

```
...
and α b = int
```

то мы получим объявление типов, которое не поддерживается у нас. Однако, так как типы "`a`" и "`b`" *не являются* по сути взаимно рекурсивными, то всё определение типов может быть переписано, что уже позволит воспользоваться нашими наработками.

## 3.5. Полиморфные вариантные типы

Мы считаем поддержку полиморфных вариантных типов [8, 9] важной части нашей работы, так как она открывает возможности композиционального определения структур данных с возможность объявления композициональных преобразований. Главным отличием между полиморфными вариантным типами и алгебраическими, является возможность *расширения* объявленных ранее полиморфных вариантных типов путём добавление новых конструкторов или комбинированием нескольких типов в один.

Нашей задачей является предоставление *бесшовной* интеграции с обобщенными возможностями. Когда несколько типов будет скомбинированы, мы должны получить все обобщенные возможности простым наследование соответствующих типов.

Как мы сказали ранее, дополнительный параметр "$\varepsilon$" вычисляется в открытую разновидность полиморфного вариантного типа. Следовательно, должно быть разрешено использовать ту же функцию обобщенного преобразования до для более *широкого* типа[4]. Это может быть достигнуто специфической формой обобщенной функции трансформации, которая производит "открытие":

```
let ⟨gcata_t⟩ ω ι subj =
  match subj with
  ...
  | C ... → ω#⟨C⟩ ι (match subj with #t as subj → subj) ...
  ...
```

Это выливается в применении методов объекта, представляющего преобразование, к открытой разновидности типа, в то время как обобщенная функция преобразования принимает замкнутый тип.

Если несколько полиморфных вариантных типов объединяются, то обобщенная функция преобразования сопоставляем значение с образцами-типами и передает управление соответствующим обобщенными функ-

[4]Мы воздерживаемся от использования термина "подтип" так как в OCaml нет настоящего подтипирования.

циям преобразования. !

# 4. Примеры

В этом разделе мы представим несколько примеров, реализованных с помощью нашей библиотеки. В этих примерах используются синтаксические расширения `camlp5`, хотя всё может быть переписано с использованием `ppxlib`. Как было сказано выше данная работа является прямым наследником [3] и все примеры из той статьи работают в этой версии. Здесь мы покажем несколько новых.

## 4.1. Типизированные логические значения

Первый пример появился во врем работы на строго типизированным логическим предметно-ориентированным языком для OCaml [16]. Одной из самых важных конструкций была унификация термов со свободными логическими переменными, работать с такими структурами данных сложно, а допустить ошибку — легко. Типичным сценарием взаимодействия между логическими и нелогическими частями программ является создание так называемых *целей вычислений* (англ. goal), содержащих структуры данных со свободными логическими переменными в них. Решением логической цели, является подстановка переменных, правые части которой в идеальном случае не содержит свободных переменных. Чтобы сконструировать цель вычислений систематически вводить логические переменные в некоторую типизированную структуру данных, а для восстановления ответа – систематически извлекать из логических представлений ответы в нелогическом представлении.

Упрощённый тип для логических переменных может быть описан следующим образом:

```
@type 'a logic =
| V     of int
| Value of 'a       with show, gmap
```

Логическое значение может быть либо свободной логической переменной (“`V`”) или каким-то другим значением (“`Value`”), которое не является свободной переменной, но потенциально может содержать свободные

переменные внутри себя. Чтобы преобразовывать в и из логических значений, можно воспользоваться следующими функциями:

```
let lift x = Value x


let reify  = function
| V     _  →  invalid_arg "Free variable"
| Value x  →  x
```

Функция "`reify`" бросает исключение для свободных переменных, так как в присутствии вхождений свободных переменных логическое значение нельзя рассматривать как обыкновенную (нелогическую) структуру данных.

Когда мы работает с логическими структурами данных, на необходима возможность вставлять логические переменные в произвольные позиции. Это означает, что мы должны переключиться на использование другого типа, который принадлежит домену логических типов. Например, для арифметических выражений, которые мы использовали как пример, нам понадобится конструировать значения вида

```
Value (
  Binop (
    V 1,
    Value (Const (V 2)),
    V 3
  )
)
```

которые будут иметь тип "`lexpr`", объявленный как

```
type expr' = Var of string logic | Const of int logic
           | Binop of lexpr * lexpr
and  lexpr = expr' logic
```

Нам также нужно реализовать две функции преобразования. Все эти определения представляют собой типичный пример однотипного (boilerplate) кода.

С измользование нашего подхода решение почти полностью декларативно[5]. Во-первых, мы абстрагируемся от интересующего нас типа, заменяя все его вхождения типовой переменной с не встречающимся ранее именем:

```
@type ('string, 'int, 'expr) a_expr =
| Var   of 'string
| Const of 'int
| Binop of 'string * 'expr * 'expr with show, gmap
```

Здесь мы абстрагировали тип от всего конкретного, но мы могли обойтись абстрагированием только от самого себя. Заметьте, что мы воспользовались двумя видами обобщенных преобразований — "show" и "gmap". Первое будет полезно для отладочных целей, а второе является необходимым для нашего решения.

Теперь мы можем объявить логические и нелогические составляющие как специализации исходного типа:

```
@type expr  = (string, int, expr) a_expr
  with show, gmap
@type lexpr = (string logic, int logic, lexpr) a_expr logic
  with show, gmap
```

Обратите внимание, что "новый" тип "expr" эквивалентен старому, следовательно, такое переписывание типов не нарушает существующий код.

Наконец, определения функций преобразования воспользуются преобразованием, полученным с помощью плагина "gmap", предоставляемого библиотекой:

```
let rec to_logic   expr = gmap(a_expr) lift  lift  to_logic  expr
let rec from_logic expr = gmap(a_expr) reify reify from_logic @@
                                       reify expr
```

Как вы видите, поддержка типовых операторов существенна для этого примера. В предыдущей реализации [3] типовые операторы не были поддержаны и их было не так просто добавить.

[5]При условии включения ключа компиляции `-rectypes`

## 4.2. Преобразование в безымянное представление

Полиморфные вариантные типы позволяют описывать структуры данных композиционально, статически типизировано и в разных модулях компоновки [9]. Объявлять преобразования таких структур данных отдельно является естественной идеей. Проблема конструирования преобразований раздельно объявленных строго типизированных компонент известна как "проблема выражений" ("The Expression Problem" [27]), которая часто используется как "лакмусовый тест" для оценивания подходов к обобщенному программированию [21, 25]. В этом разделе мы покажем решение проблемы выражений в рамках нашего подхода. В качестве конкретной задачи мы реализуем преобразование лямбда-термов в безымянное представление.

Во-первых, опишем часть языка термов без связывающих конструкций:

```
@type ('name, 'lam) lam = [
| `App of 'lam * 'lam
| `Var of 'name
] with show
```

Отделение этого типа выглядит логичной идей, так как потенциально в языке может появить множество конструкций, связывающих переменные ($\lambda$-абстракции, **let**-определения и т.д.) и комбинируя их с не связывающей частью и с ними самими можно получать множество языков с согласованным поведением.

Тип "`lam`" полиморфен. Первый параметр используется для представления имен или индексов де Брауна, второй необходим для открытой рекурсии (здесь мы следуем уже исследованному подходу по описанию расширяемых структур данных с помощью полиморфных вариантов [9]).

Как должно выглядеть преобразование в безымянное представление для такого типа? А именно, как должен выглядеть класс преобразования? Это показано ниже:

```
class ['lam, 'nameless] lam_to_nameless
```

```
  (flam : string list → 'lam → 'nameless) =
object
  inherit [string list, string, int,
           string list, 'lam, 'nameless,
           string list, 'lam, 'nameless] ⟨lam⟩
  method ⟨App⟩ env _ l r = `App (flam env l, flam env r)
  method ⟨Var⟩ env _ x = `Var (index env x)
end
```

Здесь мы используем список строк для хранения подстановки переменных и передаем его как наследуемый атрибут. Затем мы воспользуемся функцией "`index`" чтобы найти строку в подстановки, т.е. эта функция преобразуем имея в индекс де Брауна. Интересной часть преобразования является типизация общего класса предка "$\langle lam \rangle$". Первая тройка параметров описывает преобразование первого типового параметра. Как Вы видите, мы преобразуем строки в числа используя подстановку. Здесь типовая переменная "`'lam`", как мы знаем, приравняется открытой версии типа "`lam`". Наконец. результат преобразование типизируется с помощью типовой переменной "`'nameless`". Так происходит именно так потому, что, как будет понятно позднее, это будет действительно другой тип. Так как второй типовый параметр обычно ссылается рекурсивно на себя, третья тройка типовых параметров совпадает со второй.

Давайте теперь добавим связывающую конструкцию – $\lambda$-абстракцию:

```
@type ('name, 'lam) abs = [ `Abs of 'name * 'lam ] with show
```

Те же самые рассуждения применимы и тут: мы пользуется открытой рекурсией и параметризируем представление относительно имени. Класс для преобразования будет выглядеть похожим образом:

```
class ['lam, 'nameless] abs_to_nameless
  (flam : string list → 'lam → 'nameless) =
object
  inherit [string list, string, int,
           string list, 'lam, 'nameless,
```

```
                string list, 'lam, 'nameless] ⟨abs⟩
  method ⟨Abs⟩ env name term = `Abs (flam (name :: env) term)
end
```

Заметьте, что метод "$\langle Abs \rangle$" конструирует значения *другого* типа, чем любая возможная параметризация типа "`abs`". Действительно, безымянное представление типа не должно содержать никаких суррогатов имён.

Теперь мы можем объединить эти два типа, чтобы получить тип термов со связывающими конструкциями:

```
@type ('name, 'lam) term =
  [ ('name, 'lam) lam | ('name, 'lam) abs) ] with show
```

Мы можем предоставить два новых типа для именованного и безымянного представления[6]:

```
@type named    = (string, named) term with show
@type nameless = [ (int, nameless) lam | `Abs of nameless] with show
```

Наконец, мы может описать преобразование, которое превращает именованные термы в их безымянное представление:

```
class to_nameless
  (fself : string list → named → nameless) =
object
  inherit [string list, named, nameless] ⟨named⟩
  inherit [named, nameless] lam_to_nameless fself
  inherit [named, nameless] abs_to_nameless fself
end
```

Это преобразование получается путём наследования всех важных составляющих: общего класса для всех трансформаций типа "`named`" и друз конкретных преобразований его составляющих. Функция трансформации может быть получена стандартным способом:

```
let to_nameless term =
```

[6]Нам понадобится использовать ключ компиляции `-rectypes`, чтобы эти определения типов скомпилировались .

```
transform(named) (fun fself → new to_nameless fself) [] term
```

Только что мы построили реализацию, комбинируя реализации для его составляющих. Эти частичные решения могут быть раздельно скомпонованы, а вся система при этом остается строго типизированной.

## 4.3. Пример пользовательского плагина

Наконец, мы продемонстрируем использование системы плагинов на свежем примере реализации плагина. Для этой цели мы выбрали широко известное преобразование *hash-consing* [7]. Это преобразование превращает структуры данных в их максимально компактное представление в памяти, при котором структурно равные части представляются в памяти как один физический объект. Например, синтаксическое дерево выражения

```
let t =
  Binop ("+",
    Binop ("-",
      Var "b",
      Binop ("*", Var "b", Var "a")),
    Binop ("*", Var "b", Var "a"))
```

может быть переписано как

```
let t =
  let b  = Var "b" in
  let ba = Binop ("*", b, Var "a") in
  Binop ("+", Binop ("-", b, ba), ba)
```

где равные подвыражения представляются как равные поддеревья.

Наш плагин по типу "$\{\alpha_i\}$ `t`" предоставит функцию для hash-consing "`hc(t)`" с сигнатурой

$\{$ `H.t` $\rightarrow \alpha_i \rightarrow$ `H.t` $*$ $\alpha_i$ $\}$ $\rightarrow$ `H.t` $\rightarrow \{\alpha_i\}$ `t` $\rightarrow$ `H.t` $* \{\alpha_i\}$ `t`

где "`H.t`" — это гетерогенная хэш таблица для произвольных типов. Интерфейс у неё следующий:

```
module H : sig
  type t
  val hc : t → 'a → t * 'a
end
```

Функция "`H.hc`" принимает хэш таблицу и некоторое значение и возвращает потенциально обновленную хэш таблицу и значение, которое структурно эквивалентно поданному на вход. Мы не будет описывать реализацию этого модуля, а приведем пример использования в конструкторе:

```
method ⟨Binop⟩ h _ op l r =
  let h, op = hc(string) h op in
  let h, l  = fself h l in
  let h, r  = fself h r in
  H.hc h (Binop (op, l, r))
```

Этот метод принимает как наследуемый атрибут хэш таблицу "`h`", преобразуемое целиком, которое здесь не потребуется; три аргумента конструктора: "`op`" типа `string`, а также "`l`" и "`r`" типа `expr`. Мы вначале обрабатываем аргументы, что создает нам новую хэш таблицу и три новых значения тех же типов. Затем мы применяем конструктор и запускаем функцию для hash-consing ещё раз. Для обработки аргументов конструктора мы используем функции, предоставленные библиотекой:

для типа `string` это "`hc(string)`"[7], а оба подвыражения мы обработаем с помощью "`fself`".

Ещё надо разобраться с параметрами класса, наследоваться от которого мы будем в классе для преобразования типа "$\{\alpha_i\}$ `t`". Очевидно, что все наследуемые атрибуты будут иметь тип "`H.t`", а синтезированные — "`H.t` $*$ $a$" для каждого интересующего нас типа "$a$". Это приводит нас к следующему объявлению класса трансформации:

```
class [{α_i}, ε] ⟨hc_t⟩ ... = object
  inherit [{ H.t, α_i, H.t * α_i }, H.t, ε, H.t * ε] ⟨t⟩
```

[7]В общем случае, на было бы необходимо реализовать функцию hash-consing для каждого примитивного типа. В нашем же случае мы можем воспользоваться функцией "`H.hc`".

```
  ...
end
```

Для простоты мы опустили спецификацию параметров-фукнций для класса, так как их сигнатуры могут быт легко восстановлены.

Теперь надо реализовать эту логику с помощью системы плагинов.

Код инфраструктуры для плагинов указан ниже:

```
let trait_name = "hc"

module Make (AstHelpers : GTHELPERS_sig.S) = struct
  open AstHelpers
  module P = Plugin.Make (AstHelpers)
  class g tdecls = object (self : 'self)
    inherit P.with_inherited_attr tdecls
    ...
  end
end

let _ = Expander.register_plugin trait_name
          (module Make : Plugin_intf.Plugin)
```

Чтобы реализовать плагин, необходимо реализовать функтор, параметризованный дополнительным модулем, который напоминает модуль "`Ast_builder`" из библиотеки `ppxlib`, который используется для конструирования абстрактного инстаксического дерева OCaml. Нам необходимо реализовывать функтор, потому что мы поддерживаем два вида синтаксических расшируений: для `camlp5` и для `ppxlib`. Главная сущностью в теле фукнцтора является класс "`g`" ("генератор"), который мы для простоты будем наследовать от базового класса нашей бибилотеки. В данном случае мы создаем модуль с базовой реализацией плагина в модуле "`P`" на основе "`AstHelpers`" и затем наследуемся от класса "`P.with_inherited_attr`", что означает, что хотим получить плагин, где наследуемый атрибут содержательно используется (не является типом "`unit`"). Этот класс принимает объявления типов как параметры.

В конце, мы региструиуем функтор как модуль первого класса, что делает его доступным для использования.

Сейчас мы покажем как выглядят методы класса-генератора. Во-первых, надо указать какие типы будут у наследуемыех и синтезированных атрибутов:

```
method main_inh ~loc _tdecl = ht_typ ~loc

method main_syn ~loc ?in_class tdecl =
  Typ.tuple ~loc
    [ ht_typ ~loc
    ; Typ.use_tdecl tdecl
    ]

method inh_of_param tdecl _name =
  ht_typ ~loc:(loc_from_caml tdecl.ptype_loc)

method syn_of_param ~loc s =
  Typ.tuple ~loc
    [ ht_typ ~loc
    ; Typ.var ~loc s
    ]
```

Здесь мы предполагаем, что тип "`ht_typ`" объявлен как

```
let ht_typ ~loc =
  Typ.of_longident ~loc (Ldot (Lident "H", "t"))
```

Другими словами, мы объявляем, что типом наследуемого атрибут всегда будет "`H.t`", а типом синтезированного атрибута будет пара "`H.t * t`".

Следующая группа методов описывает параметры классов плагина:

```
method plugin_class_params tdecl =
  let ps = List.map tdecl.ptype_params
             ~f:(fun (t, _) → typ_arg_of_core_type t)
  in
```

```
  ps @
  [ named_type_arg ~loc:(loc_from_caml tdecl.ptype_loc) @@
    Naming.make_extra_param tdecl.ptype_name.txt
  ]

method prepare_inherit_typ_params_for_alias ~loc tdecl rhs_args =
  List.map rhs_args ~f:Typ.from_caml
```

Первый метод описывает типовые параметры класса плагина: для данного случая это типовые параметры самого объявления типа плюс дополнительный типовый параметр “$\varepsilon$”. Второй метод описывает вычисление типовых параметров для применения конструктора типа. В случае, если объявление типа выглядит как

**type** $\{\alpha_i\}$ t = $\{a_i\}$ tc

нам необходимо построить реализацию преобразовния для типа “t” из реализации оного для типа “tc”, наследуясь от правильного инстанциированного соответвующего класса. Для нашего случая класс параметризуется теми же типовыми параметрами, что и объяляемый тип, поэтому мы оставляем их как есть.

Последняя группа методов отвечает за генерацию тел методов для трансформаций конструкторов. Мы поддерживаем регулярные конструкторы алгебраических типов, где аргументами может быть и кортеж, и запись, а также записи и кортежи на верхнем уроне, преобразование которые, как правило, имеет много общих частией. Всего за это отвечают 4 метода, но здесь мы покажем только один:

```
method on_tuple_constr ~loc ~is_self_rec ~mutual_decls
                            ~inhe tdecl constr_info ts =
  ...
  match ts with
  | [] → Exp.tuple ~loc [ inhe; c [] ]
  | ts →
     let res_var_name = sprintf "%s_rez" in
     let argcount = List.length ts in
```

```
let hfhc = Exp.of_longident ~loc (Ldot (Lident "H", "hc")) in
List.fold_right
  (List.mapi ~f:(fun n x -> (n, x)) ts)
  ~init:...
  ~f:(fun (i, (name, typ)) acc ->
      Exp.let_one ~loc
        (Pat.tuple ~loc
           [ Pat.sprintf ~loc "ht%d" (i+1)
           ; Pat.sprintf ~loc "%s" @@ res_var_name name])
        (self#app_transformation_expr ~loc
           (self#do_typ_gen ~loc ~is_self_rec
                            ~mutual_decls tdecl typ)
           (if i = 0 then inhe else Exp.sprintf ~loc "ht%d" i)
           (Exp.ident ~loc name)
        )
        acc
    )
...
```

Реализация использует заранее заготовленный метод нашей библиотеки "`self#app_transformation_expr`", который генерируется применение фукцнии преобразования к соответсвующему типу.

Конечной компонентой реализации является сам модуль "`H`". Стандартный функтор "`Hashtbl.Make`" создает хэш таблицы, используя некоторую хэш фукнцию и предикат равенства, предоставленные пользователем. В целом, следуем мы следуем такому соглашению: как хэш фукнцию используем полиморфную "`Hashtbl.hash`", а качестве равенства используем физическое равенство "==". Однако, присутсвуют две сложности:

- Так как таблице гетерогенная нам необходимо использовать небезопасное приведение типов "`Obj.magic`".
- Наша реализация равенства чуть более сложная, чем обычное "==". Нам необходимо стравнивать верхнеуровневые конструкто-

ры и количества их аргументов *структурно*, а только затем сравнивать соответствующие аргументы взическим равенством. Технически, мы может считать равными структурно равные значения *различных* типов.

Мы полагаемся здесь на следующее наблюдение: hash-consing корректно использовать тольео для структур данных, которые прозрачны по ссылкам, мы предполагаем что равные структуры данных взаимозаменяемы не смотря на их типы.

Полную реализацию плагина может быть увидеть в главном репозитории. Она занимает 164 строчки кода, учитывая комментарии и пустые строки.

# 5. Обзор похожих решений

В данной работе использованы одновременно и функциональные (комбинаторы), и объектно-ориентированные возможности языка OCaml. Можно найти связанные работы одновременно и в области типизированного функционального и объектно-ориентированного программирования. Наиболее близкой, использующий язык OCaml и имеющей отношение к этой работе, библиотекой является Visitors [22], которая использует те же самые идеи, но принимает существенно другие дизайнерские решения. Детальное сравнение с Visitors вы найдете в конце данного раздела.

Во-первых, существует несколько библиотек для обобщенного программирования для OCaml, которые используют полностью генеративный подход [30, 33] — все необходимые обобщенные функции для всех типов генерируются по отдельности. Этот подход очень практичен до тех пор, пока набор предоставляемых преобразований удовлетворяет всем нуждам. Однако, если это не так, необходимо расширить кодовую базу, реализовав все отсутствующие функции заново (с потенциально очень малым переиспользовыванием кода). К тому же, те функции, которые получаются в результате, нерасширяемы. В нашем подходе, во-первых, большое количество полезных обобщенных функций может быть получено из уже сгенерированных. Во-вторых, чтобы получить полностью новый плагин, достаточно модифицировать только “интересные” части, так как функции обхода и класс для объекта преобразования библиотека создает самостоятельно.

Несколько подходов для функционального обобщенного программирования используют *представление типов* [12]. В основе лежит идея разработки универсального представления для произвольного типа, преобразования которого необходимо получить, и предоставления двух функций, выполняющих преобразование в универсальное представление и обратно, и в идеале образующих изоморфизм. Обобщенные функции преобразуют представление исходных типов данных, что позволяет реализовать все необходимые преобразования один раз. Функции транс-

ляции в универсальное представление и обратно могут быть получены (полу)автоматически, используя такие особенности системы типов как классы типов [12,25] и семейства типов [6] в языке Haskell, или используя синтаксические расширения [1] в языке OCaml. Хотя некоторые из этих подходов позволяют модификацию получаемых преобразований (например, обработка некоторых типов особым образом) и поддерживают расширяемые типы, наш подход более гибок, так как позволяет модификацию на уровне отдельных конструкторов. К тому же, мы позволяем сосуществовать нескольким видам преобразований для одного типа.

Другой подход был задействован в "Scrap Your Boilerplate" [17] (для краткости SYB), изначально разработанного для языка Haskell. Он делает возможным реализовать преобразования, которые обнаруживают вхождения конкретного типа в произвольной структуре данных. Поддерживаются два основных вида действий: *запросы*, которые выбирают значения конкретного типа данных на основе критериев, заданных пользователем, и *преобразования*, которые единообразно применяют преобразование, сохраняющее тип, в конкретной структуре данных. В последующих статьях этот подход был расширен для трансформаций, которые обходят пару структур данных одновременно [18], а также поддержкой расширения уже существующих преобразований новыми случаями [19]. Позднее, данный подход был реализован в других языках, включая OCaml [4, 31]. В отличие от нашего случая, SYB позволяет применять трансформации к конкретным типам целиком, а не отдельным конструкторам. К тому же, многообразие получающихся преобразований выглядит достаточно ограниченным. Также, потенциально, преобразования в SYB-стиле могут сломать барьер инкапсуляции, так как могут обнаруживать вхождения значений нужно типа в структуре данных *произвольного* типа. Таким образом, поведение зависит от особенностей внутренней реализации структуры данных, даже от тех, что были скрыты при инкапсуляции. Это может привести, во-первых, к возможности нежелаемой обратной разработки (reverse engineering) путём применения различных чувствительных к типу, преобразований

и анализа результатов. Во-вторых, к ненадежности интерфейсов: после модификации структуры данных реализация обобщенной функции для *старой* версии всё ещё может быть применена без статических или динамических ошибок, но с неправильным (или нежелательным) результатом.

Существует определенное сходство между нашим подходом и *алгебрами объектов* [21]. Алгебры объектов были предложены как решение проблемы выражения (expression problem) в распространенных объектно-ориентированных языках (Java, C++, C#), которые не требуют продвинутых особенностей системы типов кроме наследования и шаблонов. В оригинальном представлении алгебры объектов были преподнесены как шаблон проектирования и реализации; в последующих работах изначальная идея была улучшена различными способами [23, 24]. При использовании алгебр объектов преобразуемая структура данных также кодируется с использованием идеи "методы и варианты (конструкторы) один к одному", которая предоставляет расширяемость в обоих направлениях, а также ретроактивную реализацию. Однако, будучи разработанной для совершенно другого языкового окружения, решение с использование алгебр объектов существенно отличается от нашего. Во-первых, с использованием алгебр объектов "форма" структуры данных должна быть представлена в виде обобщенной функции, которая принимает конкретный экземпляр алгебры объектов как параметр (кодирование Чёрча для типов [12]). Применяя данную функцию к различным реализациям алгебры объектов можно получать различные преобразования (например, распечатывание). Чтобы инстанциировать саму структуру данных нужно предоставить особый экземпляр алгебры объектов — *фабрику*. Однако, после инстанциации структура данных больше не может быть трансформирована обобщенным образом. Следовательно, алгебры объектов заставляют пользователя переключиться на представление данных с помощью функций, которое может быть, а может не быть удобно в зависимости от обстоятельств. Наш же подход недеструктивно добавляет новую функциональность к уже знакомому миру алгебраических типов данных, сопоставления с

образцом и рекурсивных функций. Обобщенные реализации преобразований полностью отделены от представления данных и пользователи могут свободно преобразовывать их структуры данных привычным способом без потери возможности объявлять (и расширять) обобщенные функции. Другой особенностью OCaml, в отличии от распространенных языков объектно-ориентированного программирования, является то, что для написания расширяемого кода в основном используются полиморфные вариантные типы, а не классы. Поддержка полиморфных вариантных типов для написания расширяемых типов данных требует нового подхода.

Итого, среди уже существующих библиотек для обобщенного программирования для OCaml мы можем называть две, которые напоминают нашу: `ppx_deriving`/`ppx_traverse`, последняя версия которых находится в кодовой базе `ppxlib` [33], и Visitors [22].

`ppx_deriving` является наипростейшим подходом: объявления типов данных отображаются один к одному в рекурсивные функции, представляющие конкретный вид преобразования. Это наиболее эффективная реализация, так как функции вызываются напрямую, без позднего связывания, но нерасширяемая. Если пользователю понадобится слегка модифицировать сгенерированную функцию, то он должен будет полностью скопировать реализацию функции и изменить её. Количество работы по программированию нового преобразования может существенно увеличиться, если тип данных будет видоизменяться во время цикла разработки.

В `ppx_traverse` расширяемые трансформации также представлены как объекты. В отличие от нашего подхода, там не используется кодирование конструкторов и методов один к одному. К тому же `ppx_traverse` не использует наследуемые атрибуты, следовательно некоторые преобразования, такие как проверка на равенство и сравнение, невыразимы.

Visitors, с другой стороны, использует сходный с нашим подход, в котором были приняты многие решения, отвергнутые нами, и наоборот. Ниже мы подытожим главные различия:

- Visitors полностью объектно-ориентированы. Чтобы воспользо-

ваться преобразованием необходим создать некоторый объект и вызвать нужный метод. В нашем случае, если используются возможности, предусмотренные заранее, то можно использовать более естественный комбинаторный подход.

- Visitors реализуют некоторое количество преобразований в специфичной ad-hoc манере. В нашем случае все преобразования принадлежат некоторой обобщенной схеме. Различные трансформации можно скомбинировать с помощью наследования, если типы в схеме унифицируются. Мы также заявляем, что в нашей библиотека реализация ползовательски плагинов с трансформациями проще.

- Как и SYB, Visitors поддерживают указание способа трансформации для входящих в структуру данных типов: для каждого типа присутствует метод в объекте, представляющий трансформацию. Хотя такое представление добавляет некоторой гибкости мы осознанно отказывается от него, так как оно позволяет преодолеть инкапсуляционный барьер: изменяя методы преобразования (которые не могут быть скрыты в сигнатуре), можно получить некоторую информацию об внутреннем реализации инкапсулированной структуры данных. Более того, абстрактные структуры данных могут быть изменены способом, не предусмотренным публичным интерфейсом

- В нашем случае типовые параметры классов, представляющих трансформацию, должны быть указаны пользователем. В Visitors это работа возлагается на плечи компилятора, с помощью оригинального трюка. Однако, он не позволяет использовать Visitors в сигнатурах модулей. В нашем случае нет никаких проблем: поддерживается работа и с файлами реализации, и с файлами сигнатур.

- Visitors на сегодняшний день[8] не поддерживает полиморфные

[8]Последней доступной версией на данный момент является 20180513.

вариантные типы.

- GT поддержает произвольные применения конструкторов типов, а VISITORS и в мономорфном, и в полиморфном режиме – нет. Например, данный пример не компилируется:

```
type ('a,'b) alist = Nil | Cons of 'a * 'b
[@@deriving visitors { variety = "map"
                     ; polymorphic = true }]

type 'a list = ('a, 'a list) alist
[@@deriving visitors { variety = "map"
                     ; polymorphic = false }]
```

Более того, добавление искусственного конструктора не решает проблему:

```
type 'a list = L of ('a, 'a list) alist [@@unboxed]
[@@deriving visitors { variety = "map"
                     ; polymorphic = false }]
```

Также присутствуют сложности с переименованиями (aliases) типов в полиморфном режиме (мономорфная часть библиотеки VISITORS компилируется успешно):

```
type ('a,'b) t = Foo of 'a * 'b (* OK *)
[@@deriving visitors { variety = "map"
                     ; polymorphic = true }]

type 'a t2 = ('a, int) t
[@@deriving visitors { variety = "map"; name="somename"
                     ; polymorphic = true }]
```

Сгенерированный код можно исправить вручную, путём удаления типовых аннотаций для явного полиморфизма (explicit polymorphism) у методов, что приведет к коду, который очень напоминает генерируемый GT. Из этого мы можем заключить, что на GT можно

смотреть как перереализацию полиморфного режима библиотеки Visitors, где большее количество объявлений типов компилируется корректно.

# 6. Заключение

Существует несколько возможны направлений для дальнейшего развития проекта. Во-первых, в данной работе мы не касались вопросов производительности. Мы представляем преобразования в очень обобщенном виде, с несколькими слоями косвенности. Очевидно, что преобразования, реализованные с помощью нашей библиотеки будут работать медленнее, чем написанные вручную. Мы предполагаем, что производительность может быть улучшено с помощью, так называемого, staging [31] или, возможно, с помощью оптимизаций, специфичных для объектов.

Другим важным направлением является поддержка большего разнообразия объявлений типов, а именно GADT и нерегулярных типов. Хотя уже сделаны некоторые наработки, получившиеся решение делает интерфейс всей библиотеки чересчур сложным даже для простых случаев.

Наконец, структура с информацией о типе, которую мы генерируем, может быть использована, чтобы симитировать *ad-hoc* полиморфизм, так как они содержит реализацию функций, индексированных типами. Это в сумме с недавно предложенными расширениями [29] может открыть интересные перспективы.

# Список литературы


[1] Balestrieri Florent, Mauny Michel. Generic Programming in OCaml // Proceedings ML Family Workshop / OCaml Users and Developers workshops, Nara, Japan, September 22-23, 2016 / Ed. by Kenichi Asai, Mark Shinwell. — Vol. 285 of Electronic Proceedings in Theoretical Computer Science. — Open Publishing Association, 2018. — P. 59–100.

[2] Boulytchev Dmitry. Combinators and Type-driven Transformers in Objective Caml // Sci. Comput. Program. — 2015. — Dec. — Vol. 114, no. C. — P. 57–73. — Access mode: `https://doi.org/10.1016/j.scico.2015.07.008`.

[3] Boulytchev Dmitri. Code Reuse With Transformation Objects // CoRR. — 2018. — Vol. abs/1802.01930. — 1802.01930.

[4] Boulytchev Dmitry, Mechtaev Sergey. Efficiently Scrapping Boilerplate Code in OCaml // Workshop on ML. — ML '11. — Tokyo, Japan, 2011.

[5] Brooks Jr. Frederick P. The Mythical Man-month (Anniversary Ed.). — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 1995. — ISBN: 0-201-83595-9.

[6] Chakravarty Manuel M. T., Ditu Gabriel, Leshchinskiy Roman. Instant Generics : Fast and Easy. — 2009.

[7] Filliâtre Jean-Christophe, Conchon Sylvain. Type-safe Modular Hash-consing // Proceedings of the 2006 Workshop on ML. — ML '06. — New York, NY, USA : ACM, 2006. — P. 12–19. — Access mode: `http://doi.acm.org/10.1145/1159876.1159880`.

[8] Garrigue Jacques. Programming with Polymorphic Variants // Workshop on ML. — 1998.

[9] Garrigue Jacques. Code reuse through polymorphic variants // In Workshop on Foundations of Software Engineering. — 2000.

[10] Gibbons Jeremy. Calculating Functional Programs // Algebraic and Coalgebraic Methods in the Mathematics of Program Construction: International Summer School and Workshop Oxford, UK, April 10–14, 2000 Revised Lectures / Ed. by Roland Backhouse, Roy Crole, Jeremy Gibbons. — Berlin, Heidelberg : Springer Berlin Heidelberg, 2002. — P. 151–203. — ISBN: 978-3-540-47797-6. — Access mode: https://doi.org/10.1007/3-540-47797-7_5.

[11] Gibbons Jeremy. Datatype-generic Programming // Proceedings of the 2006 International Conference on Datatype-generic Programming. — SSDGP'06. — Berlin, Heidelberg : Springer-Verlag, 2007. — P. 1–71. — Access mode: http://dl.acm.org/citation.cfm?id=1782894.1782895.

[12] Hinze Ralf. Generics for the Masses // J. Funct. Program. — 2006. — Jul. — Vol. 16, no. 4-5. — P. 451–483. — Access mode: http://dx.doi.org/10.1017/S0956796806006022.

[13] Hutton Graham. A Tutorial on the Universality and Expressiveness of Fold // J. Funct. Program. — 1999. — Jul. — Vol. 9, no. 4. — P. 355–372. — Access mode: http://dx.doi.org/10.1017/S0956796899003500.

[14] Kiselyov Oleg, Jones Simon Peyton, Shan Chung-chieh. Fun with Type Functions // Reflections on the Work of C.A.R. Hoare / Ed. by A.W. Roscoe, Cliff B. Jones, Kenneth R. Wood. — London : Springer London, 2010. — P. 301–331. — ISBN: 978-1-84882-912-1. — Access mode: https://doi.org/10.1007/978-1-84882-912-1_14.

[15] Knuth Donald E. Semantics of context-free languages // Mathematical systems theory. — 1968. — Jun. — Vol. 2, no. 2. — P. 127–145. — Access mode: https://doi.org/10.1007/BF01692511.

[16] Kosarev Dmitry, Boulytchev Dmitry. Typed Embedding of a Relational Language in OCaml // Proceedings ML Family Workshop / OCaml Users and Developers workshops, ML/OCAML 2016, Nara,

Japan, September 22-23, 2016. — 2016. — P. 1–22. — Access mode: https://doi.org/10.4204/EPTCS.285.1.

[17] Lämmel Ralf, Jones Simon Peyton. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming // SIGPLAN Not. — 2003. — Jan. — Vol. 38, no. 3. — P. 26–37. — Access mode: http://doi.acm.org/10.1145/640136.604179.

[18] Lämmel Ralf, Jones Simon Peyton. Scrap More Boilerplate: Reflection, Zips, and Generalised Casts // Proceedings of the Ninth ACM SIGPLAN International Conference on Functional Programming. — ICFP '04. — New York, NY, USA : ACM, 2004. — P. 244–255. — Access mode: http://doi.acm.org/10.1145/1016850.1016883.

[19] Lämmel Ralf, Jones Simon Peyton. Scrap Your Boilerplate with Class: Extensible Generic Functions // SIGPLAN Not. — 2005. — Sep. — Vol. 40, no. 9. — P. 204–215. — Access mode: http://doi.acm.org/10.1145/1090189.1086391.

[20] Meijer Erik, Fokkinga Maarten, Paterson Ross. Functional Programming with Bananas, Lenses, Envelopes and Barbed Wire. — Springer-Verlag, 1991. — P. 124–144.

[21] Oliveira Bruno C. d. S., Cook William R. Extensibility for the Masses: Practical Extensibility with Object Algebras // Proceedings of the 26th European Conference on Object-Oriented Programming. — ECOOP'12. — Berlin, Heidelberg : Springer-Verlag, 2012. — P. 2–27. — Access mode: http://dx.doi.org/10.1007/978-3-642-31057-7_2.

[22] Pottier François. Visitors Unchained // Proc. ACM Program. Lang. — 2017. — Aug. — Vol. 1, no. ICFP. — P. 28:1–28:28. — Access mode: http://doi.acm.org/10.1145/3110272.

[23] Rendel Tillmann, Brachthäuser Jonathan Immanuel, Ostermann Klaus. From Object Algebras to Attribute Grammars //

SIGPLAN Not. — 2014. — Oct. — Vol. 49, no. 10. — P. 377–395. — Access mode: http://doi.acm.org/10.1145/2714064.2660237.

[24] Scrap Your Boilerplate With Object Algebras / Haoyuan Zhang, Zewei Chu, Bruno C.d. S. Oliveira, Tijs van der Storm // Proceedings of the Object-oriented Programming, Systems, Languages, and Applications (OOPSLA, 2015). — New York, United States, 2015. — Access mode: https://hal.inria.fr/hal-01261477.

[25] Swierstra Wouter. Data Types à La Carte // J. Funct. Program. — 2008. — Jul. — Vol. 18, no. 4. — P. 423–436. — Access mode: http://dx.doi.org/10.1017/S0956796808006758.

[26] Viera Marcos, Swierstra S. Doaitse, Swierstra Wouter. Attribute Grammars Fly First-class: How to Do Aspect Oriented Programming in Haskell // Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming. — ICFP '09. — New York, NY, USA : ACM, 2009. — P. 245–256. — Access mode: http://doi.acm.org/10.1145/1596550.1596586.

[27] Wadler Philip. The Expression Problem. — 1998. — Dec.

[28] 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 : ACM, 1989. — P. 60–76. — Access mode: http://doi.acm.org/10.1145/75277.75283.

[29] White Leo, Bour Frédéric, Yallop Jeremy. Modular implicits // Electronic Proceedings in Theoretical Computer Science. — 2015. — 12. — Vol. 198.

[30] Yallop Jeremy. Practical Generic Programming in OCaml // Proceedings of the 2007 Workshop on Workshop on ML. — ML '07. — New York, NY, USA : ACM, 2007. — P. 83–94. — Access mode: http://doi.acm.org/10.1145/1292535.1292548.

[31] Yallop Jeremy. Staged Generic Programming // Proc. ACM Program. Lang. — 2017. — Aug. — Vol. 1, no. ICFP. — P. 29:1–29:29. — Access mode: http://doi.acm.org/10.1145/3110273.

[32] camlp5. — https://camlp5.github.io.

[33] ppxlib. — https://github.com/ocaml-ppx/ppxlib.