УНИВЕРСИТЕТ ИТМО

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ
сентябрь–октябрь 2021 Том 21 № 5 http://ntv.ifmo.ru/
SCIENTIFIC AND TECHNICAL JOURNAL OF INFORMATION TECHNOLOGIES, MECHANICS AND OPTICS
September–October 2021 Vol. 21 No 5 http://ntv.ifmo.ru/en/
ISSN 2226-1494 (print) ISSN 2500-0373 (online)

НАУЧНО-ТЕХНИЧЕСКИЙ ВЕСТНИК УНИВЕРСИТЕТ ИТМО
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ

doi: 10.17586/2226-1494-2021-21-5-720-726
УДК 004.421.2:519.17 004.657

Обобщенное программирование с комбинаторами и объектами

Дмитрий Сергеевич Косарев[1]✉, Дмитрий Юрьевич Булычев[2]

[1] ООО «Интеллиджей ЛАБС», Санкт-Петербург, 197374, Российская Федерация
[2] Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация

[1] dmitry.kosarev@jetbrains.com✉, https://orcid.org/0000-0002-6773-5322
[2] dboulytchev@math.spbu.ru, dboulytchev@gmail.com, https://orcid.org/0000-0001-8363-7143

Аннотация

Предмет исследования. В функциональных языках программирования (например, OCaml, Haskell) распространен подход под названием «обобщенное программирование». Его сущность состоит в автоматическом построении на этапе компиляции программ некоторого кода, который выполняет преобразования значений указанного в программе типа данных. Традиционно построенные преобразования представляются с помощью функций, реализующих алгоритм преобразования. Из функций-преобразований и пользовательских функций можно получать новые преобразования с помощью композиции. Недостатком реализации преобразований с помощью функций является монолитность этих функций — невозможно изменить поведение уже объявленной функции. Если построить подходящее преобразование не представляется возможным, программист вынужден написать преобразование вручную и без переиспользования имеющегося кода, что приводит к большим затратам труда. В работе предложено автоматически строить преобразования так, чтобы их можно было изменять после объявления. **Метод.** Следуя парадигме объектно-ориентированного программирования, предложено хранить преобразования не как функции, а как объекты. Вместо вызова функции вызываются методы соответствующего объекта. Реализация преобразования разбивается на несколько методов объекта. Поддерживается расширение объектов новыми методами, изменение реализации уже имеющихся методов, и за счет этого возможна модификация автоматически построенных преобразований. **Основные результаты.** В работе предложен способ представления преобразований с помощью объектно-ориентированных возможностей языка OCaml. Объекты-преобразования имеют столько же методов, сколько альтернатив в объявлении типа входных данных преобразования. Такое количество методов позволяет реализовать достаточно разнообразные преобразования. В работе рассмотрено, в каких случаях большее количество методов нежелательно. Метод применим к полиморфным вариантным типам языка OCaml, которые не поддерживаются другими подходами по построению расширяемых преобразований. **Практическая значимость.** Подход не привязан к конкретной предметной области и позволяет использовать расширяемые преобразования в языке OCaml для произвольных программ, а также может быть перенесен на другие функциональные языки с поддержкой объектно-ориентированного программирования.

Ключевые слова

обобщенное программирование, функциональное программирование

Ссылка для цитирования: Косарев Д.С., Булычев Д.Ю. Обобщенное программирование с комбинаторами и объектами // Научно-технический вестник информационных технологий, механики и оптики. 2021. Т. 21, № 5. С. 720–726. doi: 10.17586/2226-1494-2021-21-5-720-726

Generic programming with combinators and objects

Dmitry S. Kosarev[1]✉, Dmitry Yu. Boulytchev[2]

[1] IntelliJ LABS Co. Ltd., Saint Petersburg, 197374, Russian Federation
[2] Saint Petersburg State University, Saint Petersburg, 199034, Russian Federation

[1] dmitry.kosarev@jetbrains.com✉, https://orcid.org/0000-0002-6773-5322
[2] dboulytchev@math.spbu.ru, dboulytchev@gmail.com, https://orcid.org/0000-0001-8363-7143

Abstract
The generic programming approach is popular in functional languages (for example, OCaml, Haskell). In essence, it is a compile-time generation of code that performs transformations of user-defined data types. The generated code can carry out various kinds of transformations of the values. Usually, transformations are represented as functions that implement algorithms of transformation. New transformations could be built from these transformation functions and user-defined ones. The representation based on functions has a downside: functions behave as final representations, and hence it is not possible to modify the behavior of the already built function. If the current set of transformations does not suit well, software developers are obliged to write a completely distinct transformation, even in the case when the new transformation is almost identical to the existing one. This work proposes to build transformations that are extensible after construction. The object-oriented programming paradigm will be followed, transformations will be represented not as functions but as objects. Instead of calling a transformation function, one of the object's methods will be called. The transformation itself will be split into methods. The extensibility is supported by adding new methods and overriding existing ones. In this paper, the authors propose an approach to represent transformations as objects in the OCaml functional programming language. Every alternative in data type definition has a corresponding object method. This design allows the construction of many distinct transformations. The cases where too many methods are not desirable are also discussed. The method is applicable to represent extensible transformations for polymorphic variant data types in OCaml when other methods fail to do it. The method is not bound to any particular domain. It allows the creation of extensible transformations in OCaml. It could be ported to other functional languages supporting object-oriented programming.

Keywords
generic programming, functional programming

For citation: Kosarev D.S., Boulytchev D.Yu. Generic programming with combinators and objects. *Scientific and Technical Journal of Information Technologies, Mechanics and Optics*, 2021, vol. 21, no. 5, pp. 720–726 (in Russian). doi: 10.17586/2226-1494-2021-21-5-720-726

Введение

Обобщенное программирование нацелено на придание языкам программирования большей гибкости без потери типовой безопасности [1]. В частности, на стадии компиляции программы по объявлениям типов данных предложено автоматически строить преобразования значений этих типов. Обобщенное программирование широко применяется к алгебраическим типам данных в классических типизированных функциональных языках программирования (OCaml, Haskell), а также используется в объектно-ориентированных платформах, например, .NET Source Generators[1].

При использовании обобщенного программирования бывают ситуации, когда автоматически построенные преобразования не имеют необходимой для данной задачи функциональности. В объектно-ориентированных языках программирования преобразования представляются в виде расширяемых объектов, и у программиста имеется возможность их изменять с помощью наследования. Но в функциональных языках такие преобразования конструируются в виде готовых функций, и варьировать их возможно только с помощью композиции, если необходимая вариативность была предусмотрена. Соответственно, применение обобщенного программирования для получения расширяемых преобразований в функциональных языках имеет препятствия.

В работе [2] предложена идея применения в языке OCaml элементов объектно-ориентированного программирования для решения этой проблемы. Но упомянутый подход имеет несколько недостатков: не предоставляет привычный разработчику интерфейс в виде функций-преобразований; систематически нарушает барьер абстракции, при этом в построенных объектах появляется доступ к деталям реализации, которые разработчик пожелал оставить скрытыми.

В данной работе показан метод сочетания обобщенного программирования, функциональных и объектно-ориентированных возможностей языка OCaml для построения расширяемых преобразований. Метод должен быть применим к другим языкам, где есть поддержка объектно-ориентированного программирования, например, Scala. Но в отличие от других работ, предлагаемый метод концентрируется на поддержке большего разнообразия типов данных языка OCaml, а именно полиморфных вариантных типов и типов с неограниченной рекурсией. Также реализуется минимально необходимый интерфейс доступа к преобразованию для случаев, когда реализацию исходного типа данных желательно скрыть. Функции-преобразования, построенные из объектов, имеют привычный интерфейс для пользователей, применяющих стандартные подходы к обобщенному программированию[2]. Данный метод реализован авторами в библиотеке GT[3] (Generic Transformers) для языка OCaml: способы получения стандартных преобразований, аналогичных тем, что предоставляются популярными библиотеками обобщенного программирования, а также способы получения других преобразований. Метод позволяет, в отличие от [2], поддержать полиморфные вариантные типы языка OCaml.

Метод

Предложенный метод применяет обобщенное программирование к аннотированным объявлениям типов данных и порождает во время компиляции следующие сущности.

[1] https://devblogs.microsoft.com/dotnet/introducing-c-source-generators (дата обращения: 21.09.2021).

[2] https://github.com/ocaml-ppx/ppx_deriving (дата обращения: 21.09.2021).

[3] https://github.com/JetBrains-Research/GT/tree/IFMO-2021 (дата обращения: 21.09.2021).

— Базовый виртуальный класс, который используется как общий предок для всех преобразований; один на каждый тип.
— По одному конкретному классу на каждый вид требуемого преобразования.
— Обобщенная функция преобразования (*generic catamorphism, gcata*) — одна на каждый тип данных. Для поддержки полиморфных вариантных типов потребуются незначительные изменения в типизации классов и обобщенной функции.
— С помощью объектов-преобразований и *gcata* можно получать функции-преобразования со стандартным для обобщенного программирования типом. В случае взаимно-рекурсивных определений типов получение функций-преобразований несколько усложнится, так как приходится вводить комбинатор неподвижной точки для группы типов.
— Структура данных typeinfo, которая объединяет в себе сущности, построенные на предыдущих шагах, для последующего использования.

В базовом классе объявлено некоторое количество виртуальных методов, *по одному методу на каждый конструктор* алгебраического типа. Рассмотренные в работе [2] подходы требуют объявления большего количества виртуальных методов, что иногда раскрывает детали реализации, которые программист пожелал скрыть, сделав тип абстрактным. Объявление слишком узкого интерфейса для объектов несет риск построения недостаточно выразительных преобразований. В завершении работы метод проверен на примерах, чтобы показать, что способ проектирования интерфейсов объектов позволяет получать достаточно разнообразные преобразования.

Опишем две наиболее отличительные части предлагаемого метода: типизацию преобразований и обобщенную функцию преобразования.

Типы преобразований

Метод построения расширяемых преобразований предполагает их представление с помощью классов и объектов, из которых можно получать функции-преобразования. Функции-преобразования простых типов используются при построении классов для сложных типов данных, поэтому начнем с описания типов функций-преобразований.

Основываясь на идее описания катаморфизмов [3] с помощью атрибутных грамматик [4–6] рассмотрим функции-преобразования вида $\lceil \to t \to \rfloor$, где t — тип, значения которого преобразуются, $\lceil$ и $\rfloor$ — типы *наследуемых* и *синтезируемых* атрибутов. Для описания алгоритмической части преобразований атрибутные грамматики использоваться не будут, переиспользуется только терминология для описания типов.

Обозначим с помощью $\{...\}$ множественное вхождение сущности в скобках, а с помощью $\langle t \rangle$ — некоторую модификацию имен, которая необходима для предотвращения коллизий имен, но несущественна для описываемого в работе метода.

Если тип t является n-параметрическим, то преобразование тоже будет n-параметрическим. С помощью введенной нотации следующим образом описывается обобщенная форма преобразований: $\{\lceil_i \to \to \langle_i \to \rfloor_i\} \to \lceil \to \{\langle_i\}\ t \to \rfloor$. Данная форма состоит из n функций-преобразований типовых параметров и функции-преобразования непосредственно типа `t`. Все используемые функции-преобразования действуют на соответствующие наследуемые атрибуты и возвращают синтезированные атрибуты. Общий для всех преобразований класс-предок для n-параметрического типа будет иметь $3 \times (n + 1)$ типовых параметров: тройка $\lceil_\langle, \langle, \rfloor_\langle$ для каждого типового параметра $\langle_i$, где $\lceil_i$ и $\rfloor_i$ — типовые переменные наследуемого и синтезированного атрибутов для преобразования $\langle_i$; пара дополнительных типовых переменных ι и σ для представления наследуемого и синтезированного атрибутов преобразуемого типа; дополнительная типовая переменная $\sum$, которая приравнивается к *открытому типу* $[> \{\langle_i\}\ t]$ для полиморфных вариантных типов, и к $\{\langle_i\}\ t$ для остальных.

Например, если дан двухпараметрический тип $(\langle, ®)\ t$, то заголовком общего класса-предка будет `class virtual` $[\lceil_\langle, \langle, \rfloor_\langle, \lceil_®, ®, \rfloor_®, \lceil, \sum, \rfloor]\ \langle t \rangle$.

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: $f_i : \lceil_i \to \langle_i \to \rfloor_i$;
— одна функция для реализации открытой рекурсии: *fself*: $\lceil \to \sum \to \rfloor$.

Отметим, что данные условия поддерживаются для всех типов, но для части типов некоторые компоненты могут быть излишни, например, *fself* нужен только для рекурсивных типов. Объяснение этому простое: если *используется* некоторый тип, то в общем случае его определение не известно. Следовательно, для поддержки раздельной компиляции интерфейсы всех сущностей должны иметь общую структуру.

Также необходимо описать сигнатуры методов класса для алгебраического типа данных. Метод для конструктора «`C of` a_1 `×` a_2 `× ... ×` a_k» имеет следующую сигнатуру:

$$\texttt{method virtual}\ \langle \mathrm{C} \rangle : \lceil \to \sum \to a_1 \to a_2 \to \ldots \to \to\ a_k \to \rfloor.$$

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 путем наследования от общего класса-предка. В большинстве случаев преобразование реализуется путем небольшой специализации конкретного преобразования или с использованием системы расширений библиотеки GT. В первом случае многие типовые параметры будут уже специализированы (например, для show большинство типовых параметров конкретизируется в базовые типы). Во втором система плагинов упрощает процесс правильной конкретизации типовых параметров.

Обобщенная функция преобразования

Обобщенная функция преобразования позволяет запускать преобразования, представленные как объекты, а также преобразования, расширенные путем наследования. Преобразования типов, использованных при объявлении типа $\langle t \rangle$, не попадают в интерфейсы объектов. Такой дизайн выбран для того, чтобы не предоставлять детали реализации типа в интерфейсе объекта-преобразования. Для каждого конкретного объявления типа интерфейс может быть описан универсальным образом. Другие подходы, например [3], реализуют в виде методов объектов и обобщенную функцию преобразования, и преобразования типовых параметров.

Для типа $\{\langle_i\}\ t$ обобщенная функция преобразования имеет вид:

$$\texttt{val}\ \langle \texttt{gcata}_t \rangle : [\{\lceil_{\langle i}, \langle_i, \lceil_{\langle i}\}, \lceil, \sum, \lceil] \#\langle t \rangle \to \lceil \to \{\langle_i\}\ t \to \lceil.$$

Схема реализации для типа с m конструкторами следующая:

```
let ⟨gcata_t⟩ tr  ⌈ subj = match subj with
| C_1 x_1 ... x_n1 → tr#⟨C1⟩ ⌈ subj x_1 ... x_n1
| ...
| C_m x_1 ... x_nm → tr#⟨C_m⟩ ⌈ subj x_1 ... x_nm.
```

Функция принимает объект, представляющий преобразование, у которого типовые параметры, полученные путем наследования от базового класса, соответствующим образом конкретизированы. Также она принимает наследуемый атрибут, преобразуемое значение, и возвращает синтезируемый атрибут.

Полиморфные вариантные типы

Поддержка полиморфных вариантных типов [7, 8] — важная часть настоящей работы, так как она позволяет композиционально объявлять типы данных, а также строить композициональные преобразования. Главным отличием между полиморфным вариантным и алгебраическим типами является возможность расширения без объявления нового типа данных объявленных ранее полиморфных вариантных типов двумя способами: либо добавлением новых конструкторов, либо объединением нескольких типов в один. Если несколько полиморфных вариантных типов будут скомбинированы, то нужно уметь получать все обобщенные возможности простым наследованием классов.

Для поддержки полиморфных вариантов в сигнатуры объектов введем дополнительный тип $\sum$. Для обычных типов он приравнивается к типу преобразуемых значений $\{\langle_i\}\ t$, а для полиморфных вариантных — *открытому* типу преобразуемых значений, который допускает те же значения, что и $\{\langle_i\}\ t$, а также другие. Это позволяет получать объекты-преобразования объединений типов с помощью наследования и явного указания вместо $\sum$ типа преобразуемых значений.

При использовании обобщенной функции преобразования для полиморфных вариантов переданные значения необходимо обрабатывать так, чтобы они соответствовали используемому вместо ε типу.

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

Тонкостью является применение методов объекта-преобразования, к открытой разновидности типа, в то время как обобщенная функция-преобразование принимает замкнутый тип.

Если несколько полиморфных вариантных типов объединяются, то обобщенная функция-преобразование сопоставляет значение с образцами-типами и передает управление соответствующим обобщенными функциям преобразования.

Ограничения

Поддерживаются объявления типов в языке OCaml со следующими ограничениями:

1) только регулярные алгебраические типы данных; обобщенные [9] обрабатываются как обычные алгебраические типы;
2) ограничения на типы (constraints) не учитываются;
3) расширяемые алгебраические типы данных (“..”/“=”) не поддерживаются;
4) объекты, модули и типы с ключевым словом “nonrec” не поддерживаются.

Пункты 1–3 являются стандартными ограничениями для обобщенного программирования на OCaml. Ограничение п. 4 возникает потому, что OCaml не позволяет описывать классы с одинаковым именем в одной области видимости. Опыт программирования авторов работы показал, что необходимость в ключевом слове “nonrec” возникает редко, поэтому решение данной проблемы было отложено на будущее.

Эксперименты

В ходе работы выполнены два эксперимента.

Первый эксперимент[1] проведен для определения скорости выполнения расширяемых преобразований по сравнению с нерасширяемыми. Были спроектированы преобразования с использованием четырех методов: нерасширяемые преобразования с использованием рекурсивных функций, расширяемые преобразования на основе данной работы, а также два метода с применением популярных библиотек.

Сначала были измерены количества преобразований, которые удастся осуществить за единицу времени, а затем они нормировались относительно базового уровня, в качестве которого выбрана производительность GT. Для замеров были выбраны два вида преобразований: копирование выражения и преобразование в текстовый формат. Эти преобразования применялись к ⌊-выражениям, состоящим из вложенных ⌊-абстракций (одного из трех видов ⌊-выражений). Количество таких абстракций определяет размер выражения. Для тестирования использовались различные размеры от 100 до 1000.

[1] https://github.com/JetBrains-Research/GT/tree/IFMO-2021/bench (дата обращения: 21.09.2021).

Второй эксперимент выполнен, чтобы определить какие виды расширяемых преобразований можно реализовывать с предлагаемым интерфейсом объектов-преобразований, а какие нет. Для ответа на этот вопрос были реализованы с использованием предложенного метода два примера преобразований типов данных.

Первым примером[1] выступает аугментация типа данных для использования в логическом или реляционном программировании [10]. Тип данных модифицируется таким образом, что в произвольных местах появляется возможность вставлять логические переменные вместо конкретных значений. Для легковесной интеграции с уже имеющимся кодом, использован ключ компилятора «-rectypes».

Вторым примером[2] является задача «Expression problem» [11]: объявление преобразований и абстрактного дерева синтаксиса некоторого языка, и дальнейшее добавление новых преобразований и узлов дерева *без изменения* уже имеющихся реализаций. Она часто применяется как «лакмусовый тест» для оценки подходов к обобщенному программированию. Встречаются различные решения этой задачи, рассмотрим подробнее использование обобщенного программирования и полиморфных вариантных типов языка OCaml.

```
type ('name, 'lam) lam = [ `App of 'lam * 'lam
                         | `Var of 'name ]
class [ 'lam , 'nless ] lam_to_nameless flam = object
      inherit [ string list, string, int
              , string list, 'lam, 'nless
              , string list, 'lam, 'nless   ] ⟨lam⟩
      method ⟨App⟩ env _ l r = `App (flam env l, flam env r)
      method ⟨Var⟩ env _ x = `Var (index env x)
end
type ('name , 'lam) abs = [ `Abs of 'name * 'lam ]
class [ 'lam , 'nless ] abs_to_nameless flam = object
  inherit [ string list, string, int
          , string list, 'lam, 'nless
          , string list, 'lam, 'nless ] ⟨abs⟩
  method ⟨Abs⟩ env _ name term =
      `Abs (flam (name :: env) term)
end
type ('name, 'lam) term = [ ('name, 'lam) lam
                          | ('name, 'lam) abs ]
type named = (string, named) term
type nameless = [ (int, nameless) lam | `Abs of nameless ]
class to_nameless (f: _ → named → nameless) = object
      inherit [ string list, named, nameless ] named_t
      inherit [ named, nameless ] lam_to_nameless f
      inherit [ named, nameless ] abs_to_nameless f
end
```

В приведенном примере независимо друг от друга описаны два типа данных для ⌊-выражений: `abs` для связывающих имена конструкций и `lam` для не связывающих. Отдельно описаны классы для преобразований этих конструкций в безымянное представление с индексами де Брёйна. Дополнительная функция `index` получает индекс переменной из окружения, которое передается как наследуемый атрибут. Отметим, что метод ⟨`Abs`⟩ возвращает значения другого типа, а не преобразуемого, так как аргументов конструктора меньше. Из объявленных типов описаны типы ⌊-выражений с именами и без. Преобразование в безымянное представление получено простым наследованием соответствующих частей.

Результаты. Результаты первого эксперимента приведены в таблице. Задачи второго эксперимента были успешно реализованы.

Таблица. Производительность других методов преобразований (расширяемых Visitors, не вполне расширяемых и стандартных) относительно GT (больше – лучше)

Table. Performance of other sorts of transformations (extensible Visitors, not entirely extensible and standard ones) relative to GT (more is better)

Вид преобразования	Метод реализации и его производительность относительно GT, %		
	Visitors	ppx_deriving_morphism	Стандарт
Копирование	+(14–17)	+(120–131)	+(292–305)
Форматирование	не значимо	+(2–3)	+(3–7)

[1] https://github.com/JetBrains-Research/GT/blob/IFMO-2021/sample/lists.ml (дата обращения: 21.09.2021).

[2] https://github.com/JetBrains-Research/GT/blob/IFMO-2021/sample/nameless.ml (дата обращения: 21.09.2021).

Обсуждение результатов

Выполненные в первом эксперименте измерения производительности показывают, что при построении расширяемых преобразований итоговая производительность уменьшается по сравнению с нерасширяемыми. Для первого преобразования (копирование), где измеряется производительность обхода выражения без полезной нагрузки, накладные расходы значительны для любого проверенного метода реализации. Для второго вида преобразований (форматирование) обход структуры ⌊-выражения занимает малую часть, и поэтому накладные расходы на использование расширяемых преобразований приемлемы. Для улучшения производительности предложенного в данной работе метода стоит рассмотреть применение к объектам оптимизаций на основе так называемого staging [12].

Стандартный метод построения преобразований вызывает код для обработки отдельных конструкторов напрямую, и поэтому показывает наилучшую производительность. Однако такие преобразования не являются расширяемыми.

Метод ppx_deriving_morphism[1] не использует представление в виде объектов, в нем нет накладных расходов на работу с таблицей виртуальных методов, и поэтому он показывает лучшую производительность, чем преобразования на основе объектов. Данный метод не вполне расширяем: не получится добавить новых «методов» (полей структуры), и поэтому в полной мере полиморфные вариантные типы не поддержать.

Предлагаемый метод и подход Visitors [2] представляют преобразования как объекты, но при этом обладают несколькими различиями. Во-первых, в Visitors обобщенная функция преобразования является методом объекта, а в рассматриваемом подходе — отдельная функция, что негативно влияет на производительность. Но если ее наивно реализовать как метод объекта, то не удастся правильно протипизировать объекты-преобразования для полиморфных вариантных типов. Во-вторых, предлагаемая в Visitors типизация объектов не позволяет протипизировать преобразования для типов данных с произвольной рекурсией (второй эксперимент, первый пример) и полиморфные вариантные типы (второй пример). В подходе Visitors типовые параметры объектов записываются короче, чем в GT, но их невозможно сгенерировать в файле интерфейса, что не очень хорошо для практик промышленной разработки программного обеспечения. В-третьих, у объектов Visitors больше методов, чем предлагается в этой работе, что иногда может стать недостатком. Например, если объявляемый тип «длина» использует тип int, то у объекта появится метод для преобразования (например, неотрицательных) значений типа int. Можно представить ситуацию, когда объявляемый тип в сигнатуре модуля абстрактный, а предоставление наружу объекта-преобразования позволит изменить находящиеся в нем значения типа int, тем самым нарушив барьер абстракции. Преимуществом такого количества методов является то, что с помощью Visitors представим класс преобразований «Scrap your boilerplate» [13–15], но мы их находим не очень полезными на практике.

[1] https://github.com/choeger/ppx_deriving_morphism (дата обращения: 21.09.2021).

Заключение

В работе представлен метод реализации преобразований значений типов данных с помощью объектно-ориентированных возможностей языка OCaml. По сравнению с реализацией с помощью функций объекты позволяют изменять уже построенные преобразования путем наследования. Для запуска преобразований вместо вызова функций вызываются методы объектов. Производительность расширяемых преобразований в практически полезных случаях несколько меньше по сравнению с нерасширяемыми преобразованиями из-за использования методов, а не функций.

Представленный метод позволяет получать и комбинировать преобразования в том числе и для полиморфных вариантных типов, которые являются отличительной особенностью языка OCaml. Авторам неизвестны подходы к построению расширяемых преобразований, которые полноценно поддерживают эти типы.

Существует несколько возможных направлений для дальнейшего развития проекта. Для снижения накладных расходов на реализацию расширяемых преобразований можно применять так называемый staging или оптимизации, специфичные для объектов в языке OCaml. Другим важным направлением развития является поддержка большего разнообразия объявлений типов. На данный момент обобщенные алгебраические типы (GADT) не поддерживаются, а использование нерегулярных типов не так удобно, как в Visitors.

Литература

1. Gibbons J. Datatype-generic programming // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 2007. V. 4719. P. 1–71. https://doi.org/10.1007/978-3-540-76786-2_1
2. Pottier F. Visitors unchained // Proceedings of the ACM on Programming Languages. 2017. V. 1. P. 1–28. https://doi.org/10.1145/3110272
3. Meijer E., Fokkinga M., Paterson R. Functional programming with bananas, lenses, envelopes and barbed wire // Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). 1991. V. 523. P. 124–144. https://doi.org/10.1007/3540543961_7

References

1. Gibbons J. Datatype-generic programming. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2007, vol. 4719, pp. 1–71. https://doi.org/10.1007/978-3-540-76786-2_1
2. Pottier F. Visitors unchained. *Proceedings of the ACM on Programming Languages*, 2017, vol. 1, pp. 1–28. https://doi.org/10.1145/3110272
3. Meijer E., Fokkinga M., Paterson R. Functional programming with bananas, lenses, envelopes and barbed wire. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 1991, vol. 523, pp. 124–144. https://doi.org/10.1007/3540543961_7

4. Knuth D.E. Semantics of context-free languages // Mathematical Systems Theory. 1968. V. 2. N 2. P. 127–145. https://doi.org/10.1007/BF01692511
5. Rendel T., Brachthauser J.I., Ostermann K. From object algebras to attribute grammars // ACM SIGPLAN Notices. 2014. V. 49. N 10. P. 377–395. https://doi.org/10.1145/2714064.2660237
6. Viera M., Swierstra S.D., Swierstra W. Attribute grammars fly first-class: how to do aspect oriented programming in haskell // ACM SIGPLAN Notices. 2009. V. 44. N 9. P. 245–256. https://doi.org/10.1145/1631687.1596586
7. Garrigue J. Programming with Polymorphic Variants // Proc. ACM SIGPLAN Workshop on ML. 1998.
8. Garrigue J. Code reuse through polymorphic variants // Proc. Workshop on Foundations of Software Engineering. 2000.
9. Jones S.P., Vytiniotis D., Weirich S., Washburn G. Simple unification-based type inference for GADTs // ACM SIGPLAN Notices. 2006. V. 41. N 9. P. 50–61. https://doi.org/10.1145/1160074.1159811
10. Kosarev D., Boulytchev D. Typed embedding of a relational language in OCaml // Electronic Proceedings in Theoretical Computer Science, EPTCS. 2018. V. 285. P. 1–22. https://doi.org/10.4204/eptcs.285.1
11. Wadler P. The Expression Problem [Электронный ресурс]. URL: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt, свободный. Яз. англ. (дата обращения: 26.07.2021).
12. Yallop J. Staged generic programming // Proc. of the ACM on Programming Languages. 2017. V. 1. P. 1–29. https://doi.org/10.1145/3110273
13. Lämmel R., Jones S.P. Scrap your boilerplate: A practical design pattern for generic programming // ACM SIGPLAN Notices. 2003. V. 38. N 3. P. 26–37. https://doi.org/10.1145/640136.604179
14. Lämmel R., Jones S.P. Scrap more boilerplate: Reflection, zips, and generalised casts // ACM SIGPLAN Notices. 2004. V. 39. N 9. P. 244–255. https://doi.org/10.1145/1016850.1016883
15. Boulytchev D., Mechtaev S. Efficiently Scrapping Boilerplate Code in OCaml // Workshop on ML. 2011.

4. Knuth D.E. Semantics of context-free languages. *Mathematical Systems Theory*, 1968, vol. 2, no. 2, pp. 127–145. https://doi.org/10.1007/BF01692511
5. Rendel T., Brachthauser J.I., Ostermann K. From object algebras to attribute grammars. *ACM SIGPLAN Notices*, 2014, vol. 49, no. 10, pp. 377–395. https://doi.org/10.1145/2714064.2660237
6. Viera M., Swierstra S.D., Swierstra W. Attribute grammars fly first-class: how to do aspect oriented programming in Haskell. *ACM SIGPLAN Notices*, 2009, vol. 44, no. 9, pp. 245–256. https://doi.org/10.1145/1631687.1596586
7. Garrigue J. Programming with Polymorphic Variants. *Conference ACM SIGPLAN Workshop on ML*, 1998.
8. Garrigue J. Code reuse through polymorphic variants. *Proc. Workshop on Foundations of Software Engineering*, 2000.
9. Jones S.P., Vytiniotis D., Weirich S., Washburn G. Simple unification-based type inference for GADTs. *ACM SIGPLAN Notices*, 2006, vol. 41, no. 9, pp. 50–61. https://doi.org/10.1145/1160074.1159811
10. Kosarev D., Boulytchev D. Typed embedding of a relational language in OCaml. *Electronic Proceedings in Theoretical Computer Science, EPTCS*, 2018, vol. 285, pp. 1–22. https://doi.org/10.4204/eptcs.285.1
11. Wadler P. *The Expression Problem*. Available at: https://homepages.inf.ed.ac.uk/wadler/papers/expression/expression.txt (accessed: 26.07.2021).
12. Yallop J. Staged generic programming. *Proc. of the ACM on Programming Languages*, 2017, vol. 1, pp. 1–29. https://doi.org/10.1145/3110273
13. Lämmel R., Jones S.P. Scrap your boilerplate: A practical design pattern for generic programming. *ACM SIGPLAN Notices*, 2003, vol. 38, no. 3, pp. 26–37. https://doi.org/10.1145/640136.604179
14. Lämmel R., Jones S.P. Scrap more boilerplate: Reflection, zips, and generalised casts. *ACM SIGPLAN Notices*, 2004, vol. 39, no. 9, pp. 244–255. https://doi.org/10.1145/1016850.1016883
15. Boulytchev D., Mechtaev S. Efficiently Scrapping Boilerplate Code in OCaml. *Workshop on ML*, 2011.

Авторы

Косарев Дмитрий Сергеевич — программист, ООО «Интеллиджей ЛАБС», Санкт-Петербург, 197374, Российская Федерация, SC 57205453654, https://orcid.org/0000-0002-6773-5322, dmitry.kosarev@jetbrains.com

Булычев Дмитрий Юрьевич — кандидат физико-математических наук, доцент, Санкт-Петербургский государственный университет, Санкт-Петербург, 199034, Российская Федерация, SC 23391707100, https://orcid.org/0000-0001-8363-7143, dboulytchev@math.spbu.ru, dboulytchev@gmail.com

Authors

Dmitry S. Kosarev — Software Developer, IntelliJ Labs Co. Ltd., Saint Petersburg, 197374, Russian Federation, SC 57205453654, https://orcid.org/0000-0002-6773-5322, dmitry.kosarev@jetbrains.com

Dmitry Yu. Boulytchev — PhD, Associate Professor, Saint Petersburg State University, Saint Petersburg, 199034, Russian Federation, SC 23391707100, https://orcid.org/0000-0001-8363-7143, dboulytchev@math.spbu.ru, dboulytchev@gmail.com

Статья поступила в редакцию 11.06.2021
Одобрена после рецензирования 09.09.2021
Принята к печати 02.10.2021

Received 11.06.2021
Approved after reviewing 09.09.2021
Accepted 02.10.2021