Правительство Российской Федерации

Федеральное государственное бюджетное образовательное учреждение
высшего профессионального образования

«Санкт-Петербургский государственный университет»

Кафедра системного программирования

Терешин Роман Юрьевич

Объектно-ориентированные трансформеры для OCaml: анаморфизмы

Бакалаврская работа

Допущена к защите.
Зав. кафедрой:
д. ф.-м. н., профессор Терехов А. Н.

Научный руководитель:
к. ф.-м. н., доцент Булычев Д. Ю.

Рецензент:
аспирант Подкопаев А. В.

Санкт-Петербург
2015

SAINT PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Roman Tereshin

Object-oriented transformers for OCaml: anamorphisms

Bachelor's Thesis

Admitted for defence.
Head of the chair:
professor Andrey Terekhov

Scientific supervisor:
PhD Dmitri Boulytchev

Reviewer:
Graduate Assistant Anton Podkopaev

Saint Petersburg
2015

Оглавление

Введение **4**

1. Постановка задачи **7**

2. Предметная область и связанные работы **8**

2.1. Deriving-механизм языка Haskell 8

2.2. Метапрограммирование для Haskell 9

2.3. Библиотека обобщённого программирования для OCaml 9

2.4. Библиотека управляемых типом синтаксических расширений OCaml . . 10

3. Семантика и интерфейс анаморфизмов **11**

4. Генерация шаблонного кода **15**

5. Использование **17**

6. Апробация **18**

Заключение **20**

Список литературы **21**

Введение

Один из критериев качества инструментов и методов разработки программного обеспечения — отсутствие необходимости многократного указания одной и той же информации, например, программного кода, в разных частях проекта и последующей синхронизации этих частей при внесении изменений. Факт дублирования кода не всегда очевиден, так как повторное вхождение может представлять собой не точную его копию, а код, выводимый из существующего. Частный случай такого вывода — управляемая типом генерация. Как правило, это автоматическая генерация кода, реализующего некоторую функциональность по отношению к типу данных, по определению последнего. Фокус данной работы сосредоточен на подобной поддержке алгебраических типов данных (Algebraic Data Types, ADT) и их трансформеров (преобразователей).

Алгебраические типы данных позволяют описывать графоподобные структуры данных, а также их преобразователи, в основном с помощью сопоставления с образцом их фрагментов. Такие структуры и преобразования нередко составляют существенную часть реализации различных языковых процессоров, таких как компиляторы или статические анализаторы кода, хотя их применение не ограничивается языковыми инструментами. Встроенной языковой поддержкой алгебраических типов данных обладают в первую очередь функциональные языки со статической типизацией и полиморфизмом Хиндли-Милнера, такие как Haskell[1] и диалекты языка ML, в том числе OCaml[2].

Язык программирования OCaml обладает рядом уникальных для функциональных языков возможностей, например, поддержкой полиморфных вариантов и объектно–ориентированного программирования. Существующий инструментарий OCaml включает препроцессор Camlp5[3], который позволяет расширять язык новыми конструкциями. По этим причинам в рамках данной работы мы рассмотрим поддержку алгебраических типов данных в OCaml.

Эмпирически установлено, что большинство трансформеров алгебраических типов данных попадает в одну из следующих категорий: катаморфизмы, анаморфизмы, хиломорфизмы и другие [4, 2]. Под катаморфизмом в функциональном программировании понимают обобщение так называемой свёртки списков на произвольные алгебраические типы данных, в нотации OCaml тип такого преобразователя может иметь вид:

$$(\texttt{'b} \to \texttt{'a} \to \texttt{'b}) \to \texttt{'b} \to \texttt{'a list} \to \texttt{'b}$$

Приведём несколько примеров катаморфизмов, приблизительно в порядке увеличения

[1] `https://www.haskell.org/` (дата обращения: 25.05.2015)
[2] `http://ocaml.org/` (дата обращения: 25.05.2015)
[3] `http://camlp5.gforge.inria.fr/` (дата обращения: 25.05.2015)

сложности:

- вычисление суммы элементов списка чисел;
- вычисление синтезируемых и наследуемых атрибутов деревьев разбора в соответствии с атрибутной грамматикой, например, вычисление значения абстрактного синтаксического дерева выражения в том или ином языке программирования или получение его текстового представления;
- сравнение на равенство двух экземпляров ADT;
- сериализация данных.

Анаморфизм — это преобразование, обратное катаморфизму. Здесь мы будем понимать под анаморфизмом генерацию экземпляра алгебраического типа данных по некоторому порождающему значению. Тип анаморфизма, порождающего список произвольного типа, может выглядеть следующим образом:

$$(\texttt{'b} \to (\texttt{'a * 'b})\ \texttt{option}) \to \texttt{'b} \to \texttt{'a list}$$

Первый аргумент такого трансформера является функцией, вычисляющей по порождающему значению очередной элемент генерируемого списка и новое порождающее значение, используемое для генерации остальной части списка, либо специальное значение, означающее конец генерации. Приведём несколько примеров анаморфизмов:

- генерация списка чисел от 1 до n, здесь n — порождающее значение;
- восстановление бинарного дерева поиска по его прямому (pre-order) обходу;
- генерация случайных структур данных с некоторым наперёд заданным свойством, например, абстрактных синтаксических деревьев выражений с определённым значением;
- десериализация данных.

Преобразователи допускают комбинирование. Например, вычисление значения выражения по его текстовому представлению может быть выполнено последовательным применением анаморфизма (построение абстрактного синтаксического дерева) и катаморфизма (вычисление его значения). В частности, приведённая здесь комбинация трансформеров является хиломорфизмом.

Существующая поддержка алгебраических типов данных в языках программирования обладает известным недостатком: описания преобразователей в её рамках содержат большое количество шаблонного, конвенционального кода. Наивные реализации преобразователей трудно переиспользовать или модифицировать, не прибегая

к полнотекстовому копированию. Оказалось, тем не менее, что существенная часть шаблонного кода может быть сгенерирована автоматически по определению ADT.

С целью расширения поддержки алгебраических типов данных и их преобразователей в языке OCaml Д. Ю. Булычевым была разработана инфраструктура управляемых типом объектно-ориентированных трансформеров [4]. Инфраструктура состоит из синтаксичекого расширения языка OCaml и сопутствующей библиотеки поддержки. Она выполняет управляемую типом генерацию шаблонного кода трансформеров. Реализация трансформеров является объектно–ориентированной и допускает переиспользование и частичную модификацию существующих преобразователей. Инфраструктура является расширяемой и включает реализацию ряда типовых преобразователей, например, генераторов текстового представления экземляров алгебраических типов данных [8]. Однако оригинальное решение поддерживает только один тип преобразователей — катаморфизмы. В данной работе было осуществено расширение описанного решения.

[4] `https://code.google.com/p/generic-transformers/` (дата обращения: 25.05.2015)

1. Постановка задачи

Целью данной работы является расширение покрытия описанной инфраструктурой управляемых типом трансформеров множества преобразователей путём добавления поддержки анаморфизмов. Перечислим ключевые задачи, которые решает данная работа:

- анализ существующих подходов к поддержке анаморфизмов в функциональных языках программирования;
- разработка семантики и интерфейса обобщённых анаморфизмов для алгебраических типов данных OCaml;
- реализация автоматической генерации шаблонного кода для анаморфизмов и интеграция с существующей инфраструктурой;
- апробация разработанного решения и анализ результатов.

2. Предметная область и связанные работы

Обобщенное программирование (generic programming) предлагает форму полиморфизма, промежуточную по отношению к параметрическому и т.н. ad-hoc полиморфизму [6]. Поведение параметрически полиморфной функции, такой как `List.length`, идентично для всех возможных значений типового параметра. Поведение ad-hoc полиморфной функции, например, бинарного арифметического оператора в C-подобном языке, различно для каждого из типов, для которых она определена. Поведение обобщённой функции, например предиката равенства в OCaml или Haskell, также меняется от типа к типу, но зависимым от определения типа образом. Форма (shape) определений обобщённых фукнций часто повторяет форму определений типов её агрументов [2].

Система типов предоставляет способ описать некоторые семантические свойства значений языка. В частности, тип структуры данных можно рассматривать как его семантику. Чем более выразительна система типов, тем больше свойств значения может быть определено по его типу. В языках с богатой системой типов, к которым без сомнения относится OCaml, семантика некоторых обобщённых функций может быть частично или полностью определена автоматически по определениям типов её операндов. Техника управляемого типом обобщённого программирования позволяет осуществлять такой автоматический вывод определений функций, или трансформеров, по определению типа трансформируемого значения.

Ниже мы рассмотрим несколько существующих подходов к управляемому типом обобщённому программированию в различных языках и поддержку ими анаморфизмов алгебраических типов данных.

2.1. Deriving-механизм языка Haskell

Спецификация функционального языка Haskell [3] содержит описание ряда выводимых экземпляров классов (derived instances) и ключевого слова deriving, известных как deriving-механизм. Этот механизм позволяет осуществлять автоматическую генерацию кода ряда функций, оперирующих алгебраическими типами данных, удовлетворяющих тем или иным требованиям, зависящим от класса (набора функций). Спецификация языка [3] определяет генерацию кода для ограниченного набора классов: `Eq`, `Ord`, `Enum`, `Bounded`, `Show` и `Read`. Deriving-механизм является типичным примером управляемой типом генерации кода, так как соответствующие экземпляры могут быть выведены компилятором по определению типов. Из перечисленных классов только `Enum` и `Read` содержат функции, являющиеся примерами анаморфизмов, например,

```
enumFromTo :: Enum a => a → a → [a]
```

класса Enum, осуществляющая генерацию списка последовательных элементов перечисления в заданных границах, и

```
read :: Read a => String → a
```

класса Read, выполняющая десериализацию экземпляра алгебраического типа данных из его строкового представления.

Deriving-механизм в стандартном Haskell не предусматривает расширение набора поддерживаемых классов пользователем, упомянутые анаморфизмы являются конкретными трансформерами и не допускают параметризации или модификации. В частности, представление алгебраических типов данных, используемое десериализатором Read, фиксировано разработчиками реализации языка. Совместимость реализаций deriving-механизма между различными компиляторами Haskell не гарантируется, так как его описание в спецификации языка неформально и неполно.

2.2. Метапрограммирование для Haskell

В работе [5] описано языковое расширение Template Haskell, поддерживаемое на текущий момент одним из самых популярных компиляторов Haskell GHS (Glasgow Haskell Compiler)[5]. Возможности, добавляемые им, принадлежт категории т.н. метапрограммирования [2]. Это расширение обогащает язык системой т.н. псевдоцитирований (quasi-quotations) и вклеек (splices). Первые позволяют преобразовывать фрагменты кода на Haskell в представляющие их абстрактные синтаксические деревья — Haskell–значения алгебраического типа. Вторые — выполнять обратное преобразование и вставку сгенерированного во время компиляции Haskell–кода в произвольные точки программ на Haskell. Это расширение можно рассматривать и как шаблоны для Haskell, близкие шаблонам языка C++, и как типобезопасную систему макросов, оперирующих на уровне абстрактных синтаксических деревьев. Шаблонный код, использующий эти расширения, является типизируемым — любой генерируемый им код не может не пройти проверку типов. Template Haskell, таким образом, предоставляет широкие возможности для реализации управляемой типом генерации шаблонного кода, но как макросистема, он не предоставляет готовых расширений для поддержки преобразователей, в том числе анаморфизмов.

2.3. Библиотека обобщённого программирования для OCaml

В работе [7] описана система т.н. обобщённых функций для OCaml, реализованная как препроцессор с сопутствующей библиотекой поддержки. Система фактически является попыткой реализовать для OCaml механизм, аналогичный по своим возможностям deriving в Haskell. В виду отсутствия в OCaml абстракции, аналогичной клас-

[5] `https://www.haskell.org/ghc/` (дата обращения: 25.05.2015)

сам Haskell, она имитируется с помощью модульной системы OCaml, а также путём введения в язык специального расширения, допускающего параметризацию значений (функций соответствующих модулей) типами. Например, так может выглядеть использование сериализатора и десериализатора бинарного дерева поиска:

```
Pickle.from_string<int tree> (Pickle.to_string<int tree> bst)
```

Для реализации синтаксического расширения и генерации кода модулей в этой работе используется препроцессор Camlp4.

Множество классов, являющихся анаморфизмами, совпадает с таковым для deriving-механизма в Haskell, подход обладает в точности теми же недостатками и ограничениями, что были описаны выше.

2.4. Библиотека управляемых типом синтаксических расширений OCaml

Коммерческой компанией Jane Street была разработана для OCaml свободно распространяемая библиотека type_conv[6], основной задачей которой является унификация различных синтаксических расширений языка, выполняющих управляемую типом генерацию шаблонного кода с помощью препроцессора Camlp4. Достигаемый ею результат состоит в возможности бесконфликтного сосуществования различных расширений в языке. Среди таких расширений есть сериализатор/десериализатор sexplib[7], выполняющий генерацию преобразователей значений в S-выражения и в обратном направлении для произвольного алгебраического типа данных. Однако модификация поведения этих преобразователей, например, изменение используемого представления, невозможна.

[6] `https://github.com/janestreet/type_conv` (дата обращения: 25.05.2015)
[7] `https://github.com/janestreet/sexplib` (дата обращения: 25.05.2015)

3. Семантика и интерфейс анаморфизмов

Анаморфизм — алгоритм генерации экземпляра алгебраического типа данных по некоторому порождающему значению. Рассмотрим определение ADT в OCaml:

$$
\begin{array}{l}
\underline{\texttt{type}}\ (\texttt{'}a_1,\ \ldots,\ \texttt{'}a_m)\ \texttt{t} = \\
\quad |\ C_1\ \underline{\texttt{of}}\ p_{11}\ \texttt{*}\ \ldots\ \texttt{*}\ p_{1n_1} \\
\quad \vdots \qquad\qquad \ddots \\
\quad |\ C_k\ \underline{\texttt{of}}\ p_{k1}\ \texttt{*}\ \ldots\ \texttt{*}\ p_{kn_k}\,,
\end{array}
$$

где $\texttt{'}a_1, \ldots, \texttt{'}a_m$ — типовые параметры типа t, $C_1, \ldots, C_k$ — конструкторы вариантов типа t, $p_{i1}, \ldots, p_{in_i}$ — типы аргументов конструктора C_i. Можно предположить, что в простейшем случае при $m = 0$ анаморфизм должен иметь следующий тип:

$$\texttt{'seed} \to \texttt{t}\,,$$

где 'seed — тип порождающего значения. Однако этого недостаточно, как мы покажем далее.

Мы будем различать следующие категории типов аргументов конструкторов p_{ij} (далее для краткости “аргументов”):

- $\texttt{'}a_l$ — аргумент, совпадающий с одним из типовых параметров;
- $(\texttt{'}a_1, \ldots, \texttt{'}a_m)$ t — рекурсивный аргумент, совпадающий с определяемым типом;
- другие.

Для подавляющего большинства возникающих на практике алгебраических типов справедливо, что категория “другие” содержит только конкретные типы, не являющиеся экземплярами полиморфного типа t (например, `int`, `string list`). По этой причине, в частности, в рамках нашего решения мы не будем рассматривать нерегулярные рекурсивные аргументы как рекурсивные, где нерегулярный аргумент — экземпляр полиморфного типа t, список фактических типовых параметров которого отличен от $(\texttt{'}a_1, \ldots, \texttt{'}a_m)$.

Ясно, что для генерации экземпляра ADT необходима генерация конструктора верхнего уровня и всех его аргументов на основе порождающего значения. Наша цель — позволить пользователю определить явно только один такой шаг генерации, зафиксировав следующие компоненты алгоритма:

- выбор на основе порождающего значения того или иного конструктора типа t;
- вычисление значений, достаточных для генерации всех аргументов выбранного конструктора.

Рассмотрим в качестве примера задачу восстановления бинарного дерева поиска из списка значений его узлов в прямом порядке (pre-order). Определение такого дерева может выглядеть следующим образом:

```
type 'a bst =
  | Empty
  | Node of 'a * 'a bst * 'a bst
```

Один из возможных подходов к решению этой задачи состоит в разделении списка (порождающего значения анаморфизма) на списки, соответствующие левому и правому поддеревьям и последующем рекурсивном применении анаморфизма. Несложно увидеть, однако, что такой алгоритм будем иметь временную сложность $O(n^2)$ в худшем случае, где n — число элементов в списке. Хорошо известный алгоритм восстановления бинарного дерева поиска с временной сложностью $O(n)$ в худшем случае предполагает, что генерация дерева осуществляется по списку и максимальному значению в дереве и в качестве результата имеет также остаток списка. Именно остаток списка, в частности, должен служить частью порождающего значения правого поддерева.

Таким образом, порождающее значение, используемое для генерации рекурсивных аргументов, может зависеть от результатов генерации предыдущих аргументов. В общем случае, это справедливо для всех аргументов и зависимость может быть нетривиальной[8]. Следовательно, каждый аргумент p_{ij} должен порождаться преобразователем типа

$$\texttt{'seed} \rightarrow p_{ij}\ \texttt{*}\ \texttt{'seed},$$

а тип анаморфизма не может быть тривиальнее

$$\texttt{'seed} \rightarrow \texttt{t}\ \texttt{*}\ \texttt{'seed}.$$

Для некоторых аргументов тип преобразователя, который должен определить пользователь, может быть упрощён. Если p_{ij} — рекурсивный аргумент, он может быть сгенерирован рекурсивным применением определяемого анаморфизма. Тогда порождающий его преобразователь может иметь тип

$$\texttt{'seed} \rightarrow \texttt{'seed}.$$

Если $i = 1$, предыдущих аргументов нет и нет зависимости от результатов их генерации. Тогда типы порождающих преобразователей могут иметь вид

```
'seed
```

для рекурсивного аргумента и

```
t * 'seed
```

для нерекурсивного.

Определим в качестве типа результата одного шага генерации алгебраического типа t следующий полиморфно–вариантный тип:

[8] пример соответствующего анаморфизма описан в разделе 6 (стр. 18) данной работы

$$
\begin{array}{l}
\underline{\texttt{type}}\ (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m\,,\ \texttt{'seed})\ \texttt{state} = [\\
\quad |\ \texttt{`}C_1\ \underline{\texttt{of}}\ q_{11}\ \texttt{*}\ \ldots\ \texttt{*}\ q_{1n_1}\\
\quad \vdots\qquad\qquad \ddots\\
\quad |\ \texttt{`}C_k\ \underline{\texttt{of}}\ q_{k1}\ \texttt{*}\ \ldots\ \texttt{*}\ q_{kn_k}\\
]\,,
\end{array}
$$

где q_{ij} образованы по следующим правилам:

$$
q_{ij} = \begin{cases} [\texttt{'seed -> }]_{j>1}\ \texttt{'seed}, & \text{если } p_{ij} \text{— рекурсивный аргумент} \\ [\texttt{'seed -> }]_{j>1}\ p_{ij}\ \texttt{*}\ \texttt{'seed}, & \text{в противном случае} \end{cases}.
$$

Имена конструкторов полиморфных вариантов здесь совпадают с именами конструкторов типа t. Очевидно, использование неполиморфных вариантов невозможно в силу несовпадения типов аргументов конструкторов типов `t` и `state`.

С целью поддержать возможность комбинировать анаморфизмы, мы абстрагируем их по преобразователям всех типовых параметров. В частности, функция, реализующая один шаг генерации, имеет тип:

$$
\begin{array}{l}
\underline{\texttt{type}}\ (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m\,,\ \texttt{'seed})\ \texttt{step} =\\
\quad (\texttt{'seed} \to \texttt{'}a_1\ \texttt{*}\ \texttt{'seed}) \to \ldots \to\ (\texttt{'seed} \to \texttt{'}a_m\ \texttt{*}\ \texttt{'seed}) \to\\
\quad \texttt{'seed} \to (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m\,,\ \texttt{'seed})\ \texttt{state}.
\end{array}
$$

Код обобщённого анаморфизма мы будем генерировать автоматически по определению алгебраического типа данных. Чтобы обеспечить доступ пользовательского кода к сгенерированному преобразователю мы предлагаем использовать универсальное имя для всех анаморфизмов `unfold` и допустить его параметризацию типом ADT. Таким образом, обобщённый анаморфизм имеет тип:

$$
\begin{array}{l}
\texttt{unfold(t)}\ :\\
\quad (\texttt{'seed} \to \texttt{'}a_1\ \texttt{*}\ \texttt{'seed}) \to \ldots \to\ (\texttt{'seed} \to \texttt{'}a_m\ \texttt{*}\ \texttt{'seed}) \to\\
\qquad (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m\,,\ \texttt{'seed})\ \texttt{step} \to\\
\qquad \texttt{'seed} \to (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m)\ \texttt{t}\ \texttt{*}\ \texttt{'seed}.
\end{array}
$$

Для удобства использования также будем генерировать преобразователь

$$
\begin{array}{l}
\texttt{unfold'(t)}\ :\\
\quad (\texttt{'seed} \to \texttt{'}a_1\ \texttt{*}\ \texttt{'seed}) \to \ldots \to\ (\texttt{'seed} \to \texttt{'}a_m\ \texttt{*}\ \texttt{'seed}) \to\\
\qquad (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m\,,\ \texttt{'seed})\ \texttt{step} \to\\
\qquad \texttt{'seed} \to (\texttt{'}a_1\,,\ \ldots,\ \texttt{'}a_m)\ \texttt{t}.
\end{array}
$$

В качестве примера использования приведём реализацию описанного выше анаморфизма для генерации бинарного дерева поиска по его прямому обходу:

```
open GT

let step = fun _ → function
```

```
  | ([], _) → `Empty
  | (x :: xs, max) →
      if x > max
      then `Empty
      else `Node ((x, (xs, x)), pass, fun (tail, _) → (tail, max))

let of_preorder preorder = unfold'(bst) () step (preorder, max_int)
```

В этом примере генерация узла дерева (целого типа) из элемента обхода тривиальна и дополнительный параметр step–функции, соответствующий типовому параметру, не используется. Вспомогательная функция `pass` определена в библиотеке преобразователей `GT` и является тождественной функцией.

4. Генерация шаблонного кода

Код обобщённого анаморфизма `unfold(t)` (а также `unfold'(t)`) может быть сгенерирован автоматически по определению типа `t`. Например, для типа бинарного дерева поиска `'a bst`, приведённого в примерах выше, этот код может иметь следующий вид:

```
let <автоматически сгенерированное имя> a step seed =
  let rec unfold s =
    match step a s with
      `Null → Null, s
    | `Node (arg0, arg1, arg2) →
        let (arg0_unfolded, after_arg0) = arg0 in
        let (arg1_unfolded, after_arg1) = unfold (arg1 after_arg0) in
        let (arg2_unfolded, after_arg2) = unfold (arg2 after_arg1) in
        Node (arg0_unfolded, arg1_unfolded, arg2_unfolded), after_arg2
  in
  unfold seed
```

Реализация управляемой типом генерации выполнена с помощью препроцессора Camlp5 (Caml PreProcessor-Pretty-Printer). Camlp5 включает синтаксический анализатор OCaml, который может быть расширен новыми синтаксическими конструкциями или преобразованиями синтаксических деревьев. Camlp5–расширения написаны на специальном надмножестве языка OCaml. Это надмножество, помимо стандартного OCaml, включает внутренний предметно-ориентированный язык (DSL) т.н. цитирований (quotations) и антицитирований (antiquotations), которые позволяют легко переключаться между объектным языком и метаязыком, используя синтаксис OCaml для обоих. Camlp5 можно рассматривать как систему макросов, уровень абстракции которой занимает промежуточное положение между макросами языков C и Scheme. В отличие от директивы define препроцессора C, расширения Camlp5 оперируют абстрактными синтаксическими деревьями, а не последовательностями токенов; в отличие от макросов, реализованных с помощью синтаксических правил (syntax-rules) языка Scheme, расширения Camlp5 могут осуществлять захват переменных.

Приведём фрагмент формальной грамматики в расширенной форме Бэкуса-Наура [1], используемой синтаксическим анализатором OCaml, соответствующий определениям алгебраических типов данных:

⟨*typeDefinition*⟩ ::= `type` ⟨*typedef*⟩

⟨*typedef*⟩ ::= [⟨*typeParams*⟩] ⟨*typeConstrName*⟩ ⟨*typeRepresentation*⟩

⟨*typeRepresentation*⟩ ::= '=' ['|'] ⟨*constrDecl*⟩ { '|' ⟨*constrDecl*⟩ }

⟨*constrDecl*⟩ ::= ⟨*constrName*⟩
| ⟨*constrName*⟩ `of` ⟨*typexpr*⟩ { * ⟨*typexpr*⟩ }

⟨*typeParams*⟩ ::= ⟨*typeParam*⟩
| ‘(’ ⟨*typeParam*⟩ { ‘,’ ⟨*typeParam*⟩ } ‘)’

Реализованное синтаксическое расширение вводит в язык аннотированное определение:

⟨*annotatedTypeDefinition*⟩ ::= ‘`@`’ `type` ⟨*typedef*⟩,

а также выполняет преобразование абстрактного синтаксического дерева, соответствующего такому определению, в абстрактное синтаксическое дерево определения обобщённого анаморфизма с некоторым автоматически сгенерированным именем.

Второе необходимое расширение — расширение синтаксической категории выражений OCaml:

⟨*expr*⟩ ::= `unfold` ‘`(`’ ⟨*typeConstrName*⟩ ‘`)`’
| `unfold'` ‘`(`’ ⟨*typeConstrName*⟩ ‘`)`’
| ...

и замена каждого вызова обобщённого анаморфизма вызовом соответствующей функции по ранее сгенерированному имени.

5. Использование

Реализованное решение представляет собой Camlp5–расширение языка и сопровождающую библиотеку. Для сборки расширения и модулей, использующих его, необходимы Camlp5 в сборке strict и компиляторы OCaml (ocamlc, ocamlopt). Расширение языка является объектным файлом pa_gt.cmo, для препроцессирования модулей, использующих наше решение, необходимо использовать команду

`camlp5o` <путь к pa_gt.cmo> `pr_o.cmo` <ваш OCaml модуль>

Для компиляции препроцессированного модуля необходимо дополнительно указать путь к библиотеке GT.ml, для редактирования связей — путь к сопровождающей библиотеке GT.cma.

Исходный код предлагаемого в данной работе решения можно найти в составе инфраструктуры управляемых типом объектно-ориентированных трансформеров по ссылке `https://github.com/ramntry/generic-transformers`.

6. Апробация

Для апробации предложенного подхода мы решим проблему десериализации абстрактного синтаксического дерева программы на некотором языке. В качестве метрики, необходимой для сравнения реализаций, использующей разработанный в данной работе подход, и не использующей его, мы выбрали простейшую LOC-метрику (число непустых строк кода реализации).

Десериализируемая программа в нашем примере будет представлять собой список токенов. В отличие от синтаксического разбора реализуемое преобразование полагается на синтаксическую корректность входной программы и, таким образом, является однозначным и всегда осуществимым. В качестве языка мы возьмем подмножество простого функционального языка SLL (Simple Lazy Language), различные модификации которого нередко используются в исследовательских работах [9], посвящённых языкам программирования, в сходных целях. Приведём грамматику SLL в расширенной форме Бэкуса-Наура:

⟨*program*⟩ ::= ‘`(`’ [⟨*functionDef*⟩ { ‘`,`’ ⟨*functionDef*⟩ }] ‘`)`’

⟨*functionDef*⟩ ::= ⟨*fFunctionDef*⟩
| ⟨*gFunctionDef*⟩

⟨*fFunctionDef*⟩ ::= ⟨*functionName*⟩ ⟨*variableList*⟩ ⟨*expr*⟩

⟨*gFunctionDef*⟩ ::= ⟨*functionName*⟩ ‘`(`’ ⟨*constructorName*⟩ ⟨*variableList*⟩ ‘`)`’ ⟨*expr*⟩

⟨*expr*⟩ ::= ⟨*variable*⟩
| ⟨*constructor*⟩
| ⟨*functionCall*⟩

⟨*constructor*⟩ ::= ⟨*constructorName*⟩ ‘`(`’ [⟨*expr*⟩ { ‘`,`’ ⟨*expr*⟩ }] ‘`)`’

⟨*functionCall*⟩ ::= ⟨*functionName*⟩ ‘`(`’ [⟨*expr*⟩ { ‘`,`’ ⟨*expr*⟩ }] ‘`)`’

⟨*variableList*⟩ ::= ‘`(`’ [⟨*variable*⟩ { ‘`,`’ ⟨*variable*⟩ }] ‘`)`’

⟨*variable*⟩ ::= `LowerId`

⟨*functionName*⟩ ::= `LowerId`

⟨*constructorName*⟩ ::= `UpperId`

Как можно видеть, грамматика языка обладает как минимум одним многократно повторяющимся фрагментом — списком элементов некоторого вида, что поощряет абстрагирование соответствующего десериализатора в отдельную сущность и его параметризацию десериализатором элементов такого списка. Тип абстрактного

синтаксического дерева для SLL может быть естественным образом выражен как алгебраический тип данных, отражающий отмеченную особенность:

```
@type 'a lst =
  | Nil
  | Cons of 'a * 'a lst

@type 'expr_list expr =
  | Var of string
  | Constr of string * 'expr_list
  | Call of string * 'expr_list

@type ('id_list, 'expr_list) def =
  | FDef of string * 'id_list * 'expr_list expr
  | GDef of string * string * 'id_list * 'expr_list expr

type expr' = expr' lst expr
type def' = (string lst, expr' lst) def
type program = def' lst
```

В качестве порождающего абстрактное синтаксичекое дерево значения мы используем список токенов типа:

```
type token =
  | Comma
  | OpenP
  | CloseP
  | LowerId of string
  | UpperId of string
```

Таким образом, тип реализуемого анаморфизма имеет вид:

```
token lst → program
```

Реализация была выполнена методом рекурсивного спуска [1] дважды: без использования и с использованием разработанного в данной работе подхода. В первом случае размер реализации — 31 строка кода, во втором случае — 20 строк. Как видно, использование автоматической генерации шаблонного кода анаморфизмов сокращает реализацию более, чем в 1.5 раза по метрике LOC.

Заключение

В рамках данной работы были достигнуты следующие результаты:

- в ходе анализа существующих подходов к поддержке анаморфизмов алгебраических типов данных (ADT) в функциональных языках программирования было показано, что несмотря на существование различных инструментов управляемого типом обобщённого программирования, поддержка обобщённых, расширяемых анаморфизмов ни одним из них не осуществляется;
- разработаны семантика и интерфейс обобщённых анаморфизмов ADT для OCaml, выразительность которых достаточна для реализации широкого класса анаморфизмов;
- реализована автоматическая, управляемая типом ADT генерация кода обобщённых анаморфизмов, выполнена интеграция с существующей инфрастурктурой объектно-ориентированных трансформеров для OCaml;
- проведена апробация разработанного решения, показавшая, что оно позволяет существенно (более, чем в полтора раза по метрике LOC) снизить сложность реализации нетривиальных анаморфизмов.

Существует несколько направлений развития данной работы. Можно рассмотреть проблемы реализации поддержки анаморфизмов для нерегулярных, а также обобщённых ADT (GADT); предоставления библиотеки типовых конкретных анаморфизмов; расширения поддержки ADT в OCaml другими классами преобразователей, таких как хиломорфизмы.

Список литературы

[1] Compilers: Principles, Techniques, and Tools (2nd Edition) / Alfred V. Aho, Monica S. Lam, Ravi Sethi, Jeffrey D. Ullman. — Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2006.

[2] Gibbons Jeremy. Datatype-generic programming // Spring School on Datatype-Generic Programming, volume 4719 of Lecture Notes in Computer Science. — Springer-Verlag.

[3] Marlow Simon. Haskell 2010 Language Report. — URL: https://www.haskell.org/definition/haskell2010.pdf.

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

[5] Sheard Tim, Jones Simon Peyton. Template Meta-programming for Haskell // SIGPLAN Not. — 2002. — . — Vol. 37, no. 12. — P. 60–75. — URL: http://doi.acm.org/10.1145/636517.636528.

[6] Strachey Christopher. Fundamental Concepts in Programming Languages // Higher Order Symbol. Comput. — 2000. — . — Vol. 13, no. 1-2. — P. 11–49. — URL: http://dx.doi.org/10.1023/A:1010000313106.

[7] 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. — URL: http://doi.acm.org/10.1145/1292535.1292548.

[8] Булычев Д.Ю. Компонентизация языковых процессоров на основе расширяемых типов данных и управляемых ими преобразователей. — 2012. — P. 69–89. — URL: http://www.sysprog.info/2012/04.pdf.

[9] Ключников И.Г. Суперкомпиляция: идеи и методы. — 2011. — P. 167–203. — URL: http://fprog.ru/2011/issue7/practice-fp-7-screen.pdf.