Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Реализация подхода Scrap Your Boilerplate для Ocaml

Курсовая работа студента 445 группы

Мечтаева Сергея Владимировича

Научный руководитель _____ Д. Ю. Булычев

к.ф-м.н., доцент

Санкт-Петербург

2010

Содержание

Введение

Цель

Целью работы является реализация подхода Scrap Your Boilerplate для языка Ocaml.

Scrap Your Boilerplate

Scrap Your Boilerplate [1] - это паттерн проектирования, предназначенный для обработки сложных структур данных. Он избавляет программиста от необходимости писать "boilerplate" код[1], позволяя один раз описать функции обхода для всех типов данных.

Суть подхода

Суть подхода можно показать на примере оригинальной реализации на языке Haskell. В ней используются некоторые особенности Haskell: классы типов и безопасное приведение.

Представим, что мы создаем программное обеспечение, обрабатывающее данные компаний (лист. 1):

```
data Company  = C [Dept ]
data Dept     = D Name Manager [Unit]
data Unit     = PU Employee | DU Dept
data Employee = E Person Salary
data Person   = P Name Address
data Salary   = S Float
type Manager  = Employee
type Name     = String
type Address  = String
```

Листинг 1: Представление компании

Предположим, что необходимо увеличить зарплаты всех сотрудников на 10%. Напишем функцию $increase(k, company)$, умножающую каждое значение зарплаты в компании $company$ на $(1 + k)$:

[1]boilerplate code — шаблонный код, не решающий непосредственно задачи программы

```
increase :: Float -> Company -> Company
increase k (C ds) = C (map (incD k) ds)
incD k (D nm mgr units) = D nm (incE k mgr) (map incU k units)
incU k (PU person) = PU (incE k person)
incU k (DU dept) = DU (incD k dept)
incE k (E person salary ) = E person (incS k salary)
incS k (S salary ) = S (s * (1 + k))
```

Листинг 2: Увеличение зарплаты

Такая реализация содержит яркий пример “boilerplate” кода (содержательной в ней является только последняя строка), и ей характерен ряд существенных недостатков:

- Для каждой подобной функции нужно написать большой по объему код, что вызывает неудобство и отвлекает от сути задачи
- Повышается вероятность ошибки
- При изменении структуры данных необходимо изменять все такие функции, т.к. они содержат алгоритм обхода

Частичным решением проблемы является функция *fold_right* [2] для структуры *Company*: её использование уменьшило бы объем кода при обходе данных, но все равно пришлось бы определять поведение для каждого узла (*Company Dept*, *Unit*, *Employee*), хотя их требуется просто тождественно отобразить.

Авторы Scrap Your Boilerplate предлагают следующее решение (лист. 3):

```
increase :: Float -> Company -> Company
increase k = everywhere (mkT (incS k))
```

Листинг 3: Решение проблемы

Здесь используются функции:

incS

Содержательная часть алгоритма увеличения зарплаты (лист. 2)

everywhere

Для каждого узла дерева данных *everywhere* f применяет f ко всем его дочерним узлам, затем из отображенных узлов конструируется новый узел того

[2] *fold_right* для произвольных структур данных называют также катаморфизмом [3]

же типа, что и исходный, к нему тоже применяется f и результат этого применения возвращается. Для реализации *everywhere* используется класс типов, на котором определяется функция $gmapT\ f\ x$, применяющая f к детям x

mkT

Делает из своего аргумента полиморфную функцию: проверяет с помощью класса *Typeable*, можно ли применить функцию к значению, и, если можно, то применяет, иначе — отображает значение тождественно.

Используя эти функции можно делать преобразование данных. Обобщим их, чтобы появилась возможность решать более широкий круг задач: запросы, поиск и т.д. Естественным выглядит обобщение *gmapT* до функции $gfoldl$, такой что $gfoldl\ k\ z$ узлу $Node\ child_1\ child_2$ будет сопоставлять $(z\ Node\ 'k'\ child_1)\ 'k'\ child_2$. Тогда *gmapT* выражается через $gfoldl$ следующим образом (лист. 4):

```
gmapT f = gfoldl k id
  where
    k c x = c (f x)
```

Листинг 4: Выражение *gmapT* через $gfoldl$

Релизация для Ocaml

Полноценной реализации для Ocaml не существует. Одной из причин этого является отсутствие в Ocaml классов типов. Но есть средства, которые частично реализуют возможности этого подхода с помощью расширения синтаксиса. О них будет рассказано в следующей части.

Требования

Требовалось создать такую интерпретацию Scrap Your Boilerplate, которая обеспечивала бы основные возможности этого подхода и не использовала бы расширение систаксиса. Последнее ограничение было вызвано желанием использовать полученное средство внутри другого расширения синтаксиса (композиция двух расширений невозможна).

1 Обзор существующих средств

Deriving

Deriving - это расширение синтаксиса, позволяющее получить функцию по декларации типа. Работа с ним строится в стиле классов типов, хотя оно не добавляет нового уровня абстракции, а является синтаксическим сахаром. Для создания функции необходимо написать генераторы для базовых типов (например, *int*, *char*, *string*, вариантов, списков, типов с параметрами и т.д.), тогда она будет действовать над классом, который строится следующим образом:

1. Базовые типы принадлежат классу
2. Если типы T_1, T_2, ..., T_n входят в класс C, то если к декларации типа, образованного комбинацией T_1, T_2, ..., T_n, применено расширение синтаксиса (как в лист. 5), то он тоже входит в C.

Далее приведен пример использования deriving для создания функции *show*[3], которая определена на стандартном классе *Show*, для пользовательского типа *tree* (лист. 5).

```
type 'a tree = Leaf of 'a | Branch of 'a tree * 'a * 'a tree
    deriving (Show)
```

Листинг 5: Создание функции *show* с помощью deriving

Для того, чтобы использовать эту функцию, нужно воспользоваться следующей синтаксической конструкцией (лист. 6)

```
let some_tree = Branch (Leaf 1, 2, Branch (Leaf 3, 4, Leaf 5))
let _ = print_string (Show.show<int tree> some_tree)
```

Листинг 6: Использование функции *show* с помощью deriving

Недостатки

- deriving полностью основан на расширении синтаксиса, что делает невозможным его использование внутри других расширений

[3] *show* действует аналогично *toString*

- Из сформулированных в введении проблем deriving решает лишь некоторые частично, т.к. не обеспечивает возможностей, предоставляемых классами типов в Haskell[4]

Camlp4

Camlp4 — это препроцессор для Ocaml. Он также предоставляет средства Camlp4MapGenerator и Camlp4FoldGenerator, которые можно использовать для создания функций *map* и *fold*. Далее приведен пример использования Camlp4MapGenerator (лист. 7), в котором функция *suppress_id* упрощает дерево программы, удаляя применения тождественной функции.

```
type variable = string
and term =
  | Var   of variable
  | Lam   of variable * term
  | App   of term * term
  | Const of constant
and constant =
  | CInt    of int
  | CString of string

class map = Camlp4MapGenerator.generated;;

let suppress_id =
  let f = function
    | App(Lam(v, Var(v')), t) when v = v' -> t
    | x -> x
  in
    object
      inherit map as super
      method term t = f (super#term t)
    end
```

Листинг 7: Использование Camlp4MapGenerator

[4]формально это не доказано

Camlp4MapGenerator.generated в примере 7 заменяется на класс, в котором описаны тождественные функции для типов *variable*, *term* и *constant*. Аналогично работает Camlp4FoldGenerator.

Недостатки

- Отсутствие возможности работать с полиморфными вариантами
- Сложности с использованием препроцессора camlp5, т.к. нужно избегать конфликтов с ним
- Невозможность использования типов из собираемых отдельно библиотек
- Ограничения в работе с параметрическим полиморфизмом. Например, для типовых переменных необходимо вручную описывать *map* и $fold$

Другие средства

Можно упомянуть расширение синтаксиса tywith, которое позволяет генерировать функции по декларации типа. В настоящее время проект кажется заброшенным, и возможности средства составляют подмножество возможностей deriving. Поэтому, детально этот инструмент рассматриваться не будет.

2 Решение

Варианты

Рассмотрим 3 основных направления, которые могут привести к решению задачи (рис. 1):

Расширение синтаксиса

Идея основываться на нем отбрасывается сразу, т.к. такое решение не соответствовало бы требованиям. При этом, конечно, полностью не исключается возможность его использования.

Использование возможностей языка

В Ocaml нет классов типов, но есть классы объектно-ориентированного программирования, которые предоставляют схожие возможности. Например, в Camlp4MapGenerator используется наследование (subtyping) для модификации сгенерированной функции *map*.

Рефлексия

Манипуляция данными и типами во время исполнения позволяет создать аналоги функций *everywhere* и *gmapT*, но в Ocaml эти возможности не реализованы.

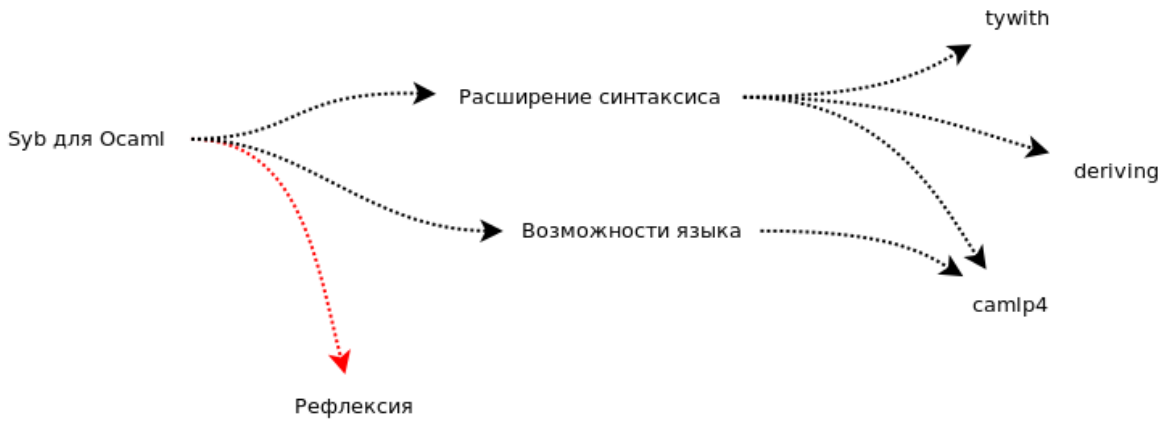

Рис. 1: Подходы к реализации

Выбор

Идея решения состоит в том, чтобы с помощью рефлексии реализовать алгоритм обхода, который бы применял некоторую функцию, определяемую пользователем, к

узлам дерева, представляющего данные. Функцию можно было бы определить как расширение базовых (полиморфных) функций, переопределяя их поведение для узлов, соответствующих заданному пользователем шаблону. Это схоже с тем, что позволяет делать с помощью наследования Camlp4FoldGenerator и Camlp4MapGenerator, но задавать поведение можно не только для узлов определенного типа, но и для произвольной подструктуры, орпеделяемой паттерном. Такая реализация сфокусирована на нуждах пользователя, ведь даже в простом примере (лист. 7) приходится выполнять pattern-matching. Ранее было сказано, что возможности рефлексии не реализованы в Ocaml, но частично восполнить данный пропуск можно используя интерфейс работы с представлением данных в памяти.

3 Детали реализации

Работа с представлением данных

В основе программы лежит работа с представлением данных. Для этого используются недокументированный модуть Obj и экспортированные функции C, которые выполняют размещение данных в памяти и некоторые другие технические функции.

Преграда, которая не позволяет сделать полноценную рефлексию в Ocaml — это то, что представление данных в памяти не дает достаточной информации о типе этих данных. Рисунок 2 демонстрирует логические представление типов: если два типа пересекаются, то один из них входит в другой (например, $int\ t \subset \alpha\ t$). Но если посмотреть на представление значений в памяти (рис. 3), оказывается, что логически несвязанные типы могут иметь пересечения (например, полиморфные варианты представляются так же, как и кортеж из двух элементов: первый — это хэш имени варианта, а второй — аргумент). Поэтому, чтобы построить обратное отображение из памяти в логическое представление требуется дополнительная информация.

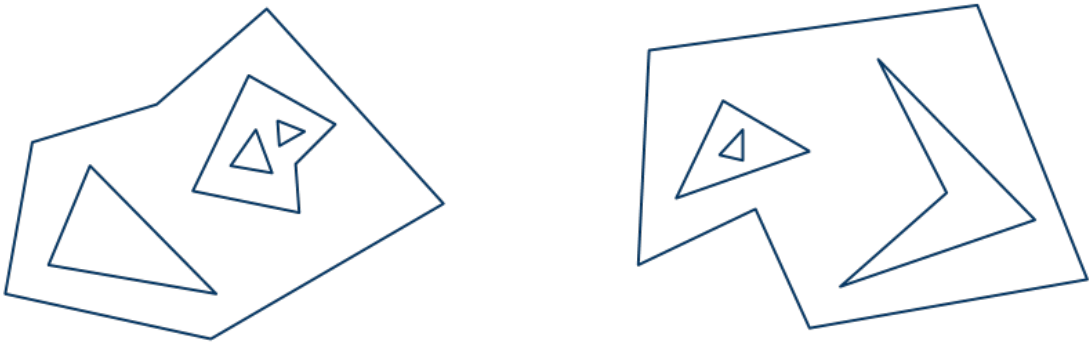

Рис. 2: Типы данных

В качестве дополнительной информации используются значения, представляющие тип. Например, тип *int list* представляется как $`List\ `Int$. С помощью анализа структуры значения в памяти и сопоставление с представлением типа определяется можно ли интерпретировать его как элемент данного типа. Такой метод допускает ошибки, например, пустой список может быть распознан как значение типа *unit*. Для решения этой проблемы можно использовать явное указание типа для функции,

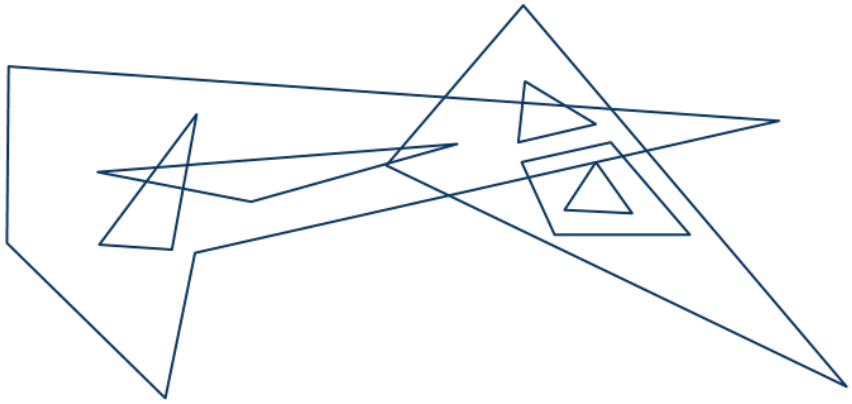

Рис. 3: Представление в памяти

применяющей описанный метод.

Модуль Catamorphism

Модуль Catamorphism содержит функцию $foldr$, которая является аналогом *everywhere* в реализации Scrap Your Biolerplate для Haskell.

Пусть требуется отобразить данные из типа, представляемого значением T_i, в тип, предстявляемый T_o в помощью функции f. ($foldr\ T_i\ T_o\ f$) - это функция, которая:

1. Строит типизированное дерево своего агрумента, используя рефлексию и T_i
2. Применяет к нему функцию f, используя комбинатор неподвижной точки
3. Проверяет соответствие результата типу T_o и возвращает его.

Комбинатор неподвижной точки имеет следующее определение (лист 8):

```
let rec fix f v = f (fix f) v
```

Листинг 8: Комбинатор неподвижной точки

Он используется для того, чтобы иметь возможность произвести рекурсивный вызов вне тела функции.

Таким образом, f принимает два аргумента - саму себя и типизированние дерево некоторого значения. Это дерево представляет из себя тройку: значение, тип этого значения и список типизированных деревьев детей корневого узла этого значения.

Модуль Mapping

В модуле Mapping определяются базовые функции для $Catamorphism.foldr$ и механизм их расширения.

Функция $id\ f\ arg$ тождественно отображает корень arg, а к его детям применяет f:

```
             A           A
id : f ->  / \    ->    / \
          B   C      f(B) f(C)
```

id является аналогом класса, который создает Camlp4MapGenerator.

Функция $foldl$ действует как $List.fold\_left$ для списка детей корневой вершины:

```
                                  A
foldl: comp -> acc -> f -> / \ -> List.fold_left compose acc [f(B); f(C)]
                             B   C
```

$foldl$ является аналогом класса, который создает Camlp4FoldGenerator.

Функция $extend\ map\ (pattern,\ function)$ переопределяет воздействие map на все узлы, удовлетворяющие паттерну $pattern$, функцией $function$. $pattern$ — это значение, которое представляет шаблон, схожий с используемыми в сопоставлениях с образцами Ocaml.

Рассмотрим $pattern$ как дерево. Его листьями являются либо сопоставления с конкретными значениями (например, $`Match\ 123$), либо $`Id$ и $`Apply$, которые сопоставляют произвольное значение. Пронумеруем дерево, обойдем его слева направо и выпишем все $`Id$ и $`Apply$ в порядке их встречаемости. Получится последовательность L_1, L_2, ..., L_n. Тогда если полученная в результате расширения функция во время обхода дерева встретит узел, сопоставляемый с образцом $pattern$, она отобразит его, применив $function$ к агрументам A_1, A_2, ..., A_n, где A_i равно сопоставленному с L_i значению, если L_i - это $`Id$, и сопоставленному с L_i значению, отображенному рекурсивно, в противном случае.

Расширения id являются преобразованиями, а $foldl$ — запросами.

Расширение синтаксиса

Для упрощения написания представлений типов сделано расширение синтаксиса, которое генерирует их автоматически по декларации. Работа расширения продемон-

стрирована на следующем примере (лист. 9 и 10):

```
type' foo = bar list
and bar = string * baz
and baz = Quux of sometype
```

Листинг 9: До препроцессирования

```
type foo = bar list
and bar = string * baz
and baz = Quux of sometype

let foo = `List bar
and bar = `Tuple [`String, `Char]
and baz = `Variant ["Quux", Some sometype]
```

Листинг 10: После препроцессирования

Пример использования

Покажем как с помощью данной реализации можно решить проблему повышения зарплаты сотрудников в Ocaml. Пусть компания представляется следующей структурой данных (лист. 11):

```
type' company = Company of dept list
and dept = Dept of name * manager * subunit list
and subunit = PUnit of employee | DUnit of dept
and employee = Employee of person * salary
and person = Person of name * address
and salary = Salary of int
and manager = Manager of employee
and name = string
and address = string
```

Листинг 11: Компания

Здесь используется расширение синтаксиса (с ключевым словом *type'*), которое создает значение *company*, представляющее тип *company*.

В таком случае функция для повышения зарплаты всем сотрудникам на 10% будет выглядеть следующим образом (лист. 12):

```
let incr_everyone : company -> company =
  Catamorphism.foldr
    company
    company
    (Mapping.extend
      Mapping.id
      (`Variant ("Salary", Some (`Child `Id)), (fun s -> Salary
        (s + s/10))))
```

Листинг 12: Повышение зарплаты

В этом примере тождественная функция на множестве компаний модифицируется изменением ее поведения для узла, соответсвующему варианту *Salary*, таким образом, что к значению зарплаты прибавляется его десятая часть. Паттерн *‘Child ‘Id* говорит о том, что расширяющая функция принимает неотображенное значение зарплаты в качестве аргумента.

Функция *find_salaries* демонструрует простоту создания запросов для сложных структур данных (лист. 13). Она получается расширением *foldl*, которая для каждого узла сцепляет списки, получающиеся отображением его детей, функцией, отображающей узел с зарплатой в список с единственным элементом: значением зарплаты. В результате *find_salaries* возвращает список значений зарплат компании.

```
let find_salaries : company -> int list =
  Catamorphism.foldr
  company
  (`List `Int)
    (Mapping.extend
      (Mapping.foldl (@) [])
      (`Variant ("Salary", Some (`Child `Id)), fun x -> [x]))
```

Листинг 13: Поиск зарплат

Заключение

Заявленная функциональность была реализована.

- Возможность писать функции трансформации данных и запросы
- Возможность использования внутри расширения синтаксиса
- Слабая зависимость от типов данных
- Отсутствие необходимости писать “boilerplate” код

Преимущества

- По сравнению с deriving: не используется расширение синтаксиса
- По сравнению Camlp4MapGenerator и Camlp4FoldGenerator: работа с полиморфными вариантами
- Про сравнению с реализациями для Ocaml: прозрачная работа с типами, позволяющая взаимодействовать с отдельно скомпилированными библиотеками
- Pattern-matching, позволяющий работать с целыми подструктурами, а не только с узлами конкретных типов

Недостатки

- Производительность ниже, т.к. требуется несколько раз обходить дерево данных
- Отсутствие статического контроля, неправильное определение функции обхода может вызвать segmentation fault
- Как и в других реализациях для Ocaml: ограничения в работе с параметрическим полиморфизмом

Выводы и дальнейшие направления деятельности

Низкоуровневая работа с данными позволила создать интерпретацию Scrap Your Biolerplate, которая достаточно близко приблизилась по своим возможностям к оригинальной реализации.

Созданное средство содержит ряд довольно серьезных недостатков. Можно предположить пути их разрешения:

Производительность

Используя возможности языка можно попытаться избавиться от части обходов структур, сведя к минимуму работу с памятью и переложив часть обязанностей на компилятор.

Статический контроль

Повысить безопасность можно уже сейчас, реализовав расширение синтаксиса для вызова функции foldr. Оно не было сделано, т.к. это противоречило бы требованиям о возможности использовать эту функцию внутри других расширений, но такой вариант вполне применим для обычного использования.

Параметричесикй полиморфизм

Вполне вероятно, в дальнейшем можно устранить этот недостаток, используя возможности языка, которые пока не были задействованы. В качестве неиболее перспективного направления можно выделить использование классов объектно-ориентированного программирования.

Список литературы

[1] Ralf Lämmel, Simon Peyton Jones, Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming, 2003.

[2] Ralf Laemmel, Simon Peyton Jones, Scrap your boilerplate with class: extensible generic functions, 2005

[3] Brian McNamara, Catamorphism, 2005.

[4] Didier Rémy, Using, Understanding, and Unraveling The OCaml Language, 2000.

[5] Xavier Leroy, The Objective Caml system release 3.11, 2008.