Санкт-Петербургский Государственный Университет

Математико-механический факультет

Кафедра системного программирования

Библиотека обобщенной обработки данных для Ocaml

Дипломная работа студента 545 группы

Мечтаева Сергея Владимировича

Научный руководитель _____ к.ф.-м.н. Булычев Д.Ю.

Рецензент _____ Сергей И.Д.

"Допустить к защите"
заведующий кафедрой _____ д.ф.-м.н., проф. Терехов А.Н.

Санкт-Петербург

2011

Saint-Petersburg State University

Mathematics and Mechanics Faculty

Software Engineering Department

Generic data processing library for Ocaml

Graduate paper by

Sergey Mechtaev

545 group

Scientific advisor _____ Dr. D.U. Bulychev

Reviewer _____ I.D. Sergey

"Approved by"

Head of Department _____ Professor A. N. Terekhov

Saint Petersburg

2011

Содержание

1 Введение

Данная работа посвящена поиску и реализации способа элиминации стереотипного кода в программах на языке Objective Caml[1]. Мотивацией послужили проблемы, с которым столкнулись при создании компилятора языка описания аппаратуры HaSCoL[2].

Стереотипным (англ. “boilerplate”) называется код, который может быть переиспользован в других контекстах без существенных изменений. Как определение стереотипного кода, так и способ элиминации могут существенно различаться в зависимости от парадигмы программирования и от того, какие возможности для этого предоставляет программное окружение. Типичными способами элимининации является использование процедур в процедурных языках или шаблонов проектирования в объектно-ориентированных [1]. В рамках данной работы под стереотипным будет пониматься код, реализующий разнообразные обходы строго типизированных структур данных с целью их модификации или сбора информации. Такой тип стереотипного кода часто возникает в функциональных языках программирования при обработке алгебраических и параметрически-полиморфных типов данных. В качестве иллюстрации рассмотрим типичный для компиляторов пример. Пусть имеется тип данных, описывающий элементарные арифметические выражения:

```
type expr =
    Add of expr * expr
  | Sub of expr * expr
  | Neg of expr
  | Var of variable
```

Здесь `expr` — имя типа данных, `Add`, `Sub`, `Neg`, `Var` — *конструкторы*, рассматриваемые как функции, которые создают представление для соответствующих типов арифметичесикх операций, `variable` — некоторый тип, содержащий информацию о переменной, входящей в выражение. Тип `expr` является рекурсивным, что позволяет строить структуры данных потенциально неограниченного размера.

Представим, что для таких выражений необходимо реализовать идентификацию переменных, т.е. некоторую процедуру, которая осуществляет связывание вхождений переменных в выражение с описанием этих переменных. Предположим, что такое

[1] http://caml.inria.fr/ocaml/

[2] http://oops.math.spbu.ru/projects/coolkit/wiki/WikiStart

связывание для одного вхождения реализуется функцией

```
val resolve : variable -> variable
```

которая для простоты преобразует тип `variable` в себя. Таким образом, фактически идентификация переменных в выражении является обобщением функции `resolve` для типа данных `expr`. Такую функцию легко написать вручную:

```
let rec resolveAll resolve = function
  | Add (x, y) -> Add (resolveAll resolve x, resolveAll resolve y)
  | Sub (x, y) -> Sub (resolveAll resolve x, resolveAll resolve y)
  | Neg e      -> Neg (resolveAll resolve e)
  | Var v      -> Var (resolve v)
```

Здесь `resolveAll` — это определяемая функция идентификации всех переменных в выражении, которая получает функцию идентификации одного вхождения (`resolve`) и рекурсивно обходит выражение, применяя ее ко всем вершинам, соответствующим вхождениям переменных.

Предположим теперь, что нам понадобилось определять список всех входящих в выражение переменных. Такую функцию тоже легко написать вручную:

```
let rec collectAll = function
  | Add (x, y) -> (collectAll x) @ (collectAll y)
  | Sub (x, y) -> (collectAll x) @ (collectAll y)
  | Neg e      -> collectAll e
  | Var v      -> [v]
```

Здесь `[v]` — список из одного элемента, "`@`" — операция конкатенации списков.

Данные примеры являются яркой иллюстрацией использования стереотипного кода. Как `resolveAll`, так и `collectAll` содержат общую семантическую часть — процедуру обхода структуры данных. Видно, что большую часть реализации этих функций занимает именно стереотипный код. Такая ситуация является типичной при наивной обработке алгебраических типов данных. Наличие стереотипного кода усложняет сопровождение и разработку программ, поскольку даже незначительное изменение типов обрабатываемых данных влечет маштабные и однообразные изменения во всей программе, приводящие к огромному количеству ошибок.

В рамках данной работы мы рассматриваем способ элиминации стереотипного кода, который основан на использовании техники обобщенного программирования, управляемого типами (*generic type-driven programming*). Данный подход помогает из-

бавиться от стереотипного кода, семантика которого полностью определяется типом обрабатываемых данных. Оба приведенных примера относятся именно к такому типу, потому что, во-первых, процедура рекурсивного обхода полностью определяется типом `expr`, а семантическое действие, специфическое для каждого из примеров, определяется типом `variable`.

Решению описанной проблемы посвящена обширная литература, обзор которой приведен в части 2. В результате изучения существующих подходов в качестве основы для данной работы был выбран подход "Scrap Your Boilerplate" (SYB) [2, 3], поддержка которого входит в стандартную библиотеку Haskell[3]. В оригинальной версии реализация SYB существенно опирается на особенности типовой системы языка Haskell и не может быть прямо перенесена в среду Objective Caml. Кроме того, более поздние исследования выявили существенное падение производительности кода при использовании SYB по сравнению с рукописным стереотипным кодом из-за повышения уровня абстракции.

В рамках данной работы была разработана адаптация подхода SYB для языка Objective Caml, обладающая сравнимой с оригиналом выразительностью и функциональностью, при этом падение производительности составило не более двух раз для содержательных примеров. Такой результат был достигнут благодаря использованию техники специализации и передачи продолжений (*continuation passing style*) [4].

[3]http://www.haskell.org

2 Обзор литературы

Существует большое количество средств, решающих проблемы стереотипного кода, для разных языков программирования [5]. Их можно разделить на три группы: средства для языка Objective Caml, средства для языка Haskell и средства для других языков. Ocaml обладал слабыми возможностями для борьбы с описанными проблемами до выхода последней версии языка, в которой была добавлена поддержка модулей первого класса (*first-class modules*)-[6, 7, 8], поэтому для него есть лишь узкий круг соответствующих решений. Для языка Haskell разработано большое число подходов для борьбы со стереотипным кодом. Кроме того, этот язык во многом похож на Ocaml. Поэтому средства для Haskell заслуживают подробного рассмотрения. Из средств для других языков можно упомянуть шаблон проектирования Visitor. Он создан для объектно-ориентированных языков, а не для функциональных, поэтому не будет рассматриваться далее.

2.1 Средства для Haskell

Средства для Haskell можно разделить на три группы: реализующие концепцию обобщенного программирования, управляемого типами (*generic type-driven programming*), использующие данные, индексированные типами (*type-indexed values* [9]) и другие подходы. Далее будут рассмотрены примеры подходов из каждой группы. Ни один из них не может быть применен в программе на Ocaml, т.к. они используют возможности Haskell, отсутствующие в Ocaml. С другой стороны, эти подходы оказали существенное влияние на разработанную библиотеку.

2.1.1 Scrap Your Boilerplate

Scrap Your Boilerplate [2, 3] — это подход (шаблон проектирования), предназначенный для обработки сложных структур данных. Он позволяет определять функции обработки, в которых трансформация узлов данных контролируется их типом.

Для демонстрации возможностей подхода покажем как он решает проблему стереотипного кода для задач идентификации и сбора переменных. Структура данных, представляющая выражение, имеет следующее определение:

```
data Expr =
       Add Expr Expr
     | Sub Expr Expr
```

```
  | Neg Expr
  | Var Variable
    deriving (Data, Typeable)
```

Здесь создаются экземпляры классов `Data` и `Typeable`. `Typeable` обеспечивает безопасное приведение типов. `Data` содержит базовые функции обхода.

Далее представлены реализации функций идентификации и сбора переменных, аналогичные приведенным в введении:

```
resolveAll :: (Variable -> Variable) -> Expr -> Expr
resolveAll resolve = everywhere (mkT resolve)


collectAll :: Expr -> [Variable]
collectAll = everything (++) ([] `mkQ` (\x -> [x]))
```

Здесь используются функции:

resolve

Работает аналогично `resolve`, описанной в введении.

everywhere, everything

Для каждого узла дерева данных `everywhere f` применяет `f` ко всем его дочерним узлам, затем из отображенных узлов конструируется новый узел того 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`. `everything` действует аналогично, но вместо трансформации узлов собирает данные и компонует их.

mkT, mkQ

`mkT` доопределяет функцию тождественно для остальных типов и, таким образом, делает её полиморфной. Для этого она проверяет с помощью класса `Typeable`, можно ли применить функцию к значению, и, если можно, то применяет, иначе — возвращает значение без изменений. `mkQ` действует аналогично, но доопределяет функцию константой.

Функции `gmapT` и `gmapQ` можно обобщить до функции `gfoldl`, такой что `gfoldl k z` узлу `Node a b` будет сопоставлять `(z Node `k` a) `k` b`; `gmapT` выражается через `gfoldl` следующим образом:

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

Описанный выше подход обладает следующими недостатками:

- использование безопасного приведения типов и классов типов делает его неприменимым в Ocaml;
- низкая производительность.

2.1.2 Instant Generics

Instant Generics [10] — это подход, основанный на использовании семейств типов [11] и разложении типов в сумму произведений (*sum-of-products view*). В терминах Haskell такое разложение может быть выражено следующим образом:

```
data Unit = Unit
data a :*: b = a :*: b
data a :+: b = Inl a | Inr b
```

Определяется семейство типов `Repr`, которое для данного типа будет задавать тип его разложения. В этом примере приведено `Repr` для списков:

```
type family Repr t
type instance Repr [a] = Unit :+: (a :*: [a])
```

Далее определяется класс типов, все члены которого населены только трансформирующимися в сумму произведений и обратно значениями:

```
class Representable a where
  type Repr a
  toRepr :: a -> Repr a
  fromRepr :: Repr a -> a
```

Если выражение находится в семействе типов `Repr`, то можно написать функцию идентификации переменных следующим образом:

```
instance Resolve Expr where
  resolveAll resolve (Var v) = Var (resolve v)
  resolveAll resolve t = fromRepr . (resolveAll resolve) . toRepr t
```

Описанный подход показывает лучшую производительность, чем Scrap Your Boilerplate, но использование семейств типов и классов типов делает невозможным его применение в Ocaml.

2.1.3 Scrap Your Scary Types

Scrap Your Scary Types [12] помогает бороться с стереотипным кодом, возникающим при обработке рекурсивных структур данных. Снова рассмотрим данные, представляющие арифметичесикие выражения. Создание простых функций, таких как сбор или идентификация переменных, требует написания обхода всех видов узлов. Для решения этой проблемы авторы метода предлагают определить класс типов `Uniplate`. Он содержит функцию `uniplate`, которая для узла возвращает пару из списка его непосредственных потомков и конструктора этого узла:

```
class Uniplate a where
  uniplate :: a -> ([a], [a] -> a)

instance Uniplate Expr where
  uniplate (Neg a) = ([a] , \[a] -> Neg a)
  uniplate (Add a b) = ([a, b], \[a, b] -> Add a b)
  uniplate (Sub a b) = ([a, b], \[a, b] -> Sub a b)
  ...
```

Через `uniplate` выражаются многие сущности, упрощающие обработку данных: `universe`, `transform` и др. Функция `universe` принимает структуру данных и возвращает все структуры того же типа, которые можно найти внутри нее. Это можно использовать, например, для сбора переменных:

```
collectAll :: Expr -> [Variable]
collectAll x = [y | Var y <- universe x ]
```

Функция `transform` обходит структуру данных в глубину и трансформирует каждую вершину при выходе. Её можно применять для идентификации переменных:

```
resolveAll resolve e = transform f e
where
  f (Var v) = Var (resolve v)
  f x = x
```

Использование расширения Haskell, которое обеспечивает пользователя мультиаргументными классами типов, делает возможным обобщить этот подход до обработки данных, состоящих из нескольких типов.

2.2 Средства для Ocaml

Средства для Ocaml можно разделить на две группы: созданные до выхода версии языка 3.12 и созданные после. Большинство средств первой группы основаны на расширениях синтаксиса, которые создают некоторые базовые функции по декларациям типов. Средства второй группы используют также добавленные в новой версии языка модули первого класса, расширяющие возможности обобщенного программирования.

2.2.1 Deriving

Deriving — это средство, имитирующее классы типов в Ocaml. В его основе лежит расширение синтаксиса, позволяющее создавать функции обработки данных, которые для каждого типа определяются как композиция некоторой трансформации для данного типа и аналогичных функций для подтипов. Для создания функции необходимо написать генераторы для базовых типов и конструкторов (например, `int`, `char`, `string`, вариантов, списков и т.д.), тогда она будет действовать над классом, который строится следующим образом:

1. базовые типы принадлежат классу;
2. если типы τ_1, τ_2, ..., τ_n входят в класс C, то если к декларации типа, образованного комбинацией τ_1, τ_2, ..., τ_n, применено расширение синтаксиса `deriving`, то он тоже входит в C.

При этом, каждый раз при вызове функции необходимо указывать контекст (тип обрабатываемых данных). Далее приведен пример использования deriving для создания функции `show`, которая определена на стандартном классе `Show`, для пользовательского типа `tree`:

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

Для того чтобы использовать эту функцию, нужно воспользоваться следующей синтаксической конструкцией:

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

Описанное средство обладает следующими недостатками:

- deriving полностью основан на расширении синтаксиса, что делает невозможным его использование внутри других расширений;
- deriving не обладает расширяемостью. Если требуется использование алгоритма, не предусмотренного авторами deriving, то необходимо вручную реализовать генераторы кода для всех базовых типов и конструкторов. По этой причине нельзя просто выразить, например, функции сбора и идентификации переменных.

2.2.2 Generics for the OCaml masses

Generics for the OCaml masses [6] — это подход, использующий модули первого класса и разложение значений в суммы произведений. В качестве произведений используются кортежи из двух элементов. Для представления сумм и пустых типов применяются следующие структуры данных:

```
type ('a, 'b) either = Left of 'a | Right of 'b
type zero = { absurd : 'a . 'a }
```

Представление типа определяется как комбинация представлений дочерних типов. В случае рекурсивных типов применяется неподвижная точка. Для создания функций обработки используются интерпретации, в которых определяется обработка узлов каждого типа:

```
module type Interpretation : sig
  type 'a tc
  val unit : unit tc
  val int  : int tc
  val ( * )  : 'a tc -> 'b tc -> ('a * 'b) tc
  (* ... *)
end
```

Предположим, что тип `variable` равен `string`. Приведем пример интерпретации, которая производит идентификацию переменных для некоторой функции `resolve`:

```
module ResolveAll : Interpretation with type 'a tc = 'a -> 'a =
struct
```

```
  type 'a tc = 'a -> 'a
  let int    = id
  let string = resolve
  let unit   = id
  let ( * ) f g (a, b) = (f a, g b)
  (* ... *)
end
```

Недостатками такого подхода является низкая производительность и ограничения в задании функций обработки, т.к. интерпретации определяются для представлений, а не для самих типов.

2.2.3 Camlp4

Camlp4[4] — это система расширяющегося синтаксиса для Ocaml. Она также предоставляет средства `Camlp4MapGenerator` и `Camlp4FoldGenerator`, которые можно использовать для создания функций `map` и `fold`. Далее приведен пример использования Camlp4MapGenerator для идентификации переменных:

```
type expr =
  | Add of expr * expr
  | Sub of expr * expr
  | Neg of expr
  | Var of variable

class map = Camlp4MapGenerator.generated;;

let resolveAll resolve =
  let f = function
    | Var v -> Var (resolve v)
    | x -> x
  in
  let o =
    object
      inherit map as super
      method expr t = f (super#expr t)
```

[4] http://caml.inria.fr/pub/old_caml_site/camlp4/index.html

```
  end
in
  o#expr
```

`Camlp4MapGenerator.generated` в этом примере заменяется на класс, в котором описаны тождественные функции обработки для типа `expr`. Аналогично работает `Camlp4FoldGenerator`.

Описанное средство обладает следующими недостатками:

- 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`.

2.2.4 Другие средства

Существуют более простые средства, решающие похожие задачи. Они обладают меньшими возможностями и большими недостатками. Расширение синтаксиса tywith позволяет генерировать функции `map`, `to_string` и `fold` по декларации типа. Такой подход обладает следующими недостатками:

- отсутствие возможности его использования внутри расширения синтаксиса;
- лишь частичное решение проблем стереотипного кода: полученные функции требуют задания преобразований для всех входящих в тип узлов, даже если их обработка не требуется.

3 Теоретические основы

В этой части будут описаны основные идеи, использованные для адаптации и улучшения SYB. Код будет приводиться на абстрактном функциональном языке. Как перевести программы с абстрактного языка на Objective Caml, будет показано в части 4.

3.1 Обозначения

Использующийся далее абстрактный язык имеет такую же динамическую семантику, как Objective Caml, но обладает более простым синтаксисом и поддерживает типы произвольного ранга [13]. Приведем соглашения, применяющиеся в обозначениях, и конструкции, нуждающиеся в объяснении:

- $f \circ g$ означает композицию функций f и g;
- fix — оператор неподвижной точки, реализация которого будет описана в ч. 4;
- $\tau_1 \wedge \tau_2$ — это тип, который населен всеми упорядоченными парами из элементов типов τ_1 и τ_2, т.е. для значений такого типа определены функции fst и snd, возвращающие первый и второй элемент соответственно;
- будем считать, что обрабатываемые данные принадлежат только типам вида $C_1\ of\ \tau_1 \wedge \tau_2 \wedge ... \wedge \tau_n \mid ... \mid C_k\ of\ \tau_1 \wedge \tau_2 \wedge ... \wedge \tau_n$ (размеченное объединение), если τ_1, τ_2, ... τ_n — это типы обрабатываемых данных, $\tau_1 \wedge \tau_2 \wedge ... \wedge \tau_n$ — естественное обобщение пар на кортежи;
- вместо (`fun x y ->` ...) будет использоваться ($\lambda\ x\ y.$ ...);
- $?v$ означает то же, что `Obj.magic v` в Ocaml (неконтролируемое изменение типа);
- имена части переменных имеют верхний индекс, совпадающий с именем некоторого типа. Это означает, что данная переменная имеет некоторое отношение к указанному типу. Например, $transform^t$ — трансформация данных типа t.

3.2 Равенство типов

Слабым равенством [14] называется значение типа $(\tau_1, \tau_2)\ eq$, для которого определены операции следующих типов:

$$refl : \forall\alpha\ .\ (\alpha,\ \alpha)\ eq$$

$$symm : \forall\alpha\ \beta\ .\ (\alpha,\ \beta)\ eq \to (\beta,\ \alpha)\ eq$$

$$trans : \forall\alpha\ \beta\ \gamma\ .\ (\alpha,\ \beta)\ eq \to (\beta,\ \gamma)\ eq \to (\alpha,\ \gamma)\ eq$$

$$coerce : \forall\alpha\ \beta\ .\ (\alpha,\ \beta)\ eq \to \alpha \to \beta$$

Значения $refl$, $symm$ и $trans$ представляют аксиомы рефлексивности, симметричности и транзитивности; *coerce* можно использовать для приведения типов.

Лейбницевское равенство [6] отличается от слабого тем, что для него определена дополнительная операция, которая позволяет из $(\tau_1, \tau_2)\ eq$ получить $(\tau_1\ c, \tau_2\ c)\ eq$ для произвольного конструктора типа c.

Значения, представляющие слабое и Лейбницевское равенство, существуют. Их реализация несущественна, поэтому не будет приводиться далее.

3.3 Маркеры типов

Введем некоторый фиксированный конструктор типа $marker$. Будем называть маркерами значения типов $\tau\ marker$ для всех τ. Определим функцию *compare*:

$$compare : \forall\alpha\ \beta\ .\ \alpha\ marker \to \beta\ marker \to (\alpha, \beta)\ eq\ option$$

$$compare = \lambda\ m_1\ m_2.\begin{cases} Some(?refl : (\tau_1, \tau_2)\ eq) & , ?m_1 \doteq ?m_2 \\ None & , not\ (?m_1 \doteq ?m_2) \end{cases}$$

где m_i имеет тип $\tau_i\ marker$, а под $\doteq$ будем пока понимать физическое равенство. Функция *compare* для пары одинаковых маркеров возвращает утверждение о равенстве типов, входящих в определение типа этих маркеров. Такое eq корректно, если не существует равных маркеров с разными типами. Для простоты далее в этой части будем считать, что между маркерами и типами установлено взаимно-однозначное соответствие.

Используя *compare* можно произвольно изменять тип трансформации, доопределяя её тождественно для данных других типов:

$$lift : \forall\alpha\ \beta\ .\ \alpha\ marker\ \to (\alpha \to \alpha) \to \beta\ marker \to (\beta \to \beta)$$

$$lift = \lambda\ m_1\ f\ m_2.\begin{cases} (coerce\ eq) \circ f \circ (coerce \circ symm\ eq) & , compare(m_1, m_2) = Some\ eq \\ id & , compare(m_1, m_2) = None \end{cases}$$

3.4 Трансформации и запросы

Подход Scrap Your Boilerplate, описанный в разделе 2.1.1, предоставляет пользователю возможность создавать функции обработки трех видов: трансформации, запросы и монадические [15] трансформации. Под трансформацией понимается функция типа $\tau \to \tau$, где τ — тип обрабатываемых данных. Трансформация строит новую структуру данных, равную старой, но с отображенными заданным способом узлами нужных типов. Запрос — это функция типа $\tau \to \rho$, где τ — тип обрабатываемых данных, а ρ — тип результата запроса. Такая функция обходит данные и для каждого узла компонует результаты запросов для его потомков. Монадическая трансформация — это функция типа $\tau \to \tau\ \mu$, где τ — тип обрабатываемых данных, а μ — конструктор типа монады. Монады можно неформально определить, как структуры данных, представляющие некоторые вычисления. В данном случае они используются, чтобы обобщить трансформацию, позволив выполнять одновременно с ней какое-либо другое действие, например, обработку ошибок.

Далее будет показано, как реализовать трансформацию, используя маркеры типов и функцию $lift$. Запросы и монадические трансформации реализуются аналогично, поэтому не будут рассматриваться подробно.

3.5 Простая трансформация

Пусть для типа t имеется маркер $marker^t : t\ marker$. Используя $lift$ можно создать простую трансформацию, управляемую типами данных:

$$
\begin{aligned}
&map^t : \forall\alpha\ .\ \alpha\ marker \to (\alpha \to \alpha) \to (t \to t)\\
&map^t =\\
&\quad \lambda\ marker\ f\ value.\\
&\qquad let\ transform^t = lift\ marker\ f\ marker^t\ in\\
&\qquad let\ transform^{t_1} = map^{t_1}\ marker\ f\ in\\
&\qquad ...\\
&\qquad let\ transform^{t_n} = map^{t_n}\ marker\ f\ in\\
&\qquad match\ value\ with\\
&\qquad \mid C_1\ (c^{t_1},\ ..., c^{t_n}) \to\\
&\qquad\quad transform^t\ (C_1\ (transform^{t_1} c^{t_1},\ ..., transform^{t_n} c^{t_n}))\\
&\qquad ...\\
&\qquad \mid C_n\ (c^{t_1},\ ..., c^{t_n}) \to\\
&\qquad\quad transform^t\ (C_n\ (transform^{t_1} c^{t_1},\ ..., transform^{t_n} c^{t_n}))
\end{aligned}
$$

Будем рассматривать только данные, которые можно представить в виде дерева. Выражение $map^t\ m\ f$ создает функцию, которая обходит дерево в глубину и, выходя из каждой вершины, копирует её и отображает доопределенной с помощью $lift$ функцией f. Получается, что она создает новое значение типа t, равное исходному, но с трансформированными функцией f узлами, тип которых помечен маркером m.

3.6 Специализация

Приведенный выше пример обладает рядом недостатков. Предположим, что требуется задать функцию обработки, которая нетривиально трансформировала бы узлы нескольких типов. Для этого нужно определить несколько трансформаций, а затем взять их композицию. Очевидно, такое решение значительно снизит производительность по сравнению с наивной реализацией, т.к. будут выполняться несколько обходов данных. Представим теперь, что необходимо обрабатывать рекурсивные данные большого объема, например, трансформировать элементы списка. Приведенная функция map^t будет производить сравнение маркеров каждый раз при обработке

очередного элемента. Это также существенно снизит производительность по сравнению с наивной реализацией.

Для решения описанных проблем используем специализацию. Специализацией назовем функцию типа $(\forall\beta\ .\ \beta\ data \to (\beta \to \beta))$, где $data$ — конструктор типа, определяемый как пара из маркера и функции map. Она будет инкапсулировать определение трансформации. Также она будет обходить структуру типа перед обработкой данных, производя необходимые сравнения маркеров, и, таким образом, вычислять трансформацию заранее. Влияние специализации амбивалентно. С одной стороны, она отделяет пользовательскую логику от алгоритмы обхода, что позволяет задавать произвольные трансформации. С другой стороны, она увеличивает производительность за счет предподсчета. Значение $data$ будет выглядеть следующим образом:

$$
\begin{aligned}
&type\ \alpha\ map = (\forall\beta\ .\ \beta\ data \to (\beta \to \beta)) \to (\alpha \to \alpha)\\
&type\ \alpha\ data = \alpha\ marker \wedge \alpha\ map\\
\\
&data^t : t\ data =\\
&\quad (marker^t,\\
&\quad \lambda\ specialize.\\
&\qquad let\ transform^{t_1} = specialize\ data^{t_1}\ in\\
&\qquad ...\\
&\qquad let\ transform^{t_n} = specialize\ data^{t_n}\ in\\
&\qquad \lambda\ value.\\
&\qquad\quad match\ value\ with\\
&\qquad\quad | \ C_1\ (c^{t_1},\ ..., c^{t_n}) \to C_1\ (transform^{t_1} c^{t_1},\ ..., transform^{t_n} c^{t_n})\\
&\qquad\quad ...\\
&\qquad\quad | \ C_n\ (c^{t_1},\ ..., c^{t_n}) \to C_n\ (transform^{t_1} c^{t_1},\ ..., transform^{t_n} c^{t_n}))
\end{aligned}
$$

где $data^{t_i}$ — это аналогичные значения для подтипов t. Такая функция map действует аналогично приведенной ранее. Но вместо обхода дерева данных она обходит граф типов и вычисляет трансформацию для типа t. Это получается за счет того, что трансформации для подтипов $(transform^{t_i})$ вычисляются заранее, а затем ис-

пользуются в замыкании, которое представляет трансформацию типа t. Заметим, что получаемая функция не отображает корень дерева значения.

Чтобы создать специализацию, можно воспользоваться, например, следующей функцией:

$$
\begin{aligned}
&make : \forall\alpha\ .\ \alpha\ marker \to (\alpha \to \alpha) \to (\forall\beta\ .\ \beta\ data \to (\beta \to \beta))\\
&make =\\
&\quad \lambda\ marker\ f.\\
&\qquad fix\ (\lambda\ specialize.\\
&\qquad\quad \lambda\ data.\\
&\qquad\qquad let\ transform = lift\ marker\ f\ (fst\ data)\ in\\
&\qquad\qquad transform\ \circ\ ((snd\ data)\ specialize))
\end{aligned}
$$

Она создает специализации для трансформаций, которые выполняют преобразование узлов, помеченных маркером $marker$, с помощью функции f. Специализация определяется как композиция трансформации корня дерева значания ($transform$) и трансформации остальных узлов ($(snd\ data)\ specialize$). Неподвижная точка определяет трансформацию одинаково для значения данного типа и для его подзначений, тип которых равен данному.

3.7 Продолжения

Приведенная выше реализация все еще содержит ряд недостатков:

- невозможно создать монадическую трансформацию;
- алгоритм обхода требует большое количество памяти на стеке, т.к. используется нехвостовая рекурсия.

Для решения этих проблем используются продолжения. Они позволяют сделать рекурсию хвостовой, кроме того, при использовании продолжений можно для каждой функции map задавать тип результата произвольным образом, т.к. он зависит не от типа аргумента, а от типа продолжения.

Приведем только типы значений $data$, т.к. изменения реализации стандартны:

$$type\ \alpha\ map = \forall\beta\ .\ \beta\ specialize \to (\alpha \to \beta) \to (\alpha \to \beta)$$

$$type\ \alpha\ specialize = \forall\beta\ .\ \beta\ data \to (\beta \to \alpha) \to (\beta \to \alpha)$$

$$type\ \alpha\ data = \alpha\ marker \wedge \alpha\ map$$

В этом примере к функции *map* и к специализации в качестве аргументов были добавлены продолжения. Поэтому необходимо также изменить определение специализации, приводившееся ранее:

$make : \forall\alpha\ .\ \alpha\ marker \to (\alpha \to \alpha) \to \beta\ specialize$

$make =$

$\lambda\ marker\ f.$

$fix\ (\lambda\ specialize.$

$\lambda\ data.$

$let\ transform = lift\ marker\ f\ (fst\ data)\ in$

$\lambda\ cont\ value.\ ((snd\ data)\ specialize)\ (cont\ \circ\ transform)\ value)$

Здесь изменился лишь способ композиции: трансформация корня дерева данных добавляется в продолжение трансформации остальных узлов дерева.

3.8 Монадические трансформации

Приведенная выше функция *map* позволяет создавать монадические трансформации типа $\tau \to \tau\ \mu$. Для создания соответствующей специализации нужно заменить слабое равенство на Лейбницевское, а затем использовать новую аксиому в определении функции $lift$. Также необходимо вместо простой композиции трансформации и продолжения в определении специализации связывать их при помощи *bind*.

3.9 Конструкторы типа

В описываемом средстве реализована поддержка типов вида $* \to *$. Для этого *data* конструктора *c* определяется как функция, которая принимает *data* аргумента (типа *t*) и возвращает *data* для типа *t c*. Для такой функции требуется, чтобы

$fst\ data^{t_1} = fst\ data^{t_2} \Rightarrow fst\ data^{t_1\ c} = fst\ data^{t_2\ c}$, иначе невозможно будет автоматически генерировать *data* для типов, в определении которых встречается применение конструкторов. Поэтому определены маркеры конструкторов, являющиеся функциями из маркеров типов в маркеры типов. Они добавляют в маркер некоторую информацию о конструкторе. Получается, что два маркера равны только если они отмечают два типа, полученные с помощью применений одних и тех же конструкторов к одним и тем же типам в одном и том же порядке.

4 Реализация

Далее будет описаны основные трудности, с которыми столкнулись при реализации идей, обсуждавшихся в части 3.

4.1 Типы высоких рангов

В начале следует объяснить, как переводить программы с абстрактного языка на Objective Caml. Единственная существенная часть перевода — это задание типов высоких рангов, т.к. в остальном языки различаются только синтаксически. Для реализации типов высоких рангов используются записи. Далее приведен пример задания типа специализации и значения *data* (для случая с продолжениями) с помощью записей:

```
type 'target specialize =
    { specialize : 'd . 'd data -> ('d -> 'target) -> ('d -> 'target) }
and 'd data =
    { marker : 'd marker;
      map : 'target . 'target specialize -> ('d -> 'target) -> ('d -> 'target) }
```

4.2 Маркеры типов

Маркеры можно задать следующим образом:

```
type 't marker = identifier
```

В этом коде `identifier` — это тип списков ссылок на unit-значения. Для сравнения таких ссылок используется физическое равенство. Это эквивалентно сравнению их адресов. Равенство списков определяется следующим образом: списки c_1 и c_2 равны тогда и только тогда, когда равны их длины и $\forall i\ c_1[i] == c_2[i]$. Такое равенство используется в определении *compare* в качестве $\doteq$. Маркеры создаются с помощью функции *make*, которая каждый раз возвращает маркер с новым значением идентификатора. Маркеры конструкторов прицепляют к маркеру-аргументу свое уникальное значение. Создание маркеров другими способами запрещено, их настоящий тип (список ссылок) скрыт интерфейсом модуля. Очевидно, в этом случае равным маркерам соответствуют одинаковые типы, поэтому определение *compare* корректно.

4.3 Неподвижная точка

Неподвижная точка используется в определении специализации. Если реализовать ее наивным способом (с помощью рекурсивной функции), то она будет зацикливаться, обходя рекурсивные типы. Для решения этой проблемы используется техника, схожая с witnessing fixed points [16]. Представим тип в виде графа. Специализация обходит граф в глубину, вычисляя функцию трансформации. Поэтому можно говорить об обратных дугах [17]. Проходя по обратной дуге, специализация возвращает функцию трансформации, в которой используется ссылка не некоторую заглушку. Затем, при выходе из каждой вершины, представляющей некоторый тип, она заменяет заглушки, соответствующие данному типу, трансформацией, вычисленной в данном узле. Таким образом, в вершинах входа в контуры она замыкает трансформации в циклы.

Идею описанного алгоритма можно показать на простом примере:

```
let fix f =
  let stub = ref (fun _ -> failwith "error") in
  let fixed = f (fun x -> !stub x) in
  stub := fixed;
  fixed
```

Здесь рассматривается упрощенная ситуация, не связанная со специализациями. Обход состоит из одного шага: в неподвижную точку (`fixed`) ставится заглушка (`stub`), а затем полученная функция замыкается с помощью присваивания. Такую неподвижную точку можно использовать, например, для вычисления факториала:

```
let fact = fix (fun fact -> function 0 -> 1 | n -> n * fact (n-1))
```

4.4 Расширение специализации

Трансформации определяются расширением тождественной специализации с помощью переопределения воздействия на узлы определенных типов. При реализации такого механизма возникает проблема: каждая из специализаций s_1, s_2, ..., s_n таких, что s_k расширяет s_{k+1}, должна специализировать потомков каждого узла с помощью s_1, но она не знает о ней, т.к. определяется раньше. Передача наиболее широкой специализации в качестве параметра s_i решает проблему, но требует изменения алгоритма подстановки заглушек, т.к. получается, что расширяющие друг друга специализации содержат общую память, и применение одной специализации может испортить

работу другой. Для нейтрализации этого эффекта необходимо удалять заглушку при выходе их каждой вершины.

4.5 Расширение синтаксиса

Реализовано расширение синтаксиса, которое генерирует значения *data* и маркер по декларации типа.

4.6 Пример использования

Предположим, имеется структура данных, представляющая программу на простом языке программирования:

```
datatype expr =
    Add of expr * expr
  | Sub of expr * expr
  | Neg of expr
  | Var of variable
and stmt =
    Assign of variable * expr
  | Seq    of stmt * stmt
  | If     of expr * stmt * stmt
  | While  of expr * stmt
```

В этом примере использовано расширение синтаксиса *datatype*, которое создает *expr_marker*, *expr_data* и т.д. Их можно использовать для создания трансформации, производящей идентификацию переменных для операторов:

```
let resolveAll resolve =
  let module T = Make_transform (Id_monad) in
  let tr = T.extend (T.return ()) variable_marker resolve in
  (tr.specializeT tr stmt_data) id
```

В приведенном примере задается тип монады, создается тождественная трансформация, затем она расширяется переопределением её воздействия на узлы типа *variable* и специализируется для типа *stmt*.

5 Результаты

В рамках работы над библиотекой удалось достичь следующих результатов:

- разработан типобезопасный подход к обработке данных, основанный на Scrap Your Boilerplate;
- создана поддерживающая его библиотека;
- показана применимость библиотеки к задачам компилятора.

5.1 Производительность

Ручная реализация часто обеспечивает выигрыш в производительности, т.к. у пользователя есть возможность оптимизировать алгоритм обхода под конкретную трансформацию. В таблице приведено сравнение времени работы функций, реализованных вручную, и функций, созданных с помощью разработанной библиотеки. Трансформация — это функция идентификации переменных для *stmt*. Запрос — функция, выделяющая все переменные из той же структуры.

	Трансформация	Запрос
Наивная реализация	5.876366	83.405213
С использованием библиотеки	7.224454	126.963935

Производительность при использовании библиотеки уменьшается в среднем в 1.5 раза. Такой результат является допустимым.

5.2 Применимость к задачам компилятора

С помощью библиотеки были реализованы функции проверки типов и идентификации переменных для компилятора простого языка. В результате код идентификации был уменьшен в 10 раз, а код проверки типов был существенно упрощен. Тем самым была показана её применимость к задачам компилятора.

5.3 Преимущества и ограничения

Основными преимуществами библиотеки перед конкурентами являются:

- прозрачная работа с монадами (нет у конкурентов);

- возможность работы с любыми типами, поддержка конструкторов (нет в Camlp4FoldGenerator);
- простота задания произвольных трансформаций и запросов (нет в deriving и Generic for Ocaml Masses);
- автоматическое обеспечение хвостовой рекурсии за счет продолжений (нет у конкурентов);
- возможность использования внутри расширения синтаксиса (нет у deriving и Camlp4FoldGenerator).

Известные недостатки:

- необходимость вручную поддерживать взаимно-однозначное соответствие между маркерами и типами (в некоторых случаях является положительным свойством);
- отсутствие возможности работать с циклическими данными.

Список литературы

1. Erich Gamma R. J. J. V., Richard Helm. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

2. Lämmel R., Jones S. P. Scrap Your Boilerplate: A Practical Design Pattern for Generic Programming // Proc. of the ACM SIGPLAN Workshop on Types in Language Design and Implementation (TLDI 2003). ACM Press, 2003. Pp. 26–37.

3. Lämmel R., Jones S. P. Scrap your boilerplate with class: extensible generic functions // Proceedings of the tenth ACM SIGPLAN international conference on Functional programming. ICFP '05. New York, NY, USA: ACM, 2005. Pp. 204–215.

4. Sussman G. J. Scheme: An interpreter for extended lambda calculus // Memo 349, MIT AI Lab. 1975.

5. Rodriguez A., Jeuring J., Jansson P. et al. Comparing libraries for generic programming in Haskell. Vol. 44. New York, NY, USA: ACM, 2008. — September. Pp. 111–122.

6. Jeremy Yallop O. K. First-class modules: hidden power and tantalizing promises. 2010.

7. Leroy X. A modular module system. Vol. 10. New York, NY, USA: Cambridge University Press, 2000. — May. Pp. 269–303.

8. Leroy X., Doligez D., Frisch A. et al. The Objective Caml system release 3.12. 2010. URL: `http://caml.inria.fr/pub/docs/manual-ocaml/`.

9. Hinze R. Generics for the masses. Vol. 39. New York, NY, USA: ACM, 2004. — September. Pp. 236–243.

10. Chakravarty M. M. T., Ditu G. C., Leshchinskiy R. Instant Generics: Fast and Easy. 2009.

11. Kiselyov O., Peyton S., chieh Shan J. C. Fun with type functions Version 2. 2009.

12. Mitchell N., Runciman C. Uniform boilerplate and list processing // Proceedings of the ACM SIGPLAN workshop on Haskell workshop. Haskell '07. New York, NY, USA: ACM, 2007. Pp. 49–60.

13. Pierce B. C. Types and Programming Languages. MIT Press, 2002.

14. More expressive GADT encodings via first class modules. 2010. URL: http://ocaml.janestreet.com/?q=node/81.

15. Wadler P. Monads for Functional Programming // Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques-Tutorial Text. London, UK: Springer-Verlag, 1995. Pp. 24–52.

16. Karvonen V. A. Generics for the working ML'er // Proceedings of the 2007 workshop on Workshop on ML. ML '07. New York, NY, USA: ACM, 2007. Pp. 71–82.

17. Cormen T. H., Leiserson C. E., Rivest R. L., Stein C. Introduction to Algorithms, second edition. 2001.

18. Kiselyov O. Smash along your boilerplate. 2007. URL: http://okmij.org/ftp/Haskell/generics.html.