Санкт-Петербургский Государственный Университет
Математико-механический факультет

Кафедра системного программирования

Башкиров Александр Андреевич

Расширение конструкции сопоставления с образцом в языке OCaml

Курсовая работа

Научный руководитель:
к. ф.-м. н., доцент С. В. Григорьев

Консультант:
программист “JetBrains Labs” Д. С. Косарев

Санкт-Петербург
2020

Оглавление

Введение

В контексте языков программирования сопоставление с образцом служит весьма эффективным и выразительным средством анализа данных. Появившееся в первых функциональных языках вкупе с алгебраическими типами данных эта конструкция довольно быстро завоевала популярность и в практически неизменном виде реализовывалась в большинстве последующих функциональных языков. Были разработаны эффективные схемы компиляции [10, 12, 22], проверки на полноту (pattern exhaustiveness) [11] и хрупкость (pattern fragility). Более того, в настоящее время сходные конструкции разрабатываются и добавляются и в некоторые лидирующие императивные языки, такие как C# [13], Kotlin [9] и Java [4].

Однако в своем наиболее распространённом варианте этот приём имеет довольно серьёзные ограничения:

- конструкторы данных должны быть открытыми (public), что не допускает абстракции определения типа данных;
- для каждого типа данных возможно лишь одно представление для использования в конструкции сопоставления с образцом;
- охранные выражения (pattern guards) либо не допускаются языком (как в языке Standard ML [14]), либо могут содержать ровно одно булево выражение (как в языке OCaml на текущий момент [24]);
- семантика сопоставления с образцом жёстко зафиксирована на уровне языка и не допускает определяемых пользователем вычислений[1].

К настоящему времени было предложено множество различных вариантов расширения сопоставления с образцом, решающих упомянутые проблемы и предоставляющих дополнительные возможности. В попытке сравнения существующих решений можно рассмотреть их классификацию по виду устраняемых ограничений, возникающих в следующих случаях.

1. *Со стороны определения образца:* лист может быть только открытым конструктором типа или константой; заметим также, что при этом, как правило, он не является значением первого порядка (first class value).
2. *Со стороны определения сопоставления:* сопоставление с образцом может являться только списком правил, а его семантика жёстко зафиксирована на уровне языка.

В этой работе мы предъявим два независимых расширения языка для устранения каждого типа ограничений. Однако отметим, что в этой работе способы модификации семантики сопоставления с образцом [23, 25] рассматриваться не будут.

[1]Изменение семантики конструкции сопоставления с образцом путем параметризации образцов типами специального класса типов представлено в секции 6.3 работы [23].

Основным подходом к расширению образцов является возможность проведения некоторых вычислений при сопоставлении, благодаря чему становятся возможными абстракция и задействование нескольких представлений при в сущности столь же выразительном синтаксисе. Многие языки программирования уже содержат подобные конструкции в том или ином виде: active patterns в F# [23], pattern synonyms в Haskell [19] и extractors в Scala [5]. В данной же статье мы предложим расширение для языка программирования общего назначения OCaml.

Постановка задачи

Целью работы является разработка расширение языка OCaml, которое бы:

- удовлетворяло требованию *консервативности* (или обратной совместимости) – синтаксис и семантика любой программы, не использующей средства расширения, должны оставаться неизменными;
- являлось полностью статически типизируемым;
- не требовало модификации системы типов языка;
- допускало абстракцию образцов;
- сохраняло возможность проверки сопоставления на полноту;
- позволяло оперировать образцами как значениями первого порядка;
- не требовало модификации существующих модулей.

Были поставлены следующие задачи:

1. Провести анализ существующих решений и выполнить сравнение по ряду объективных критериев. По результатам обосновать выбор того или иного существующего решения или представить собственное.
2. Формализовать изменения, которые необходимо внести в компилятор языка OCaml, для поддержки нового расширения:
    - расширение синтаксиса
    - правила типизации новых конструкций
    - компиляция расширения в промежуточное представление
3. Представить прототип реализации выбранного расширения.

1. Обзор

1.1. Views

Ф. Вадлер (Philip Wadler) [26] одним из первых указал на недостатки традиционной модели образцов: сопоставление с образцом является мощным выразительным средством и ясно (в некоторых случаях почти досимвольно) выражает индуктивные математические определения, но требует открытого представления типа. Абстракция данных поддерживает эффективность[2] и инкапсуляцию, но требует скрыть представление типа – таким образом программист оказывается перед дилеммой выразительного или эффективного кода.

Для решения этой проблемы Ф. Вадлер предложил концепцию *представлений* (views): определяемое представление состоит из целевого типа данных, по которому проводится сопоставление с образцом, и двух функции in и out, которые определяют преобразование в целевой тип данных и обратно соответственно. Можно определить несколько представлений с соответствующими преобразованиями и экспортировать их вместе с настоящим устройством типа или без него, при этом для пользователя они будут равноправны. Интересно заметить, что в терминах представлений могут быть выражены такие особенности сопоставления с образцом как as-образцы и охранные булевы выражения.

Сама концепция представлений оказалась довольно успешной, были представлены реализации для языков Miranda, Standard ML и Haskell. Однако имелись и серьезные недостатки: корректно определенное представление требует изоморфизма с исходным устройством типа, а функции in и out должны являться взаимно обратными. Это довольно серьезное ограничение, которое к тому же невозможно проверить средствами компилятора. Более того, наличие обеих функций вызывало проблемы с трактованием чистых (pure) вычислений как системы перезаписи (equational reasoning). Также при использовании представлений довольно сложно проследить, где будут происходить затратные вычисления. Ну и наконец содержательно одна и та же функциональность может быть реализована как функцией, так и представлением, которые подчас выражаются друг через друга, что приводит к шаблонному коду.

Поэтому в последующие несколько лет было предложено множество вариантов, устраняющих те или иные недостатки оригинальной концепции. Интересной представляется работа К. Окасаки (Chris Okasaki) [17]: автор предложил вариант представлений для языка Standard ML, представителя семейства языков ML, к которому также принадлежит и OCaml. В своей работе К. Окасаки ограничил представления до одной функции in, сняв тем самым требования изоморфизма и устранив проблемы с семантикой системы перезаписи, а так же показал, как представления встраиваются в систему модулей ML и каковы особенности использования функций с эффектами внутри таких представлений.

[2]В качестве примера можно рассмотреть представление натуральных чисел в виде эффективного машинного типа целого числа и совершенно неэффективное, но формальное и удобное представление в виде алгебраического типа данных согласно математическому определению Пеано. Подробнее см. в [26].

1.2. Pattern guards and Transformational Patterns

Примерно в это же время был предложен альтернативный подход к данной задаче: М. Эрвиг (Martin Erwig) и С. П. Джонс (Simon Peyton Jones) представили два расширения сопоставления с образцом: охранные выражения и *преобразующие* образцы (transformational patterns) [6]. Охранные выражения позволяют в одной ветке разбора проводить несколько последовательных сопоставлений. Ключевой здесь является возможность вычисления функции от привязанных переменных, результат которой также проходит сопоставление. Такое простое и по синтаксису, и по семантике, и по реализации правило автоматически позволяет абстрагироваться по любому образцу, причем зачастую с возможностью использования уже существующих функций.

К примеру, рассмотрим класс типов для простейшего списка:

```
class AbsList c where
  nil   :: c a
  cons  :: a → c a → c a
  head  :: c a → Maybe (a, c a)
```

Тогда сопоставление с образцом можно записать как

```
instance AbsList c ⇒ Functor c where
  fmap f l
    | Just (x, xs) ← head l = f x `cons` fmap f xs
    | Nothing      ← head l = nil
```

В последовательно стоящих охранных выражениях можно использовать привязки из предыдущих:

```
filterMap :: AbsList c ⇒ c a → (a → Maybe b) → c b
filterMap f l
  | Just (x, xs) ← head l,
    Just v   ← f x          = v `cons` filterMap f xs
  | Just (x, xs) ← head l,
    Nothing ← f x           = filterMap f xs
  | Nothing      ← head l   = nil
```

Таким образом любой одиночный образец для выражения типа τ может быть представлен как функция `pat :: `τ `→ Maybe` ζ, где ζ являет собой некоторый кортеж, размерность которого определяет количество привязываемых при сопоставлении переменных, а его составляющие – типы этих привязок. Отметим, что в этом смысле `Maybe ()` эквивалентен `Bool`, а неопровержимые (irrefutable) образцы могут представлены просто как `irrPat :: `τ `→` ζ. Используется `pat` как

```
f x
  | Just (...) ← pat x = ...
```

Заметим, однако, небольшую синтаксическую особенность – при таком использовании первым зачастую идет неопровержимый образец, именующий само инспектируемое выражение (scrutinee) – например, `x` в выражении `f x` выше. А уже образец, непосредственно разбирающий выражение, идет после, в охранном выражении. При

этом, как в примере с `filterMap`, если мы хотим применить функцию к одной из полученных привязок, необходимо записать это в следующем охранном выражении, что отличается от традиционной вложенной формы записи, при которой подобразцы записываются прямо на месте привязок. К тому же это вызывает дублирование отдельных охранных выражений, как с `Just (x, xs) ← head l` в примере с `filterMap`.

Для устранения этой особенности в пользу более привычной формы записи в работе был предложен синтаксический сахар – преобразующие образцы. Они позволяли записывать сопоставление с результатом функции непосредственно внутри образца. Правда, насколько нам известно, в первоначальном варианте, предложенном в работе, они так и не были реализованы, поскольку вместо них довольно быстро были добавлены так называемые *представляющие* образцы (view patterns), реализующие ту же самую идею. Используя эту нотацию `filterMap` можно переписать как

```
{-# LANGUAGE ViewPatterns  #-}

filterMap f (head  →  Just (f  →  Just v,  xs))
  = v `cons` filterMap f xs
filterMap f (head  →  Just (f  →  Nothing, xs)) = filterMap f xs
filterMap f (head  →  Nothing)                  = nil
```

Подробное сравнение охранных выражений с представлениями приведено в самой работе [6]. Говоря кратко, в общем случае ни один из подходов не может быть выражен в терминах другого, но утверждается, что охранные выражения значительно проще для определения и реализации, позволяют использовать ранее написанные функции без дополнительных определений и явно отражают места, где могут происходить затратные вычисления.

Заметим также, что коль скоро образцы представлены функциями, они могут быть дополнительно параметризованы некоторым значением. Например, в этом смысле функция `take` для класса типов `AbsList` также представляет собой образец:

```
destutter l@(take 2  →  Just [f; s]) | f = s
                                     = destutter (tail l)
destutter (head  →  Just (x, xs)) = cons x (destutter xs)
destutter (head  →  Nothing)      = nil
```

Более того, образцы могут быть параметризованы другими образцами. Поддержка языком этого свойства крайне важна для реализации комбинаторов образцов (pattern combinators) [25].

1.3. Scala Objects patterns

В работе [5] было предложено развитие идеи, что любой одиночный образец может быть представлен функцией типа `pat :: τ → Maybe ζ`. Образец в языке Scala определятся как функция `unapply(obj: τ): Option[ζ]`[3], реализованная в некотором объ-

[3]Более полное и формальное определение вы можете найти в спецификации языка Scala: `https://www.scala-lang.org/files/archive/spec/2.11/08-pattern-matching.html#extractor-patterns`

екте, именуемом *экстрактором*.

Однако, по сравнению с представляющими образцами в Haskell при сопоставлении с образцом никаких дополнительных синтаксических элементов вроде → или проверки с конструктором `Just` не требуется, что позволяет писать более краткий и выразительный код. К примеру, аналогичное определение `AbsList` и реализация функции `filterMap` на языке Scala:

```
trait AbsList[+A] {
  def nil[B]: AbsList[B]
  def cons[B, U >: B](hd: U, tl: AbsList[B]): AbsList[U]
  def head: Option[(A, AbsList[A])]
  // Extractor signatures:
  trait ConsExSig {
    def unapply[A](l: AbsList[A]): Option[(A, AbsList[A])] =
      l.head
  }

  trait NilExSig {
    def unapply[A](l: AbsList[A]): Boolean = l.head.isEmpty
  }
  // Extractors
  val Cons = new ConsExSig {}
  val Nil  = new NilExSig  {}
}

def filterMap[A, B](f: A ⇒ Option[B], l: AbsList[A])
  : AbsList[B] = {
  import l._
  l match {
    case Cons(x, xs) if f(x).isDefined
                     ⇒ cons(f(x).get, filterMap(f, xs))
    case Cons(_, xs) ⇒ filterMap(f, xs)
    case Nil()       ⇒ nil
  }
}
```

Отметим, что приведённая реализация позволяет переопределять экстракторы в классах, реализующих свойство (trait) `AbsList`.

В то же время охранные выражения в Scala могут содержать только булево выражение и не позволяют определять дополнительных привязок. При этом иногда, как и в примере выше, возникает ситуация, когда в охранном выражении нужно произвести некоторое вычисление и проверить его результат, а затем в случае успеха воспользоваться этим результатом внутри выражения – вместо этого сейчас выражение `f(x)` дублируется и более того вычисляется повторно. Чтобы избежать этой ситуации можно определить объект-обертку, реализующий `unapply` для `f`:

```
class Wrapper[A, B](f: A ⇒ Option[B]) {
    def unapply(arg: A): Option[B] = f(arg)
}

def filterMap[A, B](f: A ⇒ Option[B], l: AbsList[A])
  : AbsList[B] = {
  import l._
  val F = new Wrapper(f)
  l match {
    case Cons(F(v), xs)  ⇒  cons(v, filterMap(f, xs))
    case Cons(_, xs)     ⇒  filterMap(f, xs)
    case Nil()           ⇒  nil
  }
}
```

Экстракторами могут быть произвольные объекты времени выполнения, класс которых содержит метод `unapply` подходящего типа, что позволяет параметризацию образцов.

Также Scala предоставляет синтаксический сахар для разбора последовательностей. К примеру

```
def foo[A](l: AbsList[A]): Unit = {
  import l._
  l match {
    Cons(x, Cons(y, Cons(z, Cons(w, _)))) ⇒ () // 1
    AList(x, y, z, w, _ *)                ⇒ () // 2
    _                                     ⇒ ()
  }
}
```

Ветвь 2 представляет более краткий эквивалент ветви 1. Для применения такой нотации объект `l` должен содержать объект-экстрактор `AList`, реализующий метод `unapplySeq[A](l: AbsList[A]): Option[Seq[A]]`

Дополнительно отметим, что Scala поддерживает символьные инфиксные наименования классов, благодаря чему сопоставления с образцом также могут содержать символьную инфиксную нотацию. Например, `::` в примере ниже реализовывается стандартными средствами языка, не требуя дополнительной синтаксической поддержки, как в языке OCaml:

```
List(42) match {
  42 :: xs ⇒ true
  _        ⇒ false
}
```

Как видно, экстракторы Scala являются мощным и выразительным средством для представления абстрактных образцов, возможно, требуя некоторых простейших объектов-оберток над замыканиями. Главным неудобством же является неявное от-

ключение проверки на полноту при использовании экстракторов. Рассмотрим простейший пример

```
object MySome {
  def unapply[A](x: Option[A]): Option[A] = x
}

def unwrap(v: Option[Int]): Int =
  v match {
    case MySome(i) ⇒ i
  }
```

В этом случае мы не получаем предупреждения от компилятора о неполноте сопоставления – проблема заключается в том, что компилятор не может вывести какое значение в действительности не покрывается сопоставлением. А раз компилятор не обладает доказательством нарушения, проверка считается пройденной.

Способом указания компилятору Scala на совместную полноту экстракторов является использование в качестве реализации типа данных иерархию из базового *запечатанного* (sealed) класса и его наследников case классов, предоставляющих экстракторы по умолчанию – такое решение снова нарушает абстракцию. Использование же case классов в качестве оберток над произвольными экстракторами с целью указания на их совместную полноту является сложной задачей даже для опытного Scala разработчика и требует значительных умственных и синтаксических затрат.

1.4. F# Active Patterns

Решение, предложенное в работе [23] обладает схожей выразительностью и в то же время позволяет объединять отдельные образцы в один многозначный, сопоставление с которым может быть проверено на полноту.

Например, для того же класса типов `AbsList` (в ML-языках представляемого модулем) можно объединить образцы `Cons` и `Nil` в один неопровержимый двузначный образец:

```
let (|Cons|Nil|) l =
  match head l with
  | Some(hd, tl) → Cons(hd, tl)
  | None         → Nil
```

В качестве примера использования рассмотрим функцию `destutter`, удаляющую последовательные дубликаты элементов:

```
let rec destutter = function
  | Cons(hd, Cons(hd', _) as tl) when hd = hd' → destutter tl
  | Cons(hd, tl) → cons hd (destutter tl)
  | Nil          → nil
```

При этом будет произведена проверка на полноту, которую эта реализация пройдет благодаря второй и третьей ветвям, определяющим все возможные образцы для

входного значения.

В такой реализации используются "анонимные" типы-суммы, имена конструкторов которых не зависят от предметной области и состоят лишь из имени типа и порядкового номера. Такие типы предопределяются языком F#, к примеру

```
type ('a, 'b) choice =
  | Choice2_1 of 'a
  | Choice2_2 of 'b
```

Такое решение представляется также крайне удачным для реализации в языке OCaml, благодаря поддержке языком полиморфных вариантов, что любезно заметили авторы работы [23]. Полиморфные варианты позволяют заменить анонимные типы на более информативные и гибкие конструкторы-метки, к примеру

```
val (|Cons|Nil|) :
  'a AbsList.t →[`Cons of ('a * 'a AbsList.t) | `Nil]
```

Вместо

```
val (|Cons|Nil|) :
  'a AbsList.t →('a * 'a AbsList.t) choice
```

Помимо полных (total) образцов, предложенное расширение также позволяет определять и обычные частичные образцы в терминах все тех же функций с результирующим типом ζ `option`. При этом образцы также могут быть параметризованы различными значениями, в частности, другими образцами.

С технической же точки зрения, авторы работы предложили модификацию схемы компиляции [22], отличную от используемой компилятором OCaml [10]. Представляется интересным разработать модификацию схемы [10] для внедрения подобного расширения.

1.5. Haskell Pattern Synonyms

Пожалуй, единственным недостатком активных образцов, предложенных для F#, по сравнению с обычными образцами конструкторов типа, является тот факт, что первые можно использовать только в сопоставлении с образцом, но не для конструирования данных. Так в примере выше для разбора значения списка используется активный образец `Cons`, а для конструирования отдельная функция `cons`. При этом даже самое первое решение в виде представлений [26] позволяет использовать один и тот же терм образца в обоих случаях.

Предложенное в работе [19] решение в виде синонимов образцов позволяло определять *неявные* (implicit) или *явные двунаправленные* (explicit bidirectional) образцы, термы которых можно использовать как для конструирования, так и для разбора значений. К примеру, для класса типов `AbsList` можно определить

```
pattern Nil :: AbsList c ⇒ c a
pattern Nil ← (head → Nothing) where
  Nil = nil
```

```
pattern Cons :: AbsList c ⇒ a → c a → c a
pattern Cons x xs ← (head → Just (x, xs)) where
  Cons x xs = cons x xs
```

Тогда эквивалентное определение функции `destutter` с использованием синонимов образцов может быть записано следующим образом:

```
destutter Cons(hd, tl@Cons(hd', _)) | hd == hd' = destutter tl
destutter Cons(hd, tl) = Cons hd (destutter tl)
desturrer Nil          = Nil
```

Как видно, содержательно происходит тоже самое, но компилятор автоматически подставляет соответствующее действие на место терма. К сожалению, как подробно описано авторами в работе, цена у такого простого синтаксического сахара оказалась высокой: поскольку у одного терма может быть два определения, типы индуцированные этими определениями могут не совпадать. В работе представлен следующий пример:

```
pattern Q :: Ord a ⇒ a → a → (a, a)
pattern Q x y ← (x, y) where
  Q x y | x ⩽ y    = (x, y)
        | otherwise = (y, x)
```

Здесь для конструирования требуется ограничение `Ord`, в то время как для сопоставления нет. Сигнатура образца выводится только из определения сопоставляющей части. Поэтому в данном случае потребуется явно указать сигнатуру, поскольку выведенная окажется слишком общей: `Q :: a → b →(a, b)`.

Уникальной особенностью предложенного расширения являлись гибкие сигнатуры образцов, которые позволяли выражать такие особенности системы типов как экзистенциальные квантификаторы и GADTs. Однако для его реализации в языках семейства ML потребуется серьезная адаптация, ввиду отсутствия активно используемых в расширении классов типов и подобных ограничений, функциональность которых в OCaml представлена значительно отличающейся системой модулей и функторов.

Недостатком синонимов образцов является отсутствие возможности группировки отдельных синонимов для указания на их совместную полноту по входному значению. Т.е. в терминах активных образцов F# все синонимы образцов являются частичными (partial). Также, к сожалению, синонимы образцов не могут быть переданы напрямую в функцию и не поддерживаются в качестве составляющих класса типов, а их параметризация требует определения новых классов типов, что синтаксически чуть более тяжеловесно, нежели параметризация активного образца по месту использования.

2. Оператор возврата к сопоставлению

В данном разделе мы представим собственное расширение сопоставления с образцом. Его особенностью является минимально возможное вмешательство в язык:

- требует введения в язык всего одного ключевого слова;
- типизируется элементарно как $\forall a\ a$;
- тривиально и достаточно эффективно встраивается в любую схему компиляции;
- проверка на полноту также легко расширяется для учета новой особенности.

Вдохновением для такого дизайна послужила схема компиляции сопоставления с образцом в автомат поиска с возвратами (backtracking automata) [10, 1], оптимизированный вариант которой используется сейчас в OCaml.

В простейшем варианте такой схеме компиляции на вход поступает матрица образцов и вектор значений. Первым шагом происходит применение *разделяющего* правила (mixture rule [10]), которое некоторым образом делит матрицу на группы последовательных строк, рекурсивно компилируемых другими правилами. При компиляции группы для случая, когда сопоставление внутри этой группы не удалось, генерируется бросание *статического*[4] исключения. Разделяющее же правило ответственно за вставку статических обработчиков и линейное упорядочивание групп.

Рассмотрим пример компиляции слияния списков из [10] в промежуточное представление компилятора OCaml (также подробно описанного в [10]):

```
let merge lx ly =
  match lx, ly with
  | []   , _    → 1
  | _    , []   → 2
  | x::xs, y::ys → 3
```

```
catch
  (catch
    (switch lx with case []: 1
      default: exit)
  with (catch
    (switch ly with case []: 2
      default: exit)
  with (catch
    (switch lx with
      case (::):
        (switch ly with
          case (::) : 3
          default: exit)
      default: exit))))
with (failwith "Partial match")
```

Рис. 1: Простейшая компиляция слияния списков

[4] для которого на уровне компиляции известна точка обработки, а значит оно может быть скомпилировано в единственную инструкцию перехода

Инициализация возврата к предыдущим сопоставлениям происходит бросанием статического исключения (оператор `exit`), перехват которого (оператор `catch ... with ...`) означает переход к рассмотрению следующей ветви. Такие команды бросания исключения генерируются самой схемой компиляции и только внутри разбора ветви.

Данное расширение предлагает позволить *пользователю* инициировать такие возвраты, причем не только внутри образца, но и в теле ветви, т.е. уже после успешного разбора образцов. Для этого требуется добавить в язык *оператор возврата к сопоставлению* (далее для краткости именуемый как оператор возврата), представляемый ключевым словом `backtrack`. Идеалогически такая конструкция близка широко используемым в императивных языках операторам возврата из цикла `break` и `continue`, позволяющих ограниченное ручное управление потоком управления.

Очевидно, что такое решение позволяет не только легко эмулировать обобщенные охранные выражения, но и избавляет их от проблемы дублирования префиксов разбора! Общий префикс выносится в во внешнее сопоставление, а различные суффиксы — во внутреннее. Рассмотрим несколько модифицированный синтетический пример из [16], записанный на языке Haskell с использованием обобщенных охранных выражений:

```
foo e1 e2 =
  case e1, e2 of
    Nothing, true  → 1
    (Just x with Foo y   = lookup tbl x), false  → y
    (Just x with Bar a,b = lookup tbl x), false  → a + b
    _, true  → 3
    _, false  → 4
```

Как видно, третья ветвь синтаксически дублирует вторую, отличаясь только глубоко вложенным образцом `Bar a, b` (заметьте, что в такой вариации даже ИЛИ-образцы неприменимы). При этом нельзя перенести сопоставление с `Foo` и `Bar` во вложенное сопоставление, поскольку данные вида `(Just x with Zet = lookup tbl x), false` должны попасть в последнюю ветвь.

Оператор возврата помогает избавиться от дублирования образцов и естественно выразить задуманное:

```
let t e1 e2 =
  match[@label outer] e1, e2 with
  | None, true → 1
  | Some x, false →
      begin match lookup tbl x with
      | Foo y     → y
      | Bar(a, b) → a + b
      | _ → backtrack%outer
      end
  | _, true → 3
  | _, false → 4
```

Заметьте, что оператору `backtrack` требуется указание метки возврата, которые мы предлагаем записывать через систему атрибутов и расширений, служащих в языке OCaml специально для записи метаинформации.

Такой код будет скомпилирован в следующее промежуточное представление:

```
1  catch
2    (switch* e1 with
3      case None: (if (!= e2 0) 1 exit)
4      case Some:
5        (if (!= e2 0) exit
6          (let (x (field 0 e1))
7            (switch (apply lookup tbl x) with
8              case Foo: (field 0 x)
9              case Bar: (+ (field 0 x) (field 1 x))
10             default : exit)))
11 with
12   (if (!= e2 0) 3 4)
```

Здесь мы для краткости генерируемого кода использовали информацию о полноте `switch` в строке 2 и опустили для него метку `default`.

Обратите внимание на строку 10: стоящая там команда `exit` является результатом компиляции оператора `backtrack` и притом только она. Таким образом, мы получаем отображение `backtrack` на `exit` один к одному. В действительности не требуется более добавления никаких конструкций, даже `catch ... with ...`, которые будет введены автоматически правилом разделения, используя информацию о полноте.

В общем случае, разумеется, `exit` необходимо снабдить индексом, поскольку иногда требуется подняться на более чем 1 уровень. Но актуальная схема компиляции [10] уже обладает такими метками (введенными для оптимизаций), поэтому не требуется никакого усложнения существующего промежуточного представления.

Схему компиляции в деревья решений также несложно модифицировать для поддержки оператора возврата. Более того в действительности, многие компиляторы, такие как компилятор языка Rust [21], уже используют инструкции безусловного перехода для компиляции охранных выражений: охранные выражения компилируются внутри тела ветви, а для случая не допуска генерируется переход на сопоставление следующей ветви. Аналогичные действия предпринимаются и в компиляторе языка OCaml.

Проверку на полноту можно расширить следующим образом: для ветви, в теле которой есть команда `backtrack`, относящаяся к сопоставлению этой ветви или вышестоящему, в конец строки образцов добавляется образец на целочисленную константу 42. А в конец вектора значений неявно добавляется некоторое фиктивное значение 0. Такие действия мотивированы тем, что сопоставления с целочисленной константой всегда неполно, поскольку сигнатура типа `int` считается бесконечной. А это ровно та ситуация, которая нам требуется — для ветвей, содержащих `backtrack`, заставить механизм предупреждений [11] обрабатывать эту ветвь, как если бы для нее существовали значения, не проходящие успешного сопоставления. Можно показать, что при

таком подходе, остальные проверки вроде избыточности (redundancy) также будут обладать желаемой семантикой.

Напоследок сделаем замечание о практической применимости: как будет далее показано в разделе 3.3 оператор возврата к сопоставлению с теоретической точки зрения обладает максимально-возможной гибкостью. При этом такой оператор крайне эффективно компилируется. В действительности необходимость в таких конструкциях возникает довольно редко, но если все-таки возникает, как правило, иного способа, кроме как изменить высокоуровневую структуру кода, не остается, что может быть в свою очередь менее эффективно по производительности.

С другой стороны, оператор возврата является низкоуровневой конструкцией, противоречащей духу ML-языков, к тому же его использование как правило синтаксически довольно тяжеловесно, что впрочем можно улучшить введением синтаксического сахара, наподобие конструкции `if–let` [15] или тех же обобщенных охранных выражений, которые очевидным образом транслируются в операторы возврата.

Язык OCaml стремится достигать баланса между академичностью и производительным решением практических задач, а посему почти бесплатная (во всех отношениях) конструкция `backtrack` может считаться оправданной для последней категории.

3. Выбор конструкции для реализации

3.1. Введение объективных критериев для сравнения

Для выбора конкретного дизайна необходимо научиться объективно сравнивать существующие решения, для чего в свою очередь необходимо ввести некоторые критерии. К настоящему моменту таких критериев не было сформулировано: существующие решения сравнивались на некоторых отдельных группах примерах, оперируя такими субъективным понятиями как "краткость синтаксиса", "расширяемость", "легковесность", "более общее" и т.п.

Для введения подходящих критериев стоит обратиться к теоретической части задачи. Что такое "образец"? Что такое "сопоставление с образцом"? Что такое "конструкция сопоставления с образцом"? Какие теоретические задачи они призваны решать? Опять-таки в существующих работах подобные понятия определяются специфично для конкретного языка: вводятся некоторые синтаксические конструкции, наделяются определенной семантикой, и их называют уже перечисленными выше терминами. Поскольку сами определения как правило привязаны к конкретному языку, сравнивать конструкции между языками становится достаточно сложно.

М. Туллсен [25] первым указал на данную проблему и сосредоточился на теоретической части. Введенные им определения не зависят от конкретного языка и применимы к любому статически типизированному языку с функциями высших порядков и алгебраическими типами данных.

Так *образец* определяется как произвольная функция типа $\tau \to$ `Maybe` ζ, *сопоставление с образцом* определяется как применение данной функции к значению и декомпозиции результата (который ограничен типом `Maybe` ζ).

Для формирования образцов вводится некоторое множество *комбинаторов образцов* (определения и семантика которых подробно описаны в оригинальной работе), позволяющих определять составные образцы из более простых:

```
type ('a, 'b) pat = 'a → 'b option

(* Комбинатор ИЛИ *)
val ($|): ('a, 'b) pat → ('a, 'b) pat → ('a, 'b) pat
(* Комбинатор И *)
val ($&): ('a, 'b) pat → ('a, 'b) pat → ('a, 'b * 'b) pat
(* Комбинатор композиции *)
val ($:): ('a, 'b) pat → ('b, 'c) pat → ('a, 'c) pat
(* Комбинатор функтор-отображения *)
val ($>): ('a, 'b) pat → ('b → 'c)   → ('a, 'c) pat
(* Комбинатор параллельного разбора *)
val ($*): ('a, 'c) pat → ('b, 'd) pat → ('a * 'c, 'b * 'd) pat
```

Заметим, что исходя из самого определения образцы являются значениями первого порядка, могут применяться для значений абстрактных типов, а семантика образцом (и комбинаторов) может быть определена произвольным образом на уровне самого языка. Таким образом, данное теоретическое определение покрывает все известные

практически случаи использования образцов.

Наконец, представляется естественным определить *конструкцию сопоставления с образцом*, как встроенный язык описания предметной области (Embedded DSL), предоставляемый хост-языком для определения образцов. Заметим, что существующие языки используют данную конструкцию одновременно и для применения сопоставления с определяемым образцом.

Благодаря такому определению мы естественным образом получаем критерий для сравнения "мощности" различных дизайнов конструкций сопоставления:

Определение 1 (Максимальность)**.**
Назовем конструкцию сопоставления с образцом ***максимальной****, если она позволяет выразить произвольную композицию комбинаторов образцов.*

Также интересным представляется формализовать свойство синтаксической краткости или выразительности:

Определение 2 (Выразительность)**.**
Назовем конструкцию сопоставления с образцом ***выразительной****, если при определении произвольного образца не требуется дублирования синтаксически одинаковых образцов, за исключением, быть может, идентификаторов-привязок.*

Заметим, что для такого критерия его невыполнимость проверяется простым приведением контрпримера, а вот его выполнимость это формально технически сложное доказательство.

Крайне важно отметить, что *полнота сопоставления* еще формально не определена. Формально "полнота" это свойство именно образца, однако по исторически причинам прижился термин "полнота сопоставления". Кажется естественным определить образец полным, если при сопоставлении с ним могут быть получены только "непустые" значения, т.е. по М. Туллсену, все результирующие значения имеют тег `Just`. Несложно показать, что такое свойство *неразрешимо*. Тотальные активные образцы F# предлагают некоторое решение этой проблемы, позволяя в качестве образцов использовать также значения типа $\alpha \to \xi$, где ξ произвольный анонимный тип-сумма. В такой трактовке сопоставление с такими образцами всегда будет выдавать только "полезное" значение. Резюмируя, при достаточно мощной конструкции сопоставления в языке проверка на полноту в базовом определении неразрешима. Однако, ослабив определение образцов, можно выделить разрешимую группу полных *по определению* образцов, добившись некоторой достаточной для по крайней мере некоторых практических задач аппроксимации полноты.

3.2. Сравнение существующих решений

Произведем сравнение различных дизайнов по следующему ряду критериев:

- *максимальность*, введенная в предыдущем разделе, отражает практическую способность расширения решать поставленный круг задач;

- *выразительность*, введенная в предыдущем разделе, отражает синтаксические затраты на реализацию задачи;
- *свойство языка представлять образцы как значения первого класса* (заметим, что исходя из данного нами определения, это, к сожалению, не следует из свойства максимальности);
- *проверка на полноту*, как уже было сказана, неразрешима, тем не менее нас интересует по крайней мере наличие у расширения формальных аппроксимаций;
- *эффективность* — поскольку конструкция сопоставления с образцом является центральной для функциональных языков к ним предписываются жесткие требования по эффективной компиляции. Здесь нас интересуют наличие в литературе модификаций для эффективных схем компиляции, таких как [10, 12, 22].

Результаты представлены в таблице 1.

Дизайн	Макс-ть	Выр-ть	1п.	Проверя-ть	Эффект-ть
Views	—	—	—	+	**?**
Views for SML	—	—	—	+	+
Pattern guards	+	—	+	+	**?**
First class patterns	+	+	+	+/—	—
View patterns	+	—	+	+	**?**
Objects patterns	—	—	+	**?**	**?**
Active Patterns	+/—	—	+	+	+
Pattern Synonyms	—	—	**?**	+	**?**
Оператор возврата	+	+/—	+	+	+

Таблица 1: Сравнение конструкций сопоставления

Приведем краткие доказательства и замечания по каждому из пунктов.

3.2.1. Максимальность

Все отрицательные значения в столбцах возникают по причине, что соответствующие расширения не предоставляют возможности параметризовать часть образца значением (представления, представления SML, образцы представленные объектами и синонимы образцов), в частности значение, полученным привязкой другого образца (активные образцы). Можно рассмотреть следующий синтетический пример из [7]:

```
example :: ((String → Integer,Integer), String) → Bool
example ((f,_), f → 4) = True
```

Здесь значение `f`, привязанное в одной части образца, используется в другой (в данном случае само значение представляет собой view образец). Такое достаточно важное свойство (которое мы будет называть *параметризацией слева-направо* (left-to-right parameterization)) поддерживается только дизайнами представляющих образцов (view patterns) и обобщенных охранных выражений (pattern guards). Однако отметим,

что активные образцы также очевидным образом могут поддержать данную семантику, несмотря на то, что реализация языка F# не предлагает такой возможности. Такая дополнительная возможность никоим образом не влияет на проверку на полноту и мемоизацию [17]. Единственный объективный недостаток такой возможности заключается в том, что для схем компиляции в деревья решений мы получим очевидное ограничение на порядок столбцов, что впрочем с практической точки зрения вряд ли можно назвать существенным.

3.2.2. Выразительность

Согласно таблице критерий выразительности в текущей его формулировке оказалось совершенно нерепрезентативен — при любом расширении так или иначе существует пример, требующий дублирования кода.

Для всех расширений можно привести следующий синтетический пример дублирования, возникающий при использования комбинатора композиции и комбинатора ИЛИ.

```
type t = C1
       | C2
       | C3

let _ =
  match[@label outer] a, b with
  | _, b →
     begin match b with
     | C1 → 1
     | C2 → 2
     | C3 → backtrack%outer
     end
  | C2, C3 → 3
  | _      → 4
```

Данный пример записан с помощью использования оператора возврата к сопоставлению, однако при попытке переписать его средствами других расширений мы неизбежно получим дублирование одного из образцов (которые в общем случае могут быть произвольно сложными).

Идея такого сопоставления заключается в том, что внутри одной ветви мы можем разбирать какую-то часть данных, но при определенных обстоятельствах иметь возможность вернуться к общему сопоставлению и продолжать разбирать его в надежде, что найдется некоторая другая комбинация образцов, под которую попадут данные.

Несмотря на некоторую кажущуюся искусственность ситуации такой шаблон в действительности возникал при написании промышленного ПО. В качестве примера, мы можем привести код на языке Scala [8], преобразующий язык из одного представления в другое. При этом существует некоторое ограниченное множество исключительных ситуаций, которые по тем или иным причинам необходимо обработать особым образом. Множество ситуаций представлено одним большим сопоставлением,

при этом каждая часть конструкции также имеет смысл быть разобранной с помощью вложенного сопоставления с возможностью возврата к внешнему, что в языке Scala невыразимо в виду отсутствия оператора возврата.

Оператор возврата же в действительности не имеет этой проблемы и формально даже является выразительным расширением, однако этот дизайн также нельзя назвать таковым с практической точки зрения ввиду огромного синтаксического шума ключевых слов при попытке использования образцов над абстрактными типами данных, а также вырождения сопоставлений в последовательность и невозможности использования привычного структурного вложения образцов:

```
let rec destutter2 l =
  match head l with
  | Some(hd, tl) →
      begin match head tl with
      | Some(hd', _) when hd = hd' → destutter tl
      | _ → cons hd (destutter tl)
      end
  | None → nil
```

Сравните данную реализацию с предложенной в разделе 1.4 на основе активных образцов.

Таким образом решение данной проблемы состоит в том, чтобы совместить дизайны предоставляющие возможность возвратов (такие как pattern guards или оператор возврата) и дизайны предоставляющие абстракции на уровне структурного вложения (такие как view patterns и активные образцы). Можно заключить, что текущая экосистема Haskell, обладающая как pattern guards, так и view patterns наиболее близка к оптимальности по выразительности и не достигает ее лишь потому, что, как уже упоминалось в разделе 2 pattern guards не позволяют разбирать несколько альтернатив внутри ветви.

3.2.3. Образцы первого класса

Выполнение или невыполнение данного свойства напрямую следует из определения каждого дизайна. Особенный случай здесь представляют лишь синонимы образцов, для которых кажется очевидным возможность использования их в качестве значений первого класса, но конкретной реализации пока не предоставлено — данный вопрос оставлен открытым и в самой работе [19].

3.2.4. Проверка на полноту

Как уже было отмечено, проверка на полноту в строгом смысле неразрешима, однако для каждого из расширений представлены достаточно точные аппроксимации. Впрочем точность, сложность реализации и производительность этих проверок довольно сильно варьируются от дизайна к дизайну. Скажем, для активных образцов и представлений SML такая проверка может быть реализована достаточно просто и эффективно [11], но вот для множества расширений языка Haskell проверка весьма

затруднительна и лишь недавно [20] был предложен сложный комплекс алгоритмов, использующих к примеру SMT-решатели для повышения точности выдаваемых предупреждений.

Напомним, что для образцов первого порядка М. Туллсена, как уже упоминалось в 3.1, разумной аппроксимации можно добиться, только расширив их определение.

Для образцов представляемых объектами [5] ситуация с проверкой на полноту, как упоминалось ранее в 1.3, довольно сложна. Более того семантика языка Scala предписывает вызов функции-экстрактора на каждое сопоставление, не используя мемоизацию, и дозволяет использование эффектов, что, как описано в [17], кратно усложняет определения полноты и избыточности ветвей. Вопрос, какой аппроксимации можно добиться при таком дизайне остается открытым.

3.2.5. Эффективность

Эффективная компиляция сопоставлений над абстрактными типами данных это малоизученная и открытая область для исследований. Для активных образцов (а равно и для представлений SML, выразимых через них) в оригинальной работе [23] представлена компиляция в деревья решений [22], и авторы утверждают, что можно добиться лучших результатов, используя дополнительные возможности платформы, такие как кортежи, передаваемые на стеке (unboxed tuples). Также открытым остается вопрос, какие из расширенного ряда оптимизаций, представленных в [12], применимы для активных образцов. Для комплекса расширений языка Haskell неизвестно публикаций, раскрывающих вопросы оптимальной компиляции. Необходимо также учитывать различия стратегий вычисления и виды разрешенных алгебраических эффектов, используемых в языках, что еще более усложняет объективные сравнения.

Необходимо также сделать небольшое замечание касаемо образцов первого порядка М. Туллсена. Несмотря на то, что они представляют образцовую теоретическую модель, с практической точки зрения такая технология совершенно неприменима. Заметим, что создание образцов происходит во время исполнения (что, с одной стороны, повышает теоретическую гибкость), а сопоставление с ними вычисляется строго в соответствии с тем, как они были сконструированы, что не допускает *логических* оптимизаций. Наконец, сопоставление с образцами активно использует вызовы функций, что в общем случае не допускает *физических* оптимизаций.

Для решения этих проблем можно предложить два подхода:

- Использование *динамического компилятора*, который мог бы специальным образом обрабатывать такие конструкции и специализировать их непосредственно по время исполнения. Это позволит добиться максимально-производительных результатов с сохранением абстракций времени компиляции, однако реализация динамического компилятора является крайне сложной технической задачей.

- Введение в хост-язык специального языка описания предметной области (DSL) для формирования сопоставлений с образцом и выделение некоторого подмножества образцов, которые могут быть скомпилировано эффективно. Это значительно упростит задачу компилятору по применению специальных оптимизаций

и позволит представлять сопоставления более декларативно для пользователей. Тем не менее при активном использовании абстракций и раздельной компиляции, разумеется, компилятор не сможет применять большинство оптимизаций, и машинный код выродится в аналогичный порождаемому при использовании образцов первого порядка. Поэтому по-прежнему для генерации максимально эффективного кода необходимо использовать либо встраивание времени линковки (LTO), либо более общее динамическое встраивание времени исполнения (динамическая компиляция).

3.3. Заключительный выбор

Наконец, можно произвести окончательный, объективный насколько возможно выбор расширения для реализации.

Согласно данным таблицы 1 наиболее привлекательным для реализации представляется оператор возврата. Просто удивительно, насколько самое тривиальнейшее решение оказывается самым технически мощным! Вполне в духе языка программирования C.

Тем не менее предлагается все-таки реализовать активные образцы с поддержкой *параметризации слева-направо*, поскольку они являются своего рода синтаксическим сахаром для оператора возврата (поддерживают структурное вложение образцов) и автоматически реализовывают мемоизирующую семантику, а посему могут быть дополнительно оптимизированы компилятором. Оператор возврата является в этом смысле максимально гибким, но все же низкоуровневым инструментом, который также рекомендуется для поддержания языками ввиду своей тривиальности.

Важно отметить, что в рамках данной работы, мы не учитывали критерии совместимости расширений с продвинутыми особенностями системы типов, такими как, например, обобщенные алгебраические типы данных (GADT). Такие возможности на данный момент поддерживаются только синонимами образцов. Для активных образцов авторы также предлагали возможное развитие в области GADTS и экзистенциальных типов данных, однако этот вопрос по-прежнему остается открытым.

4. Реализация активных образцов в OCaml

В данном разделе мы представим формализацию выбранного к реализации расширения активных образцов в контексте языка OCaml.

4.1. Расширение синтаксиса

Для представления синтаксиса мы воспользуемся грамматикой из [24]. Необходимые модификации для внедрения структурированных имен следующие:

```
structured_name ::=
      # Однозначный полный образец
    '(|' capitalized_ident '|)'
      # Многозначный полный образец
  | '(|' capitalized_ident {'|' capitalized_ident} '|)'}
      # Однозначный частичный образец
  | '(|' capitalized_ident '|' '_' '|)'

value_name ::= ...
  | structured_name
```

Необходимые модификации синтаксиса образцов следующие:

```
pattern ::= ...
       # Параметризованный частичный образец
  | '<' capitalized_ident expr {expr} '>' pattern
```

Полные и не параметризованные частичные образцы уже представляются синтаксисом конструкторов-образцом. Это является желательным свойством, дабы публичное определение типа можно было бы скрыть, заменив его на экспортированный активный образец, что не потребовало бы синтаксических вмешательств в сопоставления, использующих конструкторы этого типа (однако перекомпиляция все равно потребуется во имя производительности).

Таким образом, необходимо лишь ввести дополнительный синтаксис для параметризованных частичных образцов. Заметьте, что здесь мы отходим от синтаксиса, представленного в оригинальной работе: образец вместе с параметрами должен быть заключен в угловые скобки. Причина этому следующая: синтаксически категории выражений и образцов значительно пересекаются и LALR(1)-парсер не сможет понять, стоит ли перед ним еще один параметр-выражение, либо уже образец: к примеру, является ли `Pat 42` частичным образцом с параметром 42, после которого последует что-то еще, либо образцом-конструктором, тело которого сопоставляется с константой 42? У этой проблемы есть 2 возможных решения:

1. Либо мы хотим все же различать в общем случае выражения и образцы, для чего область выражений необходимо заключить в некоторые обрамляющие символы (в нашем случае угловые скобки), дабы сгенерированный автомат знал, что пока не будет встречена закрывающая угловая скобка, следуют только выражения.

2. Либо мы синтаксически ограничиваем возможные выражения до уровня образцов, и все разбираем как образцы, а затем, на последующих фазах компиляции, преобразовываем часть результатов в категории выражений. Это подход принятый в F# и позволяющий избежать необходимости использования обрамляющих символов. Однако такой код тяжело читать и человеку, поскольку образцы и параметры вне контекста неотличимы. И ощущается довольно странным, что в качестве параметров частичного образца нельзя передавать то, что можно передавать в качестве аргументов вызова функции, например лямбды или элемент массива по индексу.

4.2. Правила типизации

К сожалению, для языка OCaml статическая семантика в литературе не представлена. Поэтому добиться математической строгости в рамках данной работы априори не получится, ввиду того, что строгость требует введения большого количества инструментов и определений. По этим, а также по другим причинам, описанным в исходной работе [23], единственным способом для презентации правил типизации является представление эталонного типизирующего анализатора, производящего типизированное абстрактное синтаксическое дерево для исходного кода.

Такой анализатор является частью прототипа, реализованного в рамках данной работы и представлен в [3].

4.3. Модификация схемы компиляции

В данном разделе мы представим необходимые модификации для оптимизированной схемы компиляции в автоматы поиска с возвратами [10].

Напомним, что для поддержания необходимых теоретических свойств активным образцам предписывается мемоизирующая семантика [17]: для каждого вхождения активного образца, соответствующая ему трансформирующая функция вызывается не более одного раза. Таким образом, необходимо для начала выписать все вхождения активных образцов, а затем для каждого из них выделить переменные для сохранения результатов вызова.

Вхождения (occurences) мы определяем формально в соответствии с работой [12]: вхождение o есть либо пустой вектор Λ, либо число k, за которым следует вхождение o': $k \cdot o'$. Вхождения задают пути к подтермам терма v в следующем смысле:

$$v/\Lambda = v$$
$$c(v_1, \ldots, v_n)/k \cdot o = v_k/o$$

Теперь необходимо дополнить схему C^* двумя параметрами — вхождением o, которое призвано отслеживать, с какой частью входного выражения сейчас производится сопоставление и *кэш-картой атар*. Обновленная кэш-карта также будет возвращаться схемой в качестве третьего элемента, в дополнение к сгенерированному коду и таблице контекстов.

Вхождение изменяется естественным образом:

- Для правил схемы компиляцией, выполняющих специализацию (таких как правило конструктора (constructor-rule)), выполняемые рекурсивные вызовы должны получать специализацию вхождения по конструктору (такая операция формально описана в работе [12]).
- Для рекурсивного вызова правила переменных (variable-rule), вхождение инкрементируется.
- Для компиляции матрицы $P'' \to L''$ правила или-образцов (orpat-rule) [10] вхождение также инкрементируется.
- В разделяющем правиле вхождение передается рекурсивным вызовам неизменным.

Кэш-карта образцов представляет собой отображение из вхождений в список пар идентификаторов активных образцов и сгенерированных имен кэш-переменных. Для одного вхождения могут быть вызваны несколько независимых активных образцов, для каждого из которых потребуется выделить свой кэш. В качестве примера рассмотрим следующее сопоставление:

```
let (|M1|M2|) x = (* ... *)
let (|C|_|)   x = (* ... *)
let (|D|_|)   x = (* ... *)

let _ =
  match (1, 2, 3) with
  | (M1 _, C _, 42) → 1
  | (M2 _, D _,  x) → 2
  | (C  _, _  ,  _) → 3
```

Сгенерированная кэш-карта будет следующей:

$$\{1\cdot \mapsto [\ ((|M1|M2|),\ z1),\ ((|C|\_|),\ z2)\ ];\quad 2\cdot \mapsto [\ ((|C|\_|),\ z3),\ ((|D|\_|),\ z4)\ ]\}$$

Здесь теги `M1` и `M2` принадлежат одному и тому же активному образцу, а теги `C` и `D` разным. По третьей координате вектора входных значений сопоставления с активными образцами не проводится.

Наконец перейдем к описанию изменений в схему компиляции.

Для разделяющего правила (mixture-rule) для случая активных образцов мы будем жадно отбирать наибольший строчный префикс матрицы, в первой позиции образцов которой стоят теги, принадлежащие одному и тому же активному образцу. Так, для примера выше, в одному группу компиляции попадут первые 2 строки.

Компиляцию двух частей необходимо дополнить правильной передачей кэш-карты, что впрочем тривиально. Разделив матрицу $P \to L$ на подматрицы $Q \to M$ и $R \to N$, необходимо выполнить следующие рекурсивные вызовы:

$$l_q, \rho_q, amap_q = C^*(\vec{x}, Q \to M, partial, (R, e); def, ctx, o, amap)$$
$$l_r, \rho_r, amap_r = C^*(\vec{x}, R \to N, ex \qquad , \qquad def, ctx, o, amap_q)$$

Кэш-картой, возвращаемой всем правилом, является $amap_r$.

Теперь необходимо добавить новое правило компиляции группы строк с активными образцами. Рассмотрим лишь случай единичного полного активного образца, для остальных видов образцов необходимые действия абсолютно аналогичны, за исключением технических деталей, таких как вычисление параметров для параметризованных образцов и сопоставление результата преобразования с тэгами анонимного типа-суммы для многозначных полных образцов.

Входная матрица $P \to L$ имеет вид $(A(q_1, \ldots, q_a)\ p_2\ \cdots\ p_n \to l)$ и активный образец `(|A|)` является единичным полным. В этом случае требуется всего лишь вызвать соответствующую преобразующую функцию. Выполнив рекурсивный вызов, получим:

$$l_c, \rho, amap = \\ C^*((y_1\ \cdots\ y_a\ x_2\ \cdots\ x_n), (q_1\ \cdots\ q_a\ p_2 \cdots p_n \to l), ex, \Downarrow(def), \Downarrow(ctx), o \cdot 0, amap)$$

Если $amap(o)$ уже содержит $(|A|)$ в качестве ключа, то достаем имя переменной кэша z. В этом случае вызов $(|A|)$ уже состоялся ранее и можно просто пользоваться значением z. Результирующий код l_r будет следующим:

```
(let (y1 (field 0     z))
       ...
     (ya (field (a—1) z))
     l_c))
```

Если $amap(o)$ не содержит $(|A|)$ в качестве ключа, то генерируем свежее имя переменной кэша z и положим $amap(o) += ((|A|), z)$. В этом случае l_r выше необходимо будет обернуть в вызов преобразующей функции:

```
(seq (setfield 0 z (apply (|A|) x1)) l_r)
```

Схема возвращает l_r, поднятое (popped) ρ и $amap$ (с учетом добавления, если таковое произошло).

Наконец, осталось лишь для начального вызова компиляции сопоставления произвести генерацию необходимых переменных кэшей. Пусть имеем $l, \_, amap = C^*(\ldots)$, $(z_1, t_1), \ldots, (z_n, t_n)$ список пар всех выделенных переменных и их типов (выведенных из типов сохранённых идентификаторов активных образцов), тогда код l необходимо обернуть в:

```
(let (z_1 = (makemutable 0 (t_1) 0))
       ...
     (z_n = (makemutable 0 (t_n) 0))
     l)
```

Внимательный читатель заметит, что в общем случае сгенерированных переменных может быть довольно большое количество. Т.к. они выделяются на стеке на производительности это особо не сказывается, но вот для не хвосто-рекурсивных функций такое дополнительное потребление памяти может оказаться весьма критичным. В общем случае некоторые переменные можно пытаться переиспользовать, однако оставим это за рамками текущей работы.

5. Заключение

В данной работе были достигнуты следующие результаты:

1. Произведён подробный разбор аналогов в большинстве наиболее передовых функциональных языков, выделены решения, которые могут служить опорой для дизайна расширений для языка OCaml:
   - Представлено расширение оператора возврата к сопоставлению.
   - Введены объективные критерии для сравнения существующих решений и произведен анализ каждого из расширений, представленных как кандидаты к реализации, обоснован выбор активных образцов как расширения для реализации.
   - Инициирован RFC [2] по реализации активных образцов в качестве расширения языка OCaml; текущие наработки по реализации должны послужить в качестве образца синтаксиса и семантики для RFC.
2. Формализованы следующие изменения, необходимые к внесению в компилятор языках OCaml для поддержки активных образцов:
   - расширение синтаксиса
   - правила типизации новых конструкций
   - модификация схемы компиляции [10] для активных образцов
3. Реализованы следующие элементы прототипа расширения:
   - синтаксический анализ
   - типизация

6. Благодарности

Автор работы благодарит программиста “JetBrains Labs” Дмитрия Косарева за всестороннюю консультацию и помощь в организации работы.

Также автор безмерно благодарен Олегу Киселёву за его бесценные статьи по внутреннему устройству компилятора OCaml [18], благодаря которым стало возможным разобраться в существующем коде типизации и провести его модификацию.

Список литературы

[1] Augustsson Lennart. Compiling pattern matching // Functional Programming Languages and Computer Architecture / Под ред. Jean-Pierre Jouannaud. — Berlin, Heidelberg : Springer Berlin Heidelberg, 1985. — С. 368–381.

[2] Bashkirov Alexander. Active patterns for OCaml. — 2020. — Режим доступа: `https://github.com/ocaml/RFCs/pull/12` (дата обращения: 01.06.2020).

[3] Bashkirov Alexander. Active patterns for OCaml implementation repository. — 2020. — Режим доступа: `https://github.com/bash-spbu/ocaml/tree/active_patterns_docker_image` (дата обращения: 01.06.2020).

[4] Brian Goetz. Data Classes and Sealed Types for Java. — 2019. — Режим доступа: `https://cr.openjdk.java.net/~briangoetz/amber/datum.html` (дата обращения: 20.05.2020).

[5] Emir Burak, Odersky Martin, Williams John. Matching Objects with Patterns // Proceedings of the 21st European Conference on Object-Oriented Programming. — ECOOP'07. — Berlin, Heidelberg : Springer-Verlag, 2007. — С. 273–298. — Режим доступа: `http://dl.acm.org/citation.cfm?id=2394758.2394779`.

[6] Erwig Martin, Peyton Jones Simon. Pattern Guards and Transformational Patterns // Haskell Workshop 2000. — 2000. — September. — Режим доступа: `https://www.microsoft.com/en-us/research/publication/pattern-guards-and-transformational-patterns/`.

[7] GHC Wiki. Haskell view patterns. — Режим доступа: `https://gitlab.haskell.org/ghc/ghc/wikis/view-patterns` (дата обращения: 20.05.2020).

[8] IntelliJ Scala. Демонстрация невыразительности сопоставления на примере конвертера из языка Scala в UAST. — Режим доступа: `https://github.com/JetBrains/intellij-scala/blob/878ef964b1bb9dc535ed0bb42ad7ea0731427065/scala/uast/src/org/jetbrains/plugins/scala/lang/psi/uast/converter/Scala2UastConverter.scala#L366` (дата обращения: 20.05.2020).

[9] Kotlin Language Documentation. Control Flow: When Expression. — 2019. — Режим доступа: `https://kotlinlang.org/docs/reference/control-flow.html#when-expression` (дата обращения: 20.05.2020).

[10] Le Fessant Fabrice, Maranget Luc. Optimizing Pattern Matching // ACM SIGPLAN Notices. — 2001. — 08. — Т. 36.

[11] Maranget Luc. Warnings for pattern matching // Journal of Functional Programming. — 2007. — 05. — Т. 17. — С. 387–421.

[12] Maranget Luc. Compiling Pattern Matching to Good Decision Trees // Proceedings of the 2008 ACM SIGPLAN Workshop on ML. — ML '08. — New York, NY, USA : ACM, 2008. — С. 35–46. — Режим доступа: http://doi.acm.org/10.1145/1411304.1411311.

[13] Microsoft Docs. C# Pattern Matching. — 2019. — Режим доступа: https://docs.microsoft.com/en-us/dotnet/csharp/pattern-matching (дата обращения: 20.05.2020).

[14] Milner Robin, Tofte Mads, Macqueen David. The Definition of Standard ML. — Cambridge, MA, USA : MIT Press, 1997. — ISBN: 0262631814.

[15] OCaml Community. if-let for OCaml. — Режим доступа: https://github.com/ocaml/ocaml/pull/194 (дата обращения: 20.05.2020).

[16] OCaml Discuss. Musings on extended pattern-matching syntaxes. — 2019. — Режим доступа: https://discuss.ocaml.org/t/musings-on-extended-pattern-matching-syntaxes/3600 (дата обращения: 20.05.2020).

[17] Okasaki Chris. Views for Standard ML // In SIGPLAN Workshop on ML. — 1998. — С. 14–23.

[18] Oleg Kiselyov. How OCaml type checker works – or what polymorphism and garbage collection have in common. — Режим доступа: http://okmij.org/ftp/ML/generalization.html (дата обращения: 20.05.2020).

[19] Pattern Synonyms / Matthew Pickering, Gergo Érdi, Simon Peyton Jones, Richard A. Eisenberg // Haskell'16. — 2016. — September. — Режим доступа: https://www.microsoft.com/en-us/research/publication/pattern-synonyms/.

[20] Peyton Jones Simon, Graf Sebastian, Scott Ryan. Lower your guards: a compositional pattern-match coverage checker. — 2020. — March. — In submission. Режим доступа: https://www.microsoft.com/en-us/research/publication/lower-your-guards-a-compositional-pattern-match-coverage-checker/.

[21] The Rust reference : Intern report / Inria ; исполн.: Community Rust : 2020. — Режим доступа: https://doc.rust-lang.org/reference/index.html.

[22] Scott Kevin D, Ramsey Norman. When Do Match-compilation Heuristics Matter? // Technical Report CS-2000-13. — 2000.

[23] Syme Don, Neverov Gregory, Margetson James. Extensible pattern matching via a lightweight language extension // Proceedings of the 12th ACM SIGPLAN international conference on Functional programming. — Association for Computing Machinery, Inc., 2007. — October. — Режим доступа: https://www.microsoft.com/en-us/research/publication/extensible-pattern-matching-via-a-lightweight-language-extension/.

[24] The OCaml system release 4.09: Documentation and user's manual : Intern report / Inria ; исполн.: Xavier Leroy, Damien Doligez, Alain Frisch и др. : 2019. — Сент. — С. 1–789. Режим доступа: https://hal.inria.fr/hal-00930213.

[25] Tullsen Mark. First Class Patterns // In 2nd International Workshop on Practial Aspects of Declarative Languages, volume 1753 of LNCS. — Springer-Verlag, 2000. — С. 1–15.

[26] Wadler P. Views: A Way for Pattern Matching to Cohabit with Data Abstraction // Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages. — POPL '87. — New York, NY, USA : ACM, 1987. — С. 307–313. — Режим доступа: http://doi.acm.org/10.1145/41625.41653.