Элиминация стереотипного кода в программах на языке Objective Caml

С. В. Мечтаев
mechtaev@gmail.com

В данной статье описывается метод элиминации стереотипного вспомогательного кода, реализующего работу со сложными структурами данных в программах на языке Objective Caml. Стереотипный код ухудшает гибкость и сопровождаемость программ, так как при незначительных изменениях данных требуется значительная работа по переделке этого кода. Метод является адаптацией для языка Objective Caml подхода Scrap Your Boilerplate (SYB), исходно реализованного для языка Haskell, — SYB не может быть «напрямую» перенесен на Objective Caml. Метод включает в себя оптимизацию исходного подхода, основанную на специализации (specification on type, specification on type and function) и передаче продолжений (continuation passing style), и позволяет добиться приемлемых потерь производительности по сравнению с «ручной» реализацией кода, отвечающего за работу со сложными структурами данных.

Ключевые слова: функциональное программирование, generic programming, стереотипный код (boilerplate code).

Введение

Разработка и сопровождение ПО, обрабатывающего сложные структуры данных, часто оказывается непростой задачей. Одной из причин этого является большое количество кода, отвечающего за работу с этими данными. Будем называть такой код *стереотипным*

(англ. boilerplate). Его негативное воздействие на программу объясняется отсутствием гибкости, т. е. небольшие изменения структур данных или алгоритмов влекут масштабные изменения в коде. Также стереотипный код обладает слабой выразительностью. Всё это приводит к большому количеству ошибок.

Основным способом решения этих проблем является переиспользование стереотипного кода. Решение этой задачи зависит от языка программирования. В процедурных языках переиспользование осуществляется посредством абстракции с помощью процедур. В языках объектно-ориентированного программирования элиминация стереотипного кода может производиться с помощью объектно-ориентированных шаблонов проектирования Visitor [3].

Источником стереотипного кода в программах на языке Objective Caml являются алгебраические типы данных. Проиллюстрируем это на примере. Пусть имеется структура данных, описывающая элементарные арифметические выражения:

```
type expression =
  | Add   of expression * expression
  | Sub   of expression * expression
  | Const of int
  | Var   of variable
```

Тип `expression` содержит значения, которые могут быть суммой или разностью двух выражений (`Add` или `Sub`) либо числовой константой (`Const`), либо переменной (`Var`). Значения типа `variable`, принимаемые в качестве параметра конструктора `Var`, содержат некоторую информацию о переменной, например имя и ссылку на декларацию.

Представим, что необходимо изменить информацию, содержащуюся во всех узлах типа `variable` некоторого выражения. Это может потребоваться, например, при реализации типичной компиляторной задачи — идентификации переменных. Пусть алгоритм трансформации переменных инкапсулируется функцией var:

```
val var : variable -> variable
```

Эта функция для данного значения типа `variable` возвращает его копию, в которую добавлена требуемая информация. Пусть

функция `resolve` обходит структуру данных выражения и строит новое выражение, равное исходному, но которое отображает с помощью `var` свои переменные. Такую функцию несложно написать «вручную»:

```
let rec resolve var = function
  | Add (x, y) -> Add (resolve var x, resolve var y)
  | Sub (x, y) -> Sub (resolve var x, resolve var y)
  | Const i    -> Const i
  | Var v      -> Var (var v)
```

Предположим теперь, что требуется получить список всех переменных, встречающихся в выражении. Эту функцию также несложно реализовать «вручную»:

```
let rec collect = function
  | Add (x, y) -> (collect x) @ (collect y)
  | Sub (x, y) -> (collect x) @ (collect y)
  | Const i    -> []
  | Var v      -> [v]
```

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

Данные примеры являются образцами использования стереотипного кода. Совершенно разные по смыслу функции `resolve` и `collect` содержат общую семантическую часть — процедуру обхода данных, реализация которой занимает бóльшую часть их кода. Более того, при увеличении сложности структуры данных (например, при добавлении новых операций) этот код будет увеличиваться, в то время как содержательная часть будет оставаться неизменной. Такая ситуация является типичной.

В рамках данной статьи рассматривается способ элиминации стереотипного кода, который основан на использовании техники обобщенного программирования, управляемого типами (*generic type-driven programming*). Данный подход помогает избавиться от стереотипного кода, семантика которого полностью определяется типом обрабатываемых данных. Приведенный выше пример относятся именно к такому случаю, потому что процедура рекурсивного обхода полностью определяется типом `expression`, а семантическое

действие, специфическое для каждого из примеров, определяется типом `variable`.

Решению описанной проблемы посвящена обширная литература [5, 6, 9, 10, 11, 14]. Изучение существующих подходов показало, что наиболее полное решение описанных проблем обеспечивает подход Scrap Your Boilerplate (SYB) [9, 10, 11], поддержка которого входит в стандартную библиотеку языка Haskell [21]. В оригинальной версии реализация SYB существенно опирается на особенности типовой системы языка Haskell и не может быть прямо перенесена в Objective Caml. Кроме того, более поздние исследования [17] выявили существенное падение производительности при использовании SYB по сравнению с рукописным кодом.

В данной статье представляется метод адаптации подхода SYB для Objective Caml, обладающий сравнимой с оригиналом выразительностью и функциональностью. Метод обеспечивает падение производительности не более чем в два раза. Такой результат достигается благодаря использованию техники специализации (*specification on type, specification on type and function*) и передачи продолжений (*continuation passing style*) [18].

1. Подход Scrap Your Boilerplate (SYB)

Основная идея данного подхода заключается в декомпозиции стереотипного кода на три части: логика, задаваемая пользователем; часть, которая является специфичной для данного типа данных; общий алгоритм обхода (рис. 1).

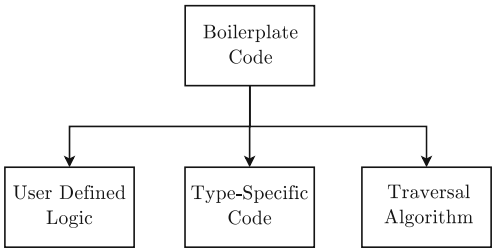

Рис. 1. Декомпозиция стереотипного кода

Данный подход предоставляет возможность создавать функции трансформаций и запросов. *Трансформация* — это функция типа $\tau \to \tau$, где τ — тип обрабатываемых данных. Трансформация обходит дерево значений в глубину и создаёт новое значение, равное исходному, но с отображенными заданным образом узлами требуемых типов. *Запрос* представляет из себя функцию типа $\tau \to \rho$, где τ — тип обрабатываемых данных, а ρ — тип результата запроса. Запрос также обходит структуру данных, но, в отличие от трансформации, собирает требуемую информацию из узлов заданных типов. Покажем, как производится описанная декомпозиция на примере трансформации. Для запросов она реализуется аналогично.

Далее будем разделять функции обработки на плоские и неплоские. Функция является *неплоской*, если она рекурсивно обходит структуру данных, применяя к узлам некоторую плоскую функцию, в противном случае она является *плоской*.

Вначале необходимо отделить пользовательскую логику от алгоритма обхода. Для этого используется лифтинг (*lifting*), т. е. доопределение пользовательской (плоской) функции, а именно: функция типа $\tau \to \tau$ доопределяется тождественно для всех типов, отличных от τ. Таким образом, она становится полиморфной функцией типа $\forall\alpha \;\; \alpha \to \alpha$.

Для реализации лифтинга используется безопасное приведение типов, предоставляемое классом `Typeable`. В этом классе определена функция `cast`, которая сравнивает тип своего аргумента с желаемым типом и производит приведение типов, если это возможно. Функция `mkT` выполняет описанный ранее лифтинг:

```
mkT :: (Typeable a, Typeable b) => (b -> b) -> a -> a
mkT f = case cast f of
       Just g  -> g
       Nothing -> id
```

Теперь можно достаточно просто создавать разнообразные функции обработки, параметризованные алгоритмом обхода, например, следующим образом:

```
everywhere :: (forall b. Typeable b => b -> b) ->
     Expression -> Expression
everywhere f (Add x y) =
```

```
      f $ Add (everywhere f x) (everywhere f y)
everywhere f (Sub x y) =
      f $ Sub (everywhere f x) (everywhere f y)
everywhere f (Const i) =
      f $ Const (f i)
everywhere f (Var v) =
      f $ Var (f v)
```

Функция `everywhere f` обходит структуру данных выражения и создает новое выражение, каждый узел которого получен применением функции `f` к соответствующему узлу исходного выражения. Другими словами, она из плоской трансформации делает неплоскую. Функцию идентификации переменных можно получить, например, следующим образом:

```
resolve :: (Variable -> Variable) ->
  Expression -> Expression
resolve var = everywhere (mkT var)
```

Приведенный пример использования `mkT` не решает проблему стереотипного кода в достаточной мере, так как для каждой структуры данных приходится создавать функции вроде `everywhere`. Если потребуется несколько алгоритмов обхода, например снизу-вверх и сверху-вниз, необходимо будет написать несколько таких функций.

Другой проблемой является сложность написания таких функций при реализации структур данных, использующих большое число различных типов и содержащих типовые переменные. Для устранения этих недостатков авторы SYB разделяют такие функции на две части: первая часть зависит от типа, вторая реализует алгоритм обхода, не зависящий от специфики типа.

Зависимая от типа часть представляется с помощью класса `Data`. Она может быть создана автоматически по декларации типа при использовании конструкции `deriving`. Класс `Data` содержит функцию `gmapT`, которая отображает непосредственных потомков своего второго аргумента первым аргументом:

```
class Typeable a => Data a where
  gmapT :: (forall b. Data b => b -> b) -> a -> a
```

Экземпляр `Data` для типа `Expression` выглядит следующим образом:

```
instance Data Expression where
  gmapT f (Add x y) = Add (f x) (f y)
  gmapT f (Sub x y) = Sub (f x) (f y)
  gmapT f (Const i) = Const (f i)
  gmapT f (Var v)   = Var (f v)
```

Используя `gmapT` можно очень просто реализовать функцию `everywhere`:

```
everywhere :: Data a => (forall b. Data b => b -> b) ->
  a -> a
everywhere f x = f (gmapT (everywhere f) x)
```

Такая реализация лишена проблем предыдущей версии `everywhere`. Она не зависит от типа, поэтому может обрабатывать сколь угодно сложные структуры данных.

Основными недостатками расмотренного подхода являются:

- ограничение сложности типа функций обработки;
- низкая производительность.

2. Предлагаемое решение

В данном разделе описывается метод адаптации подхода SYB для Objective Caml, включая оптимизацию. Под адаптацией прежде всего понимается применение декомпозиции в соответствии с рис. 1 для Objective Caml. Как и при обсуждении SYB, здесь будет приведена только реализация трансформаций. Запросы создаются аналогично.

2.1. Равенство типов

Введем конструктор типа `eq`. Следуя [15], значение типа `(a, b) eq` будем называть равенством типов `a` и `b`. Определим для них следующие типы функций:

```
val refl  : ('a, 'a) eq
val symm  : ('a, 'b) eq -> ('b, 'a) eq
val trans : ('a, 'b) eq -> ('b, 'c) eq -> ('a, 'c) eq
val cast  : ('a, 'b) eq -> 'a -> 'b
```

Здесь `refl`, `symm` и `trans` представляют аксиомы рефлексивности, симметричности и транзитивности, а `cast` может использоваться для приведения типов. Реализация этих функций стандартна, поэтому не будет приводиться в этой статье.

2.2. Маркеры типов

Для адаптации SYB нам потребуются специальные значения — маркеры типов:

```
type 'a marker = unit option ref
let make () = ref (Some ())
```

Маркеры типов будут использоваться для сравнения типов и, таким образом, заменят собой безопасное приведение. Для этого создадим функцию `compare`:

```
let compare : 'a marker -> 'b marker -> ('a, 'b) eq option =
  fun x y ->
    if x == y
    then Some (Obj.magic refl)
    else None
```

2.3. Лифтинг

Функция обхода в SYB получала в качестве параметра полиморфную функцию. Из-за этого тип функции обхода являлся типом второго ранга [16]. В языке Objective Caml отсутствует поддержка типов второго ранга, но их можно реализовать, например, с помощью записей. Поэтому определим специальный тип для полиморфных трансформаций:

```
type lifted = { f : 'a . 'a marker -> 'a -> 'a }
```

Маркер, принимаемый в качестве параметра этой функцией, можно считать аналогом контекста, передаваемого со значениями класса `Typeable`.

С помощью функции `lift` можно из мономорфной функции типа $\tau \to \tau$ сделать полиморфную, которая доопределяет ее тождественно для типов, отличных от τ:

```
let lift a func =
  { f = fun b x ->
    match compare a b with
    | None   -> x
    | Some e ->
      let forward  = cast (symm e) in
      let backward = cast e in
      backward (func (forward x)) }
```

2.4. Информация о типе

Аналогом экземпляров класса `Data` из SYB будут выступать значения типа `typeinfo`, представляющие из себя пары из маркера и функции `gmapT`:

```
type 'a typeinfo =
  { marker : 'a marker;
    gmapT  : transform -> 'a -> 'a }
and transform =
  { transform : 'a . 'a typeinfo -> 'a -> 'a }
```

Здесь `transform` — это тип, представляющий трансформации, которые, в отличии от `lifted`, принимают значения типа `typeinfo`. С помощью этого типа будут представляться неплоские трансформации. Их аргумент типа `typeinfo` можно считать аналогом контекста, который передается в `Haskell` с помощью класса типов `Data`.

Приведем пример значения типа `typeinfo` для типов `int` и `expression`, описанных во введении:

```
let int =
  let int_marker : int marker = make () in
  { marker = int_marker;
    gmapT  = (fun _ v -> v) }

let expression =
  let expr_marker : expression marker = make () in
  let rec inner () =
    { marker = expr_marker;
      gmapT  = (fun t -> function
```

```
        | Add (x, y) ->
          Add (t.transform (inner ()) x,
               t.transform (inner ()) y)
        | Neg   x    -> Neg (t.transform (inner ()) x)
        | Const x    -> Const (t.transform int x)
        | Var   x    -> Var (t.transform variable x)) }
  in inner ()
```

В этом примере используется вспомогательная функция `inner` с параметром типа `unit` для того, чтобы реализовать обход рекурсивного типа `expression`.

2.5. Создание трансформации

Определенных ранее средств достаточно для создания трансформаций, аналогичных тем, которые можно задавать, используя SYB. Аналог функции `everywhere` будет выглядеть следующим образом:

```
let everywhere f =
  let rec transform : 'a . 'a typeinfo -> 'a -> 'a =
    fun ti v ->
      f.f ti.marker (ti.gmapT { transform } v)
  in
  transform
```

С помощью функции `everywhere` можно создать, например, функцию идентификации переменных `resolve`:

```
let resolve var =
  everywhere (lift variable.marker var) expression
```

2.6. Расширение синтаксиса

Стандартная реализация языка Haskell позволяет автоматически создавать типо-зависимую часть с помощью конструкции `deriving`. Для Objective Caml мы предлагаем расширение синтаксиса, которое дает возможность автоматически конструировать значения типа `typeinfo` по декларации типа. Продемонстрируем его использование на каноническом примере увеличения зарплаты сотрудников компании из описания подхода SYB [10].

Для этого заменим ключевое слово `type` на `datatype` в описании структуры данных, представляющей компанию. В этом случае будут автоматически созданы описания значений `typeinfo` для типов из этой структуры данных. Имена значений останутся такими же, как имена соответствующих им типов. Ниже представлена функция, увеличивающая зарплаты сотрудников на 50%:

```
datatype company = C of dept list
and dept         = D of name * manager * subunit list
and subunit      = PU of employee | DU of dept
and employee     = E of person * salary
and person       = P of name * address
and salary       = S of float
and manager      = employee
and name         = string
and address      = string

let increase =
  everywhere
  (lift salary.marker (function S v -> S (v *. 1.5)))
    company
```

В приведенном примере предполагается, что уже существуют описания `typeinfo` для типов `string`, `float` и `list`.

2.7. Конструкторы типов

Наиболее естественным способом поддержки конструкторов типов (функций вида $* \to *$) кажется создание для каждого конструктора `tc` функции, которая принимает значение типа `typeinfo` для некоторого типа `t` и возвращает значение типа `typeinfo` для типа `t`. Для обеспечения прозрачности использования библиотеки необходимо, чтобы описанные функции были детерминированы с точки зрения равенства маркеров. Поэтому в качестве идентификатора маркера следует использовать список `unit`-значений, а равенство маркеров определять как равенство длин списков и равенство соответствующих элементов. Тогда при создании значения типа `typeinfo` для типа, полученного применением конструктора, можно создавать маркер этого типа, добавляя идентификатор маркера

конструктора в начало идентификатора маркера параметра конструктора. 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.

2.8. Оптимизации

«Ручная» реализация кода, работающего со сложными структурами данных, в большинстве случаев обладает значительно лучшей производительностью, чем реализация, использующая подход SYB. Далее будут приведены способы оптимизации, которые сделают падение производительности незначительным.

Основная идея оптимизации состоит в модификации комбинаторов SYB таким образом, чтобы процесс, который они порождают, максимально соответствовал бы процессу, порождаемому «ручной» реализацией.

Сравним процессы исполнения двух интерпретаций функции `resolve` — представленной во введении и SYB-версии из раздела 2.5. Первое важное отличие SYB-версии заключается в наличии дополнительного действия при обработке каждого узла — вызова функции `transform`, которая производит сравнение маркеров, вычисляет функцию обработки и рекурсивно запускает обход данных. Другое различие состоит в том, что ручная реализация не выполняет обход данных, представленных типом `int`, так как их не требуется изменять, а также применяет функцию обработки только к данным типа `variable`, в то время как SYB-версия применяет также тождественную функцию обработки к данным других типов.

Обобщение результатов сравнения указало на два основных пути оптимизации: специализация к типу и специализация к «интересной» функции.

2.8.1. Специализация к типу

Специализация функции обработки к типу данных заключается в том, что все вычисления, связанные с типами, должны выполняться до начала обработки данных, что позволяет уменьшить количество таких вычислений со значения, равного порядку размера

данных, до значения, равного порядку размера типа. Вычисления, связанные с типами данных, включают сравнение маркеров и вычисление функций обработки для узлов каждого типа.

Для реализации этого способа оптимизации нужно изменить функции `lift`, `gmapT` и `everywhere` таким образом, чтобы их результатами были замыкания, в которых уже подсчитаны требуемые значения:

```
let lift a func =
  { f = fun b ->
    match compare a b with
    | None   -> (fun x -> x)
    | Some e ->
      let forward  = cast (symm e) in
      let backward = cast e in
      (fun x -> backward (func (forward x))) }

let expression =
  let expr_marker : expression marker = make () in
  let rec inner () =
    { marker = expr_marker;
      gmapT = (fun t ->
        let expr_tr = t.transform (inner ()) in
        let int_tr  = t.transform int in
        let var_tr  = t.transform variable in
        function
          | Add (x, y) -> Add (expr_tr x, expr_tr y)
          | Neg   x    -> Neg (expr_tr x)
          | Const x    -> Const (int_tr x)
          | Var   x    -> Var (var_tr x)) }
  in inner ()

let everywhere f =
  let rec transform : 'a . 'a typeinfo -> ('a -> 'a) =
    fun ti ->
      compose (f.f ti.marker) (ti.gmapT { transform })
  in
  transform
```

Здесь `compose` — это композиция функций.

Оптимизированная функция `everywhere` вначале рекурсивно вычисляет трансформации для потомков данного узла, а затем конструирует из них трансформацию текущего узла. Описанная выше реализация такой функции приводит к зацикливанию на рекурсивных типах. Для решения данной проблемы используем технику «заглушек». Вначале заведем таблицу, которая будет сопоставлять маркеру типа соответствующую этому типу функцию трансформации. Представим тип в виде графа. Будем производить обход этого графа в глубину, и при входе в вершину, для маркера которой в таблице существует значение трансформации, будем просто возвращать это значение. Если же трансформация еще не вычислена, то будем ставить напротив этого маркера «заглушку», которую при выходе из вершины будем заменять настоящим значением с помощью присваивания. Такая техника решает проблему зацикливания, так как позволяет не производить обход по обратным дугам, которые формируют циклы. Реализация этого алгоритма выглядит следующим образом:

```
let everywhere f =
   let context = Tbl.create () in
   let rec transform :
     'a . 'a typeinfo -> ('a -> 'a) =
     fun ti ->
       let f = f.f marker in
       try
         let tr = Tbl.find context ti.marker in
         compose f (fun x -> !tr x)
       with
          _ ->
            let tr = ref stub in
            Tbl.add context ti.marker tr;
            tr := ti.gmapT { transform };
            Tbl.remove context ti.marker;
            compose f !tr
    in
    transform
```

В этом примере `stub` — описанная в алгоритме «заглушка», `Tbl` — модуль, предоставляющий требуемый ассоциативный массив. Этот массив является гетерогенным, т. е. тип значения его элемен-

та зависит от типа ключа. Его реализация с помощью функции `Obj.magic` тривиальна, поэтому не будет приводиться в данной статье.

2.8.2. Специализация к «интересной» функции

Специализация функции обработки к «интересной» функции (содержательной части, задаваемой пользователем) заключается в том, что, во-первых, производится обработка данных только для тех типов, для которых определена трансформация, а во-вторых, не будет осуществляться обход тех подструктур данных, в которых нет данных, требующих изменений.

Для реализации этого способа оптимизации потребуется добавить в тип `lifted` список маркеров типов, для которых определено преобразование, и создать следующую функцию:

```
val is_interesting : lifted -> 'a marker -> bool
```

Она для заданной функции и маркера определяет, нужно ли трансформировать данные соответствующего типа.

Для поиска подструктур, требующих обработки, воспользуемся алгоритмом, схожим с анализом потока данных. Сопоставим каждой вершине значение типа `bool`, которое показывает, достижимы ли из нее узлы, требующие обработки. Начальная разметка будет сопоставлять вершинам, которые нужно отобразить, значение `true`, а остальным — `false`. Потоковая функция будет сопоставлять каждому узлу дизъюнкцию его старого значения и значений его потомков. Итерирование такой функции от начальной разметки приведет к ее неподвижной точке, так как никакая итерация не может изменить значение в вершине с `true` на `false`. Очевидно, эта неподвижная точка будет являться решением задачи.

Проблемой на пути реализации этого алгоритма является отсутствие явного представления типа. Но если ввести соглашение об организации вычислений в функции `gmapT`, этой проблемы можно избежать. Будем считать, что все функции `gmapT` имеют структуру, сходную с той, которая приводилась в примерах применения специализации к типу: вначале вычисляются все функции трансформации для потомков, а затем возвращается замыкание, исполь-

зующее эти функции. В таком случае для определения потомков каждой вершины будем использовать следующий алгоритм. Заведем переменную `dfs_log`, в которой будет содержаться стек обхода структуры типа. Когда алгоритм обхода посещает очередную вершину, он добавляет в стек маркер этой вершины, а при завершении обработки вершины он вынимает все маркеры, лежащие выше нижнего вхождения маркера данной вершины. Тогда эти вынимаемые элементы и будут маркерами ближайших потомков.

Специализация к «интересной» функции будет являться дополнением к специализации к типу. Специализация будет состоять из двух этапов: анализ и вычисление. На этапе анализа будем производить холостые вызовы функции специализации до тех пор, пока разметка не стабилизируется. Таким образом, мы вычислим описанную выше неподвижную точку. На втором этапе будет произведена настоящая специализация, использующая полученную на первом этапе информацию. Далее приведен код этого алгоритма.

```
let everywhere f =
  let context = Tbl.create () in
  let analysis_phase = ref true in
  let m = ref (M.create ()) in
  let dfs_log = ref [] in
  let rec transform : 'a . 'a typeinfo -> ('a -> 'a) =
    fun ti ->
      let cur = is_interesting f ti.marker in
      if !analysis_phase then
        let _ =
          if List.exists
               (equals (untype ti.marker))
               !dfs_log
          then
            dfs_log := !dfs_log @ [untype ti.marker]
          else
            (let next_log =
               !dfs_log @ [untype ti.marker]
             in
             dfs_log := next_log;
             let _ =
               ti.gmapT { transform = transform }
             in
```

```
            m := M.update !m (untype ti.marker)
              (cur ||
                (List.fold_left
                  (fun acc v -> acc || M.get !m v)
                  false
                  (dfs_children
                    (untype ti.marker)
                    (!dfs_log))));
            dfs_log := next_log)
      in
      stub
    else
      let f = f.f ti.marker in
      let children = M.get !m (untype ti.marker) in
      let tr =
        try
          Tbl.find context ti.marker
        with
          _ ->
            let tr = ref stub in
            Tbl.add context ti.marker tr;
            tr := ti.gmapT { transform = transform };
            Tbl.remove context ti.marker;
            tr
      in
        if cur && children then
          compose f (fun x -> !tr)
        else if not cur && children then
          fun v -> !tr v
        else if cur && not children then
          f
        else id
in
let rec fix_interesting ti =
  let old_m = !m in
  let _ = transform ti in
  if not (M.equals !m old_m)
  then fix_interesting ti
in
fun ti ->
  fix_interesting ti;
```

```
analysis_phase := false;
transform ti
```

В данном коде `untype` приводит любой маркер к типу `untyped_marker`; `equals` сравнивает два значения типа `untyped_marker`; модуль `M` реализует функции работы со структурой данных, представляющей разметку: `equals` сравнивает две разметки, `update` создает новую разметку с одним измененным значением, `get` возвращает значение в данной вершине; функция `dfs_children` возвращает все значения списка, стоящие после первого вхождения указанного.

2.8.3. Использование продолжений

Другой проблемой является рекурсивность процесса обработки, которая приводит к необходимости использования большого количества памяти на стеке. Ее решением является использование продолжений [18]. Требуемые изменения функций стандартны, поэтому не будут приводиться в данной статье.

3. Апробация и сравнения

Для доказательства эффективности представленного метода были проведены его испытания на задачах компилятора простого языка. Как и в приведенных ранее примерах, определение обобщенной версии процедуры идентификации переменных содержало код обработчиков только тех узлов синтаксического дерева, которые представляли переменные. Наивная версия той же процедуры содержала код обработчиков всех вершин синтаксического дерева. В результате использование представленного подхода позволило сократить объем кода данной процедуры в 10 раз. Также применение подхода значительно упростило реализацию процедур проверки типов и генерации HTML-представления для дерева разбора, так как их обобщенные версии не содержали кода, отвечающего за обход данных.

Далее для анализа производительности предложенного метода пример из SYB [10] был реализован различными способами: «вручную» (hard-coded), без специализации (without specialization), с использованием специализации к типу (specialization on type), с ис-

пользованием специализацией к типу и функции (specialization on type and function). На рис. 2 приведен результат сравнения производительности этих реализаций. Горизонтальная шкала — это логарифм от размера данных, вертикальная шкала представляет время работы алгоритма в секундах. Тесты показывают, что скорость обработки при использовании описанного подхода падает в среднем в 1,5 раза. Падение производительности при использовании оригинального подхода SYB составляет в среднем 3,5 раза.

Существует ряд инструментов для языка Objective Caml, которые решают схожие проблемы: Generic for the Ocaml Masses [19], Camlp4 FoldGenerator, Deriving [22]. Был произведен анализ их возможностей в сравнении с возможностями данного подхода. Результаты анализа представлены в таблице. Первый столбец таблицы содержит список возможностей. Каждый следующий столбец показывает, какими возможностями обладает соответствующий ему подход.

Одной из причин создания данного метода послужила проблема, с которой столкнулись при реализации компилятора языка описания аппаратуры HaSCoL [1, 2, 4]. Текущая версия компилятора

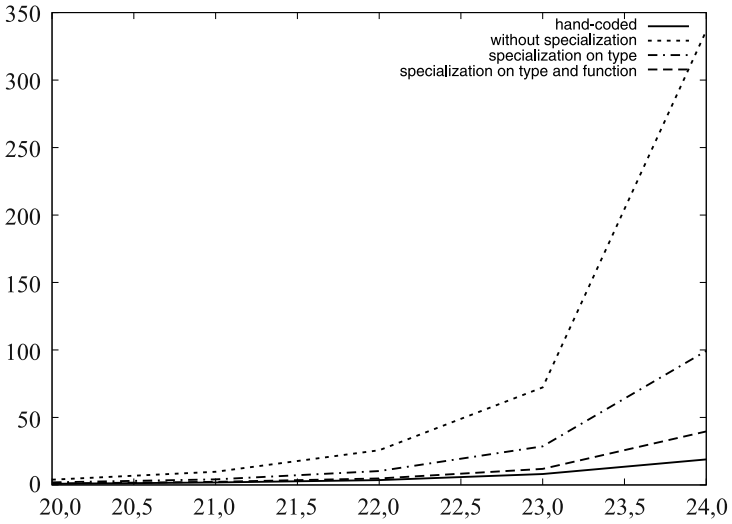

Рис. 2. Зависимость времени исполнения от размера данных для разных версий функции increase

Возможность	Данный подход	Camlp4	Deriving	Geberic for OM
Поддержка произвольных типов	Есть	Нет	Есть	Есть
Поддержка конструкторов типов	Есть	Нет	Есть	Есть
Произвольные запросы и трансформации	Есть	Есть	Нет	Нет
Обеспечение хвостовой рекурсии	Есть	Нет	Нет	Нет
Использование внутри расширения синтаксиса	Есть	Нет	Нет	Нет

содержит большое количество стереотипного кода, который затрудняет дальнейшую разработку и сопровождение. Для решения этой проблемы в следующей версии планируется использование описываемого подхода для реализации алгоритмов обработки синтаксического дерева.

К недостаткам данной реализации можно отнести отсутствие возможности обрабатывать циклические данные и необходимость вручную поддерживать взаимно-однозначное соответствие между маркерами и типами. Последний недостаток может привести к проблемам, например, в следующем случае:

```
module F (X : sig type t end) =
  struct
    type t = A of X.t | B
  end
module A = F (struct type t = int end)
module B = F (struct type t = int end)
```

Если значение типа `typeinfo` будет создано автоматически для типа `t` с помощью расширения синтаксиса, то маркеры типа `t` в модулях `A` и `B` будут различаться, так как содержимое функтора вычисляется после применения к модулю. Это может привести к ошибкам, если пользователь при создании функций обработки укажет тип не из того модуля, из которого требуется. При этом он не получит сообщения об ошибке на этапе компиляции.

Заключение

Представленные адаптация подхода SYB для языка Objective Caml и его оптимизации позволяют эффективно бороться со стереотипным кодом, связанным с обходом данных. В дальнейшем планируется развитие данного метода по трем основным направлениям. Первое включает в себя расширение возможностей за счет добавления новых базовых функций обработки, таких как `zip`, `unfold` и др. Вторым направлением является создание основанных на данном подходе средств, ориентированных на конкретные области, например сериализацию данных. Третий путь развития включает решение проблем и преодоление ограничений подхода, таких как отсутствие возможности обрабатывать циклические данные.

Список литературы

[1] *Булычев Д. Ю.* Разработка программно-аппаратных систем на основе описания макроархитектуры // Системное программирование. СПб.: Изд-во СПбГУ, 2004. С. 7–22.

[2] *Булычев Д. Ю.* Язык описания макроархитектуры для технологии совместной программно-аппаратной разработки // Системное программирование. Спб.: Изд-во СПбГУ, 2004. С. 24–48.

[3] *Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж.* Приемы объектно-ориентированного проектирования. Паттерны проектирования. СПб.: Питер, 2001. 368 с.

[4] *Boulytchev D., Medvedev O.* Hardware Description Language Based on Message Passing and Implicit Pipelining // Proceedings of the East-West Design and Test Symposium. 2009. P. 438–441.

[5] *Hinze R.* Generics for the masses // Journal of Functional Programming. 2004. P. 451–483.

[6] *Chakravarty M. M. T., Ditu G. C., Leshchinskiy R.* Instant Generics: Fast and Easy. 2009. URL: http:// www.cse.unsw.edu.au/~chak/ papers/ instant-generics.pdf

[7] *Kiselyov O.* Smash Along Your Boilerplate, 2007. URL: http://okmij.org/ftp/ Haskell/generics.html

[8] *Kiselyov O., Peyton-Jones S., Shan C.* Fun with Type Functions, 2010. URL: http:// research.microsoft.com/ en-us/ um/ people/ simonpj/ papers/assoc-types/fun-with-type-funs/typefun.pdf

[9] *Laemmel R., Peyton-Jones S.* Scrap More Boilerplate: Reflection, Zips, and Generalised Casts // Proceedings of ACM International Conference on Functional Programming, 2004. P. 244–255.

[10] *Laemmel R., Peyton-Jones S.* Scrap Your Boilerplate: A Practical Approach to Generic Programming // Proceedings of ACM SIGPLAN Workshop on Types in Language Design and Implementation. 2003. P. 26–37.

[11] *Laemmel R., Peyton-Jones S.* Scrap Your Boilerplate with Class: Extensible Generic Functions // Proceedings of ACM International Conference on Functional Programming. 2005. P. 204–215.

[12] *Leroy X., Doligez D., Frisch A. e.a.* The Objective Caml System Release 3.12, 2010. URL: http://caml.inria.fr/pub/docs/manual-ocaml/

[13] *McNamara B.* Catamorphism, 2008. URL: http://lorgonblog.wordpress.com/ 2008/04/05/catamorphisms-part-one/

[14] *Mitchell N., Runciman C.* Uniform Boilerplate and List Processing // Proceedings of the ACM SIGPLAN Workshop on Haskell Workshop (Haskell'07), 2007. P. 49–60.

[15] More Expressive GADT Encodings via First Class Modules, 2010. URL: http://ocaml.janestreet.com/?q=node/81

[16] *Pierce B. C.* Types and Programming Languages. MIT Press, 2002. 623 p.

[17] *Rodriguez A., Jeuring J., Jansson P.* et. al. Comparing Libraries for Generic Programming in Haskell // SIGPLAN Not. Vol. 44. 2008. P. 111–122.

[18] *Sussman G. J.* Scheme: An Interpreter for Extended Lambda Calculus // Memo 349, MIT AI Lab. 1975. 35 p.

[19] *Yallop J., Kiselyov O.* First-Class Modules: Hidden Power and Tantalizing Promises. 2010. URL: http:// okmij.org/ ftp/ ML/ first-class-modules/ first-class-modules.pdf

[20] *Yallop J.* Practical generic programming in OCaml // Proceedings of the Workshop on Workshop on ML, 2007. P. 83–94.

[21] Objective Caml: http://caml.inria.fr/ocaml/

[22] Haskell. URL: http://www.haskell.org

[23] Camlp4 FoldGenerator. URL: http:// caml.inria.fr/ pub/ old_caml_ site/ camlp4/ index.html