Компонентизация языковых процессоров на основе расширяемых типов данных и управляемых ими преобразователей*

Д. Ю. Булычев
dboulytchev@gmail.com

В данной статье рассматривается техника программирования, основанная на применении обобщенных управляемых типами преобразователей (generic type-driven transformers) для расширяемых типов данных. Семантика таких преобразователей полностью определяется типом преобразуемых данных, и поэтому они могут быть сгенерированы по типу автоматически. В качестве предметной области для применения этой техники мы рассматриваем языковые процессоры — программы, обрабатывающие представления других программ в виде некоторой структуры данных. Использование данного подхода позволяет «собирать» языковые процессоры из готовых, предварительно откомпилированных и статически типизированных компонент. При этом полученные процессоры в свою очередь могут играть роль компонент для расширения и настройки.

Ключевые слова: обработка данных, компиляторы и языковые процессоры, обобщенное программирование, типы данных.

Введение

Компонентная организация приложений — повсеместный, хорошо зарекомендовавший себя на практике подход. Использование готовых параметризуемых компонент позволяет радикально сократить время разработки, тестирования и сопровождения программ.

* Работа выполнена при поддержке РФФИ (грант № 10-01-00583-а).

Это в полной мере относится и к языковым процессорам — компиляторам, средствам статического анализа, реинжиниринга, понимания программ (program understanding) и пр. Однако на сегодняшний день компонентизация в области языковых процессоров используется относительно слабо — обычно сборка из готовых компонент невозможна без их найстройки и «подгонки» под каждое конкретное использование. Широко практикуется перегенерация частей реализации по декларативному или смешанному описанию, что повышает монолитность реализации и препятствует истинной компонентизации. В то же время многие современные языки программирования содержат специально разработанные механизмы, поддерживающие компонентизацию (средства раздельной трансляции, параметризованные и расширяемые типы данных и пр.), что открывает новые возможности для компонентизации в тех областях, где это было затруднительно сделать раньше. В частности, язык Objective Caml (OCaml) содержит интересные возможности для манипулирования данными, имеющими «открытый» (не полностью определенный) тип (прежде всего, полиморфные варианты).

В данной статье рассматривается прием программирования на OCaml, основанный на применении *обобщенных управляемых типами преобразователей* (*generic type-driven transformers*), и показывается, как использование таких преобразователей для расширяемых типов позволяет «собирать» языковые процессоры из готовых, предварительно откомпилированных и статически типизированных компонент. Применение данного подхода продемонстрировано на примере семейства обероноподобных языков, имеющих общего предка и получающихся синтаксическим или семантическим расширением друг из друга. Описываемый подход может лечь в основу предметно-ориентированного языка для реализации расширяемых компонентно-организованных языковых процессоров.

1. Язык Objective Caml и его особенности в контексте данной работы

Objective Caml или, кратко, OCaml [4,8] — язык программирования высокого уровня, который с 1985 года разрабатывается, поддерживается и распространяется Национальным институтом информатики и автоматизации (Institut National de Recherche en Infor-

matique et en Automatique, INRIA), Франция. Будучи членом семейства ML-языков, OCaml обладает рядом черт, свойственных статически-типизированным функциональным языкам, таких как функции высшего порядка, параметрический полиморфизм и вывод типов. Однако в дополнение к ним этот язык непротиворечиво включает в себя объектно-ориентированные конструкции — объекты и классы со структурным отношением порядка на типах, выразительную систему модулей и функторов (модулей, параметризованных другими модулями), что делает возможным его использование в рамках различных парадигм программирования. В дополнение к этому OCaml снабжен инструментами расширения синтаксиса — CamlP4/CamlP5[1] — которые позволяют добавлять в базовый язык предметно-ориентированные конструкции.

Для того чтобы подход, рассматриваемый в данной статье, был осуществим, требуется существенная поддержка со стороны языка реализации. Язык OCaml осуществляет такую поддержку, в частности, в виде *полиморфных вариантов* (*polymorphic variants*) [5]. Коротко говоря, полиморфные варианты представляют собой особый вид алгебраических типов данных, которые позволяют различным типам иметь одинаковые конструкторы, возможно, с различным количеством аргументов и различными их типами. Полиморфно-вариантный тип не обязательно предварительно описывать. Еще одной особенностью полиморфно-вариантных типов является их возможная незамкнутость — в определенном контексте может быть известен только частичный набор их конструкторов. Такие типы могут быть использованы для решения *проблемы выражения* (*expression problem*) [6, 7], которая заключается в реализации модульной и статически-типизированной обработки расширяемых структур данных.

2. Управляемые типами преобразователи

Управляемые типами преобразователи (*type-driven transformers*) — это такие преобразователи данных, семантика которых определяется типом этих данных. Поэтому такие преобразователи могут быть сконструированы автоматически, исходя только из типа преобра-

[1] http://brion.inria.fr/gallium/index.php/Camlp4,
http://pauillac.inria.fr/~ddr/camlp5/

зуемых данных. Применительно к языку OCaml также верно и обратное — тип данных может быть выведен автоматически, исходя из устройства преобразователя.

Управляемые типами преобразователи для расширяемых типов будут расширяемыми. Преобразователи для различных типов могут быть скомбинированы, образовав преобразователь для объединенного типа, который, в свою очередь, остается открытым и может быть расширен позднее. В результате может быть получен набор независимых преобразователей, каждый из которых осуществляет решение некоторой задачи для определенного подкласса структур данных.

Далее мы на примере рассмотрим конструкцию управляемых типом преобразователей для расширяемых типов в простом случае, а затем изложим два последовательных её обобщения — на случай параметризованного преобразования и на случай монадического преобразования.

2.1. Простой случай

Под простым случаем мы имеем виду ситуацию, когда управляемое типом преобразование совершенно конкретно и известно заранее.

Мы проиллюстрируем этот случай на каноническом примере — функции вычисления значения выражения. Основная идея заключается в том, чтобы представить такую функцию как некоторую композицию частичных преобразователей, каждый из которых обрабатывает только определенные вершины дерева выражения.

Рассмотрим функцию, которая обрабатывает только вершины, соответствующие каким-то бинарным операциям — например, сложению и вычитанию[2]:

```
let rec evalAddSub ext expr =
  let self = evalAddSub ext in
  match expr with
  | `Add (x, y) → self x + self y
  | `Sub (x, y) → self x − self y
  | z → ext self z
```

[2]Конструкторы полиморфно-вариантного типа отличаются от конструкторов обычных алгебраических типов на лексическом уровне — они начинаются с символа обратного апострофа.

В этом примере ext обозначает функцию расширения, т. е. такую функцию, которая обрабатывает остальные типы вершин полиморфно-вариантного значения помимо Add и Sub. Поскольку мы хотим преобразовывать потенциально рекурсивные типы, мы должны снабдить функцию расширения своей собственной функцией расширения, которая должна быть в состоянии преобразовывать вершины всех типов, включая Add и Sub. Эта функция расширения — self — получена здесь частичным применение evalAddSub к ext. Заметим, что evalAddSub — это именно преобразователь неполного или *открытого* типа, поскольку никакое конечное выражение не может быть создано с использованием только конструкторов Add и Sub.

Далее аналогичным образом конструируется преобразователь вершин, соответствующих идентификаторам. Дополнительно этот преобразователь получает состояние s, связывающее имена идентификаторов с их значениями:

```
let rec evalIdent s ext expr =
  let self = evalIdent s ext in
  match expr with
  | `Ident n → s n
  | z → ext self z
```

Отметим, что семантика этих функций полностью определена типом обрабатываемых данных. Несмотря на то, что здесь мы не используем явные описания типов, мы рассуждаем именно в терминах типов, а не их реализаций. Заметим также, что данные функции полностью независимы друг от друга — они могут быть описаны в разных (независимых друг от друга) единицах компиляции или библиотеках. Это свойство существенно отличает данный подход от подхода, описанного в [6].

Легко видеть, что определенная комбинация (именно, сумма или объединение) открытых типов, неявно определенных функциями evalIdent и evalAddSub, соответствует некоторому содержательному типу для деревьев выражений, составленных из идентификаторов, сложений и вычитаний. Хотелось бы получить преобразователь для такого типа непосредственно из преобразователей его составляющих частей, не повторяя их кода и не обрабатывая все конструкторы повторно. Это можно сделать с помощью следующе-

го комбинатора, который играет роль суммы типов:

```
let (++) left right = fun ext s →
  left (fun self → right (fun _ → ext self)) s
```

Данный комбинатор получает в качестве параметров расширяемые преобразователи открытых типов и конструирует расширяемый преобразователь для открытого типа, соответствующего их сумме. Это достигается просто предоставлением правильных функций расширений для каждого из «слагаемых». Например:

```
let evalAddSubIdent s = evalIdent s ++ evalAddSub
```

— есть (расширяемая) функция, которая вычисляет значения выражений, составленных из вершин Add, Sub и Ident. Расширяемость здесь означает, что мы всегда можем добавить что-то, выраженное в виде управляемого типом преобразователя, к данной функции. Например, если нам требуется рассмотреть новый тип вершины — скажем, умножение — мы всегда можем написать преобразователь только для этой вершины и далее скомбинировать его с уже существующими:

```
let rec evalMul ext expr =
  let self = evalMul ext in
  match expr with
  | `Mul (x, y) → self x * self y
  | z → ext self z

let evalAddSubIdentMul s = evalAddSubIndent s ++ evalMul
```

Наконец, для того чтобы применить полученные таким образом преобразователи к «замкнутому» типу, мы должны предоставить некоторую «замыкающую» функцию-расширение. Легко видеть, что данной функцией является функция применения:

```
let apply f x = f x
```

Например:

```
let x =
  evalAddSubIdent
     (function "a" → 1 | "b" → 2 | "c" → 3 | "d" → 4)
     apply
     (`Add (
```

```
    `Sub (`Ident "a",
          `Mul (`Ident "b", `Ident "c")
    ),
    `Ident "d"
  )
)
```

Здесь в результате вычислится значение выражения, равное -1.

Таким образом, мы показали, что каждое конкретное преобразование расширяемого типа данных может быть легко закодировано изложенным методом. При этом то же самое преобразование для суммы типов получается из преобразований для его слагаемых с помощью комбинатора «++».

2.2. Параметризация относительно конкретного преобразования

В предыдущих примерах мы рассматривали только некоторый конкретный преобразователь — вычислитель выражений; при таком подходе для другого преобразования потребуется отдельный преобразователь. Однако описанный подход допускает простое обобщение: разделим преобразование, полученное в простом случае, на два — *конкретное* (*specific*) и *обобщенное* (*generic*). Обобщенное преобразование будет принимать конкретное, представленное некоторым образом преобразование в качестве параметра.

Конкретное преобразование для открытого типа мы будет представлять в виде объекта, у которого столько же методов, сколько конструкторов известно для этого открытого типа. Каждый такой метод будет получать $n+1$ аргумент, где n — число аргументов соответствующего конструктора. Дополнительным первым аргументом будет само значение, к которому применяется конкретное преобразование, а остальные аргументы будут представлять собой подзначения этого значения, к которым данное преобразование уже применено.

Для примеров из предыдущего раздела обобщенные преобразователи буду выглядеть следующим образом:

```
let rec gmapAddSub t ext expr =
  let self = gmapAddSub t ext in
  match expr with
```

```
| `Add (x, y) → t#add expr (self x) (self y)
| `Sub (x, y) → t#sub expr (self x) (self y)
| z → ext self z
```

Здесь t обозначает конкретный преобразователь. Он реализован как объект с методами Add и Sub[3]. Каждый такой метод получает в качестве аргументов результаты того же преобразования, примененного к непосредственным подзначениям соответствующего значения, и само это значение как таковое и возвращает результат преобразования. Теперь вычислитель выражений может быть представлен следующим образом:

```
let evalAddSub ext expr =
  gmapAddSub (object
                method add _ x y = x + y
                method sub _ x y = x − y
              end) ext expr
```

Система типов OCaml позволяет использовать объекты без предварительного описания их типов. Другим важным свойством объектных типов является их потенциальная полиморфность. Это означает, что описание gmapAddSub не накладывает никаких искусственных ограничений на типы обрабатываемых данных или на типы локальных преобразователей. Например, преобразование копирования может быть реализовано с помощью той же функции, но другого локального преобразования, представленного объектом другого (но «совместимого») типа:

```
let copyAddSub ext expr =
  gmapAddSub (object
                method add _ x y = `Add (x, y)
                method sub _ x y = `Sub (x, y)
              end) ext expr
```

Так как каждое конкретное преобразование может быть представлено как частичное применение некоторой обобщенной функции gmapAddSub к конкретному преобразованию, представленному некоторым объектом, комбинатор суммы может быть оставлен без изменений. Например, вычислитель выражений для открытых типов, рассмотренных выше, может быть закодирован и использован следующим образом:

[3]Символ «#» в OCaml обозначает вызов метода.

```
let evalAddSubIdent s =
  gmapAddSub (object
                method add _ x y = x + y
                method sub _ x y = x - y
              end) ++
  gmapIdent (object method ident _ n = s n end)
```

Используя описанный подход, мы можем легко реализовывать преобразователи, комбинируя и расширяя предопределенный набор локальных преобразователей для каждого открытого типа.

2.3. Монадические преобразователи

Следующим обобщением описываемых преобразователей является их преобразование в *монадические*. В контексте языковых процессоров такое обобщение является совершенно естественным, поскольку довольно часто в процессе преобразования к финальному значению надо что-то «подмешать» (например, сообщение об ошибках и пр.). Монады[4] [9] являются естественным способом закодировать вычисления такого рода.

Поскольку монада представляется конструктором типа, а в OCaml единственным способом осуществить параметризацию по конструктору является организация функтора, нам необходимо «поднять» наши преобразования на модульный уровень.

Далее мы будем пользоваться следующим монадическим интерфейсом, выраженным в терминах типа модуля:

```
module type Monad =
  sig
    type 'a t
    val return : 'a → 'a t
    val (>>=)  : 'a t → ('a → 'b t) → 'b t
  end
```

Монадические версии для преобразователей типов, которые мы рассматриваем в качестве основного примера, выглядят так:

```
module AddSubMapper (M : Monad) =
```

[4]Монады — это абстрактный тип данных, интерфейс которого позволяет абстрагировать понятие вычисления с дополнительными эффектами, которые определяются позже (подробнее см. [9]).

```
  struct
    open M
    let rec gmap t ext expr =
      let self = gmap t ext in
      match expr with
      | `Add (x, y) → self x >>= (fun x →
                      self y >>= (fun y → t#add expr x y))
      | `Sub (x, y) → self x >>= (fun x →
                      self y >>= (fun y → t#sub expr x y))
      |  x          → ext self x
  end
```

и

```
module IdentMapper (M : Monad) =
  struct
    open M
    let rec gmap t ext expr =
      let self = gmap t ext in
      match expr with
      | `Ident n → t#ident expr n
      | x        → ext self t
  end
```

Обычные, «не монадические» версии преобразователей теперь могут быть получены применением этих функторов к тождественной монаде, имеющей очевидную реализацию:

```
module Id =
  struct
    type 'a t = 'a

    let return x = x
    let (>>=) x f = f x
  end
```

Например, вычисление значения выражения может быть записано теперь следующим образом:

```
let evalAddSubIdent s =
  let module AddSub = AddSubMapper (Id) in
  let module Ident  = IdentMapper  (Id) in
  AddSub.gmap (object
```

```
                method add _ x y = x + y
                method sub _ x y = x - y
              end) ++
  Ident .gmap (object
                method ident _ n = s n
              end)
```

У этой реализации есть очевидный недостаток — ее придется писать отдельно для каждой монады. Этот недостаток легко исправляется с помощью *модулей первого класса* (*first-class modules*):

```
let evalAddSubIdent m s =
  let module M = (val m : Monad) in
  let module AddSub = AddSubMapper (M) in
  let module Ident  = IdentMapper  (M) in
  AddSub.gmap (object
                method add _ x y = M.return (x + y)
                method sub _ x y = M.return (x - y)
              end) ++
  Ident .gmap (object
                method ident _ n = M.return (s n)
              end)
```

Теперь функция evalAddSubIdent получает монаду, над которой происходят все вычисления, в качестве первого аргумента. Обычное вычисление над тождественной монадой теперь может быть получено просто в виде частичного применения:

```
evalAddSub (module Id : Monad)
```

Использование монад расширяет функциональность преобразователей. Например, если мы захотим добавить в наши выражения деление, то нам потребуется как-то обрабатывать ошибочную ситуацию, которая возникает в том случае, когда делитель равен нулю. Это легко выразить, используя ту же функцию, но для другой монады, например *монады исключений* (*exception monad*):

```
module WithException =
  struct
    type 'a t = Ok of 'a | Error of string

    let return x = Ok x
```

```
          let (>>=) x f = match x with Ok x → f x | Error msg →
Error msg
        end
```

Вычисление с ошибками для конструктора деления будет, конечно, «привязано» к этой монаде:

```
let evalDiv =
  let module Div = DivMapper (WithException) in
  Div.gmap (object
              method div _ x y =
                if y = 0 then WithException.Error "zero divider"
                         else WithException.Ok (x / y)
            end)
```

Однако эту функцию можно скомбинировать обычным образом с функцией вычисления выражений предыдущего вида, которая ни к какой монаде не привязана:

```
let evalAddSubIdentDiv s =
  evalAddSubIdent (module WithException : Monad) s ++
  evalDiv
```

Таким образом, мы видим, что данный подход легко обобщается на монады с сохранением всех полезных свойств.

3. Компонентная сборка компиляторов семейства обероноподобных языков

В данном разделе мы рассмотрим применение описываемого подхода для конкретной задачи — реализации компиляторов семейства обероноподобных языков. Несмотря на то, что речь не идет о полноценных промышленных компиляторах, данная реализация позволяет оценить применимость этого подхода в модельных условиях.

Работа над данным набором компиляторов проводилась в рамках конкурса инструментов для реализации языков (2010 LDTA Tool Challenge), проводимого под эгидой семинара LDTA (Language Definitions, Tools and Applications) в рамках конференции ETAPS (European Joint Conferences on Theory and Practice of Software) в 2010 году.

3.1. Семейство обероноподобных языков и таксономия поставленных задач

В качестве отправной точки для реализации был выбран язык Oberon-0, который представляет собой императивный язык программирования, разработанный Н. Виртом в образовательных целях и использованный им в качестве эталонного языка в [10]. Этот язык содержит стандартный набор императивных черт: переменные, выражения, простейшие конструкции управления (присваивание, условие, циклы), процедуры с передачей параметров по значению и по ссылке, вложенные процедуры, агрегатные типы (массивы и структуры) с именной эквивалентностью. Таким образом, по замыслу автора, Oberon-0 является модельным языком, содержащим всё, что надо, чтобы проиллюстрировать основные задачи компиляции, но в то же время достаточно простым, чтобы эта иллюстрация была обозримой.

Далее приведена программа на языке Oberon-0. Эта программа читает из входного потока число и печатает его произведения на числа от 1 до 10 включительно:

```
MODULE Multiples;

CONST limit = 10;

VAR
  base, count, mult : INTEGER;

PROCEDURE calcmult (i          : INTEGER;
                    base       : INTEGER;
                    VAR result : INTEGER);
BEGIN
  result := i * base
END calcmult;

BEGIN
  Read (base);
  FOR count := 1 TO limit DO
    calcmult (count, base, mult);
    Write (mult);
    WriteLn
  END
```

Таблица 1. Языки семейства

Уровни языка	
L0	Константы, переменные, атомарные типы, простые выражения и присваивания
L1	Конструкции `IF` и `WHILE`
L2	Конструкции `FOR` и `CASE`
L3	Процедуры
L4	Массивы и структуры
L5	Вложенные процедуры

```
END Multiples.
```

Задача реализации компилятора для Oberon-0 была разложена на составные части по двум измерениям — в смысле языка и в смысле конкретных подзадач.

В языковом смысле Oberon-0 был представлен в виде члена семейства языков, часть которых являлись его подъязыками, а часть — надъязыками. При этом один язык семейства получался из другого расширением. Характеристика этого семейства дана в таблице . Языку Oberon-0 в этой иерархии соответствует уровень L4. L5 — это язык, синтаксически неотличимый от L4, но обладающий вложенными процедурами в полном смысле этого слова (т. е. имеющими доступ к средам объемлющих функций). Oberon-0 хоть и содержит вложенные функции как синтаксическую конструкцию, такой семантикой не обладает.

Совокупность решаемых задач компиляции показана в таблице . Она представляет собой типичный набор просмотров, которые должен реализовывать практически любой компилятор статически-типизированного языка. В качестве целевой платформы здесь выступает язык C, но с тем же успехом мог бы быть использован, например, Java-байткод или машинный язык в форме какого-либо ассемблера.

Каждая комбинация языка семейства и конкретной задачи образует некоторый законченный артефакт. Например, выбрав язык L2 и задачу T3, мы получаем артефакт, который может осуществлять типовой анализ программ, содержащих конструкции до `FOR` и `CASE` включительно, но не поддерживает процедуры или агрегатные типы и не может генерировать код.

Таблица 2. Подзадачи компиляции

Задачи компиляции	
T1	Синтаксический анализ и форматирование
T2	Анализ имен (идентификация)
T3	Анализ типов
T4	Понижение уровня (desugaring)
T5	Генерация C

Таблица 3. Таблица артефактов

Артефакт	*Язык*	*Задачи*	
A1	L2	T1–2	Базовый язык с форматированием и идентификацией
A2a	L3	T1–2	A1 плюс процедуры и их идентификация
A2b	L2	T1–3	A1 плюс анализ типов
A3	L3	T1–3	A2a и A2b
A4	L4	T1–5	Все задачи для L4
A5	L4	T1–5	Все задачи для L5

Для реализации были выбраны лишь некоторые артефакты, которые кратко характеризованы в табл. 3. Эти комбинации были определены при проведении конкурса и выбраны для того, чтобы сохранить репрезентативность задач и избежать излишней сложности.

3.2. Использование управляемых типом преобразователей для решения сформулированных задач

Управляемые типом преобразователи в массовом порядке были использованы для решения сформулированных задач. Единственными задачами, где их применить не удалось, оказались синтаксический анализ (что довольно естественно) и λ-лифтинг [11] для преобразования программы с вложенными функциями в программу без них. Во всех остальных задачах управляемые типом абстрактного синтаксического дерева преобразователи были основным инструментом.

Общий план реализации был такой.

- Для каждого языка и каждой его синтаксической категории описывался обобщенный монадический преобразователь, аналогичный тем, которые мы рассматривали для дерева выражений. Этот преобразователь определял открытый тип абстрактного синтаксического дерева для данной категории.
- При расширении языка описывался аналогичный преобразователь для «дельты», т. е. только той части языка, которой не было в базовом.
- Все задачи, которые можно было выразить через управляемые типами преобразователи, реализовывались с помощью преобразователей для соответствующих синтаксических категорий.
- На верхнем уровне решения нужных задач собирались воедино из решения этих же задач для частичных типов данных с помощью комбинатора «++».

Вкратце опишем решения некоторых задач таким способом.

Для форматированной печати текста программ была использована библиотека принтер- и парсер-комбинаторов Ostap[5]. Принтер-комбинаторы в этой библиотеке являются просто комбинаторным интерфейсом к стандартной библиотеке форматированного вывода для OCaml. Управляемые типом преобразователи были использованы, чтобы сконвертировать представление программы (её дерево) в соответствующий принтер. Например, для выражений функция печати выглядит следующим образом:

```
let gprint ps ext expr =
  let b x = hovboxed (listBySpaceBreak x) in
  let op (s, p) = string s, p in
  let w p' (x, p) = if p' < p then rboxed x else x in
  imap
    (object
       method binop _ o x y =
         let s, p = op (ps#binop o) in b [w p x; s; w p y], p
       method unop  _ o x   =
         let s, p = op (ps#unop o) in b [s; w p x], p
       method const _   x   = ps#const x
     end
```

[5] http://caml.inria.fr/cgi-bin/hump.en.cgi?contrib=513

```
  )
  ext
  expr
```

Код этой функции, в основном, состоит из использования принтер-комбинаторов. Функция imap есть в точности управляемый типом преобразователь для открытого типа выражения и тождественной монады. Её параметры — это схема печати ps, которая инкапсулирует элементы конкретного синтаксиса, используемого при выводе, и функция расширения ext, смысл которой мы уже описывали. Мы используем схему печати для того, чтобы иметь возможность осуществлять вывод в разных конкретных синтаксисах. Одним из них является синтаксис исходного языка, а другим — синтаксис языка C. Таким образом, форматированный вывод и генерация C-кода реализованы с помощью одних и тех же функций.

Функции печати и генерации C-кода для нужного языка получаются просто комбинированием (в терминах комбинатора «++») функций печати и генерации для структур данных, соответствующих представлению разных конструкций абстрактного синтаксического дерева для данного языка.

Анализ имен также реализован в виде управляемого типом преобразователя, только над монадой исключений, поскольку мы должны собрать сообщения об ошибках. Например, для дополнительных операторов, появляющихся в языке L2, код функции анализа имен таков:

```
let resolve ref cexpr expr ext stmt =
  cmap ext
    (mapT (fun _ s → Monad.Checked.return s))
    ref
    cexpr
    expr
    stmt
```

Здесь cmap — это управляемый типом преобразователь для открытого типа, представляющего данные конструкции, специализированный на монаду исключений ext — функция дальнейшего расширения; второй параметр — конкретное преобразование (заключающееся просто в возврате того же значения, поскольку анализ имен не изменяет дерево на уровне оператора). Остальные параметры —

Таблица 4. Сложность (LOC) реализации задач

	T1	T2	T3	T4	T5	Остальное	Всего
L1	227	129	56	N/A	N/A	416	828
L2	77	22	38	N/A	N/A	48	185
L3	90	148	50	N/A	N/A	1	289
L4	70	64	82	105	161	26	508
L5	0	11	0	134	0	6	151
						Всего:	1961

это просто функции, решающие ту же задачу (анализ имен) для параметров данного типа.

Важно, что анализ имен конвертирует исходное дерево в структуру данных другого типа, поскольку теперь наряду с именами переменных в нем содержится и другая информация (например, какого «сорта» эта переменная — локальная, глобальная, параметр процедуры и т. д.).

Анализ типов реализован совершенно аналогично (и использует ровно те же преобразователи, что и анализ имен). Точно так же, как и в случае анализа имен, в результате получается структура данных, имеющая отличный от исходного тип (теперь дерево аннотировано типовой информацией).

3.3. Оценка результатов реализации

В таблице 4 показана сложность реализации отдельных задач для отдельных языков. Эта сложность измерялась в числе непустых строк кода (LOC), отличных от комментариев. Колонка «Остальное» соответствует вспомогательным компонентам, таким как драйверы верхнего уровня и собственно управляемые типами преобразователи. «N/A» означает, что данная задача не была реализована для данного уровня языка; «0» означает полное переиспользование реализации данной задачи для языка предыдущего уровня.

Поскольку мы имеем дело с расширяющимся семейством языков, нам требуется почти вся функциональность реализации языка предыдущего уровня для того, чтобы реализовать язык следующе-

Таблица 5. Сложность (LOC) реализации артефактов

Артефакты	LOC
A1	919
A2a	239
A2b	94
A3	50
A4	508
A5	151
Total	1961

го. Значения в данной таблице являются *инкрементальными*, т. е. они описывают, сколько строк кода надо добавить, чтобы из решения данной задачи для языка предыдущего уровня получить решение той же задачи для следующего. Например, чтобы реализовать задачи T1–T2 для языка L2 — используя реализации тех же задач для языка L1, нам требуется 185 дополнительных строк кода.

В таблице 5 представлены те же метрики для реализованных артефактов целиком. Все значения здесь вычислены на основе анализа предыдущей таблицы. Например, A2a есть в точности L3:T1 (90) + L3:T2(148) + L3:остальное(1) = 239 и т. д.

Заключение

Подход, который мы описали, имеет как достоинства, так и недостатки. Основным недостатком является трудность «ручного» программирования в терминах полиморфных вариантов. Хорошо известно [6], что использование полиморфных вариантов на «полную мощность» может приводить к возникновению труднопонимаемых и в силу этого трудно исправимых ошибок типизации. Этот недостаток в полной мере проявляется при использовании данного подхода, поскольку мы в массовом порядке прибегаем к неявно описанным и недоопределенным типам данных. В результате даже небольшая и в сущности тривиальная ошибка может привести к выдаче компилятором совершенно невразумительной диагностики, состоящей из многих сотен строк автоматически сгенерированных типовых выражений. Для преодоления этих трудностей требуется

вдумчивое программирование и специальная дисциплина реализации.

С другой стороны, сильными сторонами описанного подхода являются следующие черты:

1. Статическая типизация: различные просмотры реализованы над данными, имеющими различные типы. Таким образом, система типов инструментального языка осуществляет контроль корректности последовательности просмотров.

2. Модульность: описанный подход позволяет собирать компилятор из переиспользуемых и перекомпилированных компонент. Эти компоненты могут быть скомбинированы практически произвольным образом (нужно лишь, чтобы их комбинация имела разумный тип).

В качестве целей дальнейшей работы можно назвать устранение «ручного» программирования в терминах полиморфных вариантов путем создания предметно-ориентированного языка, встроенного в OCaml, а также расширения функциональности управляемых типом преобразователей с целью достижения полноты в каких-либо терминах (например, в терминах атрибутных грамматик).

Данная работа была проделана в контексте исследований в области разработки средств описания универсального и специализированного аппаратного обеспечения [1–3].

Список литературы

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

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

[3] *Terekhov A.* The Main Concepts of a New HLL Computer «CAMCOH» // Computer Science Journal of Moldova. 1993. 27 p.

[4] *Doligez D., Frisch A., Garrigue J., Rémy D., Vouillon J.* The OCaml System Release 3.12. 2011. http://caml.inria.fr/pub/docs/manual-ocaml/index.html

[5] *Garrigue J.* Programming with Polymorphic Variants // ACM SIGPLAN Workshop on ML. 1998.

[6] *Garrigue J.* Code Reuse Through Polymorphic Variants // Workshop on Foundations of Software Engineering. 2000.

[7] *Wadler P.* The Expression Problem // Discussion on the Java-Genericity Mailing List. 1998. 294 p.

[8] *Rémy Using D.* Understanding, and Unraveling the OCaml Language // Applied Semantics. Advanced Lectures. LNCS 2395. Springer Verlag, 2002. P. 413–537.

[9] *Wadler P.* Monads for Functional Programming // Advanced Functional Programming, First International Spring School on Advanced Functional Programming Techniques — Tutorial Text. Springer-Verlag, 1995. P. 24–52.

[10] *Wirth N.* Compiler Construction. Addison-Wesley, 1996. 133 p.

[11] *Johnsson T.* Lambda Lifting: Transforming Programs to Recursive Equations // Proceedings of a Conference on Functional Programming Languages and Computer Architecture. 1985. P. 190–203.