Введение в программирование на Objective Caml: обзор функциональных конструкций

Д.Ю.Булычев

17 апреля 2002 г.

"Племянника своего он не знает, следовательно, и не любит <...>"
— И.А.Гончаров, "Обыкновенная история"

1 Введение

Опыт участия в разнообразных реальных проектах привел автора к выводу, что огромное большинство проблем технического характера, возникающих при разработке сложных приложений, обусловлено недостаточностью средств, которые предоставляют наиболее используемые в коммерческой разработке языки программирования. Особенно остро ощущается недостаточный уровень абстракции при описании универсальных структур данных и процедур их обработки.

Общеизвестно, что в жизненном цикле разработки ПО львиную долю занимает не собственно написание первоначального кода, а его сопровождение и поддержка, внесение изменений, доработка, добавление новой функциональности на основе уже реализованной и т.д. Поэтому, начиная большой проект, стоит задуматься, удастся ли сохранить строгое разделение функциональности между модулями, достаточной ли окажется первоначально планируемая параметризация общеиспользуемых функций и не окажется ли в конце концов в нем три разномастных реализации одной и той же структуры данных, интерферирующих между собой. С этой точки зрения основным требованием к языкам разработки оказывается полноценная поддержка полиморфизма и абстрактных типов данных. Между тем, наиболее широко используемые в практике разработки языки, такие как C++ и Java, в катастрофически недостаточной мере предоставляют эти необходимые средства, рассматривая их через призму объектности. В то же время в значительной степени параллельно развивающаяся ветвь функциональных языков достигла результатов, которые позволяют рассматривать их в качестве реальной альтернативы для традиционных языков программирования.

Расхожим аргументом против использования языков функциональной группы является невозможность их эффективной реализации. Заметим, однако, что это верно лишь в отношении функциональных языков первого поколения, которые были основаны на интерпретации и динамическом контроле типов. В отношении же современных функциональных языков этот аргумент не может быть использован, поскольку их развитие с тех пор ушло далеко вперед и по результатам новейших исследований эффективность функциональных пограмм в вычислительных задачах не уступает эффективности императивных, а в задачах обработки сложных динамических структур данных им просто нет равных.

Ниже мы перечислим основные черты современных функциональных языков:

1. параметрический полиморфизм, то есть полиморфизм, не связанный с иерархией наследования объектов;

2. поддержка абстрактных типов данных в полном объеме;

3. статическая типизация;

4. поддержка раздельной трансляции;

5. автоматическое управление динамической памятью (сборка мусора);

6. поддержка вложенных функций и функций высших порядков, то есть функций, получающих в качестве параметров функцию и возвращающих новую функцию.

Следует отметить, что необходимость языковых средств, обладающих некоторыми из этимх черт, осознается повсеместно. Об этом, в частности, свидетельствует набирающий популярность термин “*generic programming*”, который по сути означает параметрический полиморфизм.

В данной статье предпринята попытка продемонстрировать основные черты и приемы программирования на функциональном языке последнего поколения. Несмотря на обилие свободной информации по этому вопросу подавляющее большинство разработчиков программного обеспечения обходят эту тему своим вниманием (что и явилось причиной выбора эпиграфа), поэтому автор ставил своей задачей не аккуратное и последовательное изложение основ функционального программирования, а просто демонстрацию основных конструкций и парадигм на примерах. В качестве базового языка избран язык Objective Caml, в течении ряда лет разрабатывающийся в INRIA (Institut National de Recherche en Informatique et en Automatique), Франция, который обладает всеми перечисленными выше чертами.

2 Основной пример

Излюбленными примерами для демонстрации простейших функциональных программ традиционно являются функции вычисления чисел Фибоначчи и факториала. Поэтому мы начнем с чего-нибудь другого — например, программы вычисления значения многочлена в данной точке по схеме Горнера. На языке C эта программа может выглядеть так:

```
# include <stdio.h>

typedef struct {int degree; float * coeff;} polynomial;

float Gorner (polynomial p, float x)
{
  float px = 0;
  int   i;

  for (i=0; i<=p.degree; i++) px = px * x + p.coeff [p.degree-i];

  return px;
}

main ()
{
  polynomial p;

  p.degree = 2;
  p.coeff  = (float *) malloc (3 * sizeof (float));

  p.coeff [0] = 1;
  p.coeff [1] = 2;
  p.coeff [2] = 3;

  printf ("p(1)=%f\n", Gorner (p, 1));
  printf ("p(2)=%f\n", Gorner (p, 2));
  printf ("p(0)=%f\n", Gorner (p, 0));
}
```

Ничего не стоит написать “кальку” этой реализации на языке Objective Caml:

```
type polynomial = {mutable degree: int; coeff: float array}

let gorner (p:polynomial) (x:float) =
  let px = ref 0. in

  for i=0 to p.degree do
    px := !px *. x +. p.coeff.(p.degree-i)
  done;

  !px
```

```
;;

Printf.printf "p(1)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 1.);
Printf.printf "p(2)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 2.);
Printf.printf "p(0)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 0.)
```

Прокомментируем коротко Objective Caml-программу:

- Описание типа выглядит вполне традиционно. Слово `mutable` обозначает то, что значение данного поля можно изменить присваиванием. Слова `float array`, как и ожидается, обозначают массив вещественных. Тонкостью, быть может, является лишь то, что значение поля `coeff` не может быть изменено целиком (например, с помощью присваивания другого массива), так как оно не описано как `mutable`. В то же время элементы этого поля (т.е. элементы массива) могут быть изменены — таково свойство типа `array`. Наконец, в качестве интриги, сообщим, что `float array` на самом деле является типовым выражением, конкретизирующим полиморфный тип `'a array` типом `float`, но об этом позже.

- Конструкция `let что-то = чему-то` в первом приближении может быть названа аналогом описания. В данном случае "описывается" функция `gorner` (то, что в Objective Caml–программе это имя начинается со строчной буквы, имеет вполне определенный смысл). Здесь можно обратить внимание на то, что для этой функции не указан тип результата (хотя это можно было бы сделать). Конструкция `let что-то = чему-то in где-то` является выражением. Выражением же является и конструкция последовательного исполнения, а точка с запятой — соответствующей ей операцией.

- Слово `ref` указывает на то, что заведенное значение имеет тип переменной. Собственно говоря, типовое различие между переменными и константами известно давно — например, в Алголе-68 описание такой переменной в строгом синтаксисе выглядело бы как **ref float** `px =` **heap float**. Прочитать эту конструкцию можно так: `px` есть константа типа имя (глобального) вещественного. В отличие от Алгола-68, в Objective Caml–программе все подобные значения глобальны, т.е. их областью действия является вся программа.

- Конструкция цикла также довольно прозрачна. Опять–таки, подобно Алголу, счетчик цикла `i` есть не переменная, а специальная сущность. В Алголе для счетчика существует специальная семантика; семантика счетчика Objective Caml простым образом выражается в общефункциональной парадигме — об этом ниже.

- Следующая строчка также выглядит вполне традиционно, за исключением того, что разыменование выражено явно с помощью операции `!`. Подобную операцию можно встретить, например, в языках Форт (только там она обозначается как `@`) или C. Арифметические операции над вещественными обозначаются другими значками, нежели таковые для целых. Это обусловлено тем, что в Objective Caml нет приведений и перегрузки. Наконец, точка необходима для указания вырезки из массива и выборки поля структуры.

- Результатом всей функции является значение ее последнего выражения. Опять–таки, в этом нет ничего странного.

- Символ `;;` (двойная точка с запятой) обозначает конец выражения или `let`-определения. Его действительно можно вставлять в конец каждой такой конструкции, однако стоит это делать только там, где в противном случае возникла бы ошибка. О том, какая ошибка возникла бы в данном случае — ниже.

- Последние три строчки делают ровно то же, что вся функция `main`. `Printf.printf` есть упоминание функции `printf` из модуля `Printf` (имена всех модулей должны начинаться с прописной буквы.) Форматная строка выглядит обычно, за исключением того, что в Objective Caml сопоставление аргументов формату *type-safe*. При вызове функции ее аргументы не окружаются скобками и не разделяются запятыми (иногда это требует дополнительных скобок в других местах).

- В Objective Caml можно сразу указать значение структуры или массива. Это напоминает изображения Алгола-68 или соответствующие расширения GNU C. Здесь такая конструкция называется конструктором. В данном примере использованы конструкторы структуры (в виде `{поле=значение; ...; поле=значение}` и массива (в виде `[|значение`$_1$`; ...; значение`$_N$`|]`).

- Поскольку в Objective Caml нет приведений, все начальные значения вещественных переменных должны быть изображены как вещественные константы.

Таким образом, Objective Caml обладает многими хорошо известными чертами традиционных императивных языков; с другой стороны, вряд ли имело смысл затевать весь разговор, если бы любую программу на C можно было бы преобразовать в аналогичную программу на Objective Caml путем добавления дополнительных символов. Сделаем же первый шаг к истинно функциональной реализации обсуждаемого алгоритма:

```
type polynomial = {mutable degree: int; coeff: float array}

let gorner p x =
  let px = ref 0. in

  for i=0 to p.degree do
    px := !px *. x +. p.coeff.(p.degree-i)
```

```
  done;

!px
;;

Printf.printf "p(1)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 1.);
Printf.printf "p(2)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 2.);
Printf.printf "p(0)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 0.)
```

Этот первый шаг заключается, как можно видеть, в окончательном удалении аннотаций типа параметров функции `gorner`. Выше уже было замечено, что для этой функции не указан тип результата. Объяснением служит тот факт, что этот тип может быть легко выведен из ее текста. Действительно, тип результата функции — есть тип ее результата — есть тип последнего выражения ее тела – есть тип выражения `!px` — есть тип значения, именуемого `px` — есть тип выражения `!px *. x +. p.coeff.(p.degree-i)` — есть тип результата операции `+.` — есть `float`, потому что так она описана (где-то, честное слово!) и нет перегрузок. Теперь осталось заметить, что ровно такие же размышления приводят к выводу о необязательности указания типов и для параметров. Правда, существует закономерный вопрос — верно ли, что тело функции всегда содержит достаточно информации для установления окончательного ее типа? Ответ на этот вопрос — нет, и это очень хорошо. Пока же сделаем еще один шаг к функциональной реализации, а именно — изведем переменную:

```
type polynomial = {mutable degree: int; coeff: float array}

let gorner p x =
  let rec inner px i =
    if i > p.degree then px
    else inner (px *. x +. p.coeff.(p.degree-i)) (i+1)
  in
  inner 0. 0
;;

Printf.printf "p(1)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 1.);
Printf.printf "p(2)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 2.);
Printf.printf "p(0)=%f\n" (gorner {degree=2; coeff=[|1.; 2.; 3.|]} 0.);
```

(как можно увидеть, конструкция `if-then-else` появилась на правах выражения (опять как в Алголе-68)).

Может возникнуть справедливый вопрос — а чем, собственно, плоха переменная? Обсуждение этого вопроса мы отложим на будущее, а пока заметим, что невозможность присваивания приводит к необходимости сворачивания цикла в рекурсивную функцию с параметрами, которые представляют собой контекст цикла. В этом преобразовании опять-таки нет ничего выдающегося — связь рекурсии с итерацией общеизвестна. Слово rec указывает на рекурсивное определение, т.е. inner внутри inner обозначает самое себя. В противном случае это не так — каждое вхождение let–определения как бы вводит вложенную область видимости, перекрывая предыдущее определение того же имени. Таким образом,

```
let x = 0
let x = x  (* предыдущее x *)
```

(комментарии в Objective Caml ограничиваются скобками (* и *), которые могут быть вложенными.)

Заметим, что внутри функции inner использованы имена, связанные в теле объемлющей функции. Таким образом, вложенные функции в Objective Caml обладают всей полнотой доступа к объемлющему контексту. Это тоже не новость — так обстоят дела практически во всех языках со вложенными средами. Отличием же функционального подхода является то, что, как это не странно звучит, область действия вложенной функции не ограничена областью действия объемлющей и, следовательно, ее контекст также долгоживущ! Это достигается применением специального преобразования — замыкания — о чем позже.

От почти канонической функциональной реализации нас отделяют два мелких изменения. Во-первых, поскольку тип polynomial нигде не использован (точнее, нигде не использовано его имя), его описание можно удалить (хотя вообще говоря конструкция описания типа не является бесполезной в Objective Caml). Наконец, сигнатура типа array позволяет узнать количество элементов в его значении, так что нет необходимости хранить его отдельно в структуре.

С учетом всего этого данная программа будет иметь вид

```
let gorner p x =
  let n = (Array.length p) - 1 in
    let rec inner px i =
      if i > n then px
      else inner (px *. x +. p.(n-i)) (i+1)
    in
    inner 0. 0
;;

Printf.printf "p(1)=%f\n" (gorner [|1.; 2.; 3.|] 1.);
Printf.printf "p(2)=%f\n" (gorner [|1.; 2.; 3.|] 2.);
Printf.printf "p(0)=%f\n" (gorner [|1.; 2.; 3.|] 0.)
```

Здесь Array.length p опять обозначает вызов функции length модуля Array с параметром p.

3 Частичное применение и карринг

Примерно к этому моменту автору данного текста надоело выписывать конструктор массива (который представляет наш многочлен) несколько раз. Тривиальное решение состоит в том, чтобы связать его значение с каким-нибудь именем:

```
let p = [|1.; 2.; 3.|]

Printf.printf "p(1)=%f\n" (gorner p 1.);
Printf.printf "p(2)=%f\n" (gorner p 2.);
Printf.printf "p(2)=%f\n" (gorner p 0.)
```

Это выглядит неплохо, особенно если учесть, что в предыдущем варианте создавалось три разных массива, а в этом — только один. Видимо, для императивных языков это наилучшее решение. Однако оно обладает и рядом недостатков (например, идентификатор `p` попал в глобальную область видимости и, следовательно, значение именуемого им массива стало доступно для непредусмотренного изменения.) В Objective Caml лучше написать так:

```
let p = gorner [|1.; 2.; 3.|]

Printf.printf "p(1)=%f\n" (p 1.);
Printf.printf "p(2)=%f\n" (p 2.);
Printf.printf "p(2)=%f\n" (p 0.)
```

Таким образом, функцию от нескольких аргументов можно применить к неполному их списку. Такую конструкцию называют частичным применением. В частности, возможно применение функции к пустому списку аргументов, которое возвращает саму эту функцию (видимо, это единственный случай частичного применения в C[1]). В нетривиальных же случаях частичное применение возвращает функцию, которая получает остаточный список параметров и выдает то, что выдавала исходная функция.

Естественно, частичное применение может быть подвергнуто частичному применению. Например:

```
let s3 a b c = a+b+c  (* функция сложения трех целых             *)
let s2       = s3 1   (* функция сложения двух целых с единицей *)
let s1       = s2 2   (* функция сложения одного целого с       *)
                      (* единицей и двойкой                     *)
let s1'      = s3 1 2 (* она же                                 *)
let s0       = s1 3   (* сумма единицы, двойки и тройки.        *)
```

[1] В C++ слабым аналогом частичного применения может считаться конструкция указания значений параметров по умолчанию.

(знатоки сразу отметят, что апостроф может входить в состав идентификатора (но не начинать его) и что это очень удобно.)

Как и во всех предыдущих примерах, никакого указания типов здесь не требуется — окончательный тип будет выведен из контекста. Разбирая последний пример, можно заметить, что для сложения трех чисел достаточно трехкратного применения функции к одному аргументу, а именно:

$$\texttt{s0 = (s1 3) = ((s2 2) 3) = (((s3 1) 2) 3)} \qquad (1)$$

Это приводит к мысли, что можно трактовать все функции как функции одного аргумента. Например, функция `s3` может считаться

функцией от целого (`s3`), возвращающей
функцию от целого (`s2`), возвращающую
функцию от целого (`s1`), возвращающую целое.

Полный вызов такой функции будет выглядеть как раз как (1), а последовательность функций, приводящая в конечном итоге к результату, будет последовательностью применений частичных применений исходной функции к первому, второму и так далее исходному аргументу. Такая трактовка функций от нескольких аргументов, равно как и формальное преобразование, подобное (1), называется *каррингом.* В Objective Caml карринг есть, поэтому с формальной точки зрения именно (1) является каноническим способом записи вызова.

Теперь становится понятно, почему фактические параметры функции в Objective Caml не окружаются скобками и не разделяются запятыми — параметр всегда (подчеркиваем, всегда) ровно один, и в этом нет необходимости. Несколько выписанных подряд выражений всегда группируются слева направо (т.е. $e_1e_2\ldots e_n = (\ldots(e_1e_2)e_3)\ldots e_n)\ldots))$ и трактуются как последовательность применения частичных применений функции, являющейся значением выражения e_1, к значениям выражений $e_2,\ldots,e_n$. Отсюда вытекает необходимость поставить `;;` между определением функции `gorner` и ее использованием — в противном случае выражение `!px` трактуется как функция, применяемая к `Printf.printf` …. А как же функции без параметров? Функций без параметров не бывает. Система типов Objective Caml обладает типом `unit`, которому аналогичен тип `void` императивных языков. Тип `unit` обладает значением, которое порождается конструктором `()` (в Алголе-68 для этого есть специальная основа – `empty`). Таким образом,

```
let x     = a (* x есть a                         *)
let x' () = a (* x' есть функция,                 *)
              (* получающая unit и возвращающая a *)
```

Обращение `x` даст значение `a`, обращение `x'` даст функцию, возвращающую `a`. Чтобы получить `a` через `x'`, нужно `x'` передать значение типа unit: x' (). Еще одним следствием того, что unit обладает значением, является возможность построения сложных типов с его участием, например unit array и т.д.

Итак, теперь наш основной пример будет выглядеть так:

```
let gorner p x =
  let n = (Array.length p) - 1 in
    let rec inner px i =
      if i > n then px
      else inner (px *. x +. p.(n-i)) (i+1)
    in
    inner 0. 0
;;

let p = gorner [|1.; 2.; 3.|]

Printf.printf "p(1)=%f\n" (p 1.);
Printf.printf "p(2)=%f\n" (p 2.);
Printf.printf "p(0)=%f\n" (p 0.)
```

Теперь заметим, что идентификатор gorner использован лишь однажды. Это значит, что без него вполне можно обойтись [2]. Сперва внесем определение gorner внутрь определения p:

```
let p =
  let gorner p x =
    let n = (Array.length p) - 1 in
      let rec inner px i =
        if i > n then px
        else inner (px *. x +. p.(n-i)) (i+1)
      in
    inner 0. 0
  in gorner [|1.; 2.; 3.|]
```

а затем применим функцию gorner к аргументу без связывания ее с каким–либо именем:

```
let p =
  (fun p x ->
    let n = (Array.length p) - 1 in
      let rec inner px i =
        if i > n then px
```

[2] Более точно: всегда можно обойтись без идентификатора *нерекурсивной* функции — в противном случае она должна быть связана с каким–либо именем для обращения к ней внутри самой себя. Это свойство не является принципиальным. Так, можно было бы ввести специальную синтаксическую конструкцию, обозначающую вызов функции, созданной с помощью ближайшего объемлющего использования fun.

```
        else inner (px *. x +. p.(n-i)) (i+1)
      in
    inner 0. 0
  ) [|1.; 2.; 3.|]
```

(конструкция `fun аргументы -> выражение` задает функцию, не связанную с каким-либо именем. На самом деле `let f a b ... c = e` является сокращенной формой для `let f = fun a b ... c -> e`.)

Последний пример демонстрирует функциональность Objective Caml в полном смысле этого слова: `p` возвращает частичное применение своей *локальной* функции. Более того, эта локальная функция может быть частично применена не только к локальному данному объемлющей функции, но и к ее параметру:

```
let p coeffs =
  (fun p x ->
    let n = (Array.length p) - 1 in
      let rec inner px i =
        if i > n then px
        else inner (px *. x +. p.(n-i)) (i+1)
      in
    inner 0. 0
  ) coeffs
```

(тип этой функции также будет выведен из контекста: тип `coeffs` есть тип `p`, который есть массив чего-то (например, из-за использования `p` в качестве параметра `Array.length`), и это что-то есть `float` из-за использования в операции `+.`)

Таким образом, удалось написать функцию, которая по набору коэффициентов возвращает функцию, вычисляющую значение многочлена с данными коэффициентами.

4 Замыкание и функции высших порядков

Как мы видели выше, частичное применение в некотором смысле есть конкретизация исходной функции для значений некоторых ее параметров. Таким образом, эти "конкретизирующие" параметры имеют более широкую область действия, нежели обычные параметры в императивном смысле (их область действия была бы ограничена вызовом функции, в данном случае — частичным применением.) Каким образом можно "продлить" жизнь таким значениям в императивных языках?

Вернемся к нашему примеру с суммой трех чисел. На языке C функция `s3` из предыдущего раздела может выглядеть так:

```
int s3 (int a, int b, int c)
{
  return a+b+c;
}
```

Нет никакой возможности применить ее к одному параметру, и дело тут не в синтаксисе и правилах написания вызовов, а в содержательном толковании функции как таковой. Частичное применение, следовательно, должно быть выражено явно в виде другой функции:

```
int s2 (int a, int b)
{
  return s3 (1, a, b);
}
```

Однако попытка смоделировать таким образом последний пример предыдущего раздела приведет к фиаско. Действительно, функция

```
int (* (s2 (int a))) (int, int)
{
  int _s2 (int b, int c)
  {
    return s3 (a, b, c);
  }

  return _s2;
}
```

(если даже в C допускалась бы вложенность) семантически некорректна, поскольку использование во вложенной функции параметра объемлющей произойдет вне области действия последнего. Так как возвращаемые функции могут иметь произвольную вложенность, локальные данные всех функций (включая их параметры) должны иметь глобальную область действия.

Далее, ход вычислений вложенной функции зависит от значений локальных данных объемлющих (в частности, от глобальных данных) и от значений собственных параметров. Эта совокупность значений целиком определяет исполнение функции. Параметры передаются в явном виде при вызове; что же касается остального, то оно образует некий контекст функции, который можно связать с нею при создании. Таким образом, функцию `f` можно представить как пару

```
struct {t (* entry) (..., context); context c;} f;
```

а ее вызов трансформировать в

```
f.entry (p1, ..., pn, f.c)
```

(здесь `t` — тип результата, `context` — тип контекста функции, $p_1, \ldots, p_n$ — фактические параметры вызова.)

Возвращаясь к примеру с суммами, теперь мы можем написать:

```
typedef struct {int (* entry) (int, int, int); int c} function;

int call (function x, int a, int b)
{
  return x.entry (a, b, x.c);
}

int s2_entry (int a, int b, int c)
{
  return a+b+c;
}

function s2 (int a)
{
  function _s2;

  _s2.entry = s2_entry;
  _s2.c     = a;

  return _s2;
}
```

Теперь вызовы `s2` с различными значениями параметра будут приводить к возврату различных экземпляров функции `_s2` — разница будет заключаться в хранимом контексте. Влияние этого контекста учитывается путем использования дополнительного параметра при вызове функции. Наконец, можно заметить, что необходимый контекст для любой функции определяется статически в процессе идентификации. Этот контекст называется *замыканием*.

Замыкание есть обобщение многих хорошо знакомых императивных конструкций. Например, частным случаем замыкания является организация доступа ко внешним средам из вложенной функции с использованием статической цепочки. В качестве более содержательного примера использования замыкания рассмотрим функциональную реализацию объектности.

Объектом, как это принято считать с некоторого времени, является совокупность совместно инкапсулированных операций и данных — функция–член помимо своих непосредственных параметров имеет доступ к внутренности объекта. Используем аналогичное свойство замыкания для реализации объектности:

```
type
```

```
point =
{
  getx: unit -> int;
  gety: unit -> int;

  setx: int -> point;
  sety: int -> point;
}
```

(выражение `тип1 -> тип2` обозначает тип функции с параметром типа `тип1`, возвращающей значение типа `тип2`. Таким образом, `point` — это тип структуры из четырех функций.)

Это описание содержит "операционную" часть объекта — определены функции-члены (`getx`, `gety`, `setx`, `sety`). Функции, которые изменяют содержимое объекта, в функциональной реализации фактически выполняют роль дополнительных конструкторов. Остается написать основной конструктор:

```
let rec create (x, y) =
  {
    getx = (fun () -> x);
    gety = (fun () -> y);
    setx = (fun x -> create (x, y));
    sety = (fun y -> create (x, y));
  }
```

(поскольку точка с запятой является операцией, скобки вокруг `fun`–выражения необходимы.)

Определение этого конструктора содержит вторую часть обычного описания объекта — именно, определение атрибутов и собственно тел функций–членов. Как видно, функции, меняющие данные объекта, действительно свелись к вызову конструктора. В силу этого основной конструктор формально оказался рекурсивным. Видно (быть может после нескольких минут размышления) что данные описания вполне реализуют задуманный объект. Действительно, применение функции `create` к паре параметров приведет к созданию структуры с нужным набором операций. Параметры `create` попадут в ее замыкание, и, в силу использования, в замыкания нужных функций–членов.

Использование объекта, созданного таким образом, выглядит вполне традиционно:

```
let p = create (1, 2) in
let x = p.getx () in
let y = p.gety () in
let p = p.setx 10 in
let p = p.sety 20 и т.д.
```

Нетипичным является особенность основного конструктора (который формально оказался “вне” объекта) и “функциональный” вид функций–членов, модифицирующих значение объекта. Последнее исправить просто (достаточно в структуру поместить два `mutable`–поля `x` и `y` и далее использовать присваивания), но вряд ли нужно.

Наконец, наличие замыкания дает возможность реализовывать функции высших порядков, что легко позволяет выразить многие знакомые конструкции. Например, функция

```
let o f g = fun x -> f (g x)
```

возвращает функцию, которая есть композиция параметров.

Этот пример наводит на размышления. Действительно, каковы в данном случае типы параметров и результата функции `o`? Во всех предыдущих примерах тело функции позволяло однозначно определить эти типы, однако в данном случае зацепиться решительно не за что. Таким образом, мы подошли к одной из наиболее полезных черт Objective Caml.

5 Полиморфизм

Рассмотрим подробнее последний пример. Действительно, определить точный тип параметров и результата функции `o` не представляется возможным. Однако, исходя из синтаксического устройства ее тела, можно сделать следующие выводы:

1. Применение функции `o` к своим аргументам возвращает функцию (так как ее тело есть конструкция `fun`);

2. Аргументы функции `o` есть функции (так как в теле `o` встречаются их применения);

3. Тип результата `g` есть тип аргумента `f` (по той же причине и в силу отсутствия приведений);

4. Тип результата функции, являющейся результатом применения `o`, есть тип результата `f` (по тем же соображениям);

5. Тип аргумента функции, являющейся результатом применения `o`, есть тип аргумента `g` (аналогично);

Иными словами, тип `o` определяется однозначно с точностью до типов аргументов и результатов функций–параметров. Если через $\alpha \to \beta$ обозначить тип функции, которая получает значение типа α и возвращает значение типа β, то перечисленные выше условия можно переформулировать таким образом: если тип `f` есть $\beta \to \gamma$, а тип `g` есть $\alpha \to \beta$, то тип `o f g` есть $\alpha \to \gamma$. Нас, однако, интересует тип самой функции `o`. Для того, чтобы правильно его выписать, вспомним о карринге — согласно ему, `o` есть функция от одного аргумента (`f`), которая возвращает функцию от одного аргумента (`g`), которая возвращает то, что надо (`o f g`). Таким образом, тип аргумента `o` есть тип `f` (то есть $\beta \to \gamma$), а тип результата — $(\alpha \to \beta) \to (\alpha \to \gamma)$. Значит, тип `o` есть

$$(\beta \to \gamma) \to (\alpha \to \beta) \to (\alpha \to \gamma)$$

В только что разобранном примере мы сопоставили описанной функции ее тип, определяемый некоторым типовым выражением. Типовые выражения являются равноправной синтаксической конструкцией Objective Caml. Идентификаторы, начинающиеся с апострофа, называются *типовыми переменными*, а сами типовые выражения определяются с точностью до переименования типовых переменных. Если в типовом выражении есть хотя бы одна типовая переменная, то этот тип называется *полиморфным*. Именно поэтому полиморфизм данного вида называется параметрическим — тип любого значения параметризован типами его подзначений.

В синтаксисе Objective Caml полиморфный тип `o` можно представить выражением `('b -> 'c) -> ('a -> 'b) -> ('a -> 'c)`. Как следует из принципа карринга, операция -> правоассоциативна — таким образом, скобки вокруг крайне правого члена этого типового выражения излишни.

Полученное типовое выражение для суперпозиции представляется довольно громоздким, однако, как уже было заявлено ранее, в Objective Caml явное выписывание типа *не требуется*. Действительно, те рассуждения, которые мы только что провели, допускают автоматическую реализацию в виде решения задачи *вывода типа*. Неформально говоря, можно попытаться вывести тип выражения, исходя из того, какие его свойства требуются в тех контекстах, в которых оно употребляется в исходной программе. Продемонстрируем это еще одним примером — выведем тип `f`, заданного следующим образом:

```
let f () = (fun x y -> x (y ()))(fun z -> fun () -> z)
```

Для удобства перепишем это определение так, чтобы в явном виде отобразить его деревянную структуру:

```
let f () =         (* 1 *)
  (
   fun x y ->      (* 2 *)
     x (           (* 3 *)
     y             (* 4 *)
       ()          (* 5 *)
     )
  )                (* 6 *)
  (fun z ->        (* 7 *)
     fun () ->     (* 8 *)
       z           (* 9 *)
  )
```

(строчка 6 — это применение выражения, начинающего в строке 2 к выражению, начинающемуся в строке 7.)

Процесс вывода типа будет состоять из двух фаз: во-первых, мы будем ассоциировать с каждым именем тот тип, который требуется в данном контексте, во-вторых, мы будем вычислять тип выражения в контексте, определяемом типами имен, входящих в это выражение. Далее мы будем использовать стандартную для Objective Caml запись `expr : type`, которая означает, что выражение `expr` имеет тип `type`.

(* 9 *) Подвыражение — `z`. В данном случае у нас нет никаких соображений относительно типа `z`, поэтому мы считаем, что его тип описывается новой типовой переменной, скажем, `'a`. Таким образом `z : 'a`.

(* 8 *) Подвыражение — `fun () -> z`. Исходя из того, что синтаксически оно определяет функцию, мы заключаем, что его тип есть `'b -> 'c` (заметим, что мы уже не можем использовать типовую переменную `'a`, поскольку она занята для обозначения типа `z`). Однако это не все — мы можем заключить, что `'b = unit`, а `'c` — это тип `z`. Таким образом окончательно `(fun () -> z) : unit -> 'a`.

(* 7 *) Подвыражение — `fun z -> fun () -> z`. Оно снова имеет тип функции, но в данном случае тип параметра есть тип `z` (то есть `'a`), а тип результата — тип подвыражения, рассмотренного на предыдущем шаге. Таким образом, `(fun z -> fun () -> z) : 'a -> unit -> 'a`.

(* 5 *) Тут все просто — как уже говорилось, `()` имеет тип `unit`.

(* 4 *) Выражение `y ()` — это применение. Следовательно, `y : unit -> ‘c`. Таким образом, `y () : 'c`.

(* 3 *) Это тоже применение, причем тип параметра (по предыдущему пункту) есть `'c`, а про тип результата пока ничего не известно. Таким образом, `x : 'c -> 'd`, откуда следует, что `x (y ()) : 'd`.

(* 2 *) Чтобы разобраться с этим выражением, необходимо снова воспользоваться каррингом, то есть представить его в виде

```
fun x ->          (* 2a *)
  fun y ->        (* 2b *)
    x (           (* 3  *)
    y             (* 4  *)
      ()          (* 5  *)
    )
```

Что касается определения 2b, то его тип (в соответствии с аналогичными рассуждениями) есть

```
(unit -> 'c) -> 'd
```

Тип выражения `2a`, таким образом, есть

```
('c -> 'd) -> (unit -> 'c) -> 'd
```

(* 6 *) Это выражение есть применение функции, имеющей тип

```
('c -> 'd) -> (unit -> 'c) -> 'd
```

к выражению, имеющему тип

```
'a -> unit -> 'a
```

Применение функции возможно только в случае, если тип передаваемого ей фактического параметра в точности совпадает с типом формального. Иными словами, требуется проверить, верно ли, что *полиморфный* тип формального параметра

```
'c -> 'd
```

совпадает с типом фактического параметра

```
'a -> unit -> 'a
```

Для этого надо вспомнить, что, во-первых, символ `->` в типовом выражении правоассоциативен, и во-вторых, что вместо типовой переменной может быть (согласованно) подставлено любое типовое выражение. Иными словами, требуется найти подстановку, которая переводит типовое выражение

```
'c -> 'd
```

в типовое выражение

```
'a -> unit -> 'a
```

(формально говоря, речь идет об отыскании *наибольшего общего унификатора* двух термов, то есть такой согласованной подстановки других термов вместо переменных, которая данные термы переводит в графически равные, при этом любая другая обладающая этим свойством подстановка является частным случаем данной.)

Таковой подстановкой является подстановка вида

`'a` $\to$ `'c`,

`(unit -> 'a)` $\to$ `'d`

Поскольку подходящая подстановка нашлась, рассматриваемое применение корректно. Осталось вычислить тип результата применения, который получится, если мы к выражению, описывающему тип результата функции, применим указанную выше подстановку. Таким образом, окончательно тип выражения 6 есть

```
(unit -> 'a) -> (unit -> 'a)
```

Отметим, что вывод типа для применения *не затрагивает* типа самой функции. Иными словами, тип функции 2 не изменился после вывода типа для ее применения, а поиск подстановки был произведен только для проверки корректности этого применения и вывода типа его результата.

(* 1 *) Наконец, мы можем выписать и тип `f` — это

```
unit -> (unit -> 'a) -> (unit -> 'a)
```

Таким образом, система типов Objective Caml позволяет автоматически определять тип объектов программы исходя из того, какие свойства этих объектов требуются в различных контекстах. Если же желательные свойства объекта в разных контекстах *противоречат* друг другу, то это значит, что ему не соответствует никакой тип, и, следовательно, с точки зрения языка имеет место семантическая ошибка. Видно, что вывод и проверка типов происходят при компиляции программы, то есть контроль типов — статический.

В некотором смысле описанный подход, основанный на выводе типов, является противоположным по отношению к традиционным системам типов императивных языков, в которых программисту при описании типов и переменных предлагается *заранее* запастись всеми необходимыми свойствами. В качестве демонстрации этого принципиального различия рассмотрим еще один пример: пусть нам нужно написать функцию, которая применяет функцию, полученную первым аргументом, к параметру, полученному вторым аргументом. Текст на Objective Caml выглядит очевидным образом:

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

Умозрительно прокрутив процедуру вывода типов, подобно тому, как это было сделано выше, получаем следующие типы:

- тип `f` есть `'a -> 'b`
- тип `x` есть `'a`
- тип `apply` есть `('a -> 'b) -> 'a -> 'b`

Такие типы действительно вполне отражают тот минимальный набор свойств, который необходим для того, чтобы функция `apply` имела смысл.

Написать на традиционом императивном языке текст такой же степени общности не удастся. Требуемое поведение может быть эмулировано использованием объектно–ориентированных конструкций:

```
class Parameter {...};
class Result    {...};
class Function  {... public: Result Invoke (Parameter &);...};

Result apply (Function & f, Parameter & x)
{
  return f.Invoke (x);
}
```

"Элегантность" подобного решения непредвзятый читатель оценит самостоятельно.

С точки зрения реализации полиморфизм описанного типа не предполагает никакой особенной хитрости — полиморфные значения предсталяются просто указателями в область памяти. В примере выше функция `apply` "знает", что ее первый параметр — функция, а второй — ссылка на некоторое значение, тип которого совпадает с типом параметра этой функции. Больше нечего `apply` знать не нужно — возможность вызова обеспечивается исключительно статическим типовым контролем. Вызвав свой первый параметр и передав ей (по сылке) свой второй параметр, `apply` получит указатель на результат. Поскольку больше ничего с ним делать не надо, этот же указатель и будет возвращен в качестве ответа. Как хорошо видно, платой за полиморфизм является косвенность.[3]

6 Свертка циклов и fold, iter, map

Теперь, обладая таким инструментом, как полиморфизм, мы можем окончательно разобраться с нашим основным примером. Объектом пристального внимания на сей раз станет конструкция цикла, от которой мы избавились введением рекурсивной функции `inner` (напомним, что такое преобразование потребовалось для исключения из программы присваиваний). Поскольку без конструкций повторения не обойтись, понятно, что функций, подобных `inner`, придется написать множество, при этом их устройство будет во многом одинаковым. А раз так, возникает законный вопрос — нельзя ли как-нибудь избавиться от утомительного писания одинаковых функций

[3] При аккуратной реализации этой косвенности удается избежать во многих случаях, когда в вычисления не вовлекаются полиморфные значения. Так, функция `let incr x = x+1` в Objective Caml получит свой параметр по значению и вернет результат также по значению.

каждый раз, когда в программе появляется цикл? Конечно, если этого было бы сделать нельзя, данный параграф не появился бы на свет, а потому займемся сворачиваним циклов.

Рассмотрим для затравки обычный цикл с возрастающим счетчиком, то есть конструкцию вида `for (i=l; i<u; i++) S`, где `S` — группа операторов, не имеющих побочных эффектов (в частности, не меняющая `i`), считая для простоты, что в ситуации, когда `u<l`, смысл этой конструкции не определен.

Основную идею сворачивания такого цикла в рекурсивную функцию можно выразить следующим образом:

```
let rec inner i =
  if i<u
    then begin
      S;
      inner (i+1)
    end
    else ()
```

Видно, что данная функция возвращает `unit` и что удалось избавиться от присваивания счетчику цикла. Однако пока что `inner` придется каждый раз выписывать заново при каждом `S`. Для того, чтобы этого избежать, превратим `S` в *функцию* от `i` и всех переменных, от которых `S` зависит. Если рассуждать в терминах состояния памяти, которое изменяет `S`, то становится ясно, что при исполнении на некоторой итерации `S` воспринимает собственные результаты, полученные на предыдущей итерации. Иными словами, если считать `S` функцией, то ее вызовы, порождаемые циклом, устроены так:

```
S (u-1) (S (u-2) (S (u-3) ....... (S l init))...)
```

где `init` — начальное состояние памяти. Отсюда следует, что функция, которая заменит собой `S`, должна получить счетчик цикла, все значения, которые ей нужны, а возвращать — модифицированный набор этих значений. При таких условиях `inner` будет выглядеть так:

```
let rec inner i c =
  if i<u
    then begin
      inner (i+1) (s i c)
    end
    else c
```

Здесь `s` — функция, соответствующая телу цикла, `c` — совокупность всех значений, которые были нужны для исполнения `S`. Теперь `inner` возвращает нетривиальный результат, а именно — состояние после последней итерации. Напишем теперь окончательную функцию, которая будет заменять *любой* цикл рассматриваемого вида:

```
let for’ l u s c =
  let rec inner i c =
    if i = u
      then c
      else inner (i+1) (s i c)
  in inner l c
```

Возможность написания такой функции обеспечивается именно наличием полиморфизма: ее тип есть

```
int -> int -> (int -> ’a -> ’a) -> ’a
```

что и означает, что в качестве тела может быть передана любая функция, лишь бы тип ее второго параметра совпадал с типом начального контекста цикла. В качестве иллюстрации посчитаем сумму чисел от 0 до 9-ти. Императивная программа может выглядеть так:

```
s := 0;
for (i=0; i<10; i++) s += i;
```

С помощью функции `for’` это же самое можно записать следующим образом:

```
for’ 0 10 (fun i c -> c+i) 0
```

Здесь тело цикла `s += i` превратилось в функцию, которая получает счетчик и частичный результат и возвращает их сумму.

Понятно, что функцию, подобную `for’`, можно написать для каждой разновидности циклов. Таким образом, использование собственно циклических конструкций становится излишним и присваивния целиком уходят из программ.

Применим теперь полученный опыт к нашему основному примеру. Использование функции `for’` даст следующий результат:

```
let p coeffs =
  (fun p x ->
    let n = (Array.length p) - 1 in
    for’ 0 n (fun i px -> px *. x +. p.(n-i-1)) 0.
  ) coeffs
```

Теперь избавимся от частичного применения, подставив `coeffs` внутрь fun–выражения:

```
let p coeffs =
  (fun x ->
    let n = (Array.length coeffs) - 1 in
    for’ 0 n (fun i px -> px *. x +. coeffs.(n-i-1)) 0.
  )
```

Это почти хорошо, хотя функция тела может быть еще немного упрощена. Дело в том, что этой функции по существу не нужен счетчик цикла, ей нужен только элемент масива, который индексируется выражением с участием счетчика. Обозначив функцию тела через `s`, можно выписать вычисляемое выражение явно:

```
(s coeffs.(0) ... (s coeffs.(n-2) (s coeffs.(n-1) 0.)...)
```

Собственно говоря, это и есть схема Горнера. Понятно, что ситуация, в которой некоторое преобразование применяется к элементам массива, группируемых справа налево (или слева направо), будет встречаться достаточно часто, поэтому такую итеративную конструкцию имеет смысл также записать с помощью полиморфной рекурсивной функции наряду с обычными функциями для циклов:

```
let fold_right f a x =
  let n = Array.length a in
  for' 0 n (fun i c -> f a.(n-i-1) c) x
```

Здесь доступ к элементам массива "зашит" внутри функции `fold_right`, а основной пример запишется теперь так:

```
let p coeffs =
   fun x -> fold_right (fun c px -> px *. x +. c) coeffs 0.
```

Функция `fold_rigth`, как и большое количество других полезных функций, входит в состав стандартного модуля `Array` (о модулях см. ниже). Кроме нее, легко вообразить себе функцию `fold_left`, которая применяет функцию к элементам массива, группируя их слева направо:

```
fold_left f x a ∼ f (...(f (f x a.(0)) a.(1))...) a.(n-1)
```

и большое количество других полезных функций для обработки не только массивов, но любых коллекций. Основную прелесть языку придает то, что при отсутствии нужной функции ее легко можно написать.

Итак, в нынешней своей форме основной пример выглядит так:

```
let p coeffs =
   fun x -> fold_right (fun c px -> px *. x +. c) coeffs 0.
in
let p' = p [|1.; 2.; 3.|]
in
Printf.printf "p(1)=%f\n" (p' 1.);
Printf.printf "p(2)=%f\n" (p' 2.);
Printf.printf "p(0)=%f\n" (p' 0.)
```

Заметим теперь, что последние три строчки делают однотипную работу — это хороший повод, чтобы их подсократить:

```
Array.iter
 (fun x -> Printf.printf "p(%f)=%f\n" x (p' x))
 [|1.; 2.; 0.|]
```

Здесь мы завернули аргументы `p'` в массив и натравили на него функцию `iter`, которая применяет полученную первым аргументом функцию поочередно к элементам массива, полученного вторым аргументом. Хотя `iter` входит в состав модуля `Array`, ее, конечно, можно написать самим:

```
let iter f a =
  for' 0 (Array.length a) (fun i c -> f a.(i)) ()
```

Поскольку `f` не использует ничего, кроме очередного элемента массива, тип состояния в итерации есть `unit`.

В полученном тексте основного примера и `p`, и `p'` использованы по одному разу. Это значит, что можно подставить их определения вместо использования, получая сперва

```
let p coeffs =
   fun x -> fold_right (fun c px -> px *. x +. c) coeffs 0.
in
Array.iter
 (fun x -> Printf.printf "p(%f)=%f\n" x (p [|1.; 2.; 3.|] x))
 [|1.; 2.; 0.|]
```

а затем

```
Array.iter
 (fun x ->
    Printf.printf "p(%f)=%f\n"
      x
     (Array.fold_right (fun c px -> px *. x +. c) [|1.; 2.; 3.|] 0.)
 )
 [|1.; 2.; 0.|]
```

Это и есть окончательный, полностью функциональный, несокращаемый текст основного примера. В отличие от примера на C, всю программу удалось записать в виде одного выражения.

Предположим теперь, что значения многочлена в данных точках могут понадобиться для каких–нибудь дальнейших вычислений. Тогда вместо функции `Array.iter` имеет смысл использовать функцию `Array.map`, тип которой есть

```
('a -> 'b) -> 'a array -> 'b array
```

Применение `Array.map f a` возвращает массив

```
[|f a.(0); f a.(1); ...; f a.((Array.length a)-1)|]
```

то есть конструирует массив, содержащий результаты применения функции f к соответствующим элементам исходного массива. Программа, превращающая массив точек в массив значений многочлена в этих точках выглядит так:

```
Array.map
 (fun x ->
     (Array.fold_right (fun c px -> px *. x +. c) [|1.; 2.; 3.|] 0.)
 )
 [|1.; 2.; 0.|]
```

Функции fold, iter и map по существу являются общепризнанным базисом примитивов для организации итеративных вычислений и обработки коллекций. Практика показывает, что их использование почти совершенно вытеснило циклические конструкции.

7 Конструкторы и сопоставление с образцом

Данный раздел целиком посвящен вопросам создания и препарирования значений. Из предыдущих разделов мы уже знакомы с конструкторами массива и структуры. Заметим, что для структурного типа необходимо описание, которое вводит в рассмотрение имена и типы его полей. Поскольку типы в Objective Caml выводятся, имена полей различных структур должны различаться — иначе трудно определить, какой тип имеет значение, полученное выборкой поля структуры[4]. Несмотря на кажущееся неудобство, мы увидим далее, что данное ограничение практически никогда не приводит к неприятностям.

Итак, для введения в рассмотрение структурного типа требуется указать имена и типы его полей. Нигде, кроме как в описании типа структуры, это сделать нельзя — таким образом, несмотря на вывод типов, описание типа является необходимой в некоторых случаях конструкцией. Рассмотрим пример:

```
type point = {x: int; y:int; z:int}
```

Это описание вводит, как легко видеть, структурный тип из трех целых полей x, y и z, которые неизменяемы, поскольку не описаны как mutable. Если описываемый тип полиморфный, то все его типовые параметры должны быть перечислены в описании перед именем:

```
type ('a, 'b) pair = {first: 'a; second: 'b}
```

Это описание вводит тип ('a, 'b) pair — структуры из двух полей произвольного типа. После такого описания возможно создание значений экземпляров этой структуры с разными типами полей:

[4] эта трудность не является принципиальной — близкий к Objective Caml язык SML не содержит такого ограничения

```
let p  = {a=3; b="abc"}
let p' = {a=(fun x -> x); b=1}
```

Здесь тип `p` — `(int, string) pair`, тип `p'` — `('a->'a, int) pair`. Результат подстановки типовых выражений вместо типовых переменных в другое типовое выражение называется *конкретизацией*. Таким образом, `(int, int) pair` — это конкретизация полиморфного типа `('a, 'b) pair`. Конкретизация, конечно, тоже может приводить к полиморфному типу, как это видно из предыдущего примера.

Теперь понятно, что массивы, с которыми мы имели дело выше, на самом деле просто значения одного из типов стандартного вступления Objective Caml, а именно — полиморфного типа `'a array`.

Структурный тип в Objective Caml обязательно должен быть связан с именем. Иными словами, нельзя написать

```
type sphere = {center: {x: int; y:int; z:int}; radius: int}
```

поскольку тип поля `center` — *безымянная* структура. Правильным будет описание

```
type sphere = {center: point; radius: int}
```

Значение типа `point` создается с помощью конструктора структуры:

```
let p = {x=3; y=2; z=0}
```

Поскольку поля `point` неизменяемы, любое изменение значения структуры выражается через создание нового экземпляра:

```
let p' = {x=p.x+3; y=p.y; z=p.z}
```

Для такого случая в языке есть специальное синтаксическое сокращение:

```
let p' = {p with x=p.x+3}
```

Здесь после `with` перечисляются только те поля, которые изменяются.

Выборку поля, как уже было продемонстрировано, можно осуществить традиционным способом, указав его имя после точки; кроме того, Objective Caml содержит дополнительную конструкцию, которая позволяет выбрать все поля структуры одновременно:

```
let {x=a; y=b; z=c} = p
```

Здесь в правой части let–определения стоит структурное значение, а в левой — *образец*, который отличается от конструктора структуры тем, что после символов "=" в нем в свою очередь находятся образцы, которые и связываются со значением соответствующих полей. Частным случаем образца является идентификатор. Таким образом, в данном примере идентификатор `a` связывается со значением `p.x`, `b` и `c` — со значениями `p.y` и `p.z` соответственно. Если нужны не все поля, можно выбрать только некоторые из них:

```
let {x=a; y=_; z=c} = p
```

Здесь образец "_" указывает на то, что значение поля `y` не требуется связать ни с каким идентификатором.

Конструкция, которая позволяет раскрыть внутреннюю структуру значения, пользуясь для этих целей образцом, называется *сопоставлением.* Понятно, что применять сопоставление с образцом можно не только к структурам, но и к другим значениям — каждому типу соответствует свое синтаксическое устройство образца, которое в целом соответствует синтаксическому устройству конструктора. Так, образец для массива выглядит как `[|`p_1`;` p_2`; ...;` p_k`|]`, где $p_1, \ldots, p_k$ — образцы для элементов. Сопоставление с образцом позволяет определить тип значения, которое ему сопоставляется. Например, образец

```
[|_; _; [|_; a|]|]
```

соответствует типу массива массивов (`'a array array`).

Образец определяет не только тип сопоставляемого значения, но и некоторые другие его свойства. Так, в приведенном примере он накладывает требования на количество элементов в массиве (он должен состоять из трех элементов, причем последний в свою очередь должен состоять из двух элементов). Это требование в общем случае можно проверить только при исполнении программы — таким образом, сопоставление с образцом является динамически исполняемой конструкцией, которая может либо успешно завершиться, либо привести к исключению — ошибке сопоставления.

Мы видим, что пара конструктор—образец определяет в некотором смысле взаимно-обратные преобразования: конструктор создает значение из его компонент, образец — разбивает значение на компоненты в соответствии с типом. Нам уже известны два вида конструкторов (структуры и масивы) и соответствующие им образцы. Кроме этого, существует еще два вида конструкторов, один из которых чрезвычайно удобен, а другой — чрезвычайно важен.

Чрезвычайно удобным конструктором является *конструктор n-ки* (а в особенности — его частный случай *конструктор пары*). Его вид — взятое в круглые скобки перечисление выражений через запятую. Например,

```
(1, "abc", fun x y -> x+y)
```

Видно, что в n–ку можно свободно объединять разнотипные значения. Типовое выражение, используемое для обозначения типа n-ки, состоит из выражений для типов ее компонент, разделенных символом "*". Тип n-ки в данном примере описывается типовым выражением

```
int * string * (int -> int -> int)
```

Образец, с которым можно сопоставить значение типа n-ки, выглядит очевидным образом:

```
(p1, p2, ..., pk)
```

где p1,...,pk — образцы для компонент.

n-ки чрезвычайно удобны для временного объединения различных значений. Например, если нужно посчитать одновременно наибольшее и наименьшее из двух значений, то это можно сделать так:

```
let maxmin x y = if x >= y then (x, y) else (y, x)
```

Здесь первый элемент созданной пары отвечает максимальному, а второй — минимальному значению. Так же просто можно поменять местами два значения, не пользуясь явно промежуточными переменными:

```
let (a, b) = (b, a)
```

Тут уже слева от знака равенства стоит образец пары — таким образом, после данного сопоставления значения, связанные с идентификаторами a и b, поменяются местами.

Предположим теперь, что нам требуется найти индекс первого элемента массива, удовлетворяющего какому–либо условию. Полиморфизм позволит сделать это для массива элементов любого типа:

```
let lookup a p =
  let n = Array.length a in
  let rec inner i =
    if i>n or (p a.(i)) then i
    else inner (i+1)
  in
  inner 1
```

Тип этой функции — `'a array -> ('a -> bool) -> int`. Единственная неприятность заключается в том, что если искомый элемент не найден, то об этом можно догадаться по тому, что возвращенный индекс указывает за границы массива. Очевидно, что правильнее было бы вернуть такую информацию в каком–нибудь более пристойном виде. Для этой цели можно воспользоваться парой:

```
let lookup a p =
  let n = Array.length a in
  let rec inner i =
    if i>n
      then (false, i)
      else if p a.(i)
             then (true, i)
             else inner (i+1)
  in
  inner 1
```

Теперь функция `lookup` возвращает пару — “признак присутствия” и сам индекс. Это лучше, но, с другой стороны, если нужного элемента все–таки нет в массиве, второй элемент возвращаемой пары придется “выбросить”. Избежать этого можно, используя объединение (union), или, в терминах Objective Caml, *алгебраический тип*:

```
type 'a option = None | Some of 'a
```

Это описание вводит в рассмотрение зависящий от одного параметра (`'a`) полиморфный тип `option`. Создать значение типа `option` можно только с помощью одного из двух конструкторов — `None` или `Some`, при этом значение, созданое с помощью `Some`, содержит внутри себя подзначение типа `'a`. Воспользуемся этим типом `option` для наших целей:

```
let lookup a p =
  let n = Array.length a in
  let rec inner i =
    if i>n
     then None
     else if p a.(i)
            then (Some i)
            else inner (i+1)
  in
  inner 1
```

(видно, что при создании значения с помощью конструктора ему надо передать значение параметра (если он есть), при этом синтаксис вызова конструктора совпадает с синтаксисом применения функции.)

Тип этой функции — `'a array -> ('a -> bool) -> int option`. Теперь если нужный элемент не найден, вернется `None`; в противном случае индекс найденного элемента будет выдан “завернутым” в значение типа `int option` с помощью конструктора `Some`. Конструкторы алгебраических типов — это и есть те чрезвычайно важные конструкторы, которые были обещаны выше. Алгебраические типы позволяют выразить огромное разнообразие сущностей. Например, если требуется массив, каждый элемент которого является целым или строкой, то это с использованием алгебраических типов записывается так:

```
type e = Str of string | Int of int
type a = e array
```

(имена конструкторов алгебраических типов должны начинаться с прописной буквы; имена всех конструкторов всех алгебраических типов в одной и той же области видимости должны различаться.)

Здесь `e` — алгебраический тип, значение которого создается либо конструктором `Str` (и тогда хранит подзначение–строку), либо конструктором

`Int` (и тогда хранит подзначение–целое), `a` — тип массива с элементами типа `e`. Заметим, что тип `a` можно и не описывать, поскольку его описание в отличие от описаний алгебраических типов и структур не определяет никаких имен.

Конструктор алгебраического типа может принимать не более одного параметра. Если хочется "завернуть" в алгебраический тип несколько значений, то можно воспользоваться типом n-ки:

```
type rational = Whole of int | Fraction of int * int
```

Чтобы создать значение типа `rational` с помощью конструктора `Fraction`, надо сперва соорудить пару, а потом вручить ее конструктору:

```
Fraction (1, 2)
```

Наконец, полезнейшим свойством алгебраических типов является их возможная *рекурсивность*. Иными словами, значение, завернутое с помощью конструктора алгебраического типа, может быть значением того же самого типа:

```
type 'a list = Nil | Cons of 'a * 'a list
```

В данном случае конструктор `Cons` связывает пару, состоящую из значения типа `'a` и значения того же типа `'a list`, который он конструирует. Списки (а это действительно обычные односвязные списки) настолько распространены в программировании, что стандартное вступление Objective Caml содержит определение такого типа. Кроме того, в целях "экономии чернил" для конструктора `Nil` введено сокращение '`[]`', а для `Cons` — '`::`' в инфиксной нотации (таким образом, '`1 :: 2 :: 3 :: []`' — список из трех целых). Кроме того, запись '`a :: b :: ... :: []`' допускает сокращение в виде '`[a; b; ...]`'.

Таким образом, с помощью алгебраических типов могут быть описаны структуры данных, имеющие неограниченный размер, но регулярную структуру. Например,

```
type 'a tree = Node of 'a * 'a list
```

(рекурсивный алгебраический тип должен обязательно содержать хоть один конструктор — в данном случае, несмотря на отсутствие альтернатив, без конструктора `Node` обойтись нельзя.)

— дерево произвольной степени, хранящее в каждом узле значение типа `'a`. Создать такое дерево можно, например, явным выписыванием конструкторов:

```
Node (
  1,
  [
```

```
      Node (
        2,
        [
           Node (4, []);
           Node (5, [])
        ]
      );
      Node (3, [])
   ]
)
```

Понятно, что описание алгебраического типа открывает возможность использования специальных образцов для сопоставления: так, для сопоставления со значением типа `option` можно пользоваться образцами `None` и `Some p`, где `p` — в свою очередь образец для сопоставления со значением, содержащимся "внутри" `Some`. Очевидно, однако, что для того, чтобы полноценно пользоваться алгебраическими типами, необходим конструкция *условного* сопоставления, смысл которой можно выразить фразой "если значение создано таким–то конструктором, то делать это, а есть таким – то это" и т.д. Например, применив функцию `lookup`, нужно проверить, вернула ли она значение, созданное конструктором `Some`, и если да, то извлечь из него значение индекса. Для этого существует специальная синтаксическая конструкция, которая так же называется сопоставлением с образцом:

```
match lookup [|1; 2; 3|] (fun i -> i mod 2 = 0) with
  | None -> Printf.printf "Нет четного элемента!\n"
  | Some i -> Printf.printf "Элемент номер %d - четный.\n" i
```

Здесь после ключевого слова `match` идет выражение, со значением которого мы хотим разобраться, а далее после ключевого слова `with` — список альтернатив, разделенных вертикальной чертой. Каждая альтернатива состоит из образца, за которым через знак '`->`' следует выражение. Смысл конструкции сопоставления очевиден — выражение после `match` вычисляется и сопоставляется последовательно с каждым образцом в левых частях альтернатив. Как только сопоставление прошло удачно, исполняется выражение в правой части этой альтернативы, и после этого выполнение сопоставления завершается. Если значение вычисленного выражения не сопоставилось ни с одним образцом альтернатив, возбуждается исключение.

С точки зрения реализации алгебраические типы представляют собой тэгированные объединения. Дополнительным достоинством является то, что в силу косвенного представления значение алгебраического типа имеет длину, определяемую не максимумом длин всех членов объединения, а исключительно свойствами того конструктора, с помощью которого оно было создано. В частности, значение типа `'a option`, созданное с помощью конструктора `None`, короче значения того же самого типа, созданного конструктором `Some`. Конструкция же `match` представляет собой просто ветвление по значению тэга.

Разумеется, конструкция сопоставления с образцом может быть использована для раскрытия значения не только алгебраического, но и вообще любого типа:

```
match x with
  | [a]       -> ...
  | [a; b]    -> ...
  | [a; b; c] -> ...
  | _         -> ...
```

Первая альтернатива данного сопоставления содержит образец, который описывает список из одного элемента, вторая — из двух, третья — из трех. Последняя альтернатива содержит образец '_', который успешно сопоставляется с любым значением. Заметим, что, поскольку альтернативы проверяются сверху–вниз, образец '_' должен всегда быть последним, иначе идущие за ним альтернативы никогда не будут проверены.

Наличие полиморфизма и сопоставления с образцом позволяет в широких пределах расширять язык, насыщая его сруктурами данных и процедурами обработки, имеющими такую же степень общности, что и предопределенные конструкции. Например, описав структуру бинарного дерева

```
type 'a bintree = Nil | Node of 'a * 'a bintree * 'a bintree
```

можно далее легко реализовать `fold`, `iter` и `map` в обычном их смысле:

```
let rec fold_left f x t =
  match t with
    | Nil -> x
    | Node (y, l, r) -> fold_left f (fold_left (f x y) l) r

let rec fold_right f x t =
  match t with
    | Nil -> x
    | Node (y, l, r) -> fold_right f (fold_right (f x y) r) l

let rec iter f t =
  match t with
    | Nil -> ()
    | Node (y, l, r) -> f y; iter f l; iter f r

let rec map f t =
  match t with
    | Nil -> Nil
    | Node (x, l, r) -> Node (f x, map f l, map f r)
```

Типы этих функций соответственно есть

```
('a -> 'b) -> 'b -> 'a bintree -> 'b
```

```
('a -> 'b) -> 'b -> 'a bintree -> 'b
('a -> union) -> 'a bintree -> union
('a -> 'b) -> 'a bintree -> 'b bintree
```

Теперь, например, сложить числа, хранимые в узлах дерева `t` с типом `int bintree` можно с помощью выражения `fold_left (fun x y -> x+y) 0 t`, а преобразовать его в дерево с типом `string bintree`, хранящее в узлах строковые представления тех же чисел — с помощью выражения `map string_of_int t` (здесь int_of_string — функция стандартного вступления, имеющая тип `int -> string` и осуществляющая преобразование целого числа в его строковое представление).

8 Обсуждение: функциональность vs. императивность

Автор далек от мысли, что функциональные языки являются наилучшим выбором в любой ситуации — трудно рекомендовать их, например, для написания драйвера устройства. Обычно говорят, что функциональная программа менее эффективна, чем программа на C или Fortran'е. Однако при этом не учитывают, что по выразительной силе функциональные языки превосходят объектно–ориентированные (C++, Java, объектные клоны Pascal), а полиморфизм реализован в той же степени общности, что и шаблоны C++, но при этом без таких накладных расходов. Поэтому, аппелируя к эффективности необъектных языков, стоит прежде всего вспомнить, что их давно не рассматривают в качестве средства для разработки больших информационных систем.

Возможная область применения функциональных языков существенно шире, чем просто упражнения в области разработки языков программирования. Действительно, если надо написать небольшую программу в служебных целях ("программу на час"), то функциональные языки оказываются вне конкуренции — их выразительная сила и достигаемая за ее счет краткость позволяет решить практически любую простую задачу мгновенно, при этом потеря эффективности не будет иметь ровно никакого значения. Например, в одном из недавних проектов для порождения тестов автору потребовалать программа, строящая декартово произведение списков (то есть превращающая списки $[a_1;\ a_2;\ \ldots;\ a_n]$ и $[b_1;\ b_2;\ \ldots;\ b_k]$ в список пар $(a_1,\ b_1);\ (a_1,\ b_2);\ \ldots;\ (a_1,\ b_k);\ (a_2,\ b_1);\ \ldots;\ (a_n,\ b_k)$. Задача решилась в две строчки:

```
let cartesian a b =
  flatten (map (fun x -> map (fun y -> x, y) b) a)
```

причем полученная функция оказалась применимой для списков из элементов любого типа. Реализация того же самого с помощью C++ (даже используя STL) заняла бы существенно больше усилий, в том числе и на отладку. Неудивительно поэтому, что функциональные языки в настоящее время активно занимают нишу прототипирования.

С другой стороны, когда размер реализуемой система переходит через некоторый порог, накладные расходы на поддержание структур данных, представленных в простом виде, начинают превышать выигрыш в эффективности, полученный первоначально за счет этого простого представления. Таким образом, и здесь дажеэлементарные соображения производительности выводят функциональные языки на хорошие позиции.

Обсудим вкратце основные отличия императивного и функционального подходов.

Отсутствие переменных характеризует основные различия между императивным и функциональным подходами в программировании (одно из кратких определений функциональности есть "программирование без переменных"). В общих чертах, переменные плохи своей расплывчатой семантикой — в императивных языках нет однозначного способа аналитически определить значение переменной в данной точке программы. Семантика имени в императивных языках есть семантика общего вида, в то время как имя в функциональной программе обычно имеет более конкретный смысл. В общем случае значение имени в функциональной программе всегда имеет четкое аналитическое представление — это выражение, с которым это имя связано. Именно поэтому `let`–определение мы назвали лишь аналогом описания — на самом деле оно ничего не описывает, а лишь вводит имя, которое (в своей области видимости) навсегда связано с выражением в правой части `let`–определения. Исключение составляют массивы и `mutable`–поля структур. Эти конструкции, фактически реализующие возможность изменения некоего значения по месту его хранения, не считаются функциональным — таким образом, Objective Caml не является *чистым* функциональным языком.

Наличие императивных черт (таких, как присваивания и переменные) ничего не добавляет к полноте языка, однако во многих случаях помогает писать программы. В общем и целом эти конструкции при умеренном употреблении весьма практичны, хотя и создают сложности при анализе, оптимизации и работе программ. Действительно, в языках со сборкой мусора присваивание становится дорогой операцией с точки зрения эффективности. Этим и объясняется выигрыш, которого добивается типовая функциональная пограмма по сравнению с программой, например, на Java.[5] — сборка мусора в функциональном окружении менее болезненна.

Другим важным (и чуть ли не более значимым в современных условиях) отличием является подход к реализации полиморфизма. Легко видеть, что организация полиморфизма с помощью наследования привносит существенные накладные расходы — во–первых, для его достижения надо написать сравнительно много кода (описать классы, наследование, виртуальные

[5]Естественно, сравнение уместно проводить только уровне интерпретируемого байт–кода, который наряду с native–генерацией присутствует практически во всякой реализации функционального языка.

функции–члены), во–вторых, пространство для маневра ограничено виртуальными функциями (только с их помощью можно достичь нужного эффекта). Образно выражаясь, объектно–ориентированный полиморфизм нужно *моделировать*, а функциональный — встроенный, а потому более элегантный и эффективный. Легко видеть, что полиморфизм объектный и полиморфизм параметрический в основе своей реализации содержат один и тот же принцип — идентичное представление значений разных типов. Интересы проектирования больших систем требуют существенной гибкости структур данных — в результате типичным стало использование, например, RTTI (Run-Time Type Information) в объектных программах и приведение с контролем типов "против" наследования. Можно без преувеличения сказать, что типовой объектно–ориентированный код ныне содержит существенно больше динамического контроля типов, чем функциональный!

Наконец, немаловажным достоинством функциональных языков является, как это не парадоксально, относительная сложность базовых идей, лежащих в основе их проектирования. Это означает, что "функциональный" программист должен иметь некоторое представление об этих идеях, и, как следствие, более высокую квалификацию. Написанный во время интервью `fold` для представленной структуры данных — хороший тест на профпригодность.

9 Заключение

В заключении автор хотел бы перечислить некоторые ресурсы, с помощью которых заинтересованный читатель сможет продолжить свое знакомство с современным функциональным программированием. Прежде всего, это ресурс языка Objective Caml (http://cristal.inria.fr/), содержащий полное описание языка, неформальное введение, ответы на часто задаваемые вопросы, список рассылки и т.д. Оттуда же можно взять и среду программирования, бесплатно распространяемую как в исходных текстах, так и в бинарном виде для нескольких платформ (Linux, Mac OS, Windows). Помимо собственно компилятора система содержит отладчик, систему для расширения синтаксиса базового языка (CamlP4 — Caml PreProcessor Pretty Printer) и множество разнообразных библиотек.

Близким к Objective Caml языком является язык SML (Standart Meta Language), который, в отличие от Objective Caml, стандартизован. Ресурс, посвященный ему, находится по адресу http://www.lfcs.informatics.ed.ac.uk/software/ML/. Интересна также реализация SML, называемая Moscow ML (http://www.dina.kvl.dk/s̃estoft/mosml.html).

Наконец, общее знакомство с функциональным программированием можно начать с ресурса, который находится по адресу http://www.cs.nott.ac.uk/g̃mh/faq.html. Здесь собраны ссылки на ресурсы, охватывающие разнообразные стороны предмета — историю, языки, принципы использования, сравнительные характеристики и т.д.

Автор убежден, что даже поверхностное знакомство с областью функционального программирования будет чрезвычайно полезно бо́льшей часть разработчиков программного обеспечения. По крайней мере тогда незнание этих средств перестанет быть решающим аргументом против их использования.