Библиотека переводных публикаций
по функциональному программированию

Даррелл
Фергюсон,
Деуго Дуайт

Паттерны использования «call with current continuation»

Darrell Ferguson
(darrel.ferguson@ottawa.com),
Deugo Dwight (deugo@scs.carleton.ca).
Call with Current Continuation Patterns. — 2001

Проект журнала «Практика функционального программирования»

Этот перевод и другие материалы проекта «Библиотечка ПФП» доступны на fprog.ru/lib.

Библиотека переводных публикаций по функциональному программированию

Перевод: Владимир Дзюба

Ревизия: 3285 (2011-03-18)

Сайт проекта: http://fprog.ru/

Паттерны использования «call with current continuation»

Даррелл Фергюсон, Деуго Дуайт
Carleton University, Ottawa

2001

Аннотация

В этой статье описывается использование продолжений (*continuations*). В начале приводится краткий обзор продолжений, далее разбираются некоторые паттерны их использования, включая реализацию сопрограмм (*coroutines*), управляемого поиска с возвратом (*backtracking*) и многозадачности. Для примеров используется язык Scheme, поскольку в нем продолжения являются полноправными объектами.[1]

1. Введение

Часто в ходе исследования какой-либо темы (например, продолжений в Scheme) при изучении некоторой статьи мы осознаем, что наших знаний на данный момент не хватает для полного понимания всего изложенного материала. Тогда мы на время откладываем эту статью и обращаемся к статьям из списка литературы или любым другим источникам за разъяснениями; затем мы возвращаемся к исходной статье. Мы можем отложить чтение исходной статьи и вернуться к ней позднее — такую же возможность в программировании предоставляют продолжения.

Продолжения не укладываются в стандартные (читай, процедурные) представления о программировании и незнакомы большинству программистов, однако они позволяют реализовать сложные языковые конструкции без изменения интерпретатора или компилятора: например, используя только продолжения, можно реализовать поддержку многозадачности, поиск с возвратом и ряд механизмов выхода из процедур.

К сожалению, концепция полноправных (*first-class*) продолжений (которые могут быть переданы как аргументы, возвращены из функции или сохранены в какой-либо структуре данных для дальнейшего использования) не реализована в большинстве языков программирования. Scheme является исключением, именно поэтому этот язык был выбран для примеров данной статьи [1].

В начале статьи мы приводим краткий обзор продолжений для тех, кто сталкивается с ними впервые. В следующем разделе описываются два важных для понимания продолжений понятия — *контекст* и *функция выхода*. Затем мы показываем, как из этих двух понятий появляется понятие продолжения. Затем мы продемонстрируем различные области применения продолжений, включая сопрограммы [4], управляемый возврат по стеку [3] [6] и многозадачность [2].

[1]Оригинал статьи: http://repository.readscheme.org/ftp/papers/PLoP2001_dferguson0_1.pdf.

1.1. Продолжения

Рассмотрение продолжений мы начнем с определений понятий «контекст выражения» и «функция выхода». Примеры взяты из [5], там же можно найти более подробную информацию об этих понятиях.

1.1.1. Контексты

В [5] *контекст* определяется как функция одной переменной, □. Контекст выражения получается в два действия: 1) Заменить рассматриваемое выражение на □, и 2) Обернуть конструкцию, получившуюся в результате замены, в лямбда-функцию одной переменной — `(lambda (□) …)`.

Например, контекст `(+ 5 6)` в выражении `(+ 3 (* 4 (+ 5 6)))` записывается как:

```
(lambda (□) (+3 (* 4 □))).
```

Можно расширить данное определение, расширив первый шаг получения контекста: мы можем вычислять выражение с □ до тех пор, пока это возможно (т. е. не требуется вычисления □)[2]. Для демонстрации этого подхода найдем контекст выражения `(* 3 4)` в следующей конструкции:

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ (* 3 4) 5) 2))
```

Для начала заменим `(* 3 4)` на □:

```
(if (zero? 5)
    (+ 3 (* 4 (+ 5 6)))
    (* (+ □ 5) 2))
```

Продолжим первый шаг: вычислим `(zero? 5)` и оставим только альтернативную ветвь условного оператора — в результате получаем `(* (+ □ 5) 2)`. Дальнейшие вычисления невозможны, поэтому переходим ко второму шагу и получаем запись контекста □:

```
(lambda (□) (* (+ □ 5) 2))
```

1.1.2. Функции выхода

Функции выхода (*escape procedures*) — это новый тип функций. Если вызывается функция выхода, то результат этого вызова объявляется результатом всего вычисления в целом; другие функции, ожидающие его результата, игнорируются.

Примером функции выхода является функция **error**: при ее вызове все вычисления, ожидающие результата, сбрасываются, и сообщение об ошибке передается напрямую пользователю.

Теперь допустим, что существует некоторая функция **escaper**, которая принимает в качестве аргумента любую функцию и возвращает аналогично определенную функцию выхода [5]. Рассмотрим пример использования функции **escaper**:

```
(+ ((escaper *) 5 2) 3)
```

[2] Речь здесь не о ленивых вычислениях, а о том, что к моменту, когда нужно вычислить `(* 3 4)`, условие `(zero? 5)` уже будет вычислено и будет взята вторая ветка if, поэтому «оставшиеся до получения результата» вычисления не включают в себя повторный пересчет `(zero? 5)`, как было бы, если бы продолжением мы объявили `(if (zero? 5) (+ 3 (* 4 (+ 5 6))) (* (+ □ 5) 2))`. Если бы, например, вместо 2 стояло `(+ 1 1)`, то это выражение сворачивать до формирования продолжения было бы не нужно, в предположении, что аргументы функций вычисляются слева направо — *прим. ред.*

Выражение `(escaper *)` возвращает функцию выхода, которая принимает на вход произвольное число аргументов и перемножает их. Таким образом, когда вызывается `((escaper *) 5 2)`, ожидающая результата вычисления функция `+` игнорируется (из-за наличия функции выхода) и в качестве результата всего вычисления возвращается `10` — результат выполнения `(* 5 2)`.

В разделе *Выход из рекурсии и возврат в нее* мы рассмотрим определение функции **escaper** с использованием продолжений.

1.1.3. Определение продолжений

call-with-current-continuation (обычно сокращенно обозначается как **call/cc**) — функция одного аргумента, который мы будем называть *получатель (receiver)*. Получатель также должен быть функцией одного аргумента, называемого *продолжение*.

call/cc формирует продолжение, определяя контекст выражения `(call/cc receiver)` и обрамляя его в функцию выхода **escaper**, описанную в предыдущем разделе. Затем полученное продолжение передается в качестве аргумента получателю.

Пример

Рассмотрим в качестве примера выражение:

```
(+ 3 (* 4 (call/cc r)))
```

Контекстом `(call/cc r)` будет функция:

```
(lambda (□) (+ 3 (* 4 □)))
```

Таким образом, исходное выражение можно раскрыть до:

```
(+ 3 (* 4 (r (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

Допустим, **r** — это `(lambda (continuation) 6)`; тогда выражение преобразуется в:

```
(+ 3 (* 4 ((lambda (continuation) 6)
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

Продолжение (и функция выхода) не используются, поэтому результат вызова:

```
((lambda (continuation) 6) (escaper (lambda (□) (+ 3 (* 4 □)))))
```

равен 6, а результат всего выражения в целом равен 27 — (3 + 4 * 6).

Но если бы за `r` мы приняли `(lambda (continuation) (continuation 6))`, то получилось бы следующее выражение:

```
(+ 3 (* 4 ((lambda (continuation) (continuation 6))
           (escaper (lambda (□) (+ 3 (* 4 □)))))))
```

В результате применения продолжения к аргументу 6 получаем:

```
((escaper (lambda (□) (+ 3 (* 4 □)))) 6)
```

Как уже говорилось, **escaper** возвращает функцию выхода, которая сбрасывает свой контекст (как следствие, `+` и `*`, ожидающие результата вызова **escaper**, игнорируются). Хотя результатом вычисления снова оказалось 27, процесс получения результата был существенно иным.

2. Паттерны

2.1. Выход из цикла

Этот паттерн демонстрирует разделение кода на управляющую конструкцию (цикл) и логику (выполняемую на каждом шаге цикла). Продолжения используются для выхода из управляющей конструкции. До начала цикла мы формируем функцию выхода при помощи **call/cc** и функции-получателя. Таким образом, тело цикла имеет доступ к функции, которая может выйти из своего контекста и прервать цикл.

2.1.1. Контекст задачи

Нашу систему можно мысленно разделить на две части: управляющая конструкция (цикл) и логика (выполняемая на каждом шаге цикла).

2.1.2. Задача

Как мы можем выйти из цикла?

2.1.3. Факторы

- Выделение циклической структуры в обобщенную функцию облегчает повторное использование кода, но затрудняет выход из цикла.
- Копирование механизма управления циклом в каждом случае и объединение его с логикой приводит к дублированию кода.
- Подход, при котором на каждой итерации возвращаемое значение сравнивается с некоторым специальным значением, означающим необходимость выхода из цикла, ограничивает множество значений, которые могут быть возвращены из функции итерации. Если понадобится изменить это специальное значение, то придется изменить и все используемые в циклах конкретные функции, реализующие логику.
- Желательно, чтобы механизм управления циклом был отделен и независим от логики.

2.1.4. Решение

Мы можем воспользоваться продолжением для сохранения контекста перед входом в цикл, а затем, когда выполнится условие выхода, то использовать это продолжение в качестве функции выхода. В начале мы создаем функцию-получатель для продолжения (которое мы будем использовать как функцию выхода). Функция-получатель вызывает функцию организации цикла, с аргументом — функцией, реализующей логику. Функцию, реализующую логику, мы объявим внутри функции-получателя, чтобы у нее был доступ к функции выхода из цикла (т. е. чтобы она находилась в пределах лексической области видимости). Тогда внутри функции, реализующей логику, при наступлении условия выхода цикла можно будет вызвать функцию выхода (продолжение), передав ей значение, которое будет возвращено из цикла.

2.1.5. Исходный код

Бесконечный цикл можно реализовать в виде функции, принимающей на вход произвольную функцию. Внутри определяется функция управления циклом, которая: 1) вызывает переданную функцию и 2) рекурсивно вызывает саму себя.

Также мы можем поместить туда код, который должен выполняться на каждом шагу цикла, перед или после функции, реализующей основную логику (в приведенном ниже примере эти части кода выделены курсивом). Таким образом, функция управления циклом выглядит так:

```
(define infinite-loop
  (lambda (procedure)
    (letrec ((loop (lambda ()
                     …code to execute before each action …
                     (procedure)
                     …code to execute after each action …
                     (loop))))
      (loop))))
```

Теперь приведем структуру функции, реализующей логику:

```
(define action-procedure
  (lambda (args)
    (let ((receiver (lambda (exit-procedure)
                      (infinite-loop
                        (lambda ()
                          (if exit-condition-met
                              (exit-procedure exit-value)
                              action-to-be-performed)))))))
      (call/cc receiver))))
```

2.1.6. Пример

Используя приведенное выше определение **infinite-loop** (не включая выделенный курсивом код) создадим функцию, считающую до **n** и выводящую каждое число на экран. Достигнув **n**, функция выходит из цикла и возвращает **n**.

```
(define count-to-n
  (lambda (n)
    (let ((receiver (lambda (exit-procedure)
                      (let ((count 0))
                        (infinite-loop
                          (lambda ()
                            (if (= count n)
                            (exit-procedure count)
                            (begin
                              (write-line "The count is: ")
                              (write-line count)
                              (set! count (+ count 1)))))))))))
      (call/cc receiver))))
```

2.1.7. Выводы

Мы разделили исходную функцию на две части: управляющую конструкцию (цикл) и логику. Использование **call/cc** позволяет выйти из бесконечного цикла. Поскольку наш механизм управления циклом отделен от логики, мы можем использовать его с разными функциями, добавляя при этом фрагменты кода, выполняющиеся перед или после каждого шага логики, без изменения самой функции, реализующей логику.

2.2. Выход из рекурсии

Этот паттерн позволяет осуществить досрочный выход из рекурсивного вычисления. Аналогично паттерну *Выход из цикла*, мы создаем функцию выхода в контексте, предшествующем началу рекурсивного вычисления, а затем используем ее, когда условие прекращения вычислений становится истинным.

2.2.1. Контекст задачи

Мы разрабатываем рекурсивную функцию. Окончательный результат вычисления может стать известен до выполнения всех шагов вычислительного процесса.

2.2.2. Задача

Как мы можем выйти из рекурсивной функции?

2.2.3. Факторы

- Возможность прекратить вычисления (и отбросить накопившийся стек оставшихся вычислений) сделает функцию чрезвычайно эффективной для некоторых специальных случаев (возможно, позволяя полностью избежать каких-то вычислений).
- Усложнение функции может привести к потере читаемости кода.
- Рекурсивное вычисление будет временно занимать место на стеке (эта проблема не относится к языкам, поддерживающим хвостовую рекурсию).

2.2.4. Решение

Мы можем воспользоваться **call/cc** с функцией-получателем и создать функцию выхода в контексте, предшествующем началу рекурсивных вычислений. Затем мы можем выполнить обычную рекурсивную функцию, используя вспомогательную функцию, определенную внутри получателя — так, чтобы она имела доступ к продолжению. Как только выполняется условие прекращения вычислений, вызывается продолжение с необходимым возвращаемым значением.

2.2.5. Исходный код

Рассмотрим структуру рекурсивной функции, использующей **call/cc** для выхода из рекурсивного вычисления. Части кода, подлежащие заполнению, выделены курсивом.

```
(define function
  (lambda (args)
    (let ((receiver
           (lambda (exit-procedure)
             (letrec ((helper-function
                       (lambda (args)
                         (cond
                           (break-condition (exit-function exit-value))
                           other cases and recursive calls
                     (helper-function args)))))
       (call/cc receiver))))
```

2.2.6. Примеры

Пусть нужно вычислить произведение непустого списка чисел: очевидно, если список содержит 0, то в результате получится 0. Следовательно, функцию **product-list** мы можем написать так:

```
(define product-list
  (lambda (nums)
    (let ((receiver
           (lambda (exit-on-zero)
             (letrec ((product
                       (lambda (nums)
                         (cond
                           ((null? nums) 1)
                           ((zero? (car nums)) (exit-on-zero 0))
                           (else (* (car nums)
                                    (product (cdr nums))))))))
               (product nums)))))
      (call/cc receiver))))
```

При вызове **(product-list '(1 2 3 0 4 5))** мы получаем:

```
==> (product '(1 2 0 3 4))
==> (* 1 (product '(2 0 3 4)))
==> (* 1 (* 2 (product '(0 3 4))))
```

В этот момент выполняется условие выхода, и мы покидаем вычисление, возвращая 0. Заметьте, что ожидающие выполнения операции умножения были отброшены.

Можно применить этот же паттерн и к глубокой (древовидной) рекурсии. Если предположить, что в списке могут содержаться не только числа, но и другие списки, то можно переписать *product-list* так:

```
(define product-list
  (lambda (nums)
    (let ((receiver
           (lambda (exit-on-zero)
             (letrec ((product
                       (lambda (nums)
                         (cond
                           ((null? nums) 1)
                           ((number? (car nums))
                               (if (zero? (car nums))
                                   (exit-on-zero 0)
                                   (* (car nums)
                                      (product (cdr nums)))))
                           (else (* (product (car nums))
                                    (product (cdr nums))))))))
               (product nums)))))
      (call/cc receiver))))
```

2.2.7. Выводы

Использование **call/cc** для выхода из рекурсивных вычислений требует незначительных изменений в коде по сравнению с исходной рекурсивной функцией. Кроме того, оно позволяет отбрасывать ненужные вычисления (в некоторых случаях позволяя полностью избежать вычислений).

2.2.8. Следствия

Язык, который поддерживает хвостовую рекурсию, позволяет создавать рекурсивные функции, не потребляющие место на стеке (функции с хвостовой рекурсией): подобные функции сохраняют промежуточные результаты вычислений. В этом случае выгода от применения данного паттерна невелика. Однако, если выполняемые на каждом этапе рекурсии вычисления затратны, а процедура не занимает много места на стеке, данный паттерн можно использовать вместо хвостовой рекурсии.

2.3. Организация цикла через продолжения

В этом разделе рассматривается организация циклов с использованием продолжений. Используя **call/cc**, мы создаем функцию выхода в контексте, предшествующем телу цикла. Затем мы можем использовать ее, чтобы выйти из текущего контекста и вернуться в начало тела цикла для новой итерации.

2.3.1. Контекст задачи

Паттерн *Выход из цикла* позволяет выйти из бесконечного цикла; в то же время возможна ситуация, при которой необходимо итеративно выполнять часть функции или вычислительного процесса (который может быть разбросан по нескольким функциям) до тех пор, пока некоторое условие не станет истинным.

2.3.2. Задача

Как организовать цикл с использованием продолжений?

2.3.3. Факторы

- Наиболее распространенными и привычными большинству программистов механизмами организации цикла являются конструкции **for** и **while**.
- Конструкции **for** или **while** должны содержаться в одной функции.
- Распределение механизма организации цикла по нескольким функциям усложняет чтение и понимание программы.
- Может оказаться проще написать программу, если вначале написать код одной итерации, а затем добавить механизм организации цикла.
- Цикл может быть добавлен в существующий код с минимальными затратами и изменениями структуры программы.

2.3.4. Решение

Мы можем организовать цикл, получив продолжение в начале фрагмента функции, соответствующего одной итерации, и сохранив это продолжение во временную переменную. Если возникнет необходимость повторить итерацию, мы просто вызовем продолжение: таким образом мы покинем текущий контекст и вернемся в начало итерации (фрагмента). Доступ к продолжению легко получить при помощи тождественной функции: `(call/cc (lambda (proc) proc))`.

Для того, чтобы цикл завершался, нужно каким-то образом изменить состояние программы — из такого, при котором выполнение цикла продолжится, в такое, при котором цикл завершится (т. е. условие продолжения цикла станет ложным). Этого можно достичь изменением переменных, определенные вне цикла, при помощи оператора присваивания `(set!)`. Другой вариант — передавать значения в качестве аргументов продолжения.

Если код, реализующий итерацию, разбросан по нескольким функциям, и мы хотим выйти из одной из этих функций и перейти к следующему шагу цикла, можно просто передавать продолжение в качестве аргумента. Внутри этих функций при необходимости перейти к началу цикла мы будем вызывать продолжение с самим собой в качестве аргумента.

2.3.5. Исходный код

Рассмотрим структуру цикла, использующего **call/cc**. Части кода, подлежащие заполнению, выделены курсивом.

```
(define partial-loop
  (lambda (args)
    …preliminary portion …
    (let ((continue (call/cc (lambda (proc) proc))))
      …loop portion …
      (if loop-condition
          (continue continue)
      …final portion … )))
```

Основная тонкость — в вызове `(continue continue)` для перехода к следующей итерации. Вспомним принцип работы **call/cc**: сначала определяется контекст вызова **call/cc**:

```
(lambda (□)
  (let ((continue □))
          ... loop portion ...
          (if loop-condition
              (continue continue)
          ... final portion ... )))
```

Таким образом, когда мы вызываем `(continue **some argument**)`, переданный аргумент становится значением **continue**. Так как нашей целью является организация цикла — то есть многократное возвращение в данную точку выполнения программы, это значение не должно изменяться. Поэтому мы передаем предыдущее значение **continue** (нашего продолжения), что позволяет нам вернуться в ту же точку.

2.3.6. Пример

Рассмотрим в качестве примера следующую задачу: на вход передается непустой список чисел, мы хотим прибавлять к каждому элементу единицу, пока первый элемент списка не станет больше или равен 100. Мы выделим создание и инициализацию продолжения в отдельную функцию `(get-continuation-with-values)` для улучшения читаемости кода.

```
(define get-continuation-with-values
  (lambda (values)
    (let ((receiver (lambda (proc) (cons proc values))))
            (call/cc receiver))))
(define loop-with-current-value
  (lambda (values)
    (let ((here-with-values (get-continuation-with-values values)))
      (let ((continue (car here-with-values))
            (current-values (cdr here-with-values)))
          (write-line current-values)
          (if (< (car current-values 100))
              (continue (cons continue
                              (map (lambda (x) (+ x 1))
                                   current-values)))
          (write-line "Done!"))))))
```

Если вызвать `(loop-with-current-value '(1 2 3))`, первым значением **here-with-values** будет список `(*continuation* 1 2 3)`, который разделяется на **continue**, равный

continuation, и **current-values**, равный `'(1 2 3)`. При первом вызове `(continue (cons continue (map (lambda (x) (+ x 1)) current-values)))` мы свяжем **here-with-values** со значением `(*continuation* 2 3 4)`. Затем **here-with-values** будет связано с (*continuation* 3 4 5). Этот процесс будет продолжаться до тех пор, пока **here-with-values** не будет связано со значением `(*continue* 100 101 102)`: в этом момент программа выведет на экран `Done!` и завершит работу.

2.3.7. Выводы

Использование продолжений позволяет организовать цикл с минимальными изменениями кода; читаемость программы также не снижается (хотя могут потребоваться комментарии, поясняющие использование продолжений для перехода к началу цикла). При разделении кода на функции данный механизм организации цикла может быть использован повторно: для этого необходимо передавать в эти функции различные продолжения.

2.4. Выход из рекурсии и возврат в нее

Этот паттерн позволяет выйти из рекурсивного процесса, сохраняя возможность вернуться в него. Мы создаем в нужной области видимости функцию досрочного выхода, сохраняющую текущее продолжение, а затем используем ее для выхода из рекурсивного процесса. Вызов сохраненного продолжения позволяет возобновить рекурсивный процесс.

2.4.1. Контекст задачи

Мы имеем дело с рекурсией (возможно, глубокой или древовидной), из которой нам необходимо выйти, сохранив при этом возможность вернуться назад и продолжить вычисления. Это позволяет отлаживать код, определяя момент времени, когда выполняются некоторые особые условия. Этот прием также позволяет изменить какие-либо локальные или глобальные переменные с целью изменения «на лету» свойств текущего вычисления. Данный паттерн подразумевает, что повторный вход в рекурсивное вычисление инициируется программистом.

2.4.2. Задача

Как можно выйти из рекурсивного вычисления, сохранив при этом возможность продолжить его с точки выхода?

2.4.3. Факторы

- Мы хотим свести к минимуму объем изменений и добавлений в коде и сохранить его читаемость. Это позволит легко удалить данный механизм, если он использовался в отладочных целях.
- Мы хотим избежать изменения интерпретатора или компилятора для реализации данного паттерна (нам не нужен полноценный отладчик).

2.4.4. Решение

Для решения мы воспользуемся функцией выхода **escaper**, описанной в [5], чтобы покинуть текущее вычисление, и **call/cc**, чтобы сохранить контекста в момент выхода — тогда вычисление затем можно будет продолжить. Также нам потребуются доступные в глобальной области видимости функции **break** и **resume-computation**. Функция **break** должна делать две вещи: 1) сохранять текущее продолжение и 2) прекращать текущее вычисление. Функция **break** может принимать на вход значение, которое будет передано продолжению при повторном вызове. Функция **resume-computation** возобновляет прерванные вычисления. Имея в наличии такие функции, приостановить программу уже очень просто — достаточно воспользоваться **break**.

Создание функции **break** начнем со второй из ее задач. Мы уже рассмотрели функции выхода в начале статьи, а реализация **escaper** приводится в [5]. Мы можем воспользоваться этой процедурой для выхода из текущего вычисления. Первая задача функции **break** решается в несколько этапов: сперва мы сохраняем текущее продолжение (используя **call/cc** с функцией-получателем). Забегая вперед, заметим, что **resume-computation** должна вызывать сохраненное **break** продолжение с переданным изначально аргументом. Таким образом, **break** должна присваивать **resume-computation** функцию `(lambda () (continuation arg))`.

2.4.5. Исходный код

Функция **break** выглядит так:

```
(define break
  (lambda (arg)
    (let ((exit-procedure
           (lambda (continuation)
             (set! resume-computation (lambda () (continuation arg)))
             (write-line "Execution paused. Try (resume-computation)")
             ((escaper (lambda () arg))))))
      (call/cc exit-procedure))))
```

Также необходимо проинициализировать **resume-computation** временной функцией-заглушкой (она будет перезаписана при первом вызове **break**). Определим ее как `(lambda () "to be initialized")`. Теперь реализуем функцию **escape**, упомянутую в разделе 1.1.2. Во-первых, нужно сохранить продолжение (назовем его **escape-thunk**) в контексте `(lambda (□) (□))`. Для этого сначала проинициализируем **escape-thunk** функцией-заглушкой.

```
(define escape-thunk (lambda () "escape-thunk Initialized"))
```

Затем создадим функцию-получатель, которая будет присваивать **escape-thunk** данное продолжение.

```
(define escape-thunk-init
  (lambda (continue)
    (set! escape-thunk continue)))
```

Теперь остается лишь создать продолжение, воспользовавшись **call/cc** с этой функцией-получателем.

```
((call/cc escape-thunk-init))
```

Тогда можно определить **escaper** так:

```
(define escaper
  (lambda (procedure)
    (lambda args
      (escape-thunk (lambda () (apply procedure args))))))
```

2.4.6. Пример

В качестве примера данного паттерна можно рассмотреть приведенную ниже функцию **flatten-list**, которая будет приостанавливаться всякий раз, когда на вход передается аргумент, не являющийся списком.

```
(define flatten-list (lambda (arg)
  (cond
    ((null? arg) '())
    ((not (list? arg)) (list (break arg)))
    (else (append (flatten-list (car arg))
                  (flatten-list (cdr arg)))))))
```

Вызов `(flatten-list '(1 2 3))` дает следующие результаты:

```
==> (flatten-list '(1 2 3))
"Execution paused. Try (resume-computation)"
;Value: 3
==> (resume-computation)
"Execution paused. Try (resume-computation)"
;Value: 2
```

```
==> (resume-computation)
"Execution paused. Try (resume-computation)"
;Value: 1
==> (resume-computation)
;Value 3: (1 2 3)
```

2.5. Сопрограммы

Сопрограммы позволяют организовать последовательную передачу управления между функциями. Используя **call/cc**, мы получаем текущий контекст функции и сохраняем его для последующих вызовов.

2.5.1. Контекст задачи

Ход работы многих программ может рассматриваться как последовательная передача управления между несколькими сущностями — по аналогии, например, с карточными играми, где каждый игрок делает свой ход и передает управление следующему по очереди.

2.5.2. Задача

Как мы можем реализовать последовательную передачу управления между несколькими сущностями?

2.5.3. Факторы

- Размещение механизма работы каждой сущности в отдельной процедуре повышает читаемость и легкость понимания программы.
- Распределение вычислительного процесса каждой сущности по нескольким небольшим функциям, организующим последовательную передачу управления, затрудняет отслеживание логики программы и внесение изменений в управляющий механизм.
- Такие механизмы, как многопоточность или семафоры, могут повлечь высокие накладные расходы или могут быть вовсе недоступны.
- Переключение между потоками, как правило, осуществляется на более низком уровне, чем логика программы, поэтому механизм, предполагающий, что управление осуществляется через потоки, ожидающие вызова, может быть сложным для реализации и для понимания.

2.5.4. Решение

Для реализации сопрограмм можно использовать продолжения. Сопрограммы позволяют прервать выполнение функции, а также возобновить прерванную функцию. Сначала создадим по сопрограмме для каждой сущности. Функция **coroutine-maker**, описанная в [5], принимает на вход функцию, реализующую поведение сущности. Эта функция (**body-procedure**) принимает на вход два аргумента: функцию **resumer**, которая используется для передачи управления следующей по очереди сущности, и некоторое начальное значение.

coroutine-maker создает внутреннюю функцию **update-continuation**, которая используется для сохранения текущего продолжения. Когда сопрограмма вызывается с каким-либо аргументом, она передает этот аргумент своему продолжению. Перед передачей управления другой сопрограмме она обновляет свое текущее продолжение.

2.5.5. Исходный код

Определение coroutine-maker из [5] выглядит так:

```
(define coroutine-maker
  (lambda (proc)
    (let ((saved-continuation '()))
```

```
    (let ((update-continuation!
           (lambda (v)
             (write-line "updating")
             (set! saved-continuation v))))
      (let ((resumer (resume-maker update-continuation!))
            (first-time #t))
        (lambda (value)
            (if first-time
                (begin
                  (set! first-time #f)
                  (proc resumer value))
                (saved-continuation value))))))))
```

Как видно, **coroutine-maker** использует вспомогательную функцию **resume-maker**. Ее реализацию также возьмем из [5]:

```
(define resume-maker
  (lambda (update-proc!)
    (lambda (next-coroutine value)
      (let ((receiver (lambda (continuation)
                        (update-proc! continuation)
                        (next-coroutine value))))
        (call-with-current-continuation receiver)))))
```

2.5.6. Пример

Простой пример сопрограмм, использующих продолжения, приведен в [5]. Более сложный механизм и ряд расширений концепции сопрограмм могут быть найдены в [4]. Используя **coroutine-maker** из [5], мы можем создать две функции — **ping** и **pong**, передающие управление друг другу. Функции **ping-procedure** и **pong-procedure** являются *функциями логики* (передаваемыми в **coroutine-maker**) для **ping** и **pong** соответственно.

```
(define ping
  (let ((ping-procedure
          (lambda (resume value)
            (write-line "Pinging 1")
            (resume pong value)
            (write-line "Pinging 2")
            (resume pong value)
            (write-line "Pinging 3")
            (resume pong value))))
    (coroutine-maker ping-procedure)))

(define pong
  (let ((pong-procedure
          (lambda (resume value)
            (write-line "Ponging 1")
            (resume ping value)
            (write-line "Ponging 2")
            (resume ping value)
            (write-line "Ponging 3")
            (resume ping value))))
    (coroutine-maker pong-procedure)))
```

Вызов **(ping 1)** дает следующие результаты:

```
==> (ping 1)
"Pinging 1"
"Ponging 1"
"Pinging 2"
"Ponging 2"
"Pinging 3"
"Ponging 3"
;Value: 1
```

2.5.7. Выводы

Использование продолжений для создания сопрограмм позволяет разместить вычислительный процесс каждой сущности в одной функции, что повышает читаемость и легкость внесения изменений. Оно позволяет достигать того же результата, что и при использовании потоков, без изменения языка или интерпретатора.

2.6. Управляемый поиск с возвратом

Этот паттерн решает задачу перемещения к предыдущей или «будущей» точке выполнения. Для каждой такой точки сохраняется продолжение, а специальные функции, называемые «ангелы» и «дьяволы», позволяют переместиться к будущей или предыдущей точке соответственно.

2.6.1. Контекст задачи

Мы достигли некоторой точки выполнения, в которой мы хотим прервать или приостановить текущие вычисления и перейти к предыдущей, более ранней точке.

2.6.2. Задача

Как создать механизм, который позволит нам вернуться к более ранней точке выполнения программы и выбрать другую ветвь алгоритма?

2.6.3. Факторы

- Нам требуется простой и надежный механизм перемещения вперед или назад, позволяющий обрабатывать достаточно сложную логику подобных перемещений.
- Изменение программы для интеграции этого механизма должно быть простым и не должно чрезмерно повышать сложность понимания уже существующего кода.
- Если система реализует поиск с возвратом через разбиение общего вычислительного процесса на рекурсивные вычисления, для возврата просто передающие управление вызывающей процедуре, то такая система может работать неэффективно, если потребуется возврат к точке, находящейся достаточно близко к началу вычислений. Кроме того, в случае сложного поиска с возвратом понять логику такой системы будет крайне трудно.

2.6.4. Решение

Мы можем реализовать управляемый поиск с возвратом, используя **call/cc** для получения функций выхода, возвращающих управление в различные контексты; соответствующие продолжения мы сохраняем в доступной глобально структуре. В нужный момент их можно будет достать из этой структуры и вызвать, вернувшись к одной из сохраненных точек вычисления.

В [3] вводятся понятия дьяволов (*devils*), ангелов (*angels*) и вех (*milestones*). Дьявол возвращает управление в контекст, где была создана последняя веха. При этом на вход продолжения дьявол передаст свой аргумент. Его значение будет использовано так, как будто оно изначально являлось результатом исходного выражения, соответствующего данной вехе, что, возможно, приведет к выбору другой ветви алгоритма. Ангел перемещает управление вперед, к последнему вызову дьявола. Опять же, переданное ангелу значение будет, в свою очередь, передано продолжению дьявола и использовано вместо значения, изначально возвращавшегося дьяволом: это позволяет перемещаться к более поздним состояниям. Наконец, вехи сохраняют текущий контекст для последующего использования дьяволами.

В качестве аналогии рассмотрим ситуацию с изучением научных статей, описанную в начале этой работы. Мы начинаем читать первую статью и достигаем места, в которой мы понимаем, что нам необходимо расширить свои знания по теме. Мы отмечаем это место вехой и обращаемся к другим материалам. Когда мы чувствуем, что узнали достаточно для продолжения чтения исходной статьи, мы возвращаемся к ней (эквивалентно вызову дьявола). Если перед этим мы разобрались не со всеми дополнительными материалами, то после прочтения основной статьи мы можем продолжить их изучение (эквивалентно вызову ангела).

Нам потребуются две структуры данных для хранения пройденных вех и точек «в будущем» (в которых происходили вызовы дьяволов). Проще всего воспользоваться двумя стеками, если для наших целей достаточно перехода к предыдущей вехе или предыдущему вызову дьявола. Применение других структур данных также возможно и часто используется в приложениях искусственного интеллекта [3].

2.6.5. Исходный код

Реализация вех, ангелов и дьяволов приведена в [3]. Функция создания вехи должна просто сохранить текущее состояние в структуре, хранящей вехи, и возвратить исходное значение:

```
(define milestone
  (lambda (x)
    (call/cc
      (lambda (k)
        (begin (push past k)
               x)))))
```

Реализация дьявола сохраняет текущее продолжение в структуре, хранящей «точки из будущего», и возвращается к последней вехе с новым значением:

```
(define devil
  (lambda (x)
    (call/cc
      (lambda (k)
        (begin (push future k)
               ((pop past) x))))))
```

Наконец, функция ангела может быть написана так:

```
(define angel
  (lambda (x)
    ((pop future) x)))
```

Для каждой из этих функций в случае, если соответствующий стек пуст, должна возвращаться тождественная функция (а дьявол или ангел должны возвращать переданный аргумент `x`).

2.6.6. Выводы

Теперь мы можем использовать эти функции для сохранения определенных опорных точек исполнения программы, для возвращения к сохраненным состояниям программы или к незаконченным вычислениям. Мы получили простой, четко обособленный и переносимый механизм, пригодный для использования в различных приложениях. Этого механизм также можно расширить, храня состояния в более сложной структуре данных, нежели стек.

2.7. Многозадачность

Многозадачность позволяет ограничивать время выполнения отдельных вычислений. Используя планировщики (*engines*) и таймеры, мы можем реализовать прерывание (и, возможно, перезапуск) процессов. Планировщики используют **call/cc** для сохранения контекста вызывающего его вычисления (чтобы можно было вернуться к нему после завершения работы планировщика и связанных с ним функций), а также для получения текущего продолжения прерываемой функции (чтобы ее можно было возобновить позднее).

2.7.1. Контекст задачи

Во многих случаях бывает эффективно выполнить несколько вычислений параллельно, даже если на самом деле они не выполняются параллельно. Также бывает, что необходимо ограничить время выполнения некоторого вычисления. Примером может служить оператор дизъюнкции логических выражений: при параллельном выполнении, если достаточно простое выражение вычисляется достаточно быстро и оказывается истинным, все прочие вычисления можно сразу же прекратить.

2.7.2. Задача

Как организовать многозадачность (включая ограничение времени вычислений, прерывание незавершенных вычислений и их повторный запуск)?

2.7.3. Факторы

- Модификации интерпретатора/компилятора должны быть сведены к минимуму.
- Использование таких механизмов, как многопоточность или семафоры, может повлечь высокие накладные расходы или быть вовсе недоступно.
- Организация планирования задач и управления состояниями на уровне языка является более гибким, менее эффективным подходом, по сравнению с использованием низкоуровневых механизмов.

2.7.4. Решение

Мы можем обернуть функцию некоторым *планировщиком* (аналогом планировщика можно считать поток). Реализация планировщика использует **call/cc** для следующих задач: 1) получение продолжения вызывающего его вычисления для возврата к нему после завершения работы планировщика и 2) получение продолжения самого планировщика в момент истечения отведенного времени для последующего возобновления вычислений [2].

Функция **make-engine** должна принимать на вход функцию (без аргументов), реализующую логику вычислений. Возвращаемый планировщик является функцией трех аргументов: 1) отведенное на вычисления время, 2) функция, которая определяет действия планировщика в случае завершения вычислений до истечения отведенного времени (*return procedure*), и 3) функция, которая определяет действия в случае истечения отведенного времени до завершения вычислений (*expire procedure*).

В данной статье не рассматривается реализация таймера. Предполагается, что используется механизм, описанный в [2], который расширяет функцию **lambda** для учета ограничений по времени (**timed-lambda**) и использует полученную функцию для определения всех асинхронных функций. В этой реализации один тик таймера соответствует одному вызову функции.

2.7.5. Пример

Мы используем макрос **extend-syntax** для создания функции **parallel-or** с использованием планировщиков (взято из [2]):

```
(extend-syntax (parallel-or)
  ((parallel-or e ...)
   (first-true (lambda () e) ...)))

(define first-true
  (lambda proc-list
    (letrec ((engines (queue))
             (run (lambda ()
                    (and (not (empty-queue? engines))
                         ((dequeue engine)
                            1
                            (lambda (v t) (or v (run)))
                            (lambda (e) (enqueue e engines) (run)))))))
      (for-each (lambda (proc) (enqueue (make-simple-engine proc) engines))
                proc-list)
      (run))))
```

2.7.6. Выводы

Использование продолжений позволяет избежать изменений интерпретатора или компилятора (хотя нам потребовалось применить макросы Scheme). Есть небольшие накладные расходы на обеспечение работы таймера, но они могут быть снижены за счет использования синхронного подхода на некоторых участках кода. Решение задачи не требует наличия в языке поддержки потоков или семафоров.

3. Заключение

В этой работе мы представили ряд применений полноправных продолжений в различных ситуациях: от нелокальных выходов (*Выход из цикла*, *Выход из рекурсии*) и реализации собственных управляющих конструкций (*Организация цикла через продолжения*, *Выход из рекурсии и возврат в нее*) до более сложных механизмов (*Сопрограммы*, *Поиск с возвратом* и *Многозадачность*). Хотя использование продолжений зачастую может сбить с толку и затруднить понимание программы (и их, как правило, следует избегать в простых программах), они предоставляют программисту ряд продвинутых инструментов, реализация которых без использования продолжений потребовала бы изменений языка, интерпретатора или компилятора (если это вообще возможно).

Список литературы

[1] *Dybvig R. K.* The Scheme Programming Language. — Prentice Hall, 1996.

[2] *Dybvig R. K., Hieb R.* Engines from continuations. — 1989. — Pp. 109–123.

[3] *Friedman D. P., haynes D. T., Kohlbecker E. E.* Programming with continuations // *Program Transformation and Programming Environments*. — 1984. — Pp. 263–274.

[4] *Haynes c. T., Friedman D. P., Wand M.* Obtaining coroutines with continuations // *Computer Languages*. — 1986. — Pp. 143–153.

[5] *Springer G., Friedman D. P.* Scheme and the Art of Programming. — MIT Press and McGraw-Hill, 1989.

[6] *Sussman G. J., Steel G. L. J.* Scheme: An interpreter for extended lambda calculus. — 1975.