Конечные автоматы в чистых функциональных языках программирования

Я. М. Малаховски, магистрант, trojan@rain.ifmo.ru,
А. А. Шалыто, профессор, д.т.н., shalyto@mail.ifmo.ru, СПБГУ ИТМО

Автоматы и Haskell

Уровень сложности: ★☆☆

В работе рассматриваются вопросы реализации на функциональных языках программирования событийных структурных конечных автоматов, используемых в автоматном программировании. На примерах показаны решения, имеющие преимущества перед реализациями на императивных языках программирования.

Требуется знание конечных автоматов, Haskell.

Ключевые слова: конечные автоматы, автоматное программирование, функциональное программирование, Haskell

Введение

Реализация конечных автоматов [1] рассматривалась на форуме и страницах журнала *RSDN* (например, [2]). Не раз освещались также вопросы, связанные с программированием на функциональных языках программирования ([3], [4]). Однако совместное использование этих подходов не рассматривалось.

В теории конечных автоматов существуют два класса [1]: абстрактные и структурные автоматы. Абстрактные автоматы обычно используются при разработке систем генерации парсеров по контекстно-свободным грамматикам и регулярным выражениям (например, [5], [6]). Функции переходов в таких задачах обычно строятся не по диаграммам состояний, а по множеству порождающих правил.

В событийных системах управления применяются структурные автоматы [1], ранее использовавшиеся в аппаратных реализациях. В общем случае такие автоматы содержат два типа входных воздействий: события и входные переменные. Неизвестны работы, в которых описано как на функциональных языках программирования реализовать автоматы указанного класса, так как для таких автоматов требуется формирование выходных воздействий, а чистые функции не позволяют реализовать их непосредственно. В настоящей работе рассматриваются автоматы, управляемые только событиями. Реализация таких автоматов в функциональном программировании обычно не рассматривается.

Основное отличие чистых функциональных программ от императивных состоит в том, что в чистых функциональных языках отсутствуют побочные эффекты, а следовательно и переменные, в том смысле, в котором этот термин используется в императивных языках. В то же время, состояние автомата в императивных программах обычно представлено полем класса (если используется объектно-ориентированное программирование) или глобальной переменной. Из изложенного следует, что в чистых функциональных языках программирования (например, *Haskell* [7]) не удается непосредственно применять методы, используемые при реализации конечных автоматов на императивных языках программирования (например, *C++* [8]).

Кроме того, в качестве идентификаторов состояний в императивных реализациях обычно используются строки и целые числа, иногда упакованные в enum, что при ручном программировании функций переходов автомата не обеспечивает приемлемого уровня проверки корректности кода.

Реализация функций переходов на императивных и функциональных языках

Рассмотрим следующий пример: пусть требуется реализовать счетный триггер, реагирующий на события. Другими словами, требуется построить конечный автомат с двумя состояниями, кодируемыми нулем и единицей, который управляется кнопкой. Каждое нажатие кнопки порождает событие e. К выходу z конечного автомата подключена лампа. Каждое событие e переводит автомат в состояние (1 - y), где y — текущее состояние. При этом переменная состояния одновременно является выходной переменной (z = y). Таким образом, каждое нажатие кнопки будет приводить то к включению лампы, то к ее выключению. Диаграмма состояний рассматриваемого триггера представлена на рисунке 1.

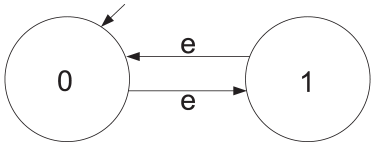

Рисунок 1. Диаграмма состояний счетного триггера.

Приведем одну из возможных реализаций этого автомата на языке C:

```
// Событие: <<Нажатие на кнопку>>.
enum event {EVENT_BUTTONCLICK};
// Состояния: <<Выключено>> и <<Включено>>.
enum state {LAMPOFF, LAMPON};

// Переменная состояния, начальное состояние --- LAMPOFF.
int current_state = LAMPOFF;
// Функция переходов.
void got_event(int event)
{
    if (event == EVENT_BUTTONCLICK)
        switch (current_state)
        {
            case LAMPOFF: current_state = LAMPON; break;
            case LAMPON: current_state = LAMPOFF; break;
        }
}

// Функция, вызываемая системой.
int main()
{
    got_event(EVENT_BUTTONCLICK);
    printf(«%i», current_state);
}
```

Заметим, что в данной реализации ничто не может уберечь от, например, проверки на равенство состояния с событием, что является ошибкой.

При реализации этого примера на языке *Haskell* в качестве идентификаторов событий и состояний возможно использование элементов алгебраических типов данных, возлагающее на компилятор проверку написанного кода:

```
-- Событие: «Нажатие на кнопку».
data Event = ButtonClick deriving Show
-- Состояния: «Выключено» и «Включено».
data State = LampOff | LampOn deriving Show

-- Функция переходов, изоморфная рисунку 1.
gotEvent :: State -> Event -> State
gotEvent LampOff ButtonClick = LampOn
gotEvent LampOn ButtonClick = LampOff

-- Функция, вызываемая системой.
-- Начальное состояние - LampOff.
main = print $ gotEvent LampOff ButtonClick
```

При использовании языка *Haskell* компилятор может не только проверить, что все элементы функции переходов корректно типизированы, но и проверить также полноту и непротиворечивость реализации функции переходов. Например, если определить лишнюю ветку в функции gotEvent, как в следующем листинге:

```
gotEvent :: State -> Event -> State
gotEvent LampOff ButtonClick = LampOn
gotEvent LampOff _ = LampOff
gotEvent LampOn ButtonClick = LampOff
```

то компилятор выдаст предупреждение.

Отметим также, что рассмотренная реализация функции переходов при помощи *pattern matching* не является единственной и вместо неё можно использовать конструкцию case, например, так:

```
gotEvent :: State -> Event -> State
gotEvent state ButtonClick = case state of
    LampOff -> LampOn
    LampOn  -> LampOff
```

Вообще говоря, это тоже pattern matching - прим.ред.

Последовательности событий

Для того чтобы добавить возможность применения последовательности событий к начальному состоянию автомата введем функцию applyEvents:

```
-- Функция, применяющая события к начальному состоянию.
applyEvents :: State -> [Event] -> State
-- Результат = состояние, если событий больше нет.
applyEvents st [] = st
-- Иначе делаем переход и
-- осуществляем рекурсивный вызов.
applyEvents st (e:es) = applyEvents (gotEvent st e) es

-- Новая функция main.
main = print $ applyEvents LampOff [ButtonClick]
```

Если бы в реализуемом примере было более одного конечного автомата, то пришлось бы писать несколько функций, аналогичных applyEvents. Поэтому можно выделить общую их часть в «библиотечный код», пригодный для написания других конечных автоматов и представляющий собой функцию высшего порядка, принимающей функцию переходов в качестве первого параметра:

```
-- Тип функции переходов.
type SwitchFunc state event = state -> event -> state

-- Функция, применяющая список событий к начальному
-- состоянию автомата при помощи функции переходов.
applyEvents :: SwitchFunc st ev -> st -> [ev] -> st
applyEvents _ st [] = st
applyEvents swF st (ev:evs) = applyEvents swF
       (swF st ev) evs
-----------------------------------------------------
-- Реализация счетного триггера
-- при помощи приведенного выше библиотечного кода.

-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show
-- Функция переходов для счетного триггера.
triggerSwF LampOff ButtonClick = LampOn
triggerSwF LampOn ButtonClick = LampOff

-- Функция, вызываемая системой.
main = print $ applyEvents triggerSwF LampOff [ButtonClick]
```

Отметим, что новая версия функции applyEvents является левой сверткой (одна из стандартных операций функционального программирования) списка событий по функции переходов. Поэтому можно заменить всё определение функции applyEvents на applyEvents = foldl.

Последовательности событий и монады

Одной из наиболее значимых встроенных конструкций языка *Haskell* являются монады. С их помощью, например, осуществляется ввод-вывод, который не нарушает функциональные основы языка. О монадах можно думать, как о способе объединения последовательности действий — в некотором смысле способ «эмулировать» императивное программирование в рамках функционального. Более подробно о монадах можно прочитать в статье [4]. Из сказанного не следует, что монады невозможно реализовать в других функциональных языках программирования, однако в *Haskell* они используются повсеместно.

Поэтому логичным развитием идеи реализации конечных автоматов на *Haskell* является попытка построить монаду, удобную в использовании.

Реализуем монаду *FSM*, предназначенную для манипуляции состояниями, а также различные вспомогательные функции и типы:

```
-- Абстрактный тип функции перехода.
type SwitchFunc state event = state -> event -> state

-- Тип монадического преобразования состояния.
newtype FSM state a = FSM ( state -> (state, a) )

-- Реализация функций класса Monad.
instance Monad (FSM state) where
```

```
    (FSM first) >>= second =
        FSM ( \s0 -> let (s1, a) = first s0
                         (FSM q) = second a in q s1 )
    return a = FSM ( \s -> (s, a) )

-- Взятие текущего состояния.
getState :: FSM state state
getState = FSM ( \was -> (was, was) )

-- Установка текущего состояния.
setState :: state -> FSM state ()
setState state = FSM ( \s -> (state, ()) )

-- Функция для применения монады.
-- Передает начальное состояние первым аргументом.
applyFSM :: s -> FSM s a -> (s, a)
applyFSM s (FSM p) = p s

-- Переход по событию.
applyEvent :: SwitchFunc state event -> event -> FSM state ()
applyEvent switchFunc event = do
    st <- getState
    setState (switchFunc st event)

-- Переход по списку событий.
applyEvents :: SwitchFunc state event ->
    [event] -> FSM state ()
applyEvents _ [] = return ()
applyEvents switchFunc (event:eventsTail) = do
    applyEvent switchFunc event
    applyEvents switchFunc eventsTail
```

Можно заметить, что монада *FSM* очень похожа на стандартную монаду *State*, однако, проигнорировав это сходство, реализуем счетный триггер, используя код из предыдущего листинга:

```
data Event = ButtonClick deriving Show
data State = LampOff | LampOn deriving Show

gotEvent :: State -> Event -> State
gotEvent state ButtonClick = case state of
    LampOff -> LampOn
    LampOn  -> LampOff

main = print $ fst $
    applyFSM LampOff $ applyEvents gotEvent
    [ButtonClick, ButtonClick]
```

Отметим, что при существенном изменении нижнего уровня абстракции (простые рекурсивные функции поменялись на монады), использование «библиотечного кода» практически не изменилось. Но если изменить функцию applyEvent, то и функция переходов будет «втянута» в монадические вычисления:

```
-- Переход по событию (альтернативная версия).
applyEvent :: SwitchFunc state event -> event -> FSM state ()
applyEvent switchFunc event = do
    st <- getState
    nst <- switchFunc st event
    setState nst

gotEvent state ButtonClick = case state of
    LampOff -> return LampOn
    LampOn  -> return LampOff
```

Однако вдумчивый читатель заметит, что это всё — переливание из пустого в порожнее.

Дело в том, что при реализации абстрактных автоматов, как правило, вообще не требуется явная функция переходов, поскольку монада для абстрактных автоматов представляет собой моноид («допустимый результат» или «недопустимое выражение»), а функция переходов строится неявно при помощи различных комбинаторов. Например, в библиотеке *Parsec* [6] проверка того, что выражение состоит из символа a или символа b, за которым следует символ c (регулярное выражение (a|b)c), может быть реализована так:

```
abctest = do
    char 'a' <|> char 'b'
    char 'c'
```

Никакого явного выделения состояний в таком коде не производится. В свою очередь, при реализации структурных автоматов программиста больше интересует *процесс* реакции на событие, нежели *конечный результат* вычисления множественного применения функции переходов, поскольку структурные автоматы предназначены для работы в реактивных системах. Поэтому неотъемлемой частью структурных автоматов являются явно выделенные состояния. Кроме того, структурным автоматам обычно не требуется механизм отката, иногда необходимый абстрактным автоматам (например, у автоматов с магазинной памятью «движение назад» осуществляется при помощи стека).

Таким образом, использование монад оправдано при реализации абстрактных автоматов, но привносит только дополнительные накладные расходы (синтаксические, и, возможно, вычислительные) при реализации структурных автоматов.

Вложенность

В предыдущих разделах было рассмотрено представление конечных автоматов, которые реагируют на внешние воздействия только сменой состояний. Однако состояние системы может содержать описание нескольких автоматов. Так можно реализовывать вложенные конечные автоматы.

В качестве задачи для реализации системы с вложенностью рассмотрим упрощенную модель цифрового устройства с четырьмя кнопками на корпусе и экраном. «Устройство» может быть включено, отображать меню, состоящее из некоторого списка элементов, и быть выключено, причем выключенное «устройство» запоминает состояние меню и при следующем входе в него отображает сохраненную позицию курсора.

Следовательно, имеется два конечных автомата («устройство» и меню), причем второй вложен в первый. Автомат, описывающий «устройство», имеет три состояния: «Выключено», «Включено» и «Меню», а автомат, описывающий меню, два: «Выбран первый пункт меню» и «Выбран второй пункт меню». Панель «устройства» содержит четыре кнопки: «Включить-выключить», «Меню», «Вверх», «Вниз». Графы переходов системы представлены на рисунке 2.

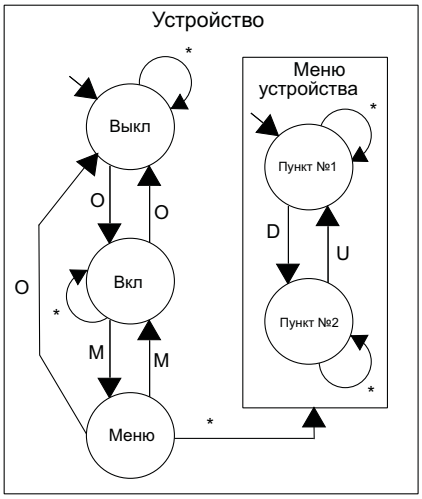

Рисунок 2. Диаграммы состояний устройства и его меню. «O», «M», «U», «D» – кнопки включения-выключения, меню, вверх и вниз соответственно.

Общую часть листингов этого раздела вынесем в отдельный листинг:

```
-- Типы событий и состояний для устройства.
data DeviceEvent = OnOffButton
                 | MenuButton
                 | UpButton
                 | DownButton deriving Show
data DeviceState = DeviceIsOff
                 | DeviceIsOn
                 | DeviceIsInMenu deriving Show

-- Типы событий и состояний для меню.
data MenuEvent = DeviceButton DeviceEvent deriving Show
data MenuState = MenuIsInPositionOne
               | MenuIsInPositionTwo deriving Show

data SystemState = SystemState
    { dstate :: DeviceState
    , mstate :: MenuState } deriving Show

applyEvents = foldl
```

Первым рассматриваемым методом реализации вложенности является реализация для всех автоматов собственных функций переходов, с той особенностью, что функции переходов внешних автоматов манипулируют состояниями всех внутренних автоматов:

```
-- Функция переходов для устройства.
deviceSwF state event = case dstate state of
    DeviceIsOff -> case event of
        OnOffButton -> state { dstate = DeviceIsOn }
        _ -> state
    DeviceIsOn -> case event of
        OnOffButton -> state { dstate = DeviceIsOff }
        MenuButton -> state { dstate = DeviceIsInMenu }
        _ -> state
    DeviceIsInMenu -> case event of
        OnOffButton -> state { dstate = DeviceIsOff }
        MenuButton -> state { dstate = DeviceIsOn }
        _ -> state { mstate =
            (menuSwF (mstate state)
                (DeviceButton event)) }

-- Функция переходов для меню.
menuSwF mstate event = case mstate of
    MenuIsInPositionOne -> case event of
        DeviceButton DownButton -> MenuIsInPositionTwo
        _ -> mstate
    MenuIsInPositionTwo -> case event of
        DeviceButton UpButton -> MenuIsInPositionOne
        _ -> mstate

-- Функция, вызываемая системой.
main = print $ applyEvents
    deviceSwF
    ( SystemState
        { dstate = DeviceIsOff
        , mstate = MenuIsInPositionOne } )
    [OnOffButton, MenuButton, DownButton, OnOffButton]
```

Во втором рассматриваемом методе устройство может посылать события, адресованные меню, через некоторый внешний канал. Следовательно, автоматы можно сделать независимыми друг от друга:

```
-- Функция переходов всей системы.
systemSwF state event =
    SystemState { dstate = ndstate, mstate = nmstate }
    where
        (ndstate, act) = deviceSwF (dstate state) event
        nmstate = case act of
            Nothing -> mstate state
            Just x -> menuSwF (mstate state) x

-- Функция переходов для устройства.
-- Возвращает кортеж (новое состояние,
-- событие, отправляемое вложенному автомату).
deviceSwF dstate event = case dstate of
    DeviceIsOff -> case event of
        OnOffButton -> (DeviceIsOn, Nothing)
        _ -> (dstate, Nothing)
    DeviceIsOn -> case event of
        OnOffButton -> (DeviceIsOff, Nothing)
        MenuButton -> (DeviceIsInMenu, Nothing)
        _ -> (dstate, Nothing)
    DeviceIsInMenu -> case event of
        OnOffButton -> (DeviceIsOff, Nothing)
        MenuButton -> (DeviceIsOn, Nothing)
        _ -> (dstate, Just $ DeviceButton event)

-- Функция переходов для меню.
menuSwF mstate event = case mstate of
    MenuIsInPositionOne -> case event of
        DeviceButton DownButton -> MenuIsInPositionTwo
        _ -> mstate
    MenuIsInPositionTwo -> case event of
        DeviceButton UpButton -> MenuIsInPositionOne
        _ -> mstate

-- Функция, вызываемая системой.
main = print $ applyEvents
    systemSwF
    ( SystemState
        { dstate = DeviceIsOff
        , mstate = MenuIsInPositionOne } )
    [OnOffButton, MenuButton, DownButton, OnOffButton]
```

Отметим, что это первый пример, реализующий выходные воздействия — посылка события автоматом устройства автомату меню. В данном случае функция переходов всей системы systemSwF реализует передачу сообщений между вложенным и внешним автоматами.

Выходные воздействия

В предыдущих примерах функции переходов были ограничены только операциями над состояниями, а выходные воздействия во внешний мир производились функцией main. На практике автоматам требуются более широкие классы выходных воздействий для того, чтобы выводить сообщения на экран, отправлять пакеты данных в сеть, обмениваться событиями и т. д.

В качестве примера реализации автомата с выходными воздействиями рассмотрим счетный триггер с четырьмя состояниями (рисунок 3).

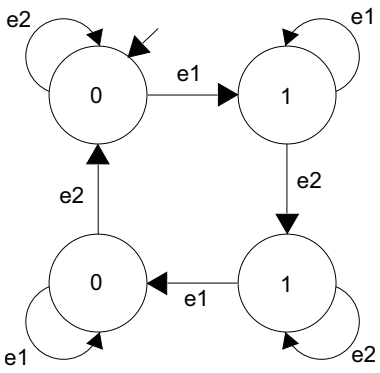

Рисунок 3.

Отличие этого триггера от предыдущего состоит в том, что лампа будет включаться или выключаться только после отпускания кнопки, а повторное нажатие или отпускание кнопки не будет производить никакого эффекта.

События и состояния для этого триггера представлены в следующем листинге:

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
```

```
                | ButtonUp deriving Show

data TriggerState = LampOffButtonUp
                | LampOffButtonDown
                | LampOnButtonUp
                | LampOnButtonDown deriving Show
```

Один из вариантов реализации выходных воздействий — изменить функцию переходов так, чтобы она возвращала не только новое состояние автомата, но и список выходных воздействий некоторого типа. Это может быть как арифметический тип данных, представляющий собой множество возможных выходных воздействий, так и тип IO(). В этом случае ответственность за выполнение выходных воздействий лежит на той части кода, которая вызывает функцию переходов автомата. Достоинством данного подхода является его гибкость.

Следующий листинг демонстрирует данный подход:

```
-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и список выходных воздействий.
type SwitchFunc state input output = state -> input
    -> (state, [output])

-- Функция applyEvents, реализованная через свертку.
applyEvents :: SwitchFunc state input output -> state
    -> [input] -> (state, [output])
applyEvents switchFunc state events =
    foldl (switchToAcc switchFunc) (state, []) events

-- Функция, «конвертирующая» функцию переходов в
-- аккумулятор выходных воздействий.
switchToAcc :: SwitchFunc state input output -> (state, [output])
    -> input -> (state, [output])
switchToAcc switchFunc (state, output) event =
    (nstate, output ++ noutput)
    where (nstate, noutput) = switchFunc state event

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case state of
    LampOffButtonUp -> case event of
        ButtonDown -> (LampOffButtonDown, [])
        _ -> (state, [])
    LampOnButtonUp -> case event of
        ButtonDown -> (LampOnButtonDown, [])
        _ -> (state, [])
    LampOffButtonDown -> case event of
        ButtonUp -> (LampOnButtonUp, [putStrLn «LampOn»])
        _ -> (state, [])
    LampOnButtonDown -> case event of
        ButtonUp -> (LampOffButtonUp, [putStrLn «LampOff»])
        _ -> (state, [])

-- Функция, вызываемая системой.
main = do
    sequence_ o
    putStrLn $ show $ s
    where
        (s, o) = applyEvents triggerSwitchFunc LampOffButtonUp
            [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
```

Проблема реализации выходных воздействий может быть решена также модификацией функции переходов. При этом в качестве возвращаемого значения она будет иметь тип IO state, где state — тип состояния автомата, а сами выходные воздействия будут выполняться непосредственно в самой функции переходов:

```
-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и осуществляющей IO.
type SwitchFunc state input output = state -> input -> IO state

-- Функция applyEvents, реализованная через IO
applyEvents :: SwitchFunc state input output -> state
    -> [input] -> IO state

applyEvents switchFunc state [] = return state
applyEvents switchFunc state (event:eventsTail) = do
    newstate <- switchFunc state event
    applyEvents switchFunc newstate eventsTail

-- Функция переходов для счетного триггера.
triggerSwitchFunc state event = case state of
    LampOffButtonUp -> case event of
        ButtonDown -> return LampOffButtonDown
        _ -> return state
    LampOnButtonUp -> case event of
        ButtonDown -> return LampOnButtonDown
        _ -> return state
    LampOffButtonDown -> case event of
        ButtonUp -> do
            putStrLn «LampOn»
            return LampOnButtonUp
        _ -> return state
    LampOnButtonDown -> case event of
        ButtonUp -> do
            putStrLn «LampOff»
            return LampOffButtonUp
        _ -> return state

-- Функция, вызываемая системой.
main = do
    result <- applyEvents triggerSwitchFunc LampOffButtonUp
        [ButtonDown, ButtonUp, ButtonDown, ButtonUp]
    putStrLn $ show $ result
```

Второй подход предоставляет больше свободы, так как в данном случае функция переходов не является чистой. Данный способ реализации является аналогом подхода, используемого в императивных языках программирования, поскольку все вычисления выполняются строго в контексте IO.

Тем не менее, в последующих примерах и рассуждениях будет применяться первый вариант (с возвратом списка воздействий) реализации выходных воздействий, поскольку в данной работе упор делается на чистые функции. Кроме того, в первом подходе также возможно использование монады IO в качестве элементов списка выходных воздействий, что и было продемонстрировано в соответствующем листинге.

Декомпозиция функции переходов

В реальных задачах очень часто возникают ситуации, когда поведение автомата в двух и более состояниях отличается только константами, известными во время компиляции. Для устранения дублирования кода в подобных ситуациях обычно реализуют комбинаторы, представляющие собой параметризуемые шаблоны, позволяющие лаконично записывать вычисления. Например, для автоматов с выходными воздействиями, часто используемыми комбинаторами, описывающими переходы, являются: «перейти в некоторое состояние, не совершая выходных воздействий» и «остаться в текущем состоянии».

Библиотечный код в следующем листинге содержит обобщения использованных в предыдущих разделах типов данных и функций, а также три базовых комбинатора для описания переходов:

```
-- Тип функции переходов.
type StateTransition state event output =
  state -> event -> (state, [output])

-- Комбинатор <<просто перейти в состояние state>>.
pure :: state -> (state, [output])
pure state = (state, [])

-- Комбинатор <<перейти в состояние state с
-- одним выходным воздействием output>>.
blot :: state -> output -> (state, [output])
blot state output = (state, [output])
```

```
-- Комбинатор <<перейти в состояние state со
-- списком выходных воздействий output>>.
dark :: state -> [output] -> (state, [output])
dark state output = (state, output)
--

-- Тип функции переходов, пригодной для использования вместе с foldl.
type FoldableTransition state event output =
  (state, [output]) -> event -> (state, [output])

-- Преобразователь из StateTransition в FoldableTransition
stt2foldable :: StateTransition state event output ->
  FoldableTransition state event output
stt2foldable transition (pstate, accumulator) event =
  (nstate, accumulator ++ output)
  where (nstate, output) = transition pstate event
```

Функцию переходов можно представить, как некоторый набор состояний, где каждое из них содержит внутри себя всю свою логику (реакцию на входные воздействия). Каждое состояние можно получить путем параметризации константами некоторого шаблона. Таким образом, функцию переходов автомата можно декомпозировать в набор параметризованных *состояний-шаблонов*, где тело функции переходов принимает решение только о том, какому из шаблонов передать управление.

В следующем листинге представлен пример реализации декомпозиции функции переходов счетного триггера с четырьмя состояниями (рисунок 3):

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
                  | ButtonUp deriving Show
data TriggerState = LampOffButtonUp
                  | LampOffButtonDown
                  | LampOnButtonUp
                  | LampOnButtonDown deriving Show

-- Шаблон состояния.
statePattern ifUp ifDown event = case event of
  ButtonUp   -> pure ifUp
  ButtonDown -> pure ifDown

-- Функция переходов. (\x -> x e) --- лямбда-функция,
-- позволяющая не добавлять « e» в конец каждой
-- ветки оператора case (используется карринг).
triggerSwF :: StateTransition TriggerState TriggerEvent ()
triggerSwF s e = (\x -> x e) $ case s of
  LampOffButtonUp   -> statePattern s LampOffButtonDown
  LampOffButtonDown -> statePattern LampOnButtonUp s
  LampOnButtonUp    -> statePattern s LampOnButtonDown
  LampOnButtonDown  -> statePattern LampOffButtonUp s

main = print $ foldl (stt2foldable triggerSwF)
  (pure LampOffButtonUp) [ButtonDown, ButtonDown]
```

Ограничением состояния-шаблона statePattern, которое используется в приведенном выше листинге, является то, что оно может быть использовано только для работы со строго определенным типом событий. Дело в том, что левая часть ветки оператора case (до стрелки) должна быть константой и не может быть представлена как аргумент шаблона.

Однако данное ограничение можно обойти, введя возможность переименования событий при помощи вспомогательных функций:

```
-- Тип событий шаблона состояния.
data StatePatternEvent = BUp | BDown

-- Шаблон состояния.
statePattern ifUp ifDown event = case event of
  BUp   -> pure ifUp
  BDown -> pure ifDown

-- Переименование событий.
evMap :: TriggerEvent -> StatePatternEvent
evMap ButtonUp = BUp
evMap ButtonDown = BDown

-- Функция переходов. (\x -> x $ evMap e) --- лямбда-функция,
-- позволяющая не добавлять « (evMap e)» в конец каждой ветки оператора case.
triggerSwF :: StateTransition TriggerState TriggerEvent ()
triggerSwF s e = (\x -> x $ evMap e) $ case s of
  LampOffButtonUp   -> statePattern s LampOffButtonDown
  LampOffButtonDown -> statePattern LampOnButtonUp s
  LampOnButtonUp    -> statePattern s LampOnButtonDown
  LampOnButtonDown  -> statePattern LampOffButtonUp s
```

В рассматриваемом листинге функция evMap применяется к каждому событию для всех веток оператора case, однако если убрать лямбда-функцию из тела triggerSwF и передавать событие каждому состоянию-шаблону непосредственно, то различные шаблоны могут работать с различными типами событий не только по отношению к функции переходов, но и по отношению друг к другу.

Таким образом, используя карринг и функции высшего порядка, функцию переходов автомата можно составлять из абсолютно независимых частей, а элементы одной функции переходов — повторно использовать в другой.

Активные автоматы

Пусть состояние автомата содержит источники, при помощи некоторой операции над которыми можно получить следующее событие. Источниками могут быть как числовые идентификаторы (например, файловый дескриптор UNIX), так и ссылки на объекты в памяти, константные списки событий и т. п.

Тогда применим операцию извлечения следующего события из источника, полученное событие передадим функции переходов автомата, выполним выходные воздействия, после этого новое состояние примем за текущее. Таким образом можно реализовать автомат, работающий в отдельном потоке (будем называть такие автоматы *активными*). Однако при этом нет ответа на вопрос о том, когда этот автомат должен остановить свое исполнение.

Есть два варианта решения данной проблемы:

- ввести специальное выходное воздействие, означающее «завершить исполнение»;
- ввести специальное состояние, попав в которое, поток завершается («конечное состояние»).

Поскольку реализация первого решения может привести к неопределенности в том случае, если в списке выходных воздействий после воздействия «завершить исполнение» находятся еще элементы, будем использовать второй вариант.

Ввести одно конечное состояние на все автоматы невозможно, так как множества их состояний представлены различными типами данных, однако можно ввести класс (в терминах языка *Haskell*) с одной операцией, которая будет определять является ли текущее состояние конечным.

Таким образом, активный автомат оперирует кортежем из множества источников и состояния автомата, функция входов и функция выходов оперируют множеством источников, а функция переходов только состоянием автомата.

Реализуем эти утверждения в следующем библиотечном коде:

```
-- Является ли текущее состояние конечным?
class AutomataState state where
  lastState :: state -> Bool

-- Структура, представляющая отдельный активный автомат.
data IOAutomata source state event output = IOAutomata
  -- Получение событий из внешнего мира.
  (source -> IO (source, [event]))
  -- Функция переходов.
  (state -> event -> (state, [output]))
```

```
  -- Функция осуществления выходных воздействий.
  (source -> output -> IO source)
  -- Стартовое состояние системы.
  (source, state)

-- Исполняет активный автомат.
runIOAutomata (IOAutomata ep stt og psystem) =
  runIOAutomata' ep stt og psystem

runIOAutomata' ep stt og (psource, pstate) = do
  (xsource, events) <- ep psource
  let (nstate, outputs) =
    foldl (stt2foldable stt) (pure pstate) events
  nsource <- foldlM og xsource outputs
  if lastState nstate
    then return ()
    else runIOAutomata' ep stt og (nsource, nstate)
  where
    foldlM _ s [] = return s
    foldlM ofunc s (o:os) = do
      ns <- ofunc s o
      foldlM ofunc ns os
```

В следующем листинге при помощи предложенного выше библиотечного кода реализован счетный триггер в виде активного автомата, получающего события со стандартного ввода приложения:

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonClick deriving Show
data TriggerState = LampOff | LampOn deriving Show

-- Тип выходных воздействий.
data TriggerOutput = SayOn | SayOff

-- Конечное состояние.
instance AutomataState TriggerState where
  lastState LampOff = True
  lastState _ = False

-- Получение событий из внешнего мира.
eventF state = do
  print «Say: click»
  x <- getLine
  return $ if x == «click»
    then blot state ButtonClick
    else pure state

-- Функция переходов.
triggerSwF state ButtonClick = case state of
  LampOff -> blot LampOn SayOn
  LampOn  -> blot LampOff SayOff

-- Функция выходов.
outF state output = do
  case output of
    SayOn  -> print «On»
    SayOff -> print «Off»
  return state

main = runIOAutomata (IOAutomata
  eventF triggerSwF outF ((), LampOff))
```

Поскольку у данного триггера источником является стандартный ввод, то структура, представляющая источники активного автомата, в данном примере — пустой кортеж.

Заметим, что в данной модели только функция переходов автомата может модифицировать состояние. Поэтому, если бы производилось рассмотрение реализации, например, клиент-серверного приложения, то в серверном автомате следовало бы хранить список клиентов в структуре источников, а в функции получения события такого автомата — просматривать этот список на наличие новых сообщений от клиентов.

Функции выходов также может потребоваться множество источников автомата, например, для того, чтобы найти файловый дескриптор клиента, которому следует отправить сообщение. Именно поэтому определение типа IOAutomata имеет вид, представленный в листинге с библиотечным кодом.

Внедрение

С использованием методов, рассмотренных в данной статье, а также библиотечного кода из предыдущего раздела, был реализован сервер обмена мгновенными сообщениями.

Заключение

В статье были рассмотрены методы реализации событийных структурных конечных автоматов на языке Haskell с возможностью декомпозиции по состояниям. Особенностью рассмотренных реализаций является проверка корректности кода функций переходов на этапе компиляции самим компилятором благодаря использованию алгебраических типов данных для представления событий, состояний и (иногда) выходных воздействий автоматов. Эффективность предложенных методов апробирована на примере реализации сервера обмена мгновенными сообщениями.

Ссылки

1. Поликарпова Н. И., Шалыто А. А. Автоматное программирование. СПб.: Питер, 2009.
2. Сацкий С. Дизайн шаблона конечного автомата на C++. //RSDN Magazine. 2003. № 1. http://rsdn.ru/article/alg/FiniteStateMachine.xml
3. Ахмечет В., Линкер Н. Функциональное программирование для всех. //RSDN Magazine. 2006. № 2. http://www.rsdn.ru/article/funcprog/fp.xml
4. Кирпичев Е. Монады //RSDN Magazine. 2008. № 3. http://www.rsdn.ru/article/funcprog/monad.xml
5. The Parser Generator for Haskell. http://www.haskell.org/happy/
6. Parsec. http://www.haskell.org/haskellwiki/Parsec
7. Hudak P., Peterson J., Fasel J. A Gentle Introduction to Haskell 98. http://www.haskell.org/tutorial/
8. Stroustrup B. The C++ Programming Language. Boston: Addison-Wesley, 2000.

❖