УДК 004.4'242

РЕАЛИЗАЦИЯ КОНЕЧНЫХ АВТОМАТОВ НА ФУНКЦИОНАЛЬНЫХ ЯЗЫКАХ ПРОГРАММИРОВАНИЯ

Я. М. Малаховски,
магистрант
А. А. Шалыто,
доктор техн. наук, профессор
Санкт-Петербургский государственный университет информационных технологий, механики и оптики

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.

Ключевые слова *— конечные автоматы, функциональное программирование, Haskell.*

Введение

Несмотря на все более широкое использование автоматных моделей при разработке программного обеспечения, в настоящее время не известно методов реализации на функциональных языках программирования [1] событийных структурных конечных автоматов, используемых в автоматном программировании [2].

Основное отличие функциональных программ от императивных состоит в том, что функциональная программа представляет собой некоторое выражение (в математическом смысле), а выполнение программы означает вычисление значения этого выражения. При этом слово «вычисление» вовсе не означает, что операции производятся только над числами.

При сравнении «чистого» (pure) функционального и императивного подходов к программированию можно отметить следующие свойства функциональных программ:

- в чистых функциональных программах отсутствуют побочные эффекты, поэтому в них не существует прямого аналога глобальным переменным и объектам императивных языков программирования;
- в функциональных программах не используется оператор присваивания;
- в функциональных программах нет циклов, а вместо них применяются рекурсивные функции;
- выполнение последовательности команд в функциональной программе бессмысленно, поскольку одна команда не может повлиять на выполнение следующей.

Отсюда следует, что при использовании чистых функциональных языков программирования (например, языка *Haskell*) не удается непосредственно применять методы, используемые при реализации конечных автоматов на императивных языках программирования (например, языка *C++*). Это объясняется тем, что состояние автомата, по сути, является глобальной переменной, а обработка последовательностей событий в императивных языках производится при помощи циклов.

В теории конечных автоматов существуют два класса: абстрактные и структурные автоматы. Абстрактные автоматы обычно используются при разработке систем генерации парсеров по контекстно-свободным грамматикам и регулярным выражениям (см., например, работы [3, 4]). Функции переходов в таких задачах обычно строятся не по диаграммам состояний, а по множеству порождающих правил. В свою очередь, реализации на императивных языках программирования, построенные по диаграммам состояний, обычно валидируются сторонними утилитами, а не компилятором.

В событийных системах управления применяются структурные автоматы, 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 ре-

ализовать 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.

Таким образом, авторами решаются следующие задачи: реализация на функциональных языках программирования событийных структурных автоматов, в которых входные переменные отсутствуют, и валидация функций переходов таких автоматов в процессе компиляции.

В работе используется язык Haskell [5–8], так как:

- в отличие от иных подобных языков, он близок к *типизированному лямбда-исчислению;*
- в нем отсутствуют побочные эффекты;
- он не нарушает функциональные концепции при вводе-выводе.

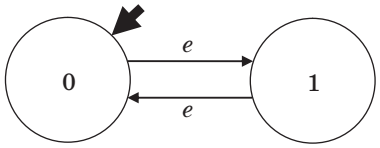
Кроме того, этот язык достаточно популярен в академической среде, и для него существует несколько качественных компиляторов.

Реализация автомата по диаграмме состояний

Суть предлагаемого подхода к реализации функции переходов на функциональных языках программирования состоит в представлении событий автоматов при помощи алгебраических типов данных, а состояний — кортежами или структурами соответствующих переменных. Продемонстрируем различные реализации счетного триггера на примерах.

Обработка одиночных событий.

Пусть требуется реализовать счетный триггер (рис. 1), 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. Исходный код, реализующий данный счетный триггер, приведен в листинге 1.



■ ***Рис. 1.*** *Диаграмма состояний счетного триггера*

Листинг 1. *Реализация счетного триггера на Haskell.*

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

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

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

Рассмотренная реализация функции переходов не является единственно возможной. Например, для этой цели можно использовать конструкции *pattern matching*. Однако в этом случае исходный код обычно получается длиннее, поскольку такая конструкция требует частичного дублирования. Поэтому в дальнейшем будет использоваться оператор *case*.

Последовательности событий.

Для того чтобы добавить возможность применения последовательности событий к начальному состоянию автомата, введем функцию *applyEvents*.

Листинг 2. *Функция applyEvents и ее использование.*

```
-- Исходный код листинга 1 за исключением функции main
...
-- Функция, применяющая события к начальному состоянию.
applyEvents :: State -> [Event] -> State
-- Результат = состояние, если событий больше нет.
applyEvents st [] = st
-- Иначе делаем переход и вызываемся рекурсивно.
applyEvents st (e:es) = applyEvents (gotEvent st e) es

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

Обобщение на произвольные типы данных для событий и состояний.

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. В листинге 3 приведен разработанный библиотечный код, а также реализация счетного триггера с его использованием.

Листинг 3. *Библиотечная функция applyEvents и ее использование.*

```
-- 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* (листинг 3) является левой сверткой (одна из стандартных операций функционального программирования) списка событий по функции переходов. Поэтому можно заменить все определение функции *applyEvents* на *applyEvents = foldl*.

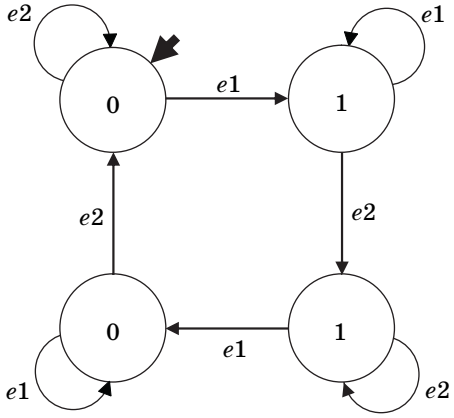
Преимущества функциональной реализации.

Алгебраические типы данных позволяют производить более строгие проверки по сравнению с конструкциями, сходными с конструкцией *enum* императивного языка C. Например, при использовании алгебраических типов данных невозможно случайно проверить на равенство элемент множества состояний с элементом множества событий. Еще одним преимуществом функционального подхода является то, что компилятор способен самостоятельно проверить полноту и непротиворечивость веток оператора *case*, тем самым осуществляя валидацию функции переходов на этапе компиляции.

Выходные воздействия

Функции переходов в предыдущих примерах самостоятельно не производили выходных воздействий, а только возвращали результирующее состояние, которое печаталось на консоль функцией *main*. На практике автоматам требуются и другие 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 (рис. 2). Его отличие от предыдущего триггера заключается в том, что лампа будет включаться или выключаться только после отпускания кнопки, а повторное нажатие или отпускание кнопки не будет производить никакого эффекта.



■ ***Рис. 2.*** *Диаграмма переходов счетного триггера с четырьмя состояниями*

Общую часть реализаций этого примера для экономии места вынесем в отдельный листинг 4.

Листинг 4. *Общая часть реализаций счетного триггера с четырьмя состояниями.*

```
-- Типы событий и состояний для счетного триггера.
data TriggerEvent = ButtonDown
          | ButtonUp deriving Show

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

Первый вариант реализации выходных воздействий.

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. Листинг 5 демонстрирует данный подход.

Листинг 5. *Функция переходов, возвращающая список выходных воздействий.*

```
-- Абстрактный тип функции переходов,
-- возвращающей новое состояние и список выходных воздей-
ствий.
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]
```

Второй вариант реализации выходных воздействий.

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 (листинг 6).

Листинг 6. *Функция переходов, возвращающая список выходных воздействий.*

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

-- Функция applyEvents, реализованная через монаду 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*.

Заключение

В работе предложены методы реализации событийных структурных конечных автоматов на языке Haskell. При этом продемонстрированы преимущества этих подходов по сравнению с реализациями на императивных языках программирования. Такими преимуществами являются строгая типизация составных частей конечного автомата и валидация функции переходов компилятором.

Литература

1. **Abelson H., Sussman G.** Structure and Interpretation of Computer Programs. — MIT Press, 1985. — 634 p.
2. **Поликарпова Н. И., Шалыто А. А.** Автоматное программирование. — СПб.: Питер, 2009. — 176 с.
3. **Parsec.** http://www.haskell.org/haskellwiki/Parsec (дата обращения: 09.07.2009)
4. **The Parser Generator for Haskell.** http://www.haskell.org/happy/ (дата обращения: 09.10.2009).
5. **Bird R.** Introduction to Functional Programming using Haskell. — NY.: Prentice Hall, 1998. — 448 p.
6. **Davie A.** Introduction to Functional Programming System Using Haskell. — Cambrige: **Cambrige University** Press, 1992. — 304 p.
7. **Hudak P., Peterson J., Fasel J.** A Gentle Introduction to Haskell 98. http://www.haskell.org/tutorial/ (дата обращения: 09.07.2009) — 64 p.
8. **Кирпичев Е.** Монады // RSDN Magazine. 2008. N 3. http://www.rsdn.ru/article/funcprog/monad.xml (дата обращения: 09.07.2009).