Статья опубликована в Научно-техническом вестнике СПбГУ ИТМО. 2010. № 6 (70), с. 73 – 77.

УДК 004.4'242

Валидация автоматов с переменными на функциональных языках программирования

Я. М. Малаховски, Г. А. Корнеев
Санкт-Петербургский государственный университет
информационных технологий, механики и оптики

В статье описан предментно-ориентированный язык программирования (*eDSL*), встроенный в язык программирования *Haskell* и предназначенный для создания автоматных программ. Предложенный язык, поддерживает повторное использование кода и валидацию функций переходов автоматов до их первого использования в программе. В рамках *eDSL* также разработано представление для логических формул над произвольными множествами переменных, не ограничивающее пользователя в выборе фиксированного логического базиса.

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

Введение

В настоящее время функциональное программирование [1] предоставляет множество возможностей для созданий вспомогательных конструкций, таких как продолжения (continuations), монады (monads) и стрелки (arrows), реализующих различные стили программирования. Перечисленные структуры имеют не только теоретическую, но и практическую значимость, предоставляя удобный «синтаксический сахар» для реализации некоторых классов программ.

В последние годы широкое распространение получили *предметно-ориентированные языки программирования* (*Domain-Specific Language*, *DSL*), предназначенные для разработки программ в четко определенных предметных областях с использованием терминов предметной области. *DSL*, основанные на синтаксисе и возможностях нижележащего языка, принято называть *встроенными предметно-ориентированными языками программирования* (*embedded DSL*, *eDSL*).

В работе [2] был предложен подход к реализации событийных конечных автоматов [3] на функциональных языках программирования. При этом для валидации функций переходов на полноту и непротиворечивость использовались свойства алгебраических типов данных и компилятора языка *Haskell* [4].

В настоящей работе рассматривается обобщение метода, предложенного в работе [2], на структурные автоматы, и язык *eDSL* его реализующий. При этом в качестве пометок на переходах автомата указываются не одиночные события, а комбинации входных переменных.

Возможность описания структурных автоматов, а также их автоматическая валидация являются ключевыми требованиями при разработке ответственных систем. Однако у большинства распространенных языков функционального программирования системы типов недостаточно выразительны для осуществления валидации функций переходов структурных автоматов. Это связано с тем, что большинство систем типов таких языков программирования сохраняют возможность реконструкции типовых аннотаций в программе и не являются Тьюринг-полными.

Язык eDSL, предлагаемый в настоящей работе, осуществляет валидацию не при помощи системы типов, а во время исполнения программы, однако при этом

гарантируется, что функции переходов, не прошедшие валидацию, не могут использоваться в программе.

Постановка задачи

Поскольку в структурных конечных автоматах переходы диаграммы состояний помечаются логическими выражениями, то для реализации автоматного *eDSL* требуются:

- синтаксическое представление для логических формул на дугах графов переходов;
- валидатор логических формул;
- синтаксическое представление для дуг и состояний автоматов.

Также при разработке синтаксиса учитывалось требование о предоставлении пользователю возможности повторного использования частей автоматов. В связи с этим, разработанный синтаксис подразумевает раздельное определение состояний, с последующим конструированием автоматов из отдельных частей.

Логические выражения

Для конструирования логических выражений требуется выбрать способ представления множества логических переменных. Для этих целей может использоваться произвольное множество элементов (например, натуральные числа), однако как показано в работе [2], использование алгебраических типов данных является более предпочтительным, поскольку они позволяют производить большее число проверок на этапе компиляции.

Кроме того, для записи логических формул над каким-то множеством переменных, необходимо выбрать логический базис. Поскольку выражения, построенные в этом логическом базисе, будут использоваться во время исполнения программы, этот базис также следует представить алгебраическим типом данных, параметризованным типом данных переменных формулы (листинг 1).

Листинг 1. Логический базис

```
data Logic v = Variable v          -- Вхождение переменной
             | Always              -- Тавтология
             | Never               -- Невыполнимая формула
             | Not (Logic v)       -- Отрицание
             | And [Logic v]       -- Конъюнкция
             | Or [Logic v]        -- Дизъюнкция
```

Пример применения предложенного логического базиса, приведен на листинге 2.

Листинг 2. Пример использования логического базиса

```
data Var = X1 | X2
f = Or [And [Variable X1, Variable X2], X2]
```

Интерпретатор таких формул, принимающий на вход отображение из имен переменных в их логические значения и возвращающий результат вычисления формулы, реализуется очевидным образом.

Валидация

В процессе валидации автомата проверяется, что из каждого состояния при любых значениях переменных будет произведен переход ровно по одной дуге диаграммы (свойство полноты и непротиворечивости).

В терминах логических формул это свойство означает, что для каждого состояния:

- не существует пары таких дуг, конъюнкция логических формул которых выполнима;
- дизъюнкция логических формул всех дуг является тавтологией.

Также обычно проверяется, что в автомате нет дуг, переходов по которым никогда не может произойти (логическая формула на дуге невыполнима).

Поскольку каждое рассматриваемое свойство автомата представимо в виде логической формулы, то его проверка может быть сведена к поиску контрпримера при помощи исчисления секвенций [5].

Особенности *eDSL*, предлагаемого в настоящей работе, позволяют исключить возможность использования некорректных функций переходов во время исполнения программы и делают процесс валидации прозрачным для пользователя.

Синтаксис

Заметим, что выражения, записанные в форме, представленной на листинге 2, весьма неудобны в использовании, поскольку обычно для логических формул используются более компактные методы записи. Поэтому синтаксис для логических формул реализован при помощи инфиксных оберток над алгебраическим типом данных из листинга 1.

Синтаксис для определения дуг и состояний реализован при помощи функций `-->`, `/./` и `state`. На листинге 3 представлен пример реализации автомата при помощи разработанного eDSL.

Листинг 3. Пример реализации автомата

```
data States = FirstState | SecondState deriving Show
data Output = Out1 | Out2 deriving Show
data X = X1 | X2 | X3 deriving (Show,Eq)

state1 = state FirstState [
    X1 &.& not' X2 --> state1 /./ Out2,
    X2 --> state1 /./ Out1,
    Star --> state1]

gotEvent = auto2Acc $ mkSimple [state1]
main = print $ foldl gotEvent FirstState [[X1], [X1, X2]]
```

В приведенном листинге при помощи функций `-->` и `/./` конструируется дуга графа переходов, функция `state` валидирует и конструирует состояние из набора дуг, а функция `mkSimple` конструирует функцию переходов автомата из набора состояний.

Следует отметить, что слева от `-->` могут располагаться как переменные, так и логические формулы. Кроме того, полезно ввести специальное выражение `Star`, обозначающее дугу, переход по которой производится только в том случае, если не выполнены логические выражения на всех остальных дугах (листинг 3).

Поскольку типы всех этих выражений различны, то в функции `-->` используется преобразование первого аргумента при помощи класса типов с функцией `elift`

(листинг 4), которая «поднимает» все возможные виды выражений до типа, содержащего, и логические формулы и `Star`.

Листинг 4. «Подъем» выражений

```
data ELifted v = Star | NoStar (Logic v)

class ELiftable v a | v -> a where
    elift :: v -> ELifted a
instance ELiftable (ELifted v) v where
    elift = id
```

Быстрые логические функции

В связи с тем, что некоторые логические функции (такие как «голосование») могут иметь большой размер, будучи выраженными в логическом базисе «и, или, не», их вычисление может занимать существенную часть времени работы функции переходов. Ускорить вычисление выражений на дугах можно двумя методами:

- упрощение логических формул в процессе конструирования автоматов;
- предоставление пользователю возможности использовать в логических выражениях произвольные логические функции языка *Haskell*.

Для получения результата вычисления выражения во втором методе не требуется интерпретации представления логического выражения, что ведет к уменьшению времени работы программы. Кроме того, функции языка *Haskell* не ограничены каким-то фиксированным базисом, а потому их метод вычисления результата может быть ближе к оптимальному, чем у любой функции, выраженной в фиксированном логическом базисе.

Рассуждая подобным образом, можно прийти к заключению, что фиксированный базис для представления логических выражений требуется только для работы валидатора. Поэтому существует возможность применить оба метода ускорения вычислений одновременно:

- упрощать логические формулы следует перед передачей выражения валидатору;
- преобразование из логических функций языка *Haskell* следует производить в пару из логического выражения в базисе валидатора и функции, вычисляющей результат по значениям входных переменных.

Это позволяет освободить пользователя от необходимости придерживаться какого-то фиксированного базиса, а также ускоряет процессы валидации и вычисления функций переходов.

Полный исходный код валидатора и всех вспомогательных функций в данной статье не приводится из-за его большого объема, пример преобразования логических функций языка *Haskell* в функции для работы с логическими выражениями представлен в листинге 5.

Листинг 5. «Подъем» функций

```
not' = lift1 not
infix 8 &.&
(&.&) = lift2 (&&)
infix 7 |.|
(|.|)  = lift2 (||)
```

Функции `lift1` и `lift2` производят описанное выше преобразование над одно- и двухаргументными логическими функциями языка *Haskell*.

Повторное использование кода

Предлагаемые методы подразумевают активное повторное использование состояний и дуг графов переходов при конструировании различных автоматов, однако описанные ранее функции эту возможность ограничивают, поскольку все части автомата должны использовать один и тот же тип входных переменных и выходных воздействий.

Метод, дающий пользователю возможность собирать автоматы из состояний, работающих с различными типами переменных, заключается в приписывании специальной функции, производящей отображения множества переменных собираемого автомата во множество переменных конкретного состояния, каждому такому состоянию. Пример использования такой функции приведен в листинге 6.

Листинг 6. Переименование переменных автомата

```
-- Этот код следует трактовать как продолжение листинга 3,
-- переопределяющее некоторые функции
data XX = XX1 | XX2 deriving (Show,Eq)

state2 = state SecondState $ [
    XX1 |.| not' XX2 --> state2 /./ Out2,
    Star --> state1]

x2xx X1 = XX1
x2xx X2 = XX2
x2xx X3 = XX2

gotEvent = auto2Acc $ mkSimple [state1, chVars x2xx state2]
main = print $ foldl gotEvent SecondState [[X1], [X1, X2]]
```

Однако теперь оказывается, что функция `state` не может возвращать прошедшее валидацию состояние, поскольку функция `chVars` должна изменить переменные внутри всех логических выражений на дугах, а валидацию нужно будет производить уже после этого преобразования. Кроме того, если одно и то же состояние используется в нескольких автоматах, то оно будет валидироваться повторно для каждого автомата. Чтобы этого избежать докажем следующую теорему.

Теорема (о корректной замене переменных). Любое отображение множества переменных состояния в логические выражения над произвольным множеством переменных (в том числе и над первоначальным множеством переменных состояния) не нарушает свойств, проверяемых валидацией.

Доказательство.

Свойство полноты требует, чтобы дизъюнкция всех формул на дугах являлась тавтологией. Свойство непротиворечивости требует, чтобы все попарные конъюнкции не были выполнимы, то есть, чтобы отрицание дизъюнкции попарных конъюнкций всех формул была тавтологией.

Рассмотрим тавтологию, представляющую интересующее нас свойство. Произведём над ней следующее изоморфное преобразование: поместим формулу в тело лямбда-абстракции, а каждую логическую переменную в формуле заменим на переменную, связанную этой абстракцией. При подстановке первоначальной переменной на место соответствующей связанной переменной будет получена первоначальная формула.

С другой стороны, можно заметить, что в теле лямбда-абстракции находится тавтология, а потому, по определению, ее можно заменить на константу «истина».

Теперь, при подстановке вместо переменных лямбда-абстракции произвольных логических комбинаций переменных из какого-то множества, результатом будет являться «истина».

Эту теорему также можно доказать при помощи структурной индукции по размеру подставляемых вместо переменных выражений, используя тот факт, что исчисление секвенций доказывает, что формула является тавтологией, тогда и только тогда, когда в конце работы алгоритма в правом и левом столбце множества переменных пересекаются [5].

Следствие (о переименовании переменных). Любая замена всех переменных, используемых в логических выражениях, надписанных над дугами состояния, на другие переменные (возможно даже, сливая различные переменные в одну) не нарушает свойств, проверяемых валидацией.

Доказательство. Данное свойство является частным случаем доказанной ранее теоремы, поскольку одинарные переменные являются частным логических выражений над множеством переменных.

Реализация предлагаемого в настоящей работе *eDSL* использует доказанный факт следующим образом: функция `state` возвращает прошедшее валидацию состояние, а функция `chVars` производит отображение переменных, не затрагивая внутренней структуры самого состояния. Таким образом, повторное использование состояния не вызывает повторной валидации.

Для предоставления еще больших возможностей использования уже написанного кода, функцию, аналогичную `chVars`, можно реализовать также и для выходных воздействий автоматов, а определяемые состояния сделать параметризуемыми шаблонами. Таким образом, получаемые части автоматов будут вообще избавлены от элементов конкретных типов данных в своих описаниях.

Заключение

В настоящей работе был разработан автоматный *eDSL* для языка *Haskell*:

- производящий валидацию функций переходов до их первого запуска;
- гарантирующий невозможность использования функций переходов, не прошедших валидацию;
- не ограничивающий пользователя в выборе логического базиса.

Доказано свойство о возможности переименования переменных в логических выражениях, надписанных над дугами в графах переходов. С учетом этого свойства в разработанный *eDSL* были добавлены средства, поддерживающие активное повторное использование уже реализованных автоматов и их частей без выполнения повторных валидаций.

Список литературы

1. *Абельсон Х., Сассман Д.* Структура и интерпретация компьютерных программ. М.: Добросвет. 2006. 608 с.
2. *Малаховски Я. М., Шалыто А. А.* Конечные автоматы в чистых функциональных языках программирования. Автоматы и Haskell // RSDN Magazine. 2009. № 3.
3. *Поликарпова Н. И., Шалыто А. А.* Автоматное программирование. СПб.: Питер. 2009. 176 с.
4. *Худак П., Петерсон Дж., Фасел Дж.* Мягкое введение в Хаскелл // RSDN Magazine. 2006. № 4; 2007. № 1. http://rsdn.ru/article/haskell/haskell_part1.xml, http://rsdn.ru/article/haskell/haskell_part2.xml
5. *Верещагин Н., Шень А.* Лекции по математической логике и теории алгоритмов. Часть 2. Языки и исчисления. М.: МЦНМО. 2002. 288 с.