4 КОМПЬЮТЕРНЫЕ СИСТЕМЫ И ИНФОРМАЦИОННЫЕ ТЕХНОЛОГИИ

УДК 004.4'2

ПРИМЕНЕНИЕ ЗАВИСИМЫХ СИСТЕМ ТИПОВ СО СТРУКТУРНОЙ ИНДУКЦИЕЙ ДЛЯ ВЕРИФИКАЦИИ РЕАКТИВНЫХ ПРОГРАММ

Я.М. Малаховски, Г.А. Корнеев

Предлагается конструктивное доказательство (в виде программы на языке Agda) того, что решение задачи о выполнимости булевой формулы при помощи исчисления секвенций и верификация с использованием темпоральных спецификаций на языке CTL могут быть представлены в разрешимом подмножестве зависимой системы типов.

Ключевые слова: задача о выполнимости булевой формулы, верификация, функциональное программирование, системы типов, зависимые типы, Agda.

Введение

Одной из современных тенденций при разработке программного обеспечения является так называемое доказательство по построению (proof by construction), когда нежелательные состояния в программе (например, некорректные состояния ресурсов) исключаются не при помощи проверок, осуществляемых внешними средствами, а при помощи свойств, которыми обладают используемые языки программирования. Одним из самых известных таких методов является RAII [1]. Также все большую популярность набирают средства аннотирования программ, такие как, например, nullness annotations, позволяющие не отделять спецификацию программы от ее кода. Это позволяет избавиться от целого класса ошибок, связанных с рассинхронизацией программы и ее спецификации в процессе разработки.

В функциональном программировании [2, 3] для решения задач доказательства программ по построению и их аннотирования применяют системы типов. В соответствии с изоморфизмом Кэрри–Ховарда различным системам типов функциональных языков программирования соответствуют различные логические системы [4]. Например, система типов простого типизированного лямбда-исчисления изоморфна минимальной логике высказываний, простые зависимые типы изоморфны логике первого порядка, а зависимые типы высших порядков изоморфны логике высших порядков [4–6]. При этом в таких системах тип терма (ти́повая аннотация терма) является утверждением, а сам терм – доказательством этого утверждения. Отмеченное свойство позволяет использовать некоторые функциональные языки программирования (имеющие в качестве системы типов зависимую или более мощную) для проверки правильности некоторых классов математических доказательств (утверждения, для доказательства которых достаточно интуиционистской логики), а также для доказательства практически произвольных утверждений о самих программах. Однако большинство распространенных функциональных языков программирования (например, Haskell [7]) не обладают такими возможностями. Это связано с тем, что эти языки сохраняют возможность автоматической реконструкции ти́повых аннотаций в программе (автоматический вывод типов всех термов), а потому их системы типов не являются Тьюринг-полными [6]. В свою очередь, использование Тьюринг-полной зависимой системы типов делает неразрешимой задачу проверки эквивалентности типов и может привести к бесконечно долгой компиляции программы [4, 5].

Интересным подходом к решению данной проблемы обладают языки Agda [8] и Coq [9]. Все программы на этих языках завершаются благодаря возможности использования в программе только структурной индукции (но не неограниченной рекурсии), а система типов совпадает по вычислительной мощности с остальной частью языка. Оказывается, что на этих языках можно выражать и проверять многие полезные свойства программ, не теряя при этом свойств разрешимости.

Другими словами, возвращаясь к задачам доказательства программ по построению и их аннотирования, можно утверждать, что в некоторых функциональных языках программирования аннотирование термов программы практически произвольными утверждениями (типами) и написание программ, корректных по построению (написание термов, имеющих необходимые типы), являются встроенными возможностями.

Таким образом, свойства программ (например, что программа удовлетворяет некоторой темпоральной спецификации [10]), обычно проверяемые при помощи сторонних утилит (например, построением модели программы и ее верификации при помощи средств проверки моделей), в языках Agda и Coq могут быть проверены средствами самих языков, если их удастся закодировать в виде утверждений в системах типов этих языков. Иначе говоря, требуемые свойства программ требуется переписать (формализовать) в утверждения, разрешимые в зависимой системе типов со структурной индукцией. Это не всегда является тривиальной задачей, поскольку, кроме кодирования самого утверждения, требуется привести и доказательство этого утверждения (также закодированное в виде терма программы).

Целью настоящей работы является конструктивное доказательство (в виде программы на языке Agda) того, что решение задачи о выполнимости булевой формулы при помощи исчисления секвенций и верификация с использованием темпоральных спецификаций на языке CTL [10] могут быть представлены в разрешимом подмножестве зависимой системы типов.

Результаты данной работы были использованы [11] при создании встроенных предметно-ориентированных языков (embedded Domain Specific Language, eDSL), предназначенных для описания реактивных программ с использованием автоматного программирования [12].

Все используемые свойства языка Agda, выходящие за рамки простого типизированного лямбда-исчисления, явно обозначены. Все функции из листингов, определения которых не приводятся, а отсутствие определения никак не комментируется, являются функциями стандартной библиотеки.

Особенности используемого языка программирования

Язык программирования Agda является чистым ленивым функциональным языком программирования со структурной индукцией [13]. Однако в качестве системы типов используется разрешимое подмножество зависимой системы типов. При этом и сам язык, и система типов совпадают по вычислительной мощности и не являются Тьюринг-полными. Классы типов [7] в языке отсутствуют, однако их можно эмулировать при помощи встроенных средств. Язык частично сохраняет возможность реконструкции ти́повых аннотаций (только тривиальные аннотации для термов non-ground типов).

Структурно-рекурсивное решение задачи о выполнимости булевой формулы

Классический метод поиска контрпримера к логическому утверждению, рассматриваемый в работе [14], не может быть непосредственно закодирован на языке Agda, поскольку перенос формулы из одного множества в другое не позволяет построить из последовательности получаемых секвенций явную убывающую последовательность. Для решения данной проблемы следует «транспонировать» формулировку алгоритма поиска решения.

Поскольку валидация при помощи исчисления секвенций использует базис И-ИЛИ-НЕ, то необходимо явно его определить:

```
data Logic (V : Set) : Set where
  Latom : V -> Logic V           -- Вхождение переменной.
  Ltrue Lfalse : Logic V         -- Тождественно истинное и
                                 -- ложное утверждения.
  Lnot : Logic V -> Logic V      -- Отрицание.
  _L∧_ _L∨_ : Logic V -> Logic V -> Logic V -- Конъюнкция и
                                            -- дизъюнкция.
```

Приведем пример применения предложенного логического базиса:

```
data Var : Set where            -- Логические
  X1 X2 X3 : Var                -- переменные

example : Logic Var
example = ((Latom X1) L∧ (Latom X2)) L∨ (Lnot (Latom X3))
```

Интерпретатор таких формул (назовем его `compute`), принимающий в качестве входных параметров формулу и функцию из имени логической переменной в ее значение (типа `Bool`) и возвращающий результат вычисления формулы, реализуется очевидным образом при помощи сопоставления с образцом (pattern matching, оператор `case`).

«Транспонирование» классического алгоритма заключается в замене в нем двух множеств секвенции на список пар вида (`сторона`, `формула`), где сторона задается элементом типа `Bool` (`true` – левая, `false` – правая). При этом множество выполняющих наборов для формулы будет списком списков таких пар.

После этого, используя то наблюдение, что множество выполняющих наборов является булевой алгеброй, введем на нем стандартные логические операции. При этом пустой список будет являться нулевым элементом, а список, содержащий пустой список – нейтральным элементом:

```
Seql : Set -> Set
Seql V = List (List V) -- Множество выполняющих наборов

zero : ∀{V} -> Seql V
zero = []
one : ∀ {V} -> Seql V
one = [ [] ]
```

```
single : ∀ {V} -> V -> Seql V
single v = [ [ v ] ]

_sor_ : ∀ {V} -> Seql V -> Seql V -> Seql V
x sor y = x ++ y
_sand_ : ∀ {V} -> Seql V -> Seql V -> Seql V
x sand y = concatMap (λ xel -> map (λ yel -> (xel ++ yel)) y) x
```

После этого функция (назовем ее `solve`), принимающая имя столбца (напомним, обозначаемое элементом типа `Bool`) и логическую формулу и возвращающая список ее выполняющих наборов, реализуется следующим образом:

```
solve : ∀ {V} -> Bool -> Logic V -> Seql (Bool × V)
solve any (Latom x) = single (any , x)
solve true   Ltrue = one
solve false  Ltrue = zero
solve true  Lfalse = zero
solve false Lfalse = one
solve any (Lnot x) = solve (not any) x
solve true  (x L∧y) = (solve true x)  sand (solve true y)
solve false (x L∧y) = (solve false x) sor  (solve false y)
solve true  (x L∨y) = (solve true x)  sor  (solve true y)
solve false (x L∨y) = (solve false x) sand (solve false y)
```

Нетрудно видеть, что в этом терме все рекурсивные вызовы производятся с меньшими аргументами, что позволяет данному коду успешно пройти тест на завершаемость, выполняемый компилятором языка Agda. Само же исчисление секвенций реализуется фильтрацией из списка выполняющих наборов невыполнимых (пустых или правый и левый столбцы которых пересекаются).

Подробная реализация всех упомянутых функций приводится в работе [11].

Грамматика языка CTL и представление верифицируемой программы

Аналогичным логическому базису образом определяется и синтаксис выражений языка CTL:

```
data CTL (A : Set) : Set where
  |atom : A -> CTL A
  |true |false : CTL A
  |not : CTL A -> CTL A
  _|∧_ _|∨_ : CTL A -> CTL A -> CTL A
  X∀ X∃ : CTL A -> CTL A
  _U∀_ _U∃_ : CTL A -> CTL A -> CTL A
```

Для реализации CTL-верификатора была выбрана функциональная версия классического алгоритма [15], представленная в работе [16]. Основная идея данной интерпретации заключается в том, что вместо вычисления подформул с использованием динамического программирования можно непосредственно вычислять верифицируемую формулу, кешируя (мемоизируя) промежуточные результаты. При этом появляется возможность отказаться от поиска компонент сильной связности, поскольку каждый путь в модели Крипке представляет собой либо некоторую последовательность состояний без циклов, либо некоторую (возможно, пустую) последовательность состояний без циклов, завершающуюся циклом. В такой формулировке формула языка CTL, вычисление которой зацикливается (т.е. ее выполнимость требует ее же выполнимости), считается невыполнимой.

Передать знание о структуре путей в модели Крипке компилятору не представляется возможным. Однако можно заметить, что путь максимальной длины для вычисления CTL-формулы в модели Крипке – гамильтонов цикл. В связи с этим для доказательства завершаемости алгоритма предлагается добавить размер модели Крипке в качестве параметра верификатора CTL-формулы и уменьшать это значение при рекурсивном вычислении той же подформулы в новом состоянии. Реализация верификатора в настоящей работе не приводится вследствие ее большого объема и необходимости рассмотреть несколько нетривиальных приемов, за подробностями направляем читателя к работе [11].

Исчисление секвенций и верификация в системе типов

Подчеркнем, что выше были описаны функции, проверяющие выполнимость булевой формулы и соответствие заданной модели Крипке некоторой темпоральной спецификации на языке программирования CTL, тотальность (totality) и завершаемость которых проходит проверку компилятором языка Agda. Поскольку в языке Agda система типов не отличается по вычислительной мощности от остальной части

языка, то для представления свойств валидации и верификации в системе типов необходимо построить соответствующие предикаты, использующие функции, разрешающие проверяемые свойства. В системе типов предикатом над данными является функция из значения какого-то типа в другой тип. При этом типом-результатом в общем случае может являться произвольный тип, описывающий свойства рассматриваемых данных.

При разработке автоматного eDSL [11] свойства, описывающие валидность и соответствие автомата некоторой темпоральной спецификации, являются внешними по отношению к функциональной программе, на которой их выражают. По этой причине информацию о них можно только стереть из типов. Таким образом, предикаты для валидации автомата (проверки полноты и непротиворечивости исходящих из состояния дуг, над которыми надписаны логические выражения) и верификации в системе типов принимают следующий вид:

```
-- S = state, V = variable, O = output action

-- Последний аргумент - список дуг переходов состояния автомата
isValid : ∀ {S V O} -> List (Logic V × (S × List O)) -> Set
isValid g = if valid g then Top else Bot

-- Последний аргумент – описание автомата
isVerified : ∀ {V} -> CTL V -> Auto -> Set
isVerified f a = if verified f a then Top else Bot
```

В приведенном листинге `Top` – тип с одним элементом, а `Bot` – тип без элементов. Другими словами, функции `isValid` и `isVerified` проверяют соответствующие свойства и стирают из типа всю информацию о проверке, за исключением ее результата. Кроме того, использование этих типов позволяет компилятору автоматически выводить значения аргументов для них, что позволяет сделать аргументы этого типа неявными аргументами функции.

После этого написание функций, проверяющих требуемые свойства автоматов на этапе компиляции, заключается в простом добавлении к ним неявного аргумента вида

```
{v : isValid l},
```

что является стандартным приемом.

Заключение

В работе показано, что решение задачи о выполнимости булевой формулы и проверка соответствия модели Крипке заданной темпоральной спецификации на языке CTL представимы в разрешимом подмножестве зависимой системы типов. Данный факт был использован в работе [11] для создания системы, поддерживающей программирование в автоматном стиле [12] на функциональных языках программирования. Описанные функции развивают методы, предложенные в работах [17, 18], но, в отличие от последних, позволяют производить валидацию и верификацию автоматов на этапе компиляции.

Таким образом, разработанные в работе термы не только являются важным теоретическим результатом в области границ применимости структурно-рекурсивных функций, но и позволяют производить проверку соответствия программы темпоральной спецификации в инструменте, предназначенном для автоматного программирования. При этом такой инструмент не требует никаких внешних средств для своей работы, а его (частичная) корректность может быть доказана на самом языке Agda.

Литература

1. Страуструп Б. Дизайн и эволюция языка C++. – СПб: Питер, 2006. – 448 с.
2. Абельсон Х., Сассман Д. Структура и интерпретация компьютерных программ. – М.: Добросвет, 2006. – 608 с.
3. Bird R. Introduction to Functional Programming Using Haskell. – Prentice-Hall, 1998. – 448 p.
4. Sørensen M.H.B., Urzyczyn P. Lectures on the Curry-Howard Isomorphism. – Elsevier, 2006. – 442 p.
5. Thompson S. Type Theory and Functional Programming. – Addison-Wesley, 1991. – 365 p.
6. Пирс Б. Типы в языках программирования [Электронный ресурс]. – Режим доступа: http://newstar.rinet.ru/~goga/tapl/, свободный. Яз. рус. (дата обращения 29.02.12).
7. Худак П., Петерсон Дж., Фасел Дж. Мягкое введение в Хаскелл // RSDN Magazine. – 2006. – № 4. – С. 35–67; 2007. – № 1. – С. 3–16.
8. The Agda Wiki [Электронный ресурс]. – Режим доступа: http://wiki.portal.chalmers.se/agda/, свободный. Яз. англ. (дата обращения 29.02.12).
9. The Coq Proof Assistant [Электронный ресурс]. – Режим доступа: http://coq.inria.fr/, свободный. Яз. англ. (дата обращения 29.02.12).
10. Вельдер С.Э., Лукин М.А., Шалыто А.А., Яминов Б.Р. Верификация автоматных программ. – СПбГУ ИТМО, 2011. – 246 с.

11. Малаховски Я.М. Применение систем типов для валидации и верификации автоматных программ. Магистерская диссертация. НИУ ИТМО, 2011 [Электронный ресурс]. – Режим доступа: http://is.ifmo.ru/papers/2011-master-malakhovski/, свободный. Яз. рус. (дата обращения 07.03.2012).
12. Поликарпова Н.И., Шалыто А.А. Автоматное программирование. – СПб: Питер. 2009. – 176 с.
13. Norrel U. Dependently Typed Programming in Agda // Advanced Functional Programming. – V. 5832. – 2009. – P. 230–266.
14. Верещагин Н.К., Шень А. Лекции по математической логике и теории алгоритмов. Ч. 2. Языки и исчисления. – М.: МЦНМО, 2002. – 288 с.
15. Clarke E.M., Emerson E.A., Sistla A.P. Automatic verification of finite-state concurrent systems using temporal logic specifications // ACM Transactions on Programming Languages and System. – 1986. – № 8. – P. 244–263.
16. CTL Model Checking in Haskell: A Classic Algorithm Explained as Memoization [Электронный ресурс]. – Режим доступа: http://www.kennknowles.com/blog/2008/05/07/ctl-model-checking-in-haskell-a-classic-algorithm-explained-as-memoization/, свободный. Яз. англ. (дата обращения 29.02.12).
17. Малаховски Я.М., Шалыто А.А. Конечные автоматы в чистых функциональных языках программирования. Автоматы и Haskell // RSDN Magazine. – 2009. – № 3. – С. 20–26.
18. Малаховски Я.М., Корнеев Г.А. Валидация автоматов с переменными на функциональных языках программирования // Научно-технический вестник СПбГУ ИТМО. – 2010. – № 6 (70). – С. 73–77.

Малаховски Ян Михайлович – Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, аспирант, trojan@rain.ifmo.ru

Корнеев Георгий Александрович – Санкт-Петербургский национальный исследовательский университет информационных технологий, механики и оптики, кандидат технических наук, доцент, kgeorgiy@rain.ifmo.ru