УДК 004.05

Проектирование сертифицированного компилятора предикатных программ

В. И. Шелехов

Компилятор со средствами дедуктивной верификации подлежит тщательной сертификации для подтверждения высокого уровня доверия к результатам дедуктивной верификации. Разработка компилятора предикатных программ на базе его модели в соответствии с модельно-ориентированной технологией повышает уровень доверия к компилятору. Главной частью модели компилятора является модель внутреннего представления программы, реализуемая в рамках третьего релиза компилятора предикатных программ. В настоящей работе описывается архитектура модели внутреннего представления программы. Важнейшим аспектом модели является анализ типов. В описании модели используется специально разработанный язык спецификации структур данных компилятора. Верификация модели и разработка программы компилятора на базе модели обеспечат высокую надежность компилятора.

Ключевые слова: дедуктивная верификация, сертификация программ, модельно-ориентированная технология.

1. Введение

Метод дедуктивной верификации обеспечивает абсолютную гарантию корректности программы относительно ее спецификации. Ввиду большой сложности и трудоемкости этот метод применяется лишь для критических фрагментов программы в приложениях, где требуется сверхвысокая надежность и безопасность.

Дедуктивная верификация намного проще и быстрее для предикатных программ, чем для аналогичных императивных программ. Преимущество по времени верификации – более чем в 5 раз. Для программы быстрой сортировки с двумя опорными элементами зафиксировано преимущество в 10 раз [1].

В соответствии со стандартами DO178C и ГОСТ Р ИСО/МЭК 15408-3 [2] сертификация программ, для которых применялась дедуктивная верификация, сопровождается сертификацией корректности сопутствующих инструментов и методов. Сертифицируется подсистема верификации вместе с содержащей ее системой программирования, отдельно front-end и back-end генерации кода. Далее в этой цепочке – операционная система, загрузчик программ и, наконец, система команд компьютера. Сертифицируется также сам метод дедуктивной верификации.

Возможность ситуации, когда в процессе компиляции в программу вносится ошибка, принципиально снижает ценность проведенной дедуктивной верификации. Поэтому сертификации компилятора отводится особое внимание. В идеале компилятор должен пройти процедуру дедуктивной верификации. Имеется лишь один такой компилятор. Это оптимизирующий компилятор CompCert [3] с языка Си, доступный с 2015 г. Компилятор состоит из 20 просмотров транслируемой программы и использует 11 промежуточных представлений про-

граммы; для каждого представления разработана формальная семантика. Затраты на дедуктивную верификацию программы компилятора CompCert в системе автоматического доказательства Coq [4] – шесть человеко-лет.

Язык предикатного программирования P [5] принципиально сложнее языка Си. Разработка и дедуктивная верификация компилятора для языка P потребовала бы на порядок больше затрат, что за границей текущих возможностей. Есть другая альтернатива – *модельно-ориентированная* технология. Ее применение при разработке компилятора с языка P позволит сертифицировать компилятор высоким уровнем доверия в соответствии со стандартом ГОСТ Р ИСО/МЭК 15408-3 [2].

Модельно-ориентированная технология в системной и программной инженерии базируется на построении и верификации *модели* создаваемого объекта (или программы). В традиционной технологии разработка большой программной системы сопровождается наличием множественных нестыковок и разломов во внутренних интерфейсах системы. В модельно-ориентированной технологии внутренние интерфейсы выстраиваются строго в соответствии с моделью (после ее верификации), что принципиально повышает надежность программной системы. За последние 20 лет сменилось три поколения модельно-ориентированной технологии – от model-based к model-driven, в которой программа генерируется по модели автоматически, далее – к интегрированной модели, единой для создаваемого объекта и встроенного в него программного обеспечения. В модели выделяются различные слои, например, связанные с надежностью, безопасностью и отказоустойчивостью. Реализуется адекватная интеграция различных слоев.

Примером успешного применения модельно-ориентированной технологии является разработка и верификация модели политики безопасности управления доступом в операционных системах [6], в частности, в Astra Linux Special Edition [7]. На языке EventB разработана формальная модель политик безопасности, доказана ее корректность и согласованность. Подтверждена корректность отображения этой модели на ядро ОС Linux размером 2 млн. строк кода. Проведена дедуктивная верификация центральной компоненты Linux Security Module. Данная работа обеспечила высокий уровень доверия к безопасности Astra Linux Special Edition, подтвержденный международной организацией Linux Foundation[1].

Сверхзадачей настоящей работы является построение полной модели компилятора предикатных программ, включающей:

- модель системы типов языка предикатного программирования;
- модель внутреннего представления транслируемой программы;
- модель оптимизирующих трансформаций предикатной программы;
- модель подсистемы дедуктивной верификации.

В настоящей работе кратко описываются модель типов и модель внутреннего представления. Их полное описание имеется в проекте [8]. Целью разработки моделей является построение компактного корректного описания, в котором разные компоненты модели адекватно согласованы между собой. На базе модели планируется разработать новую версию программы компилятора. В дальнейшем при любых изменениях необходимо будет поддерживать точное соответствие между программой и моделью.

Во втором разделе настоящей работы дается краткое описание языка предикатного программирования P [5]. В следующем разделе представлена концепция построения модели внутреннего представления предикатной программы. Далее описывается модель типов языка P [5]. В заключении суммируются основные результаты и анализируются условия, необходимые для реализации предложенного подхода.

[1] http://www.connect-wit.ru/gk-astra-linux-rasshiryaet-sotrudnichestvo-s-the-linux-foundation.html

2. Язык предикатного программирования

Предикатная программа является предикатом в форме вычислимого оператора H(x: y) с аргументами x и результатами y. Программа имеет две основы: логику и вычислимость, причем логика является первичной, главной. Минимальный полный *базис предикатных программ* определен в виде языка **P0**. Предикатная программа определяется следующей конструкцией:

<имя предиката>(<аргументы>: <результаты>) { <оператор> }.

Пусть x, y и z обозначают разные непересекающиеся наборы переменных. Набор x может быть пустым, наборы y и z не пусты. В составе набора переменных x может использоваться логическая переменная e со значениями **true** и **false**. Пусть B и C – имена предикатов, A и D – имена переменных предикатного типа. Операторами являются: *оператор суперпозиции* B(x: z); C(z: y), *параллельный оператор* B(x: y) || C(x: z), *условный оператор* **if** (e) B(x: y) **else** C(x: y), *вызов предиката* B(x: y) и *оператор каррирования*. В табл. 1 представлен полный базис вычислимых предикатов и соответствующих им операторов.

Таблица 1. Вычислимые предикаты и их программная форма

H(x: y) ≅ ∃z. B(x: z) & C(z: y)	H(x: y) { B(x: z); C(z: y) }
H(x: y, z) ≅ B(x: y) & C(x: z)	H(x: y, z) { B(x: y) || C(x: z) }
H(x: y) ≅ (e ⇒ B(x: y)) & (¬e ⇒ C(x: y))	H(x: y) { **if** (e) B(x: y) **else** C(x: y) }
H(x: y) ≅ B(x~ : y)	H(x: y) { B(x~: y) }
H(A, x: y) ≅ A(x: y)	H(A, x: y) { A(x: y) }
H(x: D) ≅ ∀y,z. D(y: z) ≡ B(x, y: z)	H(x: D) { D(y: z) {B(x, y: z)} }
H(A, x: D) ≅ ∀y,z. D(y: z) ≡ A(x, y: z)	H(A, x: D) { D(y: z) {A(x, y: z)} }

Набор x~ составлен из набора переменных x с возможным добавлением имен предикатных программ. Результатом исполнения оператора каррирования D(y: z) {B(x, y: z)} является новый предикат D, получаемый фиксацией значения набора x.

Для языка **P0** построена формальная операционная семантика ℛ(H)(x, y) и доказано тождество ℛ(H) = H [9]. На базе языка **P0** последовательным расширением и сохранением тождества ℛ(H) = H построен язык предикатного программирования **P** [5]:

$$\mathbf{P_0} \to \mathbf{P_1} \to \mathbf{P_2} \to \mathbf{P_3} \to \mathbf{P_4} = \mathbf{P}\ .$$

Каждый очередной язык в этой цепочке строится на базе предыдущего в виде системы обозначений. Одной из задач построения формальной модели внутреннего представления программы является систематическое построение формальной семантики конечного языка **P** прохождением через все цепочки обозначений.

Система типов вносится в язык **P4**. Предыдущие языки – бестиповые. В языке **P** имеется развитая система типов: подтипы, структуры, множества, алгебраические типы, предикатные типы, массивы. Тип массива является предикатным типом, его значения (массивы) являются тотальными и однозначными предикатами. Типы могут быть параметризованы переменными.

Эффективность предикатных программ достигается применением следующих *оптимизирующих трансформаций* [10], переводящих программу на язык императивного расширения P_{imp}:

- замена хвостовой рекурсии циклом;
- открытая подстановка программы на место ее вызова;
- склеивание переменных: замена всех вхождений одной переменной на другую переменную;

- кодирование объектов алгебраических типов (списков и деревьев) с помощью массивов и указателей.

Далее программа конвертируется на язык Си. В языке **P** нет циклов и указателей, вместо них используются рекурсивные программы и алгебраические типы данных.

3. Модель внутреннего представления программы

Внутреннее представление (ВП) программы, получаемое в результате работы front-end'а[2] компилятора, являясь функционально эквивалентной копией исходной программы, ориентировано на удобство работы с программой в middle- и back-end'ах[3]. Опыт разработки компиляторов показывает, что структура ВП программы обычно громоздка и неоправданно сложна, содержит много дублирующей информации. Около тридцать лет назад было большое число работ по технологии трансляции. В настоящее время разработка компилятора считается рутинной задачей. Однако классическая техника разработки компиляторов не решает проблему построения адекватной структуры ВП. В частности, не следует использовать абстрактное синтаксическое дерево программы в качестве основы для ВП. Основу ВП определяют примитивы исполнения, а не синтаксис.

В настоящей работе описывается метод построения модели ВП, абстрагированной от ее реализации на языке C++. Разрабатываемая модель ВП является частью проекта [8].

3.1. Метаязык

Для описания модели ВП используется язык абстрактных структур, содержащий следующие примитивы:

- произвольные неупорядоченные наборы (множества) объектов;
- упорядоченные последовательности объектов;
- вершины (записи) определенного именованного вида с набором полей (атрибутов);
- абстрактные указатели на объекты;
- понятия, определяющие множество структур некоторого вида.

Например,

Prog(Заголовок программы, Область локализации, Тело программы)

определяет вершину с именем Prog и тремя полями. Значение каждого поля определяется соответствующим понятием.

3.2. Архитектура внутреннего представления программы

Программа во ВП определяется в виде дерева структур. Точнее, это граф с учетом дуг, определяемых абстрактными указателями. Для всех языковых конструкций (операторов, выражений, описаний и др.) используется единое унифицированное представление в виде вершин (записей).

Всякий объект, идентифицируемый некоторым именем в тексте программы, представлен во ВП указателем на описание этого объекта. Описание *именованного объекта* представлено во ВП вершиной определенного вида. Имеются разные виды именованных объектов: простые переменные, типы, предикатные программы, формулы и др.

Совокупность имен и соответствующих им описаний одного уровня объединены в структуру вида *область локализации*, определяющую отображение имен на описания. Об-

[2] front-end реализует синтаксический и семантический анализ программы с построением ее ВП.

[3] back-end – языковый процессор, определяющий вторую часть компилятора, обычно – генератор кода. middle-end – языковый процессор, преобразующий ВП программы.

ласть локализации прикрепляется к некоторой конструкции программы. Например, область локализации, содержащая параметры типа в описании типа, прикрепляется к описанию типа.

Иерархия языковых конструкций определяет иерархию областей локализации, на базе которой проводится *идентификация* – определение соответствующего описания для каждого вхождения имени в программе в процессе семантического анализа. Дополнительно, вне иерархии областей локализации, существуют *автономные области локализации* для полей структур и интерфейсных элементов классов, причем для наследуемых классов реализуется независимая иерархия. Приведенная модель именования объектов является адекватной для проведения идентификации, в том числе и для полиморфных объектов.

Структура операторной части предикатной программы определена в виде дерева сегментов. *Сегмент* – фрагмент программы с одним входом и несколькими выходами. Каждый *выход* связан с некоторым входом другого сегмента либо с выходом программы. Отметим, что все виды операторов выбора (**switch**, **case**) [5] преобразуются в условные операторы. Вызов функции заменяется во ВП вызовом предиката и корректным образом выносится из выражения, содержащего вызов функции. В итоге во ВП программа возвращается к языку $\mathbf{P_2}$.

Во ВП устраняются все умолчания, существующие в языке P [5]. Например, вхождение переменной в позиции значения представляется вершиной Value(Переменная). Устраняется полиморфизм операций. Вместо операции «+» используются вершины вида AddNat, AddInt, AddSet, AddLists и др.

4. Анализ типов

Имена объектов и области локализации не нужны для работы с ВП в middle- и back-end'ах. Они необходимы лишь при отображении программы в исходное текстовое представление, а также при диагностике ошибок, где дополнительно требуются координаты (с точностью до символа) по входному тексту программы. Произвольный объект программы идентифицируется указателем на структуру, представляющую описание этого объекта.

Атрибутами большинства объектов программы являются *типы*. Определение типов объектов в соответствии с правилами языка P [5] и проверка разнообразных ограничений на типы требуют нетривиального анализа программы. Анализ типов (type checking), обычно называемый семантическим анализом программы, является наиболее сложной и трудоемкой частью в реализации front-end'а компилятора.

Модель ВП включает модель обширной системы типов языка P. Допускаются подтипы, определяемые как множество истинности некоторого предиката. Типы могут быть параметризованы переменными. Имеются алгебраические типы, представителями которых являются списки и деревья. Рекурсивные типы определяются через аппарат наименьшей неподвижной точки. Предикатные типы, являясь типами второго порядка, определяют типы аргументов и результатов предикатных программ. Массивы определены как частный случай предикатных типов со свойствами тотальности и однозначности.

Тип может быть параметром предикатной программы. Операции с таким типом также поставляются через параметры. Спецификация типа обычно реализуется независимой теорией в стиле алгебраических спецификаций.

Для типов, операций и предикатных программ допускается полиморфизм. Это существенно осложняет идентификацию и анализ типов. В частности, необходимо проводить качественную аппроксимацию значений переменных предикатного типа.

На типах определяются отношения *тождества*, *совместимости* (один тип является подмножеством другого) и *согласованности* (существование общей мажоранты). Эти виды отношений задействованы в правилах семантики языка P для смежных конструкций, в частности, для операндов бинарных операций. Данные виды отношений определяются индуктивно в виде системы правил.

Типовый терм – языковая конструкция, значением которой является тип. Нетривиальна проблема тождества типовых термов.

5. Заключение

Чтобы обеспечить высокий уровень доверия к результатам дедуктивной верификации предикатных программ, необходимы соответствующие гарантии корректности разрабатываемой программы компилятора с языка P. С этой целью разрабатывается модель ВП программы, на базе которой планируется выстраивать все интерфейсы компилятора.

Цель первого этапа – обеспечить адекватность ВП языку P и согласованность частей ВП. Описание структуры ВП в разд. 3.2 и в проекте [8] – это лишь форма (синтаксис) программы, которую необходимо наполнить формализованной семантикой на базе содержательного описания языка P [5]. Следующий шаг – построение формальной операционной семантики ВП программы. Полная модель ВП должна пройти процесс ее верификации.

Еще одной задачей является построение формальной модели системы типов языка P и ее верификация.

Наиболее сложной задачей следующего уровня является построение модели оптимизирующих трансформаций предикатных программ. Необходимо формально доказать корректность применения всех видов оптимизирующих трансформаций.

Практическое применение модельно-ориентированной технологии требует владения формальными методами. Эта дисциплина более десятка лет преподается в МГУ на факультете МВК и в НГУ на кафедре программирования[4].

Литература

1. *Шелехов В. И., Чушкин М. С.* Верификация программы быстрой сортировки с двумя опорными элементами // Научный сервис в сети Интернет. М.: ИПМ им. М. В. Келдыша, 2018. 26 с. URL: http://persons.iis.nsk.su/files/persons/pages/dqsort.pdf (дата обращения: 12.03.2019).
2. ГОСТ Р ИСО/МЭК 15408-3-2013. Информационная технология. Методы и средства обеспечения безопасности. Критерии оценки безопасности информационных технологий. Часть 3. Компоненты доверия к безопасности. М.: Стандартинформ, 2014. 150 с.
3. *Kästner D., Wünsche U., Barrho J., Schlickling M., Schommer B., Schmidt M., Ferdinand C., Leroy X., Blazy S.* CompCert: Practical experience on integrating and qualifying a formally verified optimizing compiler // ERTS 2018: Embedded Real Time Software and Systems. SEE, January 2018. P. 1–9.
4. Coq Proof Assistant [Электронный ресурс]. URL: https://coq.inria.fr/ (дата обращения: 12.03.2019).
5. *Карнаухов Н. С., Першин Д. Ю., Шелехов В. И.* Язык предикатного программирования Новосибирск, 2018. 42 с. URL: http://persons.iis.nsk.su/files/persons/pages/plang14.pdf (дата обращения: 12.03.2019).
6. *Девянин П. Н., Ефремов Д. В., Кулямин В. В., Петренко А. К., Хорошилов А. В., Щепетков И. В.* Моделирование и верификация политик безопасности управления доступом в операционных системах // М.: Горячая линия – Телеком, 2019. 261 с. URL: http://www.ispras.ru/publications/security_policy_modeling_and_verification.pdf (дата обращения: 21.03.2019).
7. Операционные системы Astra Linux [Электронный ресурс]. URL: http://www.astralinux.ru (дата обращения: 12.03.2019).

[4] Видеолекции по формальным методам: http://wasp.iis.nsk.su/

8. *Шелехов В. И.* Проект системы предикатного программирования. Новосибирск: ИСИ СО РАН, 2018. 24 с. URL: http://persons.iis.nsk.su/files/persons/pages/project1.pdf (дата обращения: 12.03.2019).
9. *Шелехов В. И.* Семантика языка предикатного программирования // ЗОНТ-15. Новосибирск, 2015. 13 с. URL: http://persons.iis.nsk.su/files/persons/pages/semZont1.pdf (дата обращения: 12.03.2019).
10. *Каблуков И. В., Шелехов В. И.* Реализация оптимизирующих трансформаций в системе предикатного программирования // Системная информатика. 2017. № 11. С. 21–48. Электрон. журн. 2018. URL: http://persons.iis.nsk.su/files/persons/pages/opttransform4.pdf (дата обращения: 12.03.2019).

Статья поступила в редакцию 24.07.2019;
переработанный вариант – 26.08.2019.

Шелехов Владимир Иванович

к.т.н., зав. лаб. системного программирования, Институт систем информатики имени А. П. Ершова СО РАН (630090, Новосибирск, просп. Ак. Лаврентьева, 6), доцент кафедры программирования НГУ, тел. (383) 330-27-21, e-mail: vshel@iis.nsk.su.

Certified compiler design of predicate programs

V. Shelekhov

A compiler with deductive verification facilities should be thoroughly certified to obtain high level of assurance. The development of the compiler for a predicate program using a compiler model in accordance with a model-based design method makes the compiler to be of a higher level of assurance. The main part of a compiler model is the model of inner program representation for a predicate program. The model is designed in the framework of third release of the compiler. The model architecture for inner representation of predicate program is described. Type checking is specially analyzed as an important aspect in the compiler development. The verification of the model and compiler development on the base of the model should guarantee a high level of compiler reliability.

Keywords: deductive verification, program certification, model-based design.