Модел. и анализ информ. систем. Т. **17**, № **4** (2010) 111–124

УДК 519.681+519.682.1

Пример верификации в проекте F@BOOL@, основанном на булевских решателях

Шилов Н.В.[1]

Институт систем информатики им. А.П. Ершова СО РАН,
Новосибирский государственный университет и
Новосибирский государственный технический университет

e-mail: shilov@iis.nsk.su

получена 26 октября 2010

Ключевые слова: формальная верификация программ, операционная и трансформационная семантика программ, метод верификации Флойда — Хоара, условия корректности, булевские решатели

Верифицирующий компилятор – это системная компьютерная программа, которая транслирует написанные человеком программы с языка высокого уровня в эквивалентные исполнимые программы и, кроме того, доказывает (верифицирует) специфицированные человеком математические утверждения о свойствах транслируемых программ. Цель проекта F@BOOL@ – разработка понятного для пользователей, компактного и переносимого верифицирующего компилятора аннотированных вычислительных программ, использующего эффективные и достоверные автоматические SAT-решатели в качестве средств автоматической проверки истинности условий корректности (вместо средств полуавтоматического доказательства). В период с 2006 по 2009 гг. в проекте F@BOOL@ использовался SAT-решатель zChaff. С его помощью были выполнены первые эксперименты по верификации простых программ на Mini-NIL: программы обмена переменных своими значениями, проверки троек целых чисел быть длинами сторон равностороннего или равнобедренного треугольника, и поиска одной фальшивой среди 15 монет с использованием чашечных весов. В работе рассказано об основных идеях проекта F@BOOL@ и приведены детали эксперимента по верификации программы, решающей головоломку о монетах.

1 Введение

Верифицирующий транслятор — это системная программа, переводящая программы, написанные человеком, в эквивалентные низкоуровневые программы и, кроме

[1]Работа поддержана проектом РФФИ 09-01-00361-а.

того, доказывающая, что обе программы обладают свойствами, специфицированными программистом [13]. Основная задача проекта F@BOOL@ — разработка и экспериментальная реализация простого по устройству (для пользователей) переносимого и расширяемого верифицирующего транслятора F@BOOL@ аннотированных программ, использующего эффективные булевские решатели (т.е. SAT-решатели для проверки выполнимости булевских формул в конъюнктивной нормальной форме) вместо средств автоматического доказательства условий корректности. Целевая группа пользователей F@BOOL@ — это студенты математических и программистских специальностей, изучающих комбинаторные алгоритмы, алгоритмы сортировки и поиска, основы формальных методов спецификации и анализа программ и т.д. F@BOOL@ ориентирован на верификацию свойств функций, вычисляемых программой. Для программ, вычисляющих функции, используются спецификации двух видов: частичной корректности и тотальной корректности. Нас будут интересовать спецификации частичной корректности. Они имеют вид $\{\phi\}\pi\{\psi\}$, где π — это программа, ϕ — предусловие на входные данные, а ψ — постусловие на выходные данные. Условия частичной корректности ещё называют тройками Хоара [5]. Говорят, что такая тройка $\{\phi\}\pi\{\psi\}$ истинна (и пишут $\models \{\phi\}\pi\{\psi\}$) или что программа π частично корректна относительно предусловия ϕ и постусловия ψ, если на любых входных данных, которые удовлетворяют ϕ программа π или не останавливается, или выходные данные удовлетворяют ψ [5]. Неформальный метод (не алгоритм, а именно метод) доказательства истинности троек Хоара впервые был описан в [12] и поэтому носит название метода индуктивных утверждений Флойда. Хорошо известно, что если метод Флойда удалось применить к тройке, то она истинна [5].

Цель настоящей работы — проиллюстрировать на примере подход, принятый в проекте F@BOOL@, для верификации (установления истинности) троек Хоара. Для этого мы подробно разберём один пример верификации с использованием F@BOOL@. Этот пример — программа, решающая следующую известную головоломку о 15 монетах [6]:

> • Есть 14 настоящих монет и одна фальшивая (всего 15 монет). Все настоящие монеты имеют один и тот же вес, а фальшивая монета имеет другой вес. Одна из настоящих монет помечена, а все остальные монеты (включая фальшивую) неотличимы по внешнему виду (то есть известно, что помеченная монета — настоящая). Требуется найти фальшивую монету не более чем за 3 взвешивания.

Сразу заметим, что программа, решающая эту головоломку, может быть верифицирована тестированием, так как полная система тестов для неё — это 28 возможных вариантов для номера фальшивой монетки и её веса относительно настоящей монетки. Но наша цель — проиллюстрировать на примере, как работает F@BOOL@, поэтому для нас не принципиально, что пример можно верифицировать тестированием.

В соответствии с этой целью статья имеет следующую структуру. В части 2 дан набросок основных идей и подходов проекта F@BOOL@ (по состоянию на 1 июня 2010 г.). В следующей части 3 определены синтаксис и операционная семантика малого шага для языка программирования mini-NIL(R, A); этот язык является расши-

рением базового языка программирования mini-NIL[2] посредством диапазонов значений переменных (R - "range") и статических массивов (A — "array"); в этой же части дан набросок трансформационной семантики языка mini-NIL(R, A), которая осуществляет перевод программ с mini-NIL(R, A) на базовый язык mini-NIL. В части 4 обсуждаются шаги верификации программы, решающей головоломку о 15 монетах: сначала мы трансформируем аннотированную программу на языке mini-NIL(R, A) в аннотированную программу на языке mini-NIL, по ней строим условие корректности, подлежащее проверке булевским решателем (обе программы и условие корректности вынесены в приложения). Завершает статью часть 5, в которой обсуждаются возможные направления для дальнейших исследований.

2 Экскурсия по F@BOOL@

Mini-NIL — это низкоуровневый недетерминированный язык программирования, напоминающий "классический" Basic. Он подробно документирован в [2, 3]. Каждая программа на этом языке снабжена преамбулой, в которой определяется максимальное целое значение (обычно вида 2^n для некоторого $n > 0$) и инициализированы все переменные значениями из этого интервала. Сами программы состоят из помеченных операторов присваивания и условных операторов. Все операции в программе выполняются в аддитивном упорядоченном кольце вычетов по модулю максимального целого, определённого в преамбуле. Исполнение программы начинается с инициализации переменных, затем управление передаётся оператору с меткой 0, передача управления происходит через недетерминированный переход *goto*, *then* и *else*, исполнение программы заканчивается при передаче управления любой выходной метке программы (которая встречается, но не метит ни одного оператора). Синтаксис mini-NIL имеет очень жесткий формат, т. к. это внутренний язык системы, её "автокод", а не язык высокого уровня. В аннотированных mini-NIL-программах преамбула, все выходные метки и некоторые метки операторов снабжены логическими условиями, т. н. "инвариантами" этих "точек", инвариант преамбулы называется предусловием, а инвариант выходных меток - постусловием программы. В качестве инвариантов можно использовать формулы логики первого порядка над упорядоченным кольцом вычетов по модулю максимального числа. Неформально говоря, инварианты точек — это "контракты времени исполнения":

1. предусловие проверяется на начальных (инициализированных) значениях переменных, и, если оно оказывается неверным, возникает исключение "ошибка во входных данных";

2. постусловие проверяется на результатах вычислений при достижении любой выходной метки, и, если оно оказывается неверным, возникает исключение "ошибка в результатах";

3. инвариант, сопоставленный метке, проверяется на текущих значениях переменных каждый раз при достижении этой метки, и, если он оказывается невер-

[2]NIL — это акроним для "Non-deterministic Imperative Language".

ным, возникает исключение “ошибка времени исполнения”.

Пример части аннотированной mini-NIL-программы дан в приложении B.

Статическая семантика аннотированных mini-NIL-программ — это следующий алгоритм построения условий корректности, представляющий собой вариант метода Флойда генерации условий корректности.

Предусловие. [В аннотированной mini-NIL-программе π метки всех операторов различны и в каждом условном операторе множества $then$-меток и $else$-меток не пересекаются]

1. Представить π в виде блок-схемы с метками, аннотированными их инвариантами.

2. Если в блок-схеме есть циклический участок, не содержащий ни одной аннотированной метки, то статическая семантика такой программы не определена; в противном случае — перейти к следующему шагу.

3. Для каждой аннотированной метки l построить следующее условие корректности

$$\xi_l \to \Big(\bigwedge_{\substack{k \text{ — аннотированная метка,} \\ \xi_k \text{ — её инвариант, а} \\ \pi_l^k \text{ — простой путь по блок-схеме} \\ \text{из } l \text{ до } k}} WP(\pi_l^k, \xi_k)\Big),$$

   где ξ_l — это инвариант метки l, а WP — генератор слабейшего предусловия для ациклических программ [4].

4. Множество всех построенных условий корректности является статической семантикой π.

Постусловие. [Для любого начального состояния σ программы π, если предусловие верно в σ, а все условия корректности тождественно истинны, то постусловие верно в любом заключительном состоянии, которое достижимо из σ.]

Доказательство непротиворечивости статической семантики дано в препринте [3].

Таким образом, под верификацией в проекте F@BOOL@ мы понимаем генерацию и проверку условий корректности для mini-NIL-программ. Однако следует заметить, что описанный прямой метод генерации условий корректности имеет экспоненциальную оценку сложности (по памяти и времени) из-за ветвления в $if-then-else$ и недетерминизма, а также из-за кратных вхождений переменных в выражения. Поэтому в работе [7] был разработан и обоснован другой полиномиальный алгоритм генерации условий корректности. Этот алгоритм является полным, но использует вспомогательные неинтерпретированные предикатные символы. Его сложность линейно зависит от числа операторов в программе (как это видно из сравнения аннотированной программы в приложении B и условия корректности в приложении C), но квадратично зависит от общего размера аннотированной программы (т. е. и от размера аннотаций). Именно этот алгоритм использован для

ручной генерации условий корректности для программы, решающей головоломку о 15 монетах в части 4). Однако этот алгоритм пока не реализован.

Ключевая идея проекта F@BOOL@ — это явное булевское представление всех данных (вместо булевской абстракции) и использование только булевских решателей (вместо булевских решателей в паре с разрешающими процедурами для проблемно-ориентированных теорий или программ поиска доказательства). В этом принципиальное отличие F@BOOL@ от проектов BLAST [11] и SLAM [10]. Статические анализаторы BLAST и SLAM ориентированы на подмножество языка Си и работают по одной схеме: строят конечную модель пространства конфигураций программы при помощи так называемой булевской абстракции, проверяют свойства безопасности и живости построенной модели при помощи булевских решателей, и, если эти свойства нарушены, то проверяют соответствующие контрпримеры на корректность при помощи разрешающих процедур или программ поиска доказательства.

Наоборот, в проекте F@BOOL@ мы верифицируем не свойства безопасности и живости, а функциональные свойства, генерируем условия корректности первого порядка, переводим их в булевскую форму и проверяем их при помощи булевских решателей. Для перевода условий корректности в булевскую форму используется следующий метод.

Предусловие. [θ — это формула первого порядка над аддитивной упорядоченной группой вычетов по модулю $2^n > 1$.]

1. $\xi := qbool_n(\theta)$, где $qbool_n$ — это эквивалентный перевод формул первого порядка в квантифицированные булевские формулы за счёт представления каждого значения/переменной из/над кольцом вычетов по модулю 2^n посредством n-мерного вектора булевских значений/переменных;

2. $\zeta := bool_n(\xi)$, где $bool_n$ — это эквивалентный перевод квантифицированных булевских формул в булевские формулы за счёт представления каждого квантора $\exists$ и всех кванторов $\forall$, находящихся в области действия кванторов существования, при помощи дизъюнкции и конъюнкции;

3. $\chi := cnf_3(\neg\zeta)$, где cnf_3 — это алгоритм трансформации булевской формулы в равновыполнимую 3-кнф формулу [1].

Постусловие. [Булевская формула χ выполнима тогда и только тогда, когда условие корректности θ тождественно истинно.]

Заметим, что шаг 1 этого метода имеет квадратичную временную и пространственную сложность (от размера θ), шаг 2 — экспоненциальную временную и пространственную сложность (от размера ξ), а шаг 3 — квадратичную временную, но линейную пространственную сложность (от размера ζ). Поэтому в настоящий момент в проекте F@BOOL@ запрещено использование кванторов существования в аннотациях; в этом случае описанный метод имеет полиномиальную оценку сложности по времени и памяти.

Что касается булевского решателя, то в период с 2006 по 2009 год в проекте использовался zChaff[3]. Всего было выполнено три эксперимента по верификации программ, которые перечислены ниже.

[3]http://www.princeton.edu/ chaff/zchaff.html

- Программа обмена значениями двух переменных.
- Программа проверки, что три числа являются длинами сторон равностороннего или равнобедренного треугольника.
- Программа, решающая головоломку о 15 монетах.

3 Уровни языка NIL

Как уже было сказано, синтаксис, операционная и трансформационная семантика mini-NIL(R, A) определены в [14]. Ниже дан краткий набросок синтаксиса, операционной и трансформационной семантики mini-NIL(R, A).

Язык mini-NIL(R, A) состоит из программ. Каждая программа состоит из преамбулы и тела. Преамбула состоит из объявления максимального целого, описаний переменных и массивов. Тело программы состоит из помеченных операторов присваивания переменным, операторов обновления элемента массива и условных операторов.

Преамбула программы. Объявление максимального целого числа имеет вид '$MaxInt :: M$', где M — положительное десятичное целое без знака. Описание переменной имеет вид '$VAR\ x :\ [0..r]$', где x — идентификатор (т. е. последовательность строчных букв латинского алфавита и арабских цифр, начинающаяся с буквы), а r — десятичное целое без знака из диапазона $[0..M]$ (который называется границей её области изменения). Описание массива имеет вид '$ARRAY\ a[r_1,\ ...r_n] :\ [0..r]$', где a — идентификатор, n — положительное целое число[4], а $r_1,\ ...r_n, r$ — положительные десятичные целые числа без знака из диапазона $[0..M]$; числа $r_1,\ ...r_n$ называются границами диапазона изменения индексов, а число r — границей области изменения элементов. Описание идентификатора — это описание переменной или массива с этим идентификатором. Преамбула программы — это конечная последовательность, начинающаяся с объявления максимального числа, а далее состоящая из описаний переменных и массивов такая, что каждый описанный идентификатор описан в ней ровно один раз.

Арифметические выражения и элементы массивов. Арифметические выражения и элементы массивов определяются взаимной индукцией[5].

Арифметические выражения:

- всякое десятичное целое число без знака из диапазона $[0..M]$ является (простым) выражением;
- каждая переменная является (простым) выражением;
- каждый элемент массива является (составным) выражением;
- всякая сумма или разность выражений является (составным) выражением;

[4] любое, не обязательно из диапазона $[0..M]$

[5] В дальнейшем мы часто будем для краткости говорить 'выражение' и 'элемент' вместо 'арифметическое выражение' и 'элемент массива'.

Элементы массивов имеют вид '$a[\tau_1, ...\tau_n]$', где a — это массив, n — положительное целое, а $\tau_1, ...\tau_n$ — арифметические выражения (для индексов элемента).

Тело программы. Метка — это десятичное целое число без знака (т. е. 0, 1, 2, ...). Оператор присваивания имеет вид '$l:\ x := \tau\ goto\ L$', где l — метка, x — переменная, τ — арифметическое выражение, а L — конечное множество меток[6]. Оператор обновления элемента массива имеет вид '$l:\ a[\tau_1, ...\tau_n] := \tau\ goto\ L$', где l — метка, $a[\tau_1, ...\tau_n]$ — элемент массива, τ — арифметическое выражение, а L — конечное множество меток[6]. Условный оператор имеет вид '$l:\ if\ \xi\ then\ L^+\ else\ L^-$', где l — метка, ξ — бесквантерная формула, построенная из равенств и неравенств арифметических выражений, L^+ и L^- — конечные множества меток[6]. Тело (программы) — это конечное множество операторов[7] такое, что каждая метка метит не более одного оператора. Метка '0' (ноль) называется входом или началом программы. Выходная метка, или конец программы, — это любая метка, которая встречается[8] в программе, но не метит ни одного оператора программы.

Программа. Преамбула и тело совместны, если все переменные и массивы, которые используются в теле, описаны в преамбуле, и все выражения в теле являются корректными арифметическими выражениями в соответствии с описаниями в преамбуле. Программа состоит из совместных преамбулы и тела. Если π — программа, то будем обозначать ее преамбулу $P(\pi)$, а тело — $B(\pi)$.

Таким образом синтаксис языка Mini-NIL(R, A) полностью определён. Можно считать, что этот язык является расширением языка mini-NIL [2] за счёт добавления статических массивов и различных областей изменения для разных переменных.

Операционная семантика mini-NIL(R, A) является примером семантики малого шага (Small Step Semantics) и расширяет операционную семантику mini-NIL. Неформально говоря, исполнение mini-NIL(R, A)-программы начинается передачей управления оператору, помеченному как вход (т. е. меткой '0'), и завершается передачей любой выходной метке. Исключительные ситуации во время исполнения возникают, когда переменной присваивается неопределённое значение, когда обновляется неопределённый элемент массива (т. е. элемент с неопределённым индексом[9], или когда не определена следующая метка (т. е. управление передаётся пустому множеству меток).

Семантика[10] малого шага для mini-NIL(R, A). Пусть π — произвольная программа на языке mini-NIL(R, A). Шаг (точнее — малый шаг) программы π — это срабатывание любого оператора данной программы[11]. Состояние программы π — это произвольное отображение, которое сопоставляет каждой переменной, описанной в программе, значение из её области изменения, а каждому массиву, описанному в программе, — частичную функцию из декартового произведения диапазонов изменения индексов этого массива в его область изменения элементов. Конфигурация

[6]Возможно, пустое множество.

[7]т.е. присваиваний, обновлений элементов и условных операторов.

[8]в '*goto*'-, '*then*'-, или '*else*'-множестве меток какого-либо оператора.

[9]но элемент с определённым индексом может принимать неопределённое значение.

[10]Подробнее см. в работе [14]

[11]Для сравнения: семантика большого шага допускает срабатывание сразу нескольких или даже всех операторов программы, которые не конфликтуют из-за доступа к данным.

программы π — это произвольная пара, состоящая из метки, которая встречается в программе, и состояния. Начальная конфигурация π — это конфигурация с меткой 0. Заключительная конфигурация π — это произвольная конфигурация с выходной меткой программы. Трасса π — это произвольная последовательность конфигураций, в которой любая пара соседних конфигураций — это шаг этой программы. Вычислительная трасса π — это трасса, которая начинается начальной конфигурацией, а заканчивается заключительной конфигурацией. Семантика малого шага программы π — это следующее бинарное отношение $SSS(\pi)$ на пространстве состояний π: $SSS(\pi) = \{(\sigma' , \sigma'') \in \Sigma \times \Sigma : $ существует вычислительная трасса π, которая начинается с состояния σ' и заканчивается состоянием $\sigma''\}$. Это определение совместно с операционной семантикой языка mini-NIL [2].

В работе [9] предложено выделять в компьютерных языках несколько "уровней": базовый, несколько промежуточных и полный. Базовый уровень языка должен иметь семантику виртуальной машины и быть достаточно богатым, чтобы реализовать конструкции промежуточных уровней посредством трансформаций программ с промежуточного уровня в базовый.

В проекте F@BOOL@ язык mini-NIL выполняет роль базового уровня, а язык mini-NIL(R, A) — первый пример языка промежуточного уровня, для которого должен быть определён алгоритм λ, консервативно (т.е. с "сохранением" семантики) трансформирующий программы с языка mini-NIL(R, A) в программы на языке mini-NIL, $\lambda : \pi \mapsto \lambda(\pi)$, так, что семантика малого шага $SSS(\pi)$ "совпадает" с операционной семантикой $\{(\sigma' , \sigma'') \in \Sigma \times \Sigma : \sigma'' \in \lambda(\pi)(\sigma') \}$. Трансформация λ состоит из 3 стадий, подробно описанных и обоснованных в [14]: упрощение, элиминация массивов и унификация областей.

1. Говоря неформально, упрощение — это замена всякого индекса, заданного составным выражением τ, у элемента массива в программе на новую переменную y и добавление дополнительного оператора присваивания $y := \tau$ перед местом употребления этого элемента массива. В результате получается т. н. простая программа.

2. Элиминация массивов в простой программе состоит в замене статических массивов набором новых переменных, например, массив $ARRAY\ a[2] : [0..5]$ заменяется на три новых переменных $(VAR\ x0 : [0..5])$, $(VAR\ x1 : [0..5])$, $(VAR\ x2 : [0..5])$, которые будут представлять элементы массива $a[0]$, $a[1]$ и $a[2]$, когда они определены, и ещё три новых переменных $(VAR\ y0 : [0..1])$, $(VAR\ y1 : [0..1])$, $(VAR\ y2 : [0..1])$, позволяющих моделировать ситуацию определённости или неопределённости соответствующих элементов $a[0]$, $a[1]$ и $a[2]$.

3. Идея унификации областей тоже совершенно прозрачная: если область изменения r какой-либо переменной x меньше $MaxInt$, то перед каждым присваиванием этой переменной $x := \tau$ надо выполнить дополнительную проверку, что значение τ попадает в диапазон $[0..r]$.

4 Верификация примера

Головоломка о 15 монетах — нетривиальная для человека и крайне поучительная с точки зрения преподавания программирования[12]. Поэтому здесь будет уместно привести её "человеческое" решение.

Для первого взвешивания разделим монеты на три кучки по пять монет в каждой: в первой кучке четыре "неопознанных" монеты и "эталон", а во второй и третьей кучках — по пять "неопознанных" монет. В результате первого взвешивания возможно три исхода: (<) первая чашка легче, (>) вторая чашка легче, и (=) чашки равны по весу. Рассмотрим вариант (=). В этом фальшивая монетка — в третьей кучке, а все монетки из первой и второй кучек — настоящие (их 10 штук). Тогда для второго взвешивания возьмём любую монету из третьей кучки (назовём её "а") и "эталон" на первую чашку весов, а любые две из оставшихся монет из третьей кучки (назовём их "б" и "с") — на вторую чашку весов. В результате такого второго взвешивания опять возможно три исхода: (=<) первая чашка легче, (=>) вторая чашка легче, и (==) чашки равны по весу . В случае (==) фальшивая монетка — одна из двух монеток третьей кучки, которая не участвовала во втором взвешивании; найти её за одно (третье) взвешивание не составляет труда, если использовать любую настоящую монету (из 13 имеющихся). В случае (<=) фальшивая монетка — одна из "а", "б" и "с"; тогда третьим взвешиванием сравним монетки "а" и "б" с любыми двумя настоящими монетками (из 12 имеющихся): если вес равен, то фальшивая монетка "с", если "а" и "б" легче настоящих — то фальшивая монетка "а", иначе — фальшивая монетка "б". Случай (=>) разбирается аналогично. Поиск фальшивой монетки в случае исхода (=) первого взвешивания разобран. Рассмотрим вариант (<). В этом фальшивая монетка — одна из 9 "неопознанных" монет на весах, а все монеты из третьей кучки — настоящие (их 5 штук). Тогда для второго взвешивания отделим любую из "неопознанных" монеток (назовём её "а") из первой кучки, отделим любые две монеты из второй кучки (назовём их "б" и "с"), оставшиеся 6 "неопознанных" монет из двух первых кучек положим на первую чашку весов, все 6 настоящих (5 из третьей кучки и "эталон") — на вторую чашку весов. В результате такого второго взвешивания опять возможно три исхода: (<<) первая чашка легче, (<>) вторая чашка легче, и (<=) чашки равны по весу. В случае (<=) фальшивая монетка — одна из "а", "б" и "с"; но тогда соотношение между "а + эталон" и "б + с" наследуется (то же самое), что результат первого взвешивания (т. е. <), поэтому во время третьего взвешивания поступаем как во время третьего взвешивания в случае (=<). В случае (<<) нам известно, что фальшивая монета легче настоящей, и, следовательно, искать её нужно среди трёх "неопознанных" монет, которые в первом взвешивании лежали на первой чашке весов, а найти лёгкую монетку среди трёх монет за одно (третье) взвешивание — просто. И, наконец, случай (<>) аналогичен предыдущему, только теперь фальшивая монетка тяжелее настоящих и искать её следует среди трёх монет, которые в первом взвешивании лежали на

[12]Подробно "история" с этой головоломкой рассказана в статье [6]. В той же работе [6] параметризованный вариант головоломки использован для введения динамической логики с неподвижными точками. А в работе [8] параметризованный вариант головоломки использован для сравнения трёх разных парадигм программирования — логического, функционального и императивного.

второй чашке весов. Поиск фальшивой монетки в случае исхода ($<$) первого взвешивания разобран. Вариант ($>$) разбирается аналогично только что разобранному случаю ($<$).

Часть аннотированной mini-NIL(R, A)-программы, реализующей описанный метод, приведена в приложении A. Результат трансформации этой аннотированной программы в аннотированную программу на языке mini-NIL частично приведён в приложении B. Эта трансформация была выполнена вручную в соответствии с [14], но с упрощённой элиминацией массивов (использована посылка из предусловия, что все элементы массива M[0..14] определены). Часть условия корректности θ для этой аннотированной mini-NIL-программы дана в приложении C, оно было построено вручную в соответствии с полиномиальным методом из [7]). В соответствии с описанием процесса верификации в проекте F@BOOL@, условие $cnf_3(\neg(bool_4(\theta)))$ было проверено на выполнимость при помощи решателя zChaff.

5 Заключение

В данной статье был представлен обзор проекта F@BBOL@, рассказано о состоянии дел и приведён пример верификации с использованием принятого подхода. Можно заметить, что многие шаги верификации в этом примере выполнены вручную. Поэтому первостепенная задача на ближайшее будущее — автоматизация всех шагов верификации в проекте. Конкретно речь идёт о следующих шагах.

1. Реализовать конвертор аннотированных mini-NIL(R, A)-программ в аннотированные mini-NIL программы в строгом соответствии с трансформационной семантикой mini-NIL(R, A) [14].

2. Реализовать полиномиальный генератор условий корректности, описанный в работе [7].

Разумеется, что в повестке дня стоит задача верификации новых, более интересных и сложных примеров.

Однако есть ещё одна задача, на решении которой хочется сосредоточиться в рамках проекта F@BOOL@. Речь идёт об автоматизации метатеории проекта, т. е. автоматизации доказательств о свойствах трансформаций программ при помощи инструментальных средств для автоматического доказательства теорем. Пожалуй, лучше всего мотивировать важность такого исследования следующей цитатой из объявления о 4^{rd} ACM SIGPLAN Workshop on Mechanizing Metatheory[13]:

> Researchers in programming languages have long felt the need for tools to help formalize and check their work. With advances in language technology demanding deep understanding of ever larger and more complex languages, this need has become urgent. There are a number of automated proof assistants being developed within the theorem proving community that seem ready or nearly ready to be applied in this domain. Yet, despite numerous individual

[13] `http://www.seas.upenn.edu/~sweirich/wmm/`

efforts in this direction, the use of proof assistants in programming language research is still not commonplace: the available tools are confusingly diverse, difficult to learn, inadequately documented, and lacking in specific library facilities required for work in programming languages.

Список литературы

1. Ахо А., Хопкрофт Дж., Ульман Дж. Построение и анализ вычислительных алгоритмов. М.: Мир. 1979.

2. Бодин Е.В., Калинина Н.А., Шилов Н.В. Проект верифицирующего компилятора F@BOOL@ Часть I: Общее описание проекта F@BOOL@, его место в компонентном подходе к программированию. Язык Mini-NIL — прототип языка виртуальной машины проекта. Препринт №131 Института систем информатики им. А.П. Ершова СО РАН, 2005.

3. Бодин Е.В., Калинина Н.А., Шилов Н.В. Проект верифицирующего компилятора F@BOOL@. Часть II: Логические аннотации в языке Mini-NIL, их статическая семантика и семантика времени исполнения. Препринт №138 Института систем информатики им. А.П. Ершова СО РАН, 2006.

4. Дейкстра В.Э. Дисциплина программирования. М.: Мир, 1978.

5. Грис Д. Наука программирования. М.: Мир, 1984.

6. Шилов Н.В., Бодин Е.В. Ии И. О программных логиках — просто // Системная информатика, выпуск 8. Новосибирск: Наука, 2002. С. 206–249.

7. Шилов Н.В., Ануреев И.С., Бодин Е.В. О генерации условий корректности для императивных программ // Программирование. 2008, №6. С. 1–20.

8. Шилов Н.В. Заметки о трёх парадигмах программирования // Компьютерные инструменты в образовании. 2010, №2. С. 24–37.

9. Anureev I.S., Bodin E.V., Gorodnyaya L.V., Marchuk A.G., Murzin F.A., Shilov N.V. *On the Problem of Computer Language Classification* // Joint NCC&IIS Bulletin, Series Computer Science. 2008. Vol. 28. P. 1–29.

10. Ball T., Cook B., Levin V., and Rajamani S. K. *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft* // Springer-Verlag, Berlin, LNCS. 2004. Vol. 2999. P. 1–20.

11. Beyer D., Henzinger T.A., Jhala R., and Majumdar R. *The Software Model Checker Blast: Applications to Software Engineering* // Int. Journal on Software Tools for Technology Transfer. 2007. N 9. P. 505–525.

12. Floyd R.W. *Assigning meanings to programs.* Proc. of a Symposium in Applied Mathematics. Mathematical Aspects of Computer Science, Vol. 19. American Math. Society, Providence, R. I., 1967. P. 19–32.

13. Hoare C. A. R. *The Verifying Compiler: A Grand Challenge for Computing Research.* Perspectives of Systems Informatics (PSI'2003), Springer-Verlag, Berlin, LNCS. Vol. 2890. 2003. P. 1–12.

14. Shilov N.V., Bodin Eu.V, Shilova S.O. *Fabulous arrays I: Operational and transformational semantics of static arrays in vericationproject F@BOOL@.* Bull. Nov. Comp. Center, Comp. Science. Vol. 29. 2009. P. 121–140.

A Фрагмент mini-NIL(R, A)-программы, решающей головоломку

```
MaxInt :: 14 ; // Maximal Integer.
VAR F : [0..14] ; // Program variable.
ARRAY M[14] : [0..2] ; // Program array.
VAR I : [0..14] ; VAR J : [0..14] ; // Quantifier variables.
(M[0] = 1 & E I.((M[I] = 0 V M[I] = 2) & A J.(J I => M[J] = 1)))
                                                        // Precondition.
: // Program body begin.
0: if M[0] + M[1] + M[2] + M[3] + M[4] = M[5] + M[6] + M[7] + M[8] + M[9]
                                                      then {1} else {10}
1: if M[0] + M[10] = M[11] + M[12] then {2} else {5}
2: if M[0] = M[13] then {3} else {4}
3: F:= 14 goto {33}
4: F:= 13 goto {33}
5: if M[0] + M[1] = M[10] + M[11] then {6} else {7}
.....................................................
.....................................................
28: if M[0] + M[10] = M[1] + M[8] then {29} else {30}
29: F:= 9 goto {33}
30: if M[0] + M[10] < M[1] + M[8] then {31} else {32}
31: F:= 1 goto {33}
32: F:= 8 goto {33}
: // Program body end.
M[F] ≠ 1. // Postcondition.
```

B Фрагмент mini-NIL-программы, решающей головоломку

```
14 ; // Maximal Integer.
0; 1; 0; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 0; 0;
    // ‘‘Random’’ initial values for F, M0, ... M14, I, J that meet precondition.
(M0 = 1 & (
(M0 = 0 V M0 =2 ) & ((0 ≠ 0 => M0 = 1) & (1 ≠ 0 => M1 = 1) & ...
```

```
                                                    & (14 ≠ 0 => M14 = 1)) V
(M1 = 0 V M1 =2 ) & ((0 ≠ 1 => M0 = 1) & (1 ≠ 1 => M1 = 1) & ...
                                                    & (14 ≠ 1 => M14 = 1)) V
.....................................................................
(M14 = 0 V M14 =2 ) & ((0 ≠ 14 => M0 = 1) & (1 ≠ 14 => M1 = 1) & ...
                                                  & (14 ≠ 14 => M14 = 1)) ))
                                                           // Precondition.
: // Program body begin.
0: if M0 + M1 + M2 + M3 + M4 = M5 + M6 + M7 + M8 + M9 then {1} else {10}
1: if M0 + M10 = M11 + M12 then {2} else {5}
2: if M0 = M13 then {3} else {4}
3: F:= 14 goto {33}
4: F:= 13 goto {33}
5: if M0 + M1 = M10 + M11 then {6} else {7}
.....................................................
.....................................................
28: if M0 + M10 = M1 + M8 then {29} else {30}
29: F:= 9 goto {33}
30: if M0 + M10 < M1 + M8 then {31} else {32}
31: F:= 1 goto {33}
32: F:= 8 goto {33}
: // Program body end.
( (F = 0 => M0 ≠ 1) V (F = 1 => M1 ≠ 1) V ... (F = 14 => M14 ≠ 1) ).
                                                                // Postcondition.
```

C Фрагмент условия корректности θ для mini-NIL-программы, решающей головоломку

Конъюнкция следующих формул ini, 0–32 и fin.

```
ini: precondition from Appendix B => P0
0: ( (M0 + M1 + M2 + M3 + M4 = M5 + M6 + M7 + M8 + M9) <=> Q0 ) =>
                                                ( P0 => (Q0 & P1) V (∼Q0 & P10) )
1: ( (M0 + M10 = M11 + M12) <=> Q1 ) => ( P1 => (Q1 & P2) V (∼Q1 & P5) )
2: ( (M0 = M13) <=> Q2 ) => ( P2 => (Q2 & P3) V (∼Q2 & P4) )
3: P3 => (P33 & F=14)
4: P4 => (P33 & F=13)
5: ( (M0 + M1 = M10 + M11) <=> Q5 ) => ( P5 => (Q5 & P6) V (∼Q5 & P7) )
.....................................................
.....................................................
28: ( (M0 + M10 = M1 + M8) <=> Q28 ) => ( P28 => (Q28 & P29) V (∼Q28 & P30) )
29: P29 => (P33 & F=9)
30: ( (M0 + M10 < M1 + M8) <=> Q30 ) => ( P30 => (Q30 & P31) V (∼Q30 & P32) )
31: P31 => (P33 & F=1)
32: P32 => (P33 & F=8)
```

```
fin: P33 => postcondition from Appendix B
```

F@BOOL@: experiment with a simple verifying compiler based on SAT-solvers

Shilov N.V.

Keywords: formal program verification, operational and transformational semantics, Floyd — Hoare proof technique, correctness conditions, SAT-solvers

A verifying compiler is a system computer program that translates programs written by man from a high-level language into equivalent executable programs, and besides, proves (verifies) mathematical statements specified by man about the properties of the programs being translated. The purpose of the F@BOOL@ project is to develop a transparent for users, compact and portable verifying compiler F@BOOL@ for annotated computational programs, that uses effective and sound automatic SAT-solvers (i.e. programs that check satisfiability of prepositional Boolean formulas in the conjunctive normal form) as means of automatic validation of correctness conditions (instead of semi-automatic proof techniques). The key idea is Boolean representation of all data instead of Boolean abstraction or first-order representation. (It makes difference between F@BOOL@ and SLAM.) Our project is aimed at the verification of functional properties, and it assumes generation of first-order verification conditions (from invariants), and the validation/refutation of each verification condition using SAT-solvers after "conservative" translation of the verification conditions into Boolean form. During the period from 2006 to 2009, a popular (at that time) SAT-solver zChaff was used in the F@BOOL@ project. The first three verification experiments that have been exercised with its help are listed below: swapping values of two variables, checking whether three input values are lengths of sides of an equilateral or isosceles triangle, and detecting a unique fake in a set of 15 coins. The paper presents general outlines of the project and details of the last (the most extensive) experiment.

Сведения об авторе:
Шилов Николай Вячеславович, канд. физ.-мат. наук,
ст. науч. сотр. Института систем информатики им. А.П. Ершова СО РАН,
доцент Новосибирского государственного университета
и Новосибирского государственного технического университета.