Российская академия наук
Сибирское отделение
Институт систем информатики
имени А. П. Ершова

Е. В. Бодин, Н. А. Калинина, Н. В. Шилов

ПРОЕКТ ВЕРИФИЦИРУЮЩЕГО КОМПИЛЯТОРА F@BOOL@

Часть II: Логические аннотации в языке Mini-NIL, их статическая семантика и семантика времени исполнения

Препринт
138

Новосибирск 2006

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 прозрачного для пользователей, компактного и переносимого верифицирующего компилятора, использующего эффективные и достоверные разрешающие процедуры в качестве средств автоматической проверки истинности условий корректности (вместо средств полуавтоматического доказательства) аннотированных вычислительных программ. Предполагается, что основными средствами проверки в F@BOOL@ будут автоматические SAT-решатели (т.е. программы для проверки выполнимости пропозициональных булевских формул в конъюнктивной нормальной форме). Целевая группа пользователей системы F@BOOL@ — студенты математических и программистких специальностей, желающие понять принципы верификации программ и приобрести первоначальный практический опыт по верификации.

Данный препринт является второй частью документации по проекту F@BOOL@. Он содержит неформальное введение в верификацию нерезидентных программ по Р. Флойду, формальное описание синтаксиса, статической и динамической (времени исполнения) семантики логических аннотаций для программ на языке Mini-NIL (который является прототипом языка виртуальной машины в проекте F@BOOL@).

Работа поддержана грантами РФФИ № 05-07-90162 и 06-01-00464-а.

© Институт систем информатики имени А. П. Ершова СО РАН, 2006

Siberian Division of the Russian Academy of Sciences
A. P. Ershov Institute of Informatics Systems

E. Bodin, N. Kalinina, N. Shilov

VERIFYING COMPILER F@BOOL@

Part II: Logical annotations in Mini-NIL, their static and run-time semantics

Preprint
138

Novosibirsk 2006

Verifying Compiler is a system program that proves specified program properties besides translation of a computer program.

Project F@BOOL@ is aimed to development of a transparent, compact, and portable verifying compiler. It is assumed that F@BOOL@ will exploit efficient decision procedures for validation of correctness conditions (instead of semi-automatic theorem provers). In particular, F@BOOL@ will extensively use SAT-solvers for validation of Boolean encoding of correctness conditions. The target group of F@BOOL@ users comprises Computer Science, Information Technology and Mathematics students willing to comprehend program verification.

The technical report is the second part of F@BOOL@ documentation. It presents informal introduction to Floyd-style verification of program, formal syntax, static and run-time semantics of Logical Annotations for a toy programming language Mini-NIL. (This language is a prototype of F@BOOL@ virtual machine.)

© A. P. Ershov Institute of Informatics Systems, 2006

НЕФОРМАЛЬНОЕ ВВЕДЕНИЕ В ФОРМАЛЬНУЮ СПЕЦИФИКАЦИЮ ПРОГРАММ

Цель этого раздела документации — дать интуитивно-ясное представление о том, что такое логические аннотации и какова их роль в верификации. Поэтому в этом разделе вместо конкретного языка программирования будет использоваться так называемый «псевдокод». Вообще говоря, псевдокод не имеет строго формального определения, однако, можно перечислить ряд общих требований (носящих рекомендательный характер) на используемые в псевдокоде программные конструкции и типы данных. Ниже перечислен вариант этих требований.

1. Программа на псевдокоде включает в себя определения типов данных и описания переменных, а также тело программы.
2. Каждый тип данных определяется набором значений и операций, в которых значения этого типа могут участвовать в качестве аргументов или результатов.
   - 2.1) в качестве значений типов данных разрешается использовать обычные математические объекты — булевские значения (TRUE и FALSE), числа (целые, рациональные, вещественные и т.п.), точки (на прямой, плоскости и пространстве), геометрически определенные линии и плоскости, графы (и их вершины и ребра), множества математических объектов и отображения множеств, и т.д.;
   - 2.2) в качестве конструкторов новых типов разрешается использовать математическую конструкцию множества и программистскую конструкцию массива[1];
   - 2.3) для всех типов данных все используемые операции должны быть определены в максимально формальной (математической) форме.
3. Для каждой переменной должен быть определен ее тип.
4. Тело программы состоит из операторов присваивания (записанных с использованием Паскалевского значка «:=»), при исполнении которых не возникает побочных эффектов, а только изменяется значение переменной, и операторов вызова процедур; тело программы строится при помощи следующих конструкций.
   - 4.1) «;» для последовательного исполнения,

[1] А также некоторые другие конструкции, например, очередь и список, магазин и стек.

4.2) «IF-THEN-ELSE» для выбора по условию,

4.3) «WHILE-DO-OD» и «DO-UNTIL» для итерации по условию.

Для группировки операторов разрешается использовать пару «BEGIN-END». Допускается использование меток, оператора перехода «GO TO», циклов «FOR» со счетчиком[2]. Кроме того, возможно использование операторов прерывания «BREAK» и/или «STOP», когда это не вызывает недоразумений, какую именно конструкцию или оператор они прерывают и как (на каком уровне вложенности, например).

Однако программа на псевдокоде описывает преобразования данных, но не описывает назначение этих преобразований. Назначение программы описывают спецификации. Для того что бы придать более точный смысл спецификациям программ наряду с языком псевдокода для представления алгоритмов, необходимы формализованные средства спецификации программ.

Так как большинство программ преобразует набор входных данных в выходные, то для спецификации программ используют в основном два вида спецификаций. Эти виды спецификаций называются

- условиями частичной корректности,
- условиями тотальной корректности.

Они схематически записываются в виде $\{\varphi\}\pi\{\psi\}$ и $[\varphi]\pi[\psi]$ соответственно, где π — алгоритм, предусловие φ — это условие на входные данные, которое требуется перед исполнением программы π, а постусловие ψ — это условие на выходные данные, которое гарантируется после исполнения алгоритма π.

Из определения смысла условий корректности следуют утверждения:

- Программа π частично корректна по отношению к предусловию φ и постусловию ψ (обозначение $\models\{\varphi\}\pi\{\psi\}$), если на любых входных данных, которые удовлетворяют свойству φ, программа π или не останавливается (зацикливается, зависает и т.п.), или останавливается с выходными данными, которые удовлетворяют свойству ψ.
- Программа π тотально корректна по отношению к предусловию φ и постусловию ψ (обозначение $\models[\varphi]\pi[\psi]$), если на любых входных данных, которые удовлетворяют свойству φ, программа π всегда останавливается с выходными данными, которые удовлетворяют свойству ψ.

---

[2] Позже мы будем использовать обобщенный цикл «FOR», в котором «счетчик» будет перебирать в произвольном порядке (но без повторений) элементы конечного множества.

Таким образом, оба вида спецификаций устанавливают связь между свойствами входных и выходных данных, но разница между частичной и тотальной корректностью состоит в завершаемости программы: частичная корректность допускает неостановку алгоритма, а тотальная корректность неостановку запрещает. Отсюда следует, что частичная корректность всегда следует из тотальной корректности. (Обратное в общем случае неверно.)

Приведем несколько примеров спецификации программ на псевдокоде.

Пусть LOOP — это никогда не завершающийся бесконечный цикл WHILE TRUE DO $x := x + 1$, где x — целая переменная. Тогда для этой программы верно любое условие частичной корректности, и в частности, $\models$\{TRUE\}LOOP\{FALSE\}, но неверно $\models$[TRUE]LOOP[TRUE].

Следующий пример — задача на вычисление целой части квадратного корня из натурального числа. Неформально алгоритм, который решает эту задачу, можно специфицировать так: по целому неотрицательному числу n вычислить $[\sqrt{n}]$. Некто предложил следующую программу ISR[3], которая (якобы) решает поставленную задачу (причём без использования операции умножения и без вложенных циклов):

```
VAR x,y,n: INT;
x:=0; y:=0;
WHILE y<=n DO
        (y:=y+x+x+1; x:=x+1);
x:=x−1
```

В таком случае условие частичной корректности

$$\{n \geq 0\}\mathrm{ISR}\{x^2 \leq n \ \& \ (x + 1)^2 > n\}$$

означало бы декларацию, что при любом натуральном начальном значении N переменной n, если алгоритм ISR когда-либо завершает работу, то заключительное значение X переменной x будет такое, что $X^2 \leq N \ \& \ (X+1)^2 > N$, т.е. равно $[\sqrt{n}]$. А условие тотальной корректности

$$[n \geq 0]\mathrm{ISR}[x^2 \leq n \ \& \ (x + 1)^2 > n]$$

[3] Integer Square Root

означало бы дополнительно, что программа ISR обязательно завершает свою работу. К обоснованию истинности обоих утверждений корректности мы вернемся позже.

Третий пример — программа решения следующей комбинаторной геометрической задачи.

> На плоскости дано N черных и N (столько же) белых точек. Никакие три точки из данных не лежат на одной прямой. Требуется попарно соединить черные и белые точки отрезками прямых так, чтобы эти отрезки не пересекались.

Целевой «объект» задачи — множество из N отрезков с концами (разного цвета) в заданных N чёрных и N белых точках, причём, не содержащее пересекающихся отрезков. Поэтому имеет смысл ввести специальный тип данных СОЕДинение, значения которого — всевозможные множества из N отрезков с концами (разного цвета) в заданных N чёрных и N белых точках. Среди всех значений типа СОЕД нас интересуют такие множества отрезков, которые не содержат пересечений. Поэтому кажется вполне оправданным ввести булевскую функцию ХОР(X: СОЕД) : BOOL на значениях этого типа, которая возвращает значение TRUE тогда и только тогда, когда множество отрезков X не содержит пересекающихся отрезков. Заметим, что тип СОЕД состоит из N! различных значений: первая белая точка может быть соединена с любой из N чёрных точек, вторая белая — с любой из оставшихся (N–1) чёрных точек и т.д. Поэтому можно считать, что перед нами перечислимый тип, среди значений которого есть первый элемент, доступный благодаря константе ПЕРВ: СОЕД, и есть функция следования СЛЕД(X: СОЕД) : СОЕД, которая по каждому элементу (кроме последнего), переданному в качестве значения фактического параметра, возвращает следующий элемент.

В терминах типа данных СОЕД геометрическая задача интерпретируется следующим простым способом: среди всех возможных значений типа СОЕД начиная с ПЕРВ найти при помощи СЛЕД (если есть) такой, который удовлетворяет ХОР.

Однако вместо полного перебора имеет смысл попробовать применить какую-нибудь эвристику. В данном случае для повышения эффективности речь идёт о «локальном» устранении пересечений в множестве из N отрезков с разными концами в чёрных и белых точках, в надежде, что это позволит в конце концов устранить все пересечения. Эта идея проиллюстрирова-

на на рис. 1, на котором «облако» изображает отдельно взятую пару пересекающихся маршрутов. «Работоспособность» этой идеи проиллюстрирована на рис. 2, где «облако» представляет уже всю совокупность черных и белых точек (т.е. N = 5).

Для программной реализации этой эвристики на псевдокоде вместо функции следования СЛЕД(X: СОЕД) : СОЕД, которая позволяет осуществить перебор всех значений типа СОЕД, будет использоваться функция ЩЁЛК(X: СОЕД) : СОЕД. Эта функция реализует желаемую эвристику, локально устраняя пересечения отрезков, т.е. выбирает некоторую пару (если такая существует) пересекающихся отрезков [B’,W’] ∩ [B”,W”] ≠ ∅ из множества отрезков X и «035переЩЁЛКивает» их, заменяя на пару отрезков [B’,W”] и [B”,W’], т.е. возвращает множество отрезков (X \ {[B’,W’] , [B”,W”]})∪ {[B’,W”] , [B”,W’]}, как это показано на рис. 1.

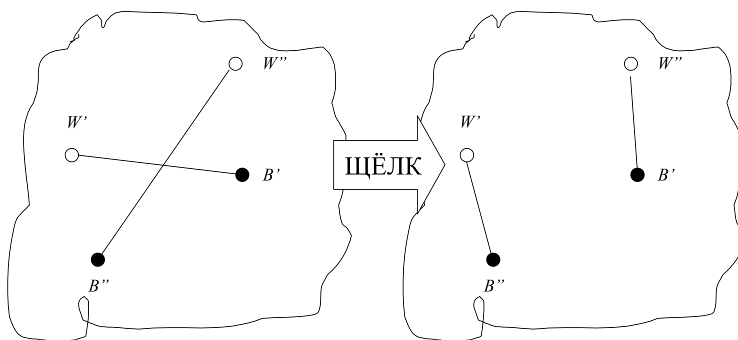

Рис. 1

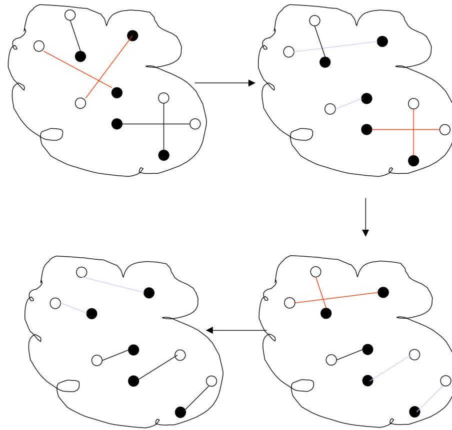

Рис. 2

Теперь программа эвристического перебора соединений может быть реализована на псевдокоде следующим образом:

```
VAR X: СОЕД;
       X:= ПЕРВ;

WHILE  NOT(ХОР(X))
     DO
        X:= ЩЁЛК(X)
     OD;

OUTPUT (X)
```

Обозначим эту программу EVR. Тогда неформально ее можно было бы специфицировать так: если никакие три точки не лежат на одной прямой, то EVR обязательно найдет хорошее (взаимно-однозначное без пересечений) соединение черных и белых точек отрезками прямых. Очевидно, что «обязательно найдет» в данном случае означает, что EVR «обязательно завер-

шит работу». Поэтому самая подходящая спецификация для этой программы — следующее условие тотальной корректности

[NOT(КО_ЛИН)]EVR[ХОР(X)],

где КО_ЛИН — это следующее предусловие: среди данных точек существуют три, которые лежат на одной прямой.

НЕФОРМАЛЬНОЕ ВВЕДЕНИЕ В ФОРМАЛЬНУЮ ВЕРИФИКАЦИЮ ПРОГРАММ

Методы доказательства корректности алгоритмов связаны с именем другого классика науки программирования Р. Флойда (1936–2001) (рис. 3). Метод доказательства частичной корректности алгоритмов, предложенный им, так и называется «метод Флойда», а метод доказательства тотальной корректности — «метод потенциалов». Оба метода впервые были описаны в [3]. Для представления обоих методов используют «псевдоалгоритмический» формат, а для описания применимости методов — условия частичной корректности.

Метод для условий частичной корректности состоит в следующем.

Предусловие метода: $\{\varphi\}\pi\{\psi\}$ — условие частичной корректности.

1. Представить программу π графически в виде блок-схемы и выбрать множество контрольных точек на дугах, включающее начало и конец программы, такое, что любой цикл по графу блок-схемы содержал хотя бы одну контрольную точку.
2. Сопоставить началу программы предусловие φ, концу программы — постусловие ψ, а каждой из оставшихся контрольных точек I — некоторое условие ξ_I, которое будем называть инвариантом этой контрольной точки[4].
3. Для каждой пары контрольных точек I и J, для каждого ациклического пути из точки I в точку J по графу блок-схемы доказать: если условие ξ_I истинно перед началом исполнения этого участка (соответствующего выбранному пути от I к J), то условие ξ_J истинно после исполнения этого участка.

---

[4] Термин инвариант будем употреблять также по отношению к предусловию и постусловию.

Рис. 3

Постусловие метода: $\{\varphi\}\pi\{\psi\}$ — верное условие частичной корректности.

Заметим, что метод Р. Флойда для условий частичной корректности сам является частично корректным по отношению к своим предусловию и постусловию: если его удается применении к тройке $\{\varphi\}\pi\{\psi\}$, то $\models\{\varphi\}\pi\{\psi\}$. Однако его завершаемость не гарантирована, так как необходимо придумать и доказать инварианты (шаги 2 и 3).

Корректность метода можно обосновать следующим образом.

Пусть φ имеет место перед применением программы π к некоторым входным данным. Если программа завершается при этих начальных данных, то рассмотрим его «протокол», т.е. последовательность его шагов и данных во времени (рис. 4):

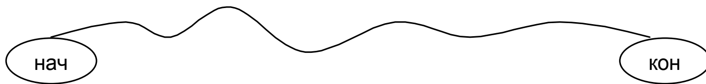

Рис. 4

Выделим в этом протоколе все контрольные точки (на рис. 5 для определённости их всего 2) и выпишем условия, соответствующие каждой из этих контрольных точек:

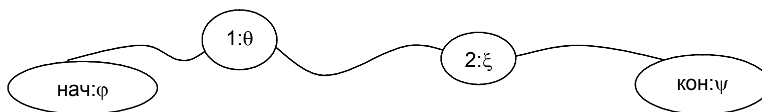

Рис. 5

По построению, участки протокола

- от начала до точки 1,
- от точки 1 до точки 2,
- от точки 2 до конца

не содержат повторений шагов алгоритма. В силу шага (3) метода,

- если φ верно в точке начала, то θ верно в точке 1;
- если θ верно в точке 1, то ξ верно в точке 2;
- если ξ верно в точке 2, то ψ верно в точке конца.

Следовательно, ψ имеет место после завершения алгоритма π. Таким образом, $\models\{\varphi\}\pi\{\psi\}$.

Метод потенциалов для доказательства завершаемости детерминированных программ в простейшей форме был сформулирован следующим образом.

- выбирается конечное множество числовых значений, которые называются потенциалами;
- каждому набору значений используемых переменных сопоставляется некоторый потенциал.

Тогда если каждое исполнение тела любого цикла в программе уменьшает значение потенциала, то программа завершает работу.

Более строго метод потенциалов для доказательства условий тотальной корректности может быть сформулирован следующим образом.

Предусловие метода:

$[\varphi]\pi[\psi]$ — условие тотальной корректности детерминированной программы.

1. Применить к условию частичной корректности $\{\varphi\}\pi\{\psi\}$ метод Р.Флойда.
2. Сопоставить каждой контрольной точке I алгоритма π (выделенной во время доказательства частичной корректности) отображение F_I, которое каждому набору значений используемых переменных сопоставляет некоторое неотрицательное целое число (которое называется потенциалом).
3. Для каждой пары контрольных точек I и J, для каждого пути из точки I в точку J, в котором нет повторений шагов алгоритма, доказать, что если при каком-либо наборе значений переменных D этот путь приводит к набору значений переменных T, то $F_I(D) > F_J(T)$ (т.е. потенциал убывает).

Постусловие метода: $[\varphi]\pi[\psi]$ — верное условие тотальной корректности.

Метод для условий тотальной корректности сам является только частично корректным по отношению к своим предусловию и постусловию, т.е. при его применении к тройке $[\varphi]\pi[\psi]$, (даже в случае успешного применения метода Флойда к $[\varphi]\pi[\psi]$) его завершаемость не гарантирована, так как требуется придумать отображения в множество потенциалов и доказать убывание потенциалов (шаги 2 и 3).

Мы не будем обсуждать применимость метода потенциалов по двум причинам:

- во-первых, он применим только к программам, в которых нет недетерминированных переходов на множество меток, состоящее из более чем одной метки, в то время как в программах на Mini-NIL всякий переход может быть недетерминированным;
- во-вторых, всякая программа на Mini-NIL «завершается», т.е. всякое ее вычисление (даже зацикливающееся) использует только некоторое конечное множество конфигураций, так как все вычисления происходят по модулю некоторого числа.

ПРИМЕРЫ НА ПРИМЕНЕНИЕ МЕТОДА ФЛОЙДА

Рассмотрим простой пример применения метода Флойда к задаче на вычисление целой части квадратного корня из натурального числа при помощи программы ISR.

Утверждается, что программа вычисляет в переменной x целую часть квадратного корня из неотрицательного целого числа n. Поэтому в качестве начального варианта спецификации этого алгоритма можно принять следующее условие частичной корректности: $\{n \geq 0\}\text{SR}\{x = [\sqrt{n}\ ]\}$. Однако постусловие $x = [\sqrt{n}\ ]$ можно заменить на равносильное ($x^2 \leq n$ & $(x + 1)^2 > n$), которое более приспособлено для этого алгоритма. Поэтому окончательный вид спецификации алгоритма следующий: $\{n \geq 0\}\text{SR}\{x^2 \leq n \ \&\ (x + 1)^2 > n\}$.

Применим для доказательства этого условия частичной корректности метод Флойда.

- Представим ISR в виде блок-схемы и выберем множество контрольных точек 1, 2 и 3 в соответствии с методом Флойда. Сопоставим начальной точке 1 предусловие ($n \geq 0$), конечной точке 3 — постусловие ($x^2 \leq n$ & $(x + 1)^2 > n$), а оставшейся контрольной точке — инвариант ($y = x^2$ & $(x \div 1)^2 \leq n$), где «÷» операция вычитания без перехода через 0 (т.е. $0 \div 1 = 0$) (рис. 6).
- Определим все участки программы между всеми парами контрольных точек, которые не содержат повторений шагов алгоритма: (1..2), (2+2) и (2–3), где «..» обозначает отсутствие условных операторов, а «+» и «–» — положительную и отрицательную ветви условного оператора[5].
- Для каждого из этих участков докажем, что если условие, сопоставленное началу участка, верно перед началом исполнения участка, то условие, сопоставленное концу участка, верно после исполнения этого участка.
  - Участок (1..2). Если $n \geq 0$, то после присваиваний $x := 0$ и $y := 0$ очевидным образом ($y = x^2$ & $(x \div 1)^2 \leq n$).
  - Участок (2+2). Пусть значение переменной n есть N, а значения переменных x и y перед исполнением этого участка есть X и Y. Пусть ($\text{Y} = \text{X}^2$ & $(\text{X}\div 1)^2 \leq \text{N}$) перед исполнением участка. После исполнения присваивания $y := y + x + x + 1$ значение переменной y станет $(\text{X} + 1)^2$, а после исполнения присваивания $x := x + 1$ значение пе-

[5] Эту же систему обозначений для участков блок-схемы будем использовать во всех случаях.

ременной x станет (X + 1). Следовательно, новые значения переменных x и y после исполнения участка опять удовлетворяют соотношению $y = x^2$. Кроме того, так как участок начинается с положительной ветви условия $y \le n$, то $Y \le N$ и, следовательно, $X^2 \le N$. Так как значение переменной x после исполнения участка есть (X + 1), то следовательно, это значение удовлетворяет неравенству $(x - 1)^2 \le n$.

- Участок (2–3). Пусть значение переменной n есть N, а значения переменных x и y перед исполнением этого участка есть X и Y. Пусть $(Y = X^2 \& (X \div 1)^2 \le N)$ перед исполнением участка. После исполнения присваивания $x := x - 1$ значение переменной x станет $(X \div 1)$. Поэтому после исполнения этого участка $x^2 \le n$. Кроме того, так как участок начинается с отрицательной ветви условия $y \le n$, то $y > n$ и, следовательно, $X^2 > n$. Так как в конце этого участка значение переменной x равно (X – 1), то следовательно, после исполнения участка $(x + 1)^2 > n$.

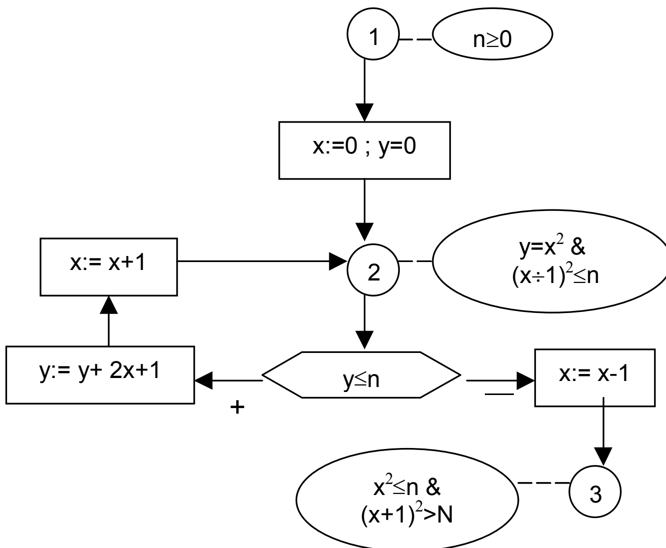

Рис. 6

Таким образом, $\models\{n \geq 0\}\mathrm{SR}\{x^2 \leq n\ \&\ (x + 1)^2 > n\}$ — верное утверждение частичной корректности, и следовательно, алгоритм SR частично корректен по отношению к предусловию $n \geq 0$ и постусловию $x = [\sqrt{n}\,]$. Иными словами, для любого начального значения $\mathrm{N} \geq 0$ переменной n, если алгоритм SR закончит вычисления, то заключительное значение переменной x будет $[\sqrt{n}\,]$.

СИНТАКСИС И НЕФОРМАЛЬНАЯ СЕМАНТИКА АННОТАЦИЙ В ЯЗЫКЕ MINI-NIL

Язык Mini-NIL описан в первой части документации проекта F@BOOL@ [1]. Он состоит из программ, которые строятся из операторов, меток, переменных и констант. Программы и все их составные части на Mini-NIL имеют строгий формат, так как задача этого языка — не удобство программирования, а проверка жизнеспособности идеи проекта F@BOOL@. Переменных в этом языке 27 — это строчные латинские буквы от «a» до «z». Метки — это десятичные целые числа без знака (и ведущих нулей).

Отличие аннотированных программ на языке Mini-NIL от неанотированных состоит в том, что к преамбуле, некоторым из операторов и выходным меткам «прицеплены» логические аннотации (которые отделяются точкой с запятой). Неформально говоря, аннотации — это логические формулы, построенные из равенств и неравенств арифметических выражений при помощи обычных логических операций отрицания «#», конъюнкции «&», дизъюнкции «V», импликации «=>», эквивалентности «<=>», квантора всеобщности «A» и квантора существования «E». Использование именно такой нотации вместо традиционной математической «¬», «∧», «∨», «→», «↔», «∀» и «∃» или программисткой нотации «NOT», «AND», «OR» и «FOR_ALL» объясняется двумя причинами: во-первых, желанием использовать только клавиатурные символы, а во-вторых, стремлением подчеркнуть, что аннотация не программа, имеет другую семантику и, соответственно, использует другой синтаксис.

Формальное определение контекстно-свободного синтаксиса аннотированных программ языка Mini-NIL в нотации Бэкуса—Наура следует ниже[6].

[6] За исключением переменных и меток, которые уже были определены неформально.

Новые (по сравнению с [1]) определения, связанные с аннотациями, выделены курсивом.

⟨assignment⟩ ::= ⟨label⟩:~ ⟨variable⟩:=⟨expression⟩~ goto~{⟨list_of_labels⟩}
⟨expression⟩ ::= ⟨prime_expression⟩ |
⟨prime_expression⟩⟨operation⟩⟨prime_expression⟩
⟨prime_expression⟩ ::= ⟨variable⟩ | ⟨decimal_integer⟩ | M[7]
⟨operation⟩ ::= + | — | *

⟨test⟩ ::= ⟨label⟩ :~ if~ ⟨condition⟩~ then ~{⟨list_of_labels⟩}~
else ~{⟨list_of_labels⟩}
⟨condition⟩ ::= ⟨prime_expression⟩ ⟨relation⟩ ⟨prime_expression⟩
⟨relation> ::= = | < | >

⟨list_of_labels⟩ ::= ⟨empty_list⟩ | ⟨non_empty_list⟩
⟨empty_list⟩ ::=
⟨non_empty_list⟩ ::= ⟨label⟩ | ⟨label⟩,~⟨non_empty_list⟩

⟨preamble⟩ ::= ⟨decimal_integer⟩¶ | ⟨decimal_integer⟩,~⟨preamble⟩

⟨operator⟩ ::= ⟨assignment⟩¶ | ⟨test⟩¶ |
⟨assignment⟩*;~⟨annotation⟩¶* |
⟨test⟩*;~⟨annotation⟩¶*
⟨body⟩ ::= ⟨operator⟩ | ⟨operator⟩ ⟨body⟩

⟨program⟩ ::= ⟨preamble⟩*:~⟨precondition⟩¶* ⟨body⟩*:~⟨postcondition⟩¶*

⟨precondition⟩::= ⟨annotation⟩
⟨postcondition⟩::= ⟨annotation⟩

⟨annotation⟩ ::= ⟨formula⟩
⟨formula⟩ ::= ⟨boolean_constants⟩ | ⟨predicate⟩ | ⟨negated_formula⟩ |
| ⟨conjunction⟩ | ⟨disjunction⟩ | ⟨implication⟩ | ⟨equivalence⟩
| ⟨quantified_formula⟩

⟨boolea_constants⟩ ::= TRUE | FALSE

[7] M обозначает максимальное целое число.

⟨predicate⟩ ::= ⟨term⟩ ⟨relation⟩ ⟨term⟩
⟨term⟩ ::= ⟨prime_expression⟩ | (⟨term⟩ ⟨operation⟩ ⟨term⟩)

⟨negated_formula⟩ ::= (#~⟨formula⟩)
⟨conjunction⟩ ::= (⟨formula⟩ ~&~ ⟨formula⟩)
⟨disjunction⟩ ::= (⟨formula⟩~V~⟨formula⟩)
⟨implication⟩ ::= (⟨formula⟩ ~=> ⟨formula⟩)
⟨equivalence⟩ ::= (⟨formula⟩<=>⟨formula⟩)
⟨quantified_formula⟩ ::= (A⟨variable⟩.~⟨formula⟩) | (E⟨variable⟩.~⟨formula⟩)

Здесь символ «~» использован для явного представления пробела, а «¶» — для явного представления неизобразимого символа перевода строки. Пробелы и переводы строки, кроме разрешенных явно, запрещены.

По соображениям удобства определение семантики аннотаций, налагается требование, что все операторы, помеченные меткой «0», не аннотированы. Другие контекстные ограничения на аннотированный язык Mini-NIL следующие.

1. Переменные, которые используются в операторах присваивания и условных операторах, не могут быть связанны в аннотациях кванторами «E» и/или «A».
2. Количество чисел в преамбуле должно быть на одно больше, чем количество переменных, которые используются в операторах присваивания и условных операторах программы[8].

Неформально говоря, выполнение аннотированной программы на языке Mini-NIL происходит также, как и неаннотированной программы [1]. Некоторая разница состоит в следующем:

1) аннотация ⟨**precondition**⟩ проверяется на входных данных (значения которых заданны в преамбуле), и в случае, когда эта аннотация оказывается неверной, генерируется прерывание «ошибка во входных данных»;
2) аннотация ⟨**postcondition**⟩ проверяется на каждом наборе выходных результатов, и в случае, когда эта аннотация оказывается неверной, генерируется прерывание «ошибка в результатах вычислений»;

[8] Таким образом уточняется единственное контекстное ограничение, которое было введено для неаннотированных программ на Mini-NIL в [1].

3) перед выполнением оператора, к которому «прицеплена» ⟨**annotation**⟩, она проверяется на текущих значениях переменных, и в случае, когда эта аннотация оказывается неверной, генерируется прерывание «ошибка времени исполнения».

Таким образом (неформально) определено понятие *динамической семантики* логических аннотаций. Роль логических аннотаций в динамической семантике — контракты времени исполнения [2][9]. Впервые понятие «контрактов времени исполнения» было введено Б. Мейéром (вдохновленное идеями формальной верификации [3, 4][10]). Если говорить кратко, то контракты времени исполнения — это условия трех типов на

- допустимые входные данные (⟨**precondition**⟩),
- гарантированные результаты вычислений (⟨**postcondition**⟩),
- что является неизменным во время вычислений (остальные ⟨**assertion**⟩),

которые проверяются непосредственно во время выполнения программы, и или не влияют на вычисления, если «контракт выполнен», или генерируют исключение в противном случае. Поэтому в контексте динамической семантики мы будем называть все ⟨**precondition**⟩, ⟨**postcondition**⟩ и ⟨**assertion**⟩ контрактами.

Статическая семантика логических аннотаций — это инварианты контрольных точек на графике блок-схемы программы, с которыми метод Флойда доказывает истинность в любом кольце вычетов целых чисел условия частичной корректности $\{\varphi\}\alpha\{\psi\}$, где φ — предусловие, ψ — постусловие, а α — тело программы.

ДИНАМИЧЕСКАЯ СЕМАНТИКА АННОТАЦИЙ В ЯЗЫКЕ MINI-NIL

Выберем произвольно и зафиксируем (для определенности) аннотированную программу α на языке Mini-NIL. Пусть φ и ψ — ее предусловие и постусловие. Сначала определим семантику и истинность формул, которые встречаются в α, а затем — динамическую и статическую семантику логических аннотаций. При этом будем использовать понятие конфигурации, определенное в [1] для операционной вычислительной семантики неанно-

[9] Для быстрой справки можно обратиться к Design by contract. From Wikipedia, the free encyclopedia at http://en.wikipedia.org/wiki/Design_by_contract.

[10] См. также книги [5, 6] на русском языке.

тированных Mini-NIL программ[11]. Но понятия начальной и заключительной конфигураций уточняются следующим образом.

- Начальная конфигурация — это такая конфигурация INI, в которой L = 0, значения программных переменных a, b, c, … — это начальные значения переменных V_a, V_b, V_c, … в соответствии с преамбулой, значения всех подкванторных переменных равны нулю, а значение специальной метапеременной EXP есть пустое множество.
- Заключительная конфигурация — это произвольная конфигурация, в которой L — произвольная метка, которая не метит ни одного оператора в α, а значения всех подкванторных переменных равны нулю.

Каждая конфигурация CNF = (L, V_a, V_b, V_c, …) позволяет приписать каждому выражению TER из ⟨**term**⟩ или FRM из ⟨**formula**⟩ целое (по модулю (M+1)) CNF(TER) и булевское CNF(FRM) значения соответственно:

1. (значения термов)
   1.1) значение каждого десятичного числа — это остаток от деления этого числа на (M + 1); значение каждой переменной Y, которая встречается в программе, — это целое число V_Y;
   1.2) CNF(TER′+TER″) = (CNF(TER′) + CNF(TER″))mod(M+1), CNF(TER′−TER″) = (CNF(TER′) — CNF(TER″))mod(M+1), CNF(TER′*TER″) = (CNF(TER′) × CNF(TER″))mod(M+1).
2. (значения формул)
   2.1) CNF(TRUE) — всегда истина, а CNF(TRUE) — всегда ложь;
   2.2) CNF(TER′ = TER″) есть истина тогда и только тогда, когда значения CNF(TER)′ и CNF(TER″) равны;
   2.3) CNF(TER′ < TER″) есть истина тогда и только тогда, когда значение CNF(TER′) меньше[12] значения CNF(TER″);
   2.4) CNF(TER′ > TER″) есть истина тогда и только тогда, когда значение CNF(TER′) больше[11] значения CNF(TER″);
   2.5) CNF(# FRM) есть истина тогда и только тогда, когда CNF(FRM) есть ложь;
   2.6) CNF(FRM′ & FRM″) есть истина тогда и только тогда, когда CNF(FRM′) есть истина и CNF(FRM″) есть истина;
   2.7) CNF(FRM′ V FRM″) есть истина тогда и только тогда, когда CNF(FRM′) есть истина или CNF(FRM″) есть истина;

---

[11] Теперь, однако, следует иметь в виду, что «все переменные, которые встречаются в α», включают и переменные, связанные кванторами в аннотациях.

[12] Два числа из кольца вычетов по модулю (M + 1) сравниваются как натуральные числа.

2.8) CNF(FRM′ =>FRM′′) есть истина тогда и только тогда, когда CNF(FRM′) есть ложь или CNF(FRM′′) есть истина;
2.9) CNF(FRM′<=>FRM′′) есть истина тогда и только тогда, когда CNF(FRM′) и CNF(FRM′′) одновременно истина или одновременно ложь;
2.10) CNF(A VAR. FRM) есть истина тогда и только тогда, когда для любой конфигурации CNF′, которая отличается от CNF только значением переменной VAR, истина CNF′(FRM);
2.11) CNF(E VAR. FRM) есть истина тогда и только тогда, когда существует конфигурация CNF′, которая отличается от CNF только значением переменной VAR, что CNF′(FRM) есть истина.

Понятие срабатывания оператора остается прежним.

- Пусть L**:** X**:=**EXP **goto** {NEXT} — произвольный помеченный оператор присваивания из α, где L — метка, X — переменная, EXP — выражение, а NEXT — список меток. Срабатывание этого оператора присваивания — это произвольная пара конфигураций (L, V_a, V_b, V_c, …) (L′, V'_a, V'_b, V'_c, …) такая, что L′ — метка из NEXT, V'_X= (L, V_a, V_b, V_c, …)(EXP), и V'_Y=V_Y для любой переменной Y отличной от X.
- Пусть L**:** **if** CON **then** {NEXT_TRUE} **else** {NEXT_FALSE} — произвольный помеченный условный оператор из α, где L — метка, CON — условие, а NEXT_TRUE и NEXT_FALSE — списки меток. Срабатывание этого условного оператора — это произвольная пара конфигураций (L, V_a, V_b, V_c, …) (L′, V'_a, V'_b, V'_c, …) такая, что $V_Y = V'_Y$ для любой переменной Y, и L′$\in$ NEXT_TRUE, если (L, V_a, V_b, V_c, …)(CON) есть истина, или L′$\in$ NEXT_FALSE, если (L, V_a, V_b, V_c, …)(CON) есть ложь.

Заметим, что срабатывание любого оператора не изменяет значений квантифицированных переменных.

Понятие трассы (инициальной трассы, финальной трассы и вычислительной трассы) программы α переносится со случая неаннотированных программ[13].

[13] Однако следует помнить, что понятия начальной и заключительной конфигураций несколько изменились из-за использования подкванторных переменных, которые должны принимать нулевые значения.

А вот определение результатов вычислений программы α изменяется следующим образом. Для набора начальных значений *программных* переменных (V_a, V_b, V_c, …) пусть $\alpha(V_a, V_b, V_c, \ldots)$ — это множество всех таких наборов значений *программных* переменных (V'_a, V'_b, V'_c, …), для которых существует вычислительная трасса, начинающаяся со значений (V_a, V_b, V_c, … 0, … 0), а заканчивающаяся значениями (V'_a, V'_b, V'_c, …, 0, … 0), где нулями означены все подкванторные переменные программы.

Для того чтобы определить динамическую семантику аннотаций программы α, нам понадобится понятие аварийной конфигурации и аварийного флага[14]. Аварийная конфигурация — это такая конфигурация CNF = (L, V_a, V_b, V_c, …), для которой

- или метка L есть «0», и CNF(φ) есть ложь[15];
- или L не метит ни одного оператора α, и CNF(ψ) есть ложь[16];
- или в α есть аннотированный оператор «L: op; θ», и CNF(θ) есть ложь.

Формулы φ, ψ и θ соответственно называются аварийными флагами конфигурации.

Динамическая семантика аннотаций программы α (или семантика времени исполнения) — это множество abnorm(α,V_a, V_b, V_c, …) всех аварийных конфигураций, которые возникают в вычислительных трассах программы α, начинающихся со значений (V_a, V_b, V_c, … 0, … 0). Неформально говоря, динамическая семантика аннотаций — это те конфигурации программы, которые возникли во время вычислений, в которых контракт (соответствующая аннотация) не был выполнен (т.е. оказалась ложной).

КОРРЕКТНЫЙ МЕТОД ПОСТРОЕНИЯ ДИНАМИЧЕСКОЙ СЕМАНТИКИ АННОТАЦИЙ

В [1] описан корректный алгоритм обхода расширенных семантических деревьев и сформулировано утверждение его корректности. Дополним этот алгоритм так, что наряду с вычислением семантики программы он вычислял бы и все аварийные конфигурации. Новые (по сравнению с [1]) определения, описания и шаги алгоритма выделены курсивом.

Предусловие:

---

[14] Exception (англ.).

[15] Напомним, φ — предусловие программы α.

[16] Напомним, ψ — постусловие программы α.

α — синтаксически правильная *аннотированная* программа на языке Mini-NIL,
φ и ψ — ее предусловие и постусловие,
$LB(\alpha)$ — множество меток, которые встречаются в α,
$VR(\alpha)$ — множество переменных, которые встречаются в α,
$(V_a, V_b, V_c, \ldots)$ — начальные значения переменных α.

```
TYPE VALS = [0..M]^VR(α);
TYPE CONF = LB(α)×[0..M]^VR(α);
VAR result:= ∅ : SET OF VALS ;
VAR exceptions:= ∅; SET OF CONF;
VAR curn, next : CONF ;
VAR que:= <«начальная конфигурация»> : QUEUE OF CONF ;
VAR visited:= ∅ : SET OF CONF ;

BEGIN
DO curn:= head(que) ; que:= tail(que) ;
  IF  curn — аварийная конфигурация THEN exceptions:= exceptions ∪{curn} ;
  IF curn — заключительная конфигурация α
  THEN { LET (V′a, V′b, V′c, …) — набор значений переменных в curn
             IN  result:= result ∪ {(V′a, V′b, V′c, …)} }
  ELSE
    FOR EACH {next : next∉visited, next∉que и (curn, next) — срабатывание α}
                                                                DO que:= que^next ;
  visited:= visited ∪{curn}
UNTIL (que пуста)
END .
```

Постусловие:
result = $\alpha(V_a, V_b, V_c, \ldots)$ и exceptions = $abnorm(\alpha, V_a, V_b, V_c, \ldots)$.

Следующее утверждение формально обобщает утверждение о корректности алгоритма обхода расширенных семантических деревьев из [1], но фактически является его простым следствием, так как приведенный алгоритм построения динамической семантики аннотаций просто содержит один дополнительный оператор присваивания, который изменяет переменные, не влияющие на поток управления и данных.

Утверждение 1

Пусть выполнено предусловие алгоритма обхода семантического дерева ЕК(α), а highth: TR(α) $\rightarrow$ INT — функция, которая по каждой вершине в дереве возвращает её высоту (расстояние от корня). Тогда алгоритм удовлетворяет следующим условию безопасности[17] (1) и условию прогресса[18] (2).

1. В любой момент исполнения алгоритма (кроме начального момента) имеют место пять перечисленных ниже свойств:
   a) множество exceptions состоит из всех аварийных конфигураций, которые встречаются в TR(α) до высоты (highth(curn) -1) включительно, и некоторых аварийных конфигураций, которые встречаются в TR(α) на высоте highth(curn);
   b) множество visited состоит из всех конфигураций, которые встречаются в TR(α) до высоты (highth(curn) $-$ 1) включительно, и некоторых конфигураций, которые встречаются в TR(α) на высоте highth(curn);
   c) множество result состоит из наборов значений переменных, которые встречаются в заключительных конфигурациях, причем все наборы, которые встречаются в заключительных конфигурациях из TR(α) до высоты (highth(curn) -1) включительно, входят в result;
   d) очередь que не имеет повторов и состоит из некоторых конфигураций, которые не входят в visited, но встречаются в TR(α) на высотах highth(curn) или (highth(curn) +1);
   e) для любой пары конфигураций CNF$'$ и CNF$''$ программы α, если CNF$'\in$visited, а CNF$''$является наследником CNF$'$ в TR(α), то CNF$''\in$visited или CNF$''\in$que, а в противном случае CNF$''$ = curn.
2. Для любой конфигурации CNF$\in$TR(α) наступит такой момент исполнения алгоритма, в который curn = CNF.

СТАТИЧЕСКАЯ СЕМАНТИКА АННОТАЦИЙ В ЯЗЫКЕ MINI-NIL

Статическая семантика аннотаций — это следующий метод доказательства невозможности аврийных конфигураций (т.е. пустоты множества abnorm). Он конкретизирует для языка Mini-NIL метод доказательства час-

---

[17] Условиями безопасности принято называть утверждения о том, что ничего плохого некогда не произойдет, т.е. всегда будет выполняться нечто правильное и хорошее.

[18] Условиями прогресса или живости принято называть утверждения о том, что что-то хорошее рано или поздно наступит, произойдет.

тичной корректности Р. Флойда, рассмотренный выше. Также как и сам метод Р. Флойда, статическая семантика аннотаций представлена в «алгоритмоподобном» виде и снабжена пред- и постусловиями.

Предусловие:

α — синтаксически правильная *аннотированная* программа на языке Mini-NIL,

в которой каждый оператор имеет уникальную метку, а в каждом условном операторе then-список и else-список не пересекаются[19],

φ и ψ — ее предусловие и постусловие,

$(V_a, V_b, V_c, \ldots)$ — начальные значения переменных α.

1. Расстановка контрольных точек и инвариантов: представить программу α в виде блок-схемы так что
   1.1) началу программы сопоставляется контрольная точка с инвариантом φ (предусловие), причем, все дуги передачи управления на операторы с меткой «0» проходят в блок-схеме через эту контрольную точку;
   1.2) что всякому аннотированному оператору «L: op; θ» сопоставляется контрольная точка с инвариантом θ, а все дуги передачи управления на этот оператор проходят в блок-схеме через эту контрольную точку;
   1.3) концу программы сопоставляется контрольная точка с инвариантом ψ (постусловие), причем, все дуги передачи управления на метки, не метящие ни одного оператора, ведут в блок-схеме на эту контрольную точку.
2. Если хотя бы один цикл по графу блок-схемы не содержит ни одной контрольной точки, то построение статической семантики аннотаций немедленно завершается с неопределенным результатом, иначе — продолжается в соответствии с шагом 3.

[19] В таком случае говорят, что эти списки дизъюнктивны.

3. Генерация и доказательство условий корректности: для каждой контрольной точки I доказать тождественную истинность[20] условия корректности $\theta \Rightarrow \underset{\substack{J-контрольная\\ точка,\\ \xi-ее_инвариант,\\ W-ациклический_путь,\\ начинающийся_дугой,\\ идущей_от_I,\\ закнчивающийся_дугой,\\ идущей_к_J}}{\&} wp(W,\xi)$

   где θ — инвариант точки I, а *wp* — это функция, которая по пути[21] по блок-схеме и формуле вычисляет формулу по следующим правилам:
   3.1) $wp(\rightarrow^{22}, \chi) = \chi$,
   3.2) $wp(W\rightarrow y{:}= t^{23} \rightarrow, \chi) = wp(W\rightarrow, \chi_{t/x})$, где $\chi_{t/x}$ — это результат замены всех не связанных кванторами вхождений переменной *y* на выражение *t*,
   3.3) $wp(W\rightarrow \omega$ «+» $\rightarrow, \chi) = wp(W\rightarrow, (\omega \Rightarrow\chi))$,
   3.4) $wp(W\rightarrow \omega$ «–» $\rightarrow, \chi) = wp(W\rightarrow, (\omega \vee \chi))$.

Постусловие:
если в начальной конфигурации (0, V_a, V_b, V_c, … 0, …) истинно предусловие φ, то abnorm(α,V_a, V_b, V_c, …) = ∅ и, в частности, во всякой заключительной конфигурации TR(α) истинно постусловие ψ.

Для формулировки следующей леммы нам понадобиться понятие T-абстракции и F-абстракции условного оператора. Пусть if φ then L^+ else L^- — условный оператор. Тогда его T-абстракция (if φ then L^+ else $L^-)^T$ — это условный оператор if TRUE then L^+ else L^-, а его F-абстракция (if φ then L^+ else $L^-)^F$ — это условный оператор if FALSE then L^+ else L^-.

Лемма

Пусть α — программа, в которой каждый оператор имеет уникальную метку, а в каждом условном операторе then-список и else-список не пересекаются. Пусть ψ — какая-либо формула Mini-NIL. Пусть W — путь по графу блок-схемы α, начинающийся и заканчивающийся дугой[24], а n ≥ 0 —

[20] т.е. истинность во всех конфигурациях.
[21] который начинается и заканчивается дугой.
[22] «→» — путь, состоящий только из одной дуги и не содержащий никаких вершин.
[23] *y* и *t* — произвольные программная переменная и простое выражение.
[24] возможно, состоящим из одной дуги.

число операторов в W: $W \equiv \to op_1\sigma_1\to \ldots op_n\sigma_n\to$, где $\sigma_1, \ldots\sigma_n$ — это или «+», или «–», или пусто в соответствии с видом предшествующего оператора[25]. Пусть $CNF_0 \ldots CNF_n$ — последовательность конфигураций, удовлетворяющая следующему свойству: для любого $i\in[1..n]$,

- если op_i — оператор присваивания, а σ_i пусто, то $CNF_{(i-1)}CNF_i$ — срабатывание op_i;
- если op_i — условный оператор, а σ_i есть «+», то $CNF_{(i-1)}CNF_i$ — срабатывание T-абстракции $(op_i)^T$;
- если op_i — условный оператор, а σ_i есть «–», то $CNF_{(i-1)}CNF_i$ — срабатывание F-абстракции $(op_i)^F$.

Тогда следующие утверждения эквивалентны:

- $CNF_0(wp(W, \psi))$ есть истина;
- если $CNF_0 \ldots CNF_n$ является трассой α, то $CNF_n(\psi)$ есть истина.

Доказательство проведем индукцией по $n \geq 0$.

База индукции при $n = 0$ очевидна. Действительно,

- путь W не содержит ни одного оператора, поэтому $wp(W, \psi)\equiv\psi$;
- $CNF_0 \ldots CNF_n$ состоит из единственной конфигурации $CNF_0\equiv CNF_n$.

Поэтому $CNF_0(wp(W, \psi)) = CNF_n(\psi)$.

Предположение индукции: пусть утверждение верно для $n = k \geq 0$.

Шаг индукции: докажем утверждение для $n = (k+1)$. В таком случае $W \equiv \to op_1\sigma_1\to \ldots op_k\sigma_k\to op_{(k+1)}\sigma_{(k+1)}\to$, а последовательность конфигураций имеет вид $CNF_0 \ldots CNF_k\ CNF_{(k+1)}$. Поэтому надо рассмотреть три случая, которые соответствуют виду последнего оператора $op_{(k+1)}$ (присваивание или условный) и знаку $\sigma_{(k+1)}$ (пусто, или «+», или «–»). Рассмотрим эти случаи по отдельности.

Пусть $op_{(k+1)}$ — это присваивание, а знак $\sigma_{(k+1)}$ — это пусто. Для определенности пусть $op_{(k+1)}$ — это оператор присваивания «$y := t$», а $CNF_k = (L', V_a', V_b', V_c', \ldots)$ и $CNF_{(k+1)} = (L'', V_a'', V_b'', V_c'', \ldots)$. Так как $CNF_kCNF_{(k+1)}$ — срабатывание op_n, то значения переменных $V_a'', V_b'', V_c'', \ldots$ отличаются от значений $V_a', V_b', V_c', \ldots$ только для переменной y: $V_y''= CNF_k(t)$. Имеем $CNF_n(\psi) = CNF_k(\psi_{t/y})$, так как слева значения переменной y в формуле ψ есть $CNF_k(t)$, а справа в формулу ψ вместо всех несвязанных вхождений y

[25] «+» или «–» следуют за условными операторами, а пусто — за оператором присваивания.

подставлен терм t, который принимает значение $CNF_k(t)$. Поэтому следующие утверждения эквивалентны:

- если $CNF_0 \ldots CNF_k\, CNF_{(k+1)}$ является трассой α, то $CNF_n(\psi)$ есть истина;
- если $CNF_0 \ldots CNF_k$ является трассой α, то $CNF_k(\psi_{t/y})$ есть истина.

Но согласно предположению индукции последнее утверждение эквивалентно тому, что

- $CNF_0(wp(\rightarrow op_1\sigma_1\rightarrow \ldots op_k\sigma_k\rightarrow, \psi_{t/y}))$ есть истина.

В силу того что $wp(\rightarrow op_1\sigma_1\rightarrow \ldots op_k\sigma_k\rightarrow, \psi_{t/y}) = wp(\rightarrow op_1\sigma_1\rightarrow \ldots op_k\sigma_k\rightarrow op_{(k+1)}\sigma_{(k+1)}\rightarrow, \psi_{t/y})$ и $W \equiv \rightarrow op_1\sigma_1\rightarrow \ldots op_k\sigma_k\rightarrow op_{(k+1)}\sigma_{(k+1)}\rightarrow$, получаем, что все эти утверждения эквивалентны следующему утверждению:

- $CNF_0(wp(W, \psi))$ есть истина.

Таким образом, случай, когда $op_{(k+1)}$ — это присваивание, а $\sigma_{(k+1)}$ — это пусто, доказан.

Пусть $op_{(k+1)}$ — это тест, а знак $\sigma_{(k+1)}$ — это «+». Для определенности пусть $op_{(k+1)}$ — это условный оператор «if ξ then L^+ else L^-», а $CNF_k = (L', V_a', V_b', V_c', \ldots)$ и $CNF_{(k+1)} = (L'', V_a'', V_b'', V_c'', \ldots)$. Так как $CNF_k CNF_{(k+1)}$ — срабатывание $(op_n)^T$, то значения переменных $V_a'', V_b'', V_c'', \ldots$ совпадают со значениями $V_a', V_b', V_c', \ldots$ Поэтому $CNF_k(\theta) = CNF_n(\theta)$ для любой формулы θ и, в частности, для ξ и ψ. Кроме того, $CNF_0 \ldots CNF_k\, CNF_{(k+1)}$ является трассой α тогда и только тогда, когда $CNF_0 \ldots CNF_k$ является трассой α, а $CNF_k CNF_{(k+1)}$ — срабатывание op_n (так как в силу уникальности меток в CNF_k мог сработать только оператор if ξ then L^+ else L^-, а $CNF_k(\xi)$ = ложь не возможно в силу дизъюнктивности L^+ и L^-).

Поэтому имеем следующую цепочку эквивалентных утверждений:

- $CNF_0(wp(W, \psi))$ есть истина;
- так как $wp(W, \psi) = wp(\rightarrow op_1\sigma_1\rightarrow \ldots op_k\sigma_k\rightarrow, (\xi\rightarrow\psi))$, то по предположению индукции, если $CNF_0 \ldots CNF_k$ является трассой α, то $CNF_k(\xi\rightarrow\psi)$ есть истина;
- если $CNF_0 \ldots CNF_k\, CNF_{(k+1)}$ является трассой α, то $CNF_n(\psi)$ есть истина.

Таким образом, случай, когда $op_{(k+1)}$ — это условный оператор, а $\sigma_{(k+1)}$ — это «+», доказан.

Случай когда $op_{(k+1)}$ — это тест, а знак $\sigma_{(k+1)}$ — это «–» доказывается аналогично рассмотренному случаю, когда $op_{(k+1)}$ — это тест, а знак $\sigma_{(k+1)}$ — это «+». ■

Следствием этой леммы является следующее утверждение о корректности статической семантики относительно динамической.

Утверждение 2

Статическая семантика (т.е. метод доказательства невозможности аварийных конфигураций) является частично корректной относительно своих предусловия и постусловия.

Доказательство. Предположим, что нам удалось успешно применить метод к аннотированной программе α с пред- и постусловиями φ и ψ и начальными значениями переменных (V_a, V_b, V_c, …). В частности, это означает, что всякий цикл по блок-схеме программы содержит хотя бы одну контрольную точку (см. шаг 2 метода). Далее, для любой пары контрольных точек I и J, любого ациклического пути W по блок-схеме от I к J является тождественно истинной формула $\theta \rightarrow wp(W,\xi)$, где θ и ξ — аннотации точек I и J (см. шаг 3). В силу Леммы это означает, что для каждого ациклического участка между любыми двумя контрольными точками доказано, что если выполнен инвариант, сопоставленный началу участка, следует, что после исполнения этого участка инвариант, сопоставленный его концу, принимает истинное значение.

Пусть $CNF_0(\varphi)$ есть истина, где CNF_0 — начальная конфигурация программы. Если взять произвольную вычислительную трассу $CNF_0 \ldots CNF_n$, то она «распадается» на трассы, соответствующие исполнению ациклических участков между контрольными. А так как $CNF_0(\varphi)$ есть истина, то в соответствии с обоснованием метода Флойда для доказательства частичной корректности, во всякой конфигурации CNF из этой вычислительной трассы, которая соответствует какой-либо контрольной точке, $CNF(\xi)$ есть истина, где ξ — инвариант, соответствующий этой контрольной точке. Поэтому вдоль этой трассы нет аварийных конфигураций. В силу произвольного выбора вычислительной трассы заключаем, что аварийных конфигураций в расширенном семантическом дереве вообще нет, т.е. $abnorm(\alpha, V_a, V_b, V_c, \ldots) = \varnothing$, что и требовалось доказать. ■

Под задачей верификации программ на языке Mini-NIL мы будем понимать выполнение описанного метода доказательства невозможности аварийных ситуаций.

СПИСОК ЛИТЕРАТУРЫ

1. 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. — Новосибирск, 2005. — (Препр. / Ин-т систем информатики имени А.П. Ершова СО РАН; № 131).
2. Meyer B. Applying «Design by Contract» // Computer (IEEE). — 1992. — Vol. 25, N 10. — P.40–51.
3. Floyd R. W. Assigning meanings to programs // Mathematical Aspects of Computer Science. — 1967. — P. 19–32.
4. Hoare C.A.R. An axiomatic basis for computer programming // Communs. of the ACM. — 1969. — N 12. — P. 576–580.
5. Непомнящий В.А., Рякин О.М. Прикладные методы верификации программ. — М.: Радио и связь, 1988.
6. Грис Д. Наука программирования. — М.: Мир, 1984.

Е. В. Бодин, Н. А. Калинина, Н. В. Шилов

ПРОЕКТ ВЕРИФИЦИРУЮЩЕГО КОМПИЛЯТОРА F@BOOL@

Часть II: Логические аннотации в языке Mini-NIL, их статическая семантика и семантика времени исполнения

Препринт
138

Рукопись поступила в редакцию 11.12.06
Рецензент И.С. Ануреев
Редактор З. В. Скок

Подписано в печать 28.12.06
Формат бумаги 60 × 84 1/16 Объем 1.8 уч.-изд.л., 2.0 п.л.
Тираж 60 экз.

Центр оперативной печати «Оригинал 2», г. Бердск, 49-а, оф. 7, тел./факс 8 (241) 5 38 77