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 I: Общее описание проекта F@BOOL@, его место в компонентном подходе к программированию. Язык Mini-NIL — прототип языка виртуальной машины проекта

Препринт
131

Новосибирск 2005

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. Работа представляет общее описание проекта верифицирующего компилятора F@BOOL@, дает теоретическое обоснование его реализуемости, методическую проработку его места и роли в компонентном (сборочном) подходе к созданию больших программных систем.

Кроме того, работа фиксирует формальный синтаксис и операционную семантику модельного и учебного языка программирования Mini-NIL, который является прототипом языка виртуальной машины проекта F@BOOL@, на котором будут апробированы основные идеи системы F@BOOL@. Вынесенное в название работы указание «Часть I» подразумевает, что данная публикация является первой частью документации по проекту F@BOOL@ и что по мере проработки проекта будет опубликованы следующие части документации.

Работа поддержана грантом РФФИ № 05-07-90162.

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 I: Outlines of F@BOOL@ project in the context of component-based programming. Mini-NIL: a prototype of F@BOOL@ virtual machine language

Preprint
131

Novosibirsk 2005

Verifying Compiler is a system program that proves specified program properties besides translation of program. Paper represents overview of the research project that is entitled to design and implement a verifying compiler F@BOOL@, provides some theoretical background for feasibility of the project, discusses importance of verifying compilers for component-based approach to programming.

At the same time paper fixes formal syntax and operational semantics of a toy programming language Mini-NIL. This language is designed for approbation of basic concepts of the project and is a prototype of F@BOOL@ virtual machine. Publication is the first part of F@BOOL@ documentation.

ВВЕДЕНИЕ И ПОСТАНОВКА ЗАДАЧИ

Верифицирующий компилятор — это системная компьютерная программа, которая

1) переводит (транслирует) написанные человеком программы с языка высокого уровня в эквивалентные исполнимые программы;
2) доказывает (верифицирует) специфицированные человеком математические утверждения о свойствах транслируемых программ.

Проблема создания верифицирующего компилятора является одной из фундаментальных проблем современного программирования. По-видимому, одним из первых, кто чётко сформулировал эту проблему, стал классик программирования А. Хоар [1].

Роль верифицирующего компилятора при компонентном (сборочном) подходе к программированию [2] трудно переоценить. Действительно, суть компонентного программирования — «сборка» большой программы из прежде сделанных компонентов, написанных и отлаженных разными людьми, с разной программисткой культурой, образованием и квалификацией. В принципе, при компонентной сборке игровой или учебной программы, где вопросы безопасности не являются приоритетными, многократное тестирование (в том числе и публичное) компонентов подтверждает удовлетворительную для пользователя надежность программы. Однако, если вопрос безопасности и надежности является важным, то кроме тестирования компонентов желательно выполнить их формальную верификацию[1]. В таком случае сертифицировать надежность верифицированного компонента может или выполненное заранее доказательство корректности компонента, поставляемое вместе с компонентом, или легко переносимый полностью автоматический верифицирующий компилятор, способный верифицировать этот компонент.

Подход, который предполагает снабжать исполняемый код компонента заранее выполненным доказательством его корректности (безопасности, по крайней мере), начал развиваться с середины 1990-х годов в работах американского ученого Г. Некулы и получил название «кода с доказательст-

[1] Формальная верификация не исключает тестирования, а тестирование не заменяет формальную верификацию: доказать можно неверное утверждение про неправильную программу, а тестирование никогда не является исчерпывающим.

вом» («Proof-carrying code») [3]. К сожалению, главный недостаток этого подхода — огромный размер доказательства по сравнению с размером исполнимого кода. Г. Некула в шутку характеризует его как «доказательство с кодом» («Proof, carrying code»).

Альтернативой коду с доказательством может стать легко переносимый полностью верифицирующий компилятор, способный автоматически верифицировать достаточно сложные компоненты по их внешним спецификациям (только по пред- и пост- условиям, например). Именно таким компилятором для подмножества C должна стать предлагаемая нами система F@BOOL@. Ключевая идея этого верифицирующего компилятора — представление значений всех типов данных в виде наборов булевских переменных и использование эффективных булевских решателей для проверки условий корректности вместо их дедуктивного вывода с использованием программ поиска доказательства. По-видимому, проект F@BOOL@ наиболее близок к проекту SLAM, который разрабатывает фирма Microsoft (`http://research.microsoft.com/slam/`) [4]. SLAM автоматически по программе на (подмножестве) C строит ее конечную модель (которая является программой, работающей над булевским типом) и верифицирует свойства безопасности и живости этой модельной (булевозначной) программы. Однако разница состоит в том, что наш проект сразу нацелен на верификацию широкого спектра свойств программ и предполагает возможность использования аннотаций таких программ (инвариантов циклов и т.п.) для генерации безпрограммных условий корректности.

Традиционно направление программирования, которое занимается доказательством математических теорем с помощью ЭВМ, называется автоматическим доказательством теорем, а направление, которое занимается автоматическим доказательством свойств программ — формальной верификацией. Первые автоматические доказатели возникли, фактически, одновременно с появлением первых компьютеров. Однако первые программы-верификаторы стали появляться значительно позже — только в середине 1970-х годов. Исторически эти направления развивались независимо разными группами исследователей с различным научным и финансовым потенциалом.

К сожалению, системы автоматического доказательства теорем до сих пор носят, скорее, экспериментальный характер, хотя были попытки создания (полу)коммерческих и промышленных систем такого рода (например, Computational Logic или SRI International). Однако у научной общественности не вызывает сомнения перспективы автоматического доказательства в области формальной верификации компьютерных программ и оборудова-

ния. А что касается экспериментальных универсальных систем автоматического доказательства, то лидерами среди таких систем является, повидимому, Vampire, HOL и PVS.

В 1990-е годы произошёл бурный рост экспериментальных исследований по верификации компьютерных программ и оборудования с использованием булевского представления моделей программ и оборудования в формате так называемых (упорядоченных) бинарных разрешающих диаграмм (OBDD). Толчком к этому послужил успех экспериментальной системы SMV символической верификации конечных моделей. Основа высокой эффективности SMV — формат OBDD, который позволяет проверять выполнимость и истинность булевских формул, а также допускает (с использованием динамического программирования) эффективно преобразовывать булевские формулы. Для спецификации свойств программ и оборудования система SMV использовала одну из популярных программных логик — логику дерева вычислений CTL. (Отметим, однако, что разные системы проверки моделей отличаются друг от друга не только форматом представления моделей, но также и языком спецификаций, выбранным для описания свойств моделируемых программ и оборудования.)

ПРОЕКТ F@BOOL@: ЕГО ЦЕЛИ, ЭТАПЫ И ПОДХОДЫ

Проект нацелен на разработку, экспериментальную реализацию и отладку эффективного верифицирующего компилятора для программ на подмножестве языка C. Итогом проекта должна стать специализированная система F@BOOL@, ориентированная на конструирование, преобразование и доказательство свойств и верификацию программ и устройств моделируемых при помощи дискретных систем с конечным числом состояний.

Предлагаемая система должна поддерживать

3) представительное подмножество языка C для программ (с формальной операционной и аксиоматической семантиками), вычисляющих булевозначные функции;
4) библиотеку автоматических средств для преобразования форматов представления булевозначных функций, для решения систем булевых функциональных уравнений и неравенств;
5) логический язык для спецификаций, вычисляющих булевозначные функции, использующий средства темпоральных и программных логик, а также некоторые средства высших порядков (например, вычисления неподвижных точек);

6) автоматическую генерацию условий корректности по программам и спецификациям; автоматическое доказательство условий корректности с использованием (элементов) логического вывода и решателей булевских уравнений.

Мы надеемся достигнуть намеченной функциональности системы F@BOOL@ за счёт использования следующих подходов и средств:

7) апробацию и проработку основных идей системы F@BOOL@ сначала на модельном языке NIL и учебном уровне, который в дальнейшем будет использоваться в качестве виртуальной машины для C-программ;
8) использование иерархии подмножеств языка C, разработанной в [5] таким образом, что
   - на базовом уровне используются только конструкций, допускающие «прозрачную» формализацию в виде структурной операционной семантики Плоткина и совместной аксиоматической семантики Флойда-Хоара,
   - более высокие уровни транслируются на базовый уровень таким образом, что специфицированные свойства программ допускают одновременное и эквивалентное преобразование;
9) использование для спецификаций и аннотации программ формализм программных, временных и эпистимических пропозициональных логик с кванторами второго порядка по пропозициональным переменным;
10) использование для доказательства (вместо программ автоматического доказательства) беспрограммных условий корректности эффективных SAT-решателей (например, ZCHAFF [6] и UnitWalk [7]), а для программно-зависимых условий корректности — символические верификаторы конечных моделей (как SMV[8]).

Мы надеемся достигнуть эффектности алгебраической части системы F@BOOL@ за счёт использования многовариантности внутреннего представления булевозначных функций и многопотоковости преобразований (когда все используемые варианты обрабатываются параллельно). Так, например, наряду с конъюнктивными и дизъюнктивными нормальными формами будут использоваться бинарные разрешающие диаграмы (OBDD) и другие варианты представления булевских функций.

Надёжность системы F@BOOL@ будет гарантирована за счёт применения строгой дисциплины программирования, многовариантного программирования, формальной (ручной и автоматической) верификации основных

алгоритмических модулей и высокой степенью тестированности. Многовариантность программирования возникает в силу ориентации на многовариантность внутреннего представления и многопотоковость при обработке (когда разные варианты представления будут использовать разные алгоритмы преобразований) . Для формальной верификации основных алгоритмических модулей будет использоваться логика Флойда—Хоара. А высокая степень тестированности включает, в частности, тестирование системы в целом на представительной выборке из библиотек тестов SATLIB [9].

Проект F@BOOL@ рассчитан на 2005—2007 гг. Теоретические исследования по теме проекта предполагается вести на протяжении всего периода. Что касается реализационной части проекта, то в данный момент нам видится следующее распределение задач по годам.

2005 год. Разработка модельного и учебного языка аннотированных программ Mini-NIL, предназначенного для вычислений без использования структурных типов данных над целыми числами по некоторому модулю и ориентированного на верификацию посредством SAT-решателей и символьных верификаторов конечных моделей.

2006 год. Разработка языка аннотированных программ NIL, предназначенного для вычислений над целыми числами по некоторому модулю и ориентированного на верификацию посредством SAT-решателей и символьных верификаторов конечных моделей. Разработка архитектуры системы F@BOOL@ и транслятора аннотированных программ из подмножества языка C-light в аннотированные программы на языке NIL.

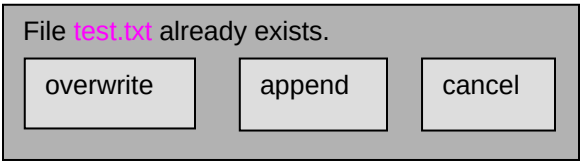
2007 год. Разработка, прототипирование, верификация и тестирование основных модулей системы F@BOOL@.

MINI-NIL: НА ПУТИ К ЯЗЫКУ ВИРТУАЛЬНОЙ МАШИНЫ СИСТЕМЫ F@BOOL@

Недетерминизм играет огромную роль в теории программирования [10]. Однако заметим, что у большинства программистов-практиков и большинства студентов недетерминизм вызывает скорее неприятие и непонимание, желание дать ему вероятностную интерпретацию посредством случайного выбора. В то же время и в реальном программировании сплошь и рядом возникают типично недетерминированные ситуации, когда несколько вариантов являются «равнодопустимыми», но не «равновероятными». Например, легендарное сообщение «Press any key to continue». Чаще всего «any

key» — это или перевод строки, или пробел. Но в принципе это любая из более чем 100 символов клавиатуры.

В приведенном выше примере вероятностная интерпретация «any key», в принципе, вполне имеет смысл. Но вероятностная интерпретация не имеет никакого смысла, когда, например, надо реалистично «промоделировать» поведение пользователя, которому надо выбрать один из трех пунктов следующего всплывающего меню:



В таком случае не имеет смысла говорить, например, о вероятности выбора кнопки «append», а надо просто признать, что эта кнопка просто «равнодопустима» на ряду другими кнопками «overwrite» и «cancel». Таким образом, реалистичная модель пользователя — это недетерминированный выбор одной из трех кнопок меню.

Аббревиатура NIL — это сокращение от Nondeterministic Imperative Language. Язык Mini-NIL — это NIL в миниатюре, предназначенный для апробации идеи проекта F@BOOL@, в котором NIL будет языком виртуальной C-машины. Ниже дано полное описание синтаксиса и операционной семантики языка Mini-NIL.

Язык Mini-NIL 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@.

Переменные (точнее — программные переменные) в этом языке — это строчные латинские буквы[2] «a», … «z». Метки — это десятичные целые числа без знака (и ведущих нулей). Операторы бывают двух видов — присваивания и условные; все операторы обязательно помечены метками. Формальное определение в нотации Бэкуса—Наура следует ниже.

[2] Таким образом, в Mini-NIL всего 27 переменных.

⟨assignment⟩ ::= ⟨label⟩:~ ⟨variable⟩:=⟨expression⟩~ goto~{⟨list_of_labels⟩}¶
⟨expression⟩ ::= ⟨prime_expression⟩ |
⟨prime_expression⟩⟨operation⟩⟨prime_expression⟩
⟨prime_expression⟩ ::= ⟨variable⟩ | ⟨decimal_integer⟩ | M
⟨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⟩
⟨body⟩ ::= ⟨operator⟩ | ⟨operator⟩ ⟨body⟩

⟨program⟩ ::= ⟨preamble⟩ ⟨body⟩

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

Программа на языке Mini-NIL состоит из преамбулы и списка помеченных метками операторов присваивания и условных операторов (причем, каждая метка может метить несколько операторов сразу). Предполагается, что каждая программа на языке Mini-NIL использует переменные «без пропусков», т.е. она может использовать только одну переменную, и тогда эта переменная «a»; только две переменных, и тогда это переменные «a» и «b»; только три переменных, и тогда это переменные «a», «b» и «c», и т.д.

Единственный разрешенный тип данных в языке Mini-NIL — это «целые», которые на самом деле представляют собой кольцо вычетов по модулю некоторого целого числа (M+1) > 0. Значение этого числа M во время выполнения программы можно получить в качестве значения выражения **M**, а транслятор (интерпретатор) программ на языке Mini-NIL узнает зна-

чение числа (M+1) из преамбулы программы: (M+1) — это первое число преамбулы.

Выполнение программы основано на обновлении значений переменных посредством операторов присваивания, ветвления по условию в условных операторах и недетерминированной передачи управления в **goto**, **then** и **else**. Выполнение программы всегда начинается с одного из операторов, помеченных меткой **0**, и начальных значений переменных, которые интерпретатор программ на языке Mini-NIL узнает из преамбулы программы: начальное значение первой переменной (т.е. «a») — это второе число преамбулы, начальное значение второй переменной (т.е. «b») — это третье число преамбулы, начальное значение третьей переменной (т.е. «c») — это четвертое число преамбулы, и т.д.

Из сказанного выше про определение значения числа (M+1) и начальных значений переменных следует, что количество чисел в преамбуле должно быть на одно больше, чем количество переменных, которые встречаются в программе. Данное контекстно-зависимое ограничение является единственным[3].

Каждое присваивание **⟨variable⟩ := ⟨expression⟩ goto {⟨list_of_labels⟩}** приводит к тому, что значение одной переменной **⟨variable⟩** изменяется на значение **⟨expression⟩** по модулю числа (M+1), а управление передается на одну из меток в **⟨list_of_labels⟩**.

Каждый условный оператор **if ⟨condition⟩ then {⟨first_list_of_labels⟩} else {⟨second_list_of_labels⟩}** не изменяет значений ни одной переменной, но передает управление недетерминированно на одну из меток из **⟨first_list_of_labels⟩**, если условие **⟨condition⟩** истинно, или недетерминированно на одну из меток из **⟨second_list_of_labels⟩**, если условие **⟨condition⟩** ложно.

Вычисление завершается как только управление передано на какую-либо метку, которая не метит никакого оператора в программе.

ОПЕРАЦИОННАЯ ВЫЧИСЛИТЕЛЬНАЯ СЕМАНТИКА MINI-NIL

Выберем произвольно программу α на языке Mini-NIL и зафиксируем ее (для определенности).

[3] Заметим, что оно вообще говоря не выводит Mini-NIL из класса регулярных языков.

Конфигурация α — это произвольный вектор (L, V_a, V_b, V_c, …), где L — метка (которая встречается в α), а $V_a, V_b, V_c, \ldots \in [0..M]$ — целочисленные значения переменных (которые встречаются в α). Начальная конфигурация — это конфигурация, в которой L = 0, а значения переменных V_a, V_b, V_c, … — это начальные значения переменных в соответствии с преамбулой. Заключительная конфигурация — это произвольная конфигурация, в которой L — произвольная метка, которая не метит ни одного оператора в α.

Каждая конфигурация (L, V_a, V_b, V_c, …) позволяет приписать каждому выражению **⟨expression⟩**, в котором используются только переменные из программы, целое значение (по модулю (M+1)), а каждому условию **⟨condition⟩** — булевское значение:

- значение каждого десятичного числа — это остаток от деления этого числа на (M+1); значение каждой переменной Y, которая встречается в программе, — это целое число VY;
- значения выраженииий EXP′+EXP″, EXP′–EXP″or EXP′×EXP″ (где EXP′ и EXP″ являются ⟨prime_expression⟩) — это сумма, разность и произведение (по модулю (M+1)) значений EXP′ and EXP″;
- если EXP′ и EXP″ — это ⟨prime_expression⟩, то
    - EXP′ = EXP″ истинно тогда и только тогда, когда значения EXP′ и EXP″ равны,
    - EXP′ < EXP″ истинно тогда и только тогда, когда значение EXP′ меньше значения EXP″,
    - EXP′ > EXP″ истинно тогда и только тогда, когда значение EXP′ больше значения EXP″.

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′$\in$ NEXT и для любой переменной Y имеет место

$$V'_Y = \begin{cases} V_Y, \text{ если Y } - \text{ это не X,} \\ \text{значение EXP в конфигурации (L, } V_a, V_b, V_c, \ldots) \text{ в противном случае.} \end{cases}$$

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

$$L' \in \begin{cases} \text{NEXT\_TRUE, если CON есть TRUE в конфигурации } (L, V_a, V_b, V_c, \ldots), \\ \text{NEXT\_FALSE, если CON есть FALSE в конфигурации } (L, V_a, V_b, V_c, \ldots). \end{cases}$$

Трасса (вычислительная трасса программы α) — это произвольная конечная последовательность конфигураций CNF_0 … CNF_n такая, что всякая смежная пара конфигураций из этой последовательности CNF_i $CNF_{(i+1)}$ является срабатыванием некоторого оператора из α. Инициированная трасса α — это произвольная трасса, которая начинается начальной конфигурацией. Финальная трасса α — это произвольная трасса, которая заканчивается заключительной конфигурацией. Вычислительная трасса α — это произвольная инициированная заключительная трасса.

Для набора начальных значений переменных (V_a, V_b, V_c, …) пусть $\alpha(V_a, V_b, V_c, \ldots)$ — это множество всех таких наборов значений переменных (V'_a, V'_b, V'_c, …), для которых существует вычислительная трасса, начинающаяся со значений (V_a, V_b, V_c, …), а заканчивающаяся значениями (V'_a, V'_b, V'_c, …).

ФОРМАЛЬНАЯ СПЕЦИФИКАЦИЯ КОМПИЛЯТОРА ДЛЯ ЯЗЫКА MINI-NIL

Компилятор для языка Mini-NIL — это программа, которая получает на вход текстовый ASCII-файл с расширением «nil», завершает работу за конечное время и формирует два выходных текстовых ASCII-файла с тем же именем, что и входной файл, но с расширением «log» и «out» соответственно.

- Если входной nil-файл не содержит синтаксически правильную программу на языке Mini-NIL, то в соответствующий выходной log-файл заносится информация о несоответствии nil-файла синтаксису языка, а в выходной out-файл заносится одно слово «UNDONE».
- Если входной nil-файл содержит синтаксически правильную программу на языке Mini-NIL, а (V_a, V_b, V_c, …) — начальные значения переменных, взятые из преамбулы, то в соответствующий выходной log-файл заносится одно слово «CORRECT», а в выходной out-файл заносится множество $\alpha(V_a, V_b, V_c, \ldots)$ построчно (в каждой строке — один набор значений (V'_a, V'_b, V'_c, …)$\in \alpha(V_a, V_b, V_c, \ldots)$), отсортированное в алфавитном порядке, за которым в отдельной строке следует слово «DONE».

ПРИМЕР «СДЕЛКА» (BARGAIN)

Разберем простой пример правильной программы на Mini-NIL. Пусть входной файл bargain.nil содержит следующую программу:

```
5,1,2,3
0: a:=M−1 goto {1}
1: if a<b then {2} else {2, 3}
2: a:=a+1 goto {1, 2}
0: a:=0 goto {1}
3: if a>c then {} else {4, 6}
4: a:=a−1 goto {4, 5}
5: if a>c then {4} else {4, 1}
```

Соответствующие выходные файлы bargain.log и bargain.out. Файл bargain.log содержит ровно одно слово «CORRECT», а файл bargain.out — следующие три строчки

```
2, 2, 3
3, 2, 3
DONE
```

Синтаксическая корректность программы bargain.nil очевидна. Поэтому разберем только на неформальном уровне, как работает данная программа и почему получается именно такой файл bargaim.out.

Неформальная идея, положенная в основу этого примера — торговые переговоры между Продавцом и Покупателем. Продавец пробует повысить цену товара, а Покупатель — понизить. У Продавца есть некоторая минимальная цена, ниже которой он не продаст товар, но продолжит торговаться. А у покупателя есть некоторая максимальная цена, больше которой он никогда не даст за товар, а немедленно прекратит переговоры. Переговоры завершаются, как только после предложения цены Продавцом предложенная цена устраивает Покупателя.

Программа bargain.nil использует следующие переменные:

- «a» для переговорной цены,
- «b» для минимальной приемлемой цены продавца,
- «c» для максимальной приемлемой цены покупателя.

Остановка программы соответствует заключению сделки.

Преамбула программы определяет величину M = 4 (так как первое число в преамбуле — это значение (M+1)), начальное значение 1 для переменной «a», начальное значение 2 для переменной «b», и начальное значение 3 для переменной «c».

Во время инициализации вычислений поток управления сразу начинает ветвится, так как сразу два оператора помечено меткой 0:

0: a:=M goto {1}

0: a:=0 goto {1} .

Первый из этих операторов соответствует ситуации, когда Продавец начинает обдумывать цену с «запредельной» цены 4 за товар, а второй — когда он начинает с мысли, отдать товар даром (т.е. по цене 0).

Следующая «ветвь» программы моделирует «размышления» Продавца о цене, которую предложить Покупателю:

1: if a<b then {2} else {2, 3}

2: a:=a+1 goto {1, 2} .

Здесь ситуация развивается следующим образом.

- Во время выполнения условного оператора, помеченного меткой 1, Продавец сравнивает предлагаемую цену «a» с минимальной приемлемой для него ценой «b»:
  - ⇒ если предлагаемая цена мала, то он ее увеличивает, переходя к исполнению оператора, помеченного меткой 2;
  - ⇒ если же предлагаемая цена немала, то он (недетерминировано) решает, увеличить ли прелагаемую цену еще, переходя к исполнению оператора, помеченного меткой 2, или предложить товар за эту цену Покупателю, переходя к исполнению оператора, помеченного меткой 3.
- Во время выполнения оператора присваивания, помеченного меткой 2, Продавец увеличивает цену, а потом (недетерминировано) решает, увеличить ли предлагаемую цену еще, повторив исполнение этого же оператора, помеченного меткой 2, или сравнить эту цену с минимальной, переходя к исполнению оператора, помеченного меткой 1.

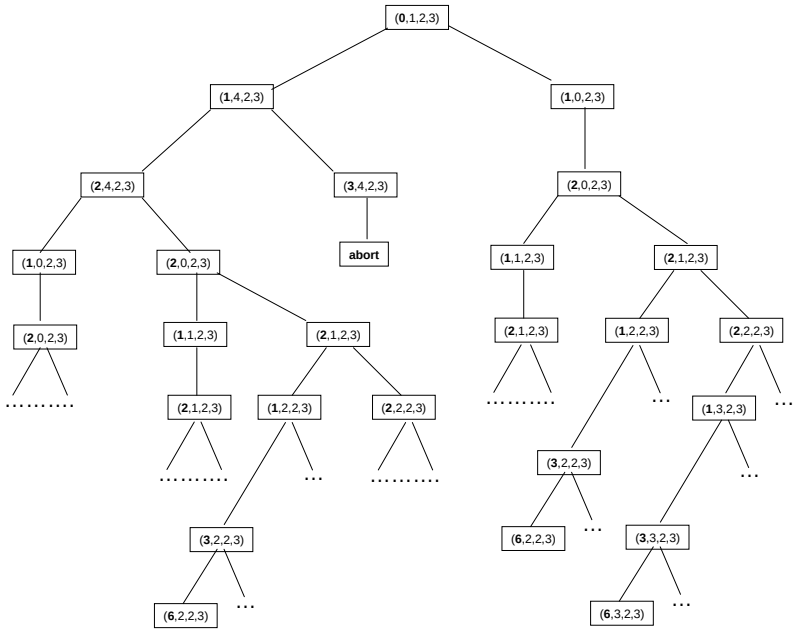
Следующая «ветвь» программы моделирует «размышления» Покупателя о цене, которую предложил Продавец:

3: if a>c then {} else {4, 6}
4: a:=a−1 goto {4, 5}
5: if a>c then {4} else {4, 1} .

Здесь ситуация развивается следующим образом.

- Во время выполнения условного оператора, помеченного меткой 3, Покупатель сравнивает предлагаемую цену «a» с максимальной приемлемой для него ценой «c»:
  - ⇒ если предлагаемая цена велика, то он прекращает переговоры, что представлено переходом в «никуда» на пустое множество меток;
  - ⇒ если же предлагаемая цена не выходит за пределы разумного, то он (недетерминировано) решает, уменьшать ли прелагаемую цену еще, переходя к исполнению оператора, помеченного меткой 4, или принять предложенную цену и завершить сделку, что соответствует переходу на метку 6, которая не метит ни одного оператора в программе.
- Во время выполнения оператора присваивания, помеченного меткой 4, Покупатель уменьшает цену, а потом (недетерминировано) решает, уменьшать ли предлагаемую цену еще, повторив исполнение этого же оператора, помеченного меткой 4, или сравнить эту цену с максимальной, переходя к исполнению оператора, помеченного меткой 5.
- Во время выполнения условного оператора, помеченного меткой 5, Покупатель сравнивает предлагаемую цену «a» с максимальной приемлемой для него ценой «c»:
  - ⇒ если предлагаемая цена велика, то он уменьшает ее, переходя к исполнению оператора, помеченного меткой 4;
  - ⇒ если же предлагаемая цена не выходит за пределы разумного, то он (недетерминировано) решает, уменьшать ли предлагаемую цену еще, переходя к исполнению оператора, помеченного меткой 4, или сообщить свое предложение Продавцу, что соответствует передаче управления оператору, помеченному меткой 1.

То, что bargain.nil(1,2,3) = {(2,2,3), (3,2,3)}, следует из представления всех возможных вычислительных трасс этой программы в виде дерева, в котором «склеены» совпадающие начала разных. Это дерево изображено ниже.



СЕМАНТИЧЕСКИЕ ДЕРЕВЬЯ

Дерево, которое возникло при обсуждении примера bargain.nil, можно обобщить на случай произвольной программы на языке Mini-NIL, а именно: все множество вычислительных трасс программы, которые начинаются в начальной конфигурации, можно представить в виде так называемого семантического дерева, «склеив» общие префиксы.

Определение семантического дерева TR(α).

Пусть α — программа на языке Mini-NIL. Тогда

- вершины дерева TR(α) — это конфигурации α, а корень — начальная конфигурация;
- вершина CNF′ является наследником вершины CNF, когда (CNF, CNF′) является срабатыванием некоторого оператора в α.

Пример семантического дерева был приведен выше для программы bargain.nil. В этом примере, как и общем случае, очевидно, что $\alpha(V_a, V_b, V_c, \ldots) = \{ (V'_a, V'_b, V'_c, \ldots) : (L, V'_a, V'_b, V'_c, \ldots)$ — лист $TR(\alpha)$, причем L не метит никакого оператора в $\alpha\}$. Поэтому задача компилятора синтаксически корректной программы на Mini-NIL — это просто обойти все листья $TR(\alpha)$, которые являются заключительными конфигурациями.

КОРРЕКТНЫЙ МЕТОД ОБХОДА РАСШИРЕННЫХ СЕМАНТИЧЕСКИХ ДЕРЕВЬЕВ

Рассмотрим подробнее, как именно можно обойти семантическое дерево. Заметим, что вообще говоря, расширенное семантическое дерево имеет виртуальную, а не статическую природу, так как оно строится во время исполнения программы. В принципе, известны два метода обхода (виртуальных) деревьев:

- метод отката (обход в глубину),
- метод ветвей и границ (обход в ширину).

Так как для любой Mini-NIL программы множество конфигураций конечно[4], то оба метода, в принципе, применимы. Мы, однако, рассмотрим здесь только метод ветвей и границ.

Для того что бы адаптировать метод ветвей и границ для обхода семантических деревьев, надо ответить на вопрос, каковы целевое и граничное условия? В качестве целевого условия мы примем следующее: «Лист является заключительной конфигурацией, которая ещё не посещалась во время обхода». Тогда в качестве граничного условия имеет смысл принять следующее: «Вершина является конфигурацией, которая уже посещалась во время обхода». Поэтому общая схема метода ветвей и границ преобразуется в следующий алгоритм обхода семантических деревьев.

Предусловие:
α — синтаксически правильная программа на языке Mini-NIL,
$LB(\alpha)$ — 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.

[4] Оно не превосходит (число используемых меток)$\times(M+1)^{(\text{число используемых переменных})}$.

TYPE VALS = $[0..M]^{VR(\alpha)}$;
\\ Тип для наборов значений переменных программы α.

TYPE CONF = $LB(\alpha)\times[0..M]^{VR(\alpha)}$;
\\ Тип для вершин $TR(\alpha)$, т.е. для конфигураций программы α.

VAR result:= $\varnothing$: SET OF VALS ;
\\ Переменная для множества наборов значений переменных программы α,
\\ инициализированная пустым множеством.

VAR curn, next : CONF ; \\ Переменные для конфигураций программы α.

VAR que:=<«начальная конфигурация»> : QUEUE OF CONF ;
\\ Переменная для очереди активных вершин дерева,
\\ инициализированная начальной конфигурацией.

VAR visited:= $\varnothing$: SET OF CONF ;
\\ Переменная для множества посещенных вершин дерева,
\\ инициализированная пустым множеством.

BEGIN
DO curn:= head(que) **;** que:= tail(que) **;**

IF curn — заключительная конфигурация α
THEN { LET (V'_a, V'_b, V'_c, …) — набор значений переменных в curn
IN result:= result $\cup$ {(V'_a, V'_b, V'_c, …)} }
ELSE
FOR EACH { next : next$\notin$visited, next$\notin$que и (curn, next) — срабатывание α }
DO que:= que^next ;
visited:= visited $\cup${curn}
UNTIL (que пуста)
END .

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

Следующее утверждение фактически формулирует инвариант цикла и условие завершаемости алгоритма обхода семантического дерева и, тем самым, является доказательством его корректности.

Утверждение.

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

1. В любой момент исполнения алгоритма или это начальный момент, или имеют место три свойства, перечисленные ниже:
   a) 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);
   b) 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;
   c) 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);
   d) 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.

В настоящий момент реализовано несколько вариантов компилятора (интерпретатора) для языка Mini-NIL, основанных либо на приведенном выше алгоритме, либо на методе отката. Эти интерпретаторы имеют разную эффективность, но пока еще не прошли сравнительное тестирование.

СПИСОК ЛИТЕРАТУРЫ

1. Hoare C.A.R. The Verifying Compiler: A Grand Challenge for Computing Research // Lect. Notes Comput. Sci. — 2003. — Vol.2890. — P. 1–12.
2. Городняя Л.В. Банк улучшаемых компонентов информационных систем. — 30 с. — Новосибирск, 2005. — (Преп. / ИСИ СО РАН; № 130).
3. Necula G. C. Proof-carrying code // Proc. of Conf. Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages.1997. — P. 106–119.
4. Ball T., Rajamani S.K. The SLAM Project: Debugging System Software via Static Analysis // Proc. of the 29th ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages. — 2002. — P. 1–3.
5. Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В. Ориентированный на верификацию язык C-light. // Системная информатика. Вып. 9. — Новосибирск: Изд-во СО РАН, 2004. — С. 51–134.
6. Moskewicz M., Nadigan C., Zhao Y., Zhang L., Malik S. Chaff: Engineering an efficient SAT solver // Proc. of 39th Design Automation Conf. (DAC 2001), Las Vegas, 2001.
7. Hirsch E., Kojevnikov A. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination // Annals of Mathematics and Artificial Intelligence. 2005. — Vol.43 (1–4). — P. 91–111.
8. Кларк Э.М., Грумберг О., Пелед Д. Верификация моделей программ: Model Checking. — М.: МЦНМО, 2002. — 416 с.
9. Hoos H.H., Stutzle T. SATLIB: An Online Resource for Research on SAT. In: Gent I.P., Maaren H.V., Walsh T. Editors // SAT 2000. — IOS Press, 2000. — P. 283–292.
10. Дейкстра Е.В. Дисциплина программирования. — М.: Мир, 1978.

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

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

Рукопись поступила в редакцию 15.12.05
Редактор З. В. Скок

Подписано в печать 29.12.05
Формат бумаги 60 × 84 1/16 Объем 1.4 уч.-изд.л., 1.5 п.л.
Тираж 60 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383) 330-72-02