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

А.В. Промский

ПРИМЕНЕНИЕ ТРЕХУРОВНЕВОГО ПОДХОДА К ВЕРИФИКАЦИИ ПРОГРАММ НА ЯЗЫКЕ C#-LIGHT

Препринт
139

Новосибирск 2006

В работе дается описание применения нового трехуровневого подхода к верификации программ на языке C#-light, который включает все основные последовательные конструкции языка C#. На первом этапе язык C#-light транслируется в промежуточный язык C#-kernel. На втором этапе, посредством аксиоматической семантики языка C#-kernel, порождаются «ленивые» условия корректности, включающие специальные функциональные символы, представляющие отложенное уточнение инвариантов меток и отложенные вызовы методов и делегатов. На третьем этапе эти условия уточняются с использованием алгоритмов операционной семантики. Цель работы — применить этот подход к верификации некоторых программ (изначально написанных на языке Java), вызвавших проблемы в системе ESC/Java. Также затронут вопрос упрощения условий корректности.

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

A.V. Promsky

APPLICATION OF THREE-LEVEL APPROACH TO C#-LIGHT PROGRAM VERIFICATION

Preprint
139

Novosibirsk 2006

This paper describes an application of the new three-level approach to verification of programs in C#-light, which is a significant sequential C# subset. At the first stage, C#-light is translated into an intermediate language C#-kernel. At the second stage, lazy verification conditions are generated by means of axiomatic semantics of C#-kernel. The verification conditions are lazy because they can include symbols representing postponed extractions of label invariants as well as postponed invocations of methods and delegates. At the third stage, lazy verification conditions are refined using some algorithms of operational semantics. The paper's goal is to apply our approach to verification of some programs (originally written in Java) which proved to be verification challenges in the system ESC/Java. In addition, the issue of verification condition simplification is considered.

ВВЕДЕНИЕ

Задача верификации программ была и остается одной из важнейших и самых трудных задач программирования. Особенно уровень сложности повысился с повсеместным использованием объектно-ориентированных (ОО) языков программирования. Семантические концепции таких языков требуют использования новых формальных методов и языков спецификаций, поскольку сочетание классической логики Хоара и языка первого порядка становится неадекватным поставленной задаче.

Данная работа посвящена разрабатываемому многоуровневому методу верификации программ на языке C#-light, являющемся широким подмножеством последовательного C# [5]. Заметим, что отсутствие многопоточного исполнения не умаляет достоинств метода, поскольку переход к параллелизму невозможен без решения проблем верификации последовательных однопоточных ОО-программ.

На концепцию нашего метода повлияло то, что преследовались две взаимоисключающие цели. Во-первых, необходим широкий охват языка C#, иначе метод не будет иметь практического (а тем более, коммерческого) интереса. Поэтому язык C#-light покрывает б**о**льшую часть языка C# за исключением, помимо упомянутой многопоточности, ряда реализационно-зависимых конструкций. Формальным определением языка C#-light является структурная операционная семантика (СОС). Во-вторых, хотелось бы основать метод на хорошо исследованном и зарекомендовавшем себя аксиоматическом подходе Хоара [6, 7, 13]. Поэтому в языке C#-light было выделено компактное ядро C#-kernel, для которого разработана логика Хоаровского типа. Первый уровень нашего метода состоит в трансляции исходной C#-light программы в язык C#-kernel. Корректность трансляции гарантируется тем, что формально определенный набор правил перевода допускает доказательство сохранения эквивалентности относительно СОС. На втором этапе в аксиоматической семантике языка C#-kernel происходит вывод «ленивых» (lazy) условий корректности (УК). «Ленивость» условий связана с тем, что они не являются окончательными формулами языка спецификаций и содержат специальные термы, связанные с нахождением инвариантов меток и динамически вызываемых функций. Доопределение или уточнение этих термов, невозможное средствами самой логики Хоара, происходит на третьем этапе с помощью некоторых алгоритмов СОС.

Для проверки возможностей нашего метода проводилась верификация программ из коллекции, предложенной в [15]. Фактически эти

программы (вернее, семантические концепции, стоящие за ними) были названы «вызовами» (challenges) для верификации Java-программ. Некоторые из примеров не поддаются верификации в системе ESC/Java либо требуют применения методов, выходящих за рамки этой системы. Языки Java и C# очень близки, поэтому примеры с минимальными изменениями подходят для C#-light. Также изучались примеры, характерные для C#, но не Java. Примеры были разбиты по следующим категориям (в работе описывается по одному примеру из каждой категории):

- передача управления и побочные эффекты;
- наследование;
- статическая инициализация;
- новые возможности C#: делегаты, события и т.п.

В ходе отработки нашего метода возник вопрос упрощения условий корректности. Проблема разрастания условий известна давно [10, 18], но в нашей модели языка спецификаций она становится еще острее. Был предложен и отработан ряд стратегий упрощения.

Работа имеет следующую структуру. В разд. 1 даются сведения, необходимые для изложения примеров. В частности, кратко описаны входные языки и используемые методы. В разд. 2 рассмотрены стратегии упрощения УК до этапа их доказательства. Основной материал работы — описание верификации пяти примеров излагается в разд. 3. Наиболее интересный из них рассмотрен подробно.

Данная работа была частично поддержана Microsoft Research в рамках проекта ROTOR в 2002–2003 гг., Лаврентьевским молодежным грантом СО РАН N 14 и грантом 04-01-00114a Российского Фонда Фундаментальных Исследований.

1. ПРЕДВАРИТЕЛЬНЫЕ СВЕДЕНИЯ

Рассмотрим кратко необходимые для понимания дальнейшего материала сведения. Детальную информацию по всем вопросам можно найти в [1, 21, 2, 22].

1.1. Используемые языки

Входными языками нашего метода являются язык C#-light и язык спецификаций, используемый для записи утверждений о свойствах C#-

light-программ. C#-kernel является внутренним языком, напрямую с которым пользователь не работает.

1.1.1. Язык C#-light

Как уже говорилось, язык C#-light не поддерживает многопоточное исполнение. Помимо этого запрещены следующие классы конструкций:

- атрибуты;
- деструкторы;
- оператор `using`;
- операции и операторы `checked` и `unchecked`;
- небезопасный код;
- директивы препроцессора.

Заметим, что все перечисленные конструкции либо требуют знаний о конкретной реализации платформы .Net[1], либо незначительно расширяют выразительную мощь языка C#, и поэтому ими можно пожертвовать.

Вместе с тем, язык C#-light остается достаточно мощным подмножеством C#. В частности, в нем присутствуют свойства, события, делегаты и индексаторы, отсутствующие в Java.

Кроме того, в C#-light вводится синтаксис для записи аннотаций. *Аннотация* — это комментарий вида

```
/// <a> R </a>
```

где `R` — это формула языка спецификаций (см. следующий разд.).

1.1.2. Язык спецификаций

Важную роль в методе Хоара играет язык спецификаций, используемый для записи утверждений о свойствах программ, в частности для пред- и постусловий и инвариантов циклов. Фактически многие свойства системы Хоара зависят от выразительной мощи этого языка. Как отмечалось ранее, классический язык первого порядка недостаточен для современных языков программирования. Существуют различные подходы к расширению языка первого порядка [6, 24, 26]. В нашем случае добавлены функции высших порядков и некоторые элементы λ-исчисления.

Типы. Допустимые типы языка спецификаций включают универсум $\mathcal{U}$, множество C# идентификаторов $\mathcal{N}$, множество абстрактных имен

[1] Т.е. достаточно низкоуровневой информации.

типов $\mathcal{T}$, функции $T \to T'$ и декартовы произведения $T \times T'$, где T, T' — одни из допустимых типов. Заметим, что $\mathcal{U}$ включает, по крайней мере, все C# литералы, множество $\mathcal{L}$ ячеек, множество $\mathcal{N}at$ натуральных чисел (включая ноль) и неопределенное значение ω.

Выражения. Допустимые выражения языка спецификаций включают константы, переменные, функциональные термы вида $s(s_1, \ldots, s_n)$ и λ-термы $\lambda(x, s)$ с обычной семантикой.

Мы также фиксируем функцию *upd* со следующей стандартной интерпретацией: если s — выражение типа $T \to T'$, e_1, e_2 — выражения типов T и T', соответственно, то $upd(s, e_1, e_2)$ — выражение типа $T \to T'$, значение которого совпадает с s всюду, кроме, может быть e_1, и $upd(s, e_1, e_2)(e_1) = e_2$.

Заметим, что описанный синтаксис выражений имеет префиксный вид, но для простоты далее часто будет использована привычная инфиксная запись.

1.1.3. Язык C#-kernel

Строго говоря, C#-kernel не является синтаксическим подмножеством языка C#-light, поскольку включает новый класс конструкций — метаинструкции. Однако абстрактная машина C#-light может их обрабатывать, поэтому семантически мы остаемся в рамках C#-light. Кратко C#-kernel можно описать как язык, удовлетворяющий следующим требованиям и ограничениям:

- в языке C#-kernel запрещены следующие конструкции C#-light:
  - пространства имен,
  - `using`-директивы,
  - операторы `break`, `continue`, `return`, `goto case`, `goto default`, `throw`, `try`, `switch`, `while`, `do`, `for`, `foreach`, операторы-объявления,
  - операции `||`, `&&`, `?:`, `new` и все операции присваивания;
- статический функциональный член может быть вызван только при условии, что соответствующие класс или структура уже проинициализированы[2];
- все метки, имена локальных переменных и имена локальных констант должны быть уникальны;

[2]Это верно для любого ОО языка, но в C#-kernel статическая инициализация производится явно в тексте программы, поэтому и требование формулируется явно!

- множества меток, имен типов, имен локальных переменных и имен локальных констант не пересекаются;
- разрешены только выражения следующего вида:
  - константа (литерал) c,
  - переменная x,
  - нормализованный[3] вызов $x.y(z_1, ..., z_n)$ или $y(z_1, ..., z_n)$, где x, y — имена, z_1, ..., z_n — имена или литералы,
  - метаинструкция[4] x `:=` e,
  - метаинструкция `new_instance()`,
  - метаинструкция `Init(`C`)`,
  - метаинструкции `catch(`T`,` x`)` и `catch(`x`)`;
- условием оператора `if` может быть только булевская переменная или результат вызова метаинструкций `catch`;
- в объявлениях полей и констант классов и структур запрещены инициализаторы. Вместо них, для каждого класса и структуры резервируются специальные методы `SFI` и `IFI`, в которых инициализируются статические и instance-поля соответственно.

Для читателя, знакомого с языком C#, идея перевода в C#-kernel может показаться чересчур экзотичной. Особенно удивляет присутствие метаинструкций, тем более, что в нашем подходе по верификации C-программ язык C-kernel является собственным подмножеством C-light [3, 4]. Чтобы прояснить ситуацию, предлагается следующая аналогия. В языке C можно использовать ассемблерные вставки. Также можно транслировать C в язык ассемблера, который, в свою очередь, транслируется в машинные команды. Задавая машинно-командную семантику ассемблерной программы, мы задаем и семантику исходной C-программы. Образно говоря, C#-kernel является языком ассемблера нашей абстрактной C#-light машины. И если для процедурного языка C-light мы обходились без всякого ассемблера, оставаясь в рамках языка C, то для объектно-ориентированного C#-light приходится изобретать и объектно-ориентированный ассемблер.

Замечание 1. Приведенная аналогия с ассемблерными вставками вероятно вызовет у читателя вопрос: можно ли делать то же самое в C#-light. Теоретически — да, более того, доказательство корректности перевода основано на способности абстрактной машины исполнять оба

[3]Понятие нормализованного вызова описано в стандарте [5].

[4]Операций такого вида нет в C#, к тому же они относятся к металогике программ, поэтому выбрано такое название.

языка, но на практике это никому не нужно. В нашем подходе использование C#-kernel является совершенно прозрачным, и в идеале пользователь не обязан знать что-либо о премежуточном представлении.

Замечание 2. Другой вероятный вопрос — почему в качестве ассемблера не используется язык CIL, тем более, что в ряде работ по верификации Java-программ верифицируется именно байт-код. Ответ таков: язык C#-kernel гораздо абстрактнее CIL и логика Хоара для него проще.

1.2. Используемые методы

В основном мы остаемся в рамках классического подхода, когда формальным определением языка является достаточно низкоуровневая операционная семантика, а для вывода утверждений о программах используется высокоуровневая логика Хоаровского типа. Наличие операционной семантики позволяет проводить исследования непротиворечивости и полноты логики Хоара.

1.2.1. Абстрактная машина языка C#-light

Как обычно, операционная семантика задается в терминах манипуляций над состояниями при интерпретации программных конструкций. В классическом подходе [6, 7], развитом для модельного языка while-программ и для языка Pascal, состояние — это отображение из множества программных переменных в их значения. Следовательно, имена переменных понимались как уникальные метки ячеек памяти, и состояние по метке выдавало содержимое ячейки. Есть две основные причины, по которым этот подход не работает для языка C#. Во-первых, разрешены локальные переменные с ограниченным временем жизни, следовательно, область определенности состояния может динамически изменяться. Во-вторых, наличие ссылок означает, что доступ к содержимому конкретной ячейки памяти может осуществляться не через единственную метку, а через произвольное количество меток. Поэтому изменение состояния через одну из меток должно автоматически вызывать и изменение через все оставшиеся ссылки для данной ячейки.

Для решения этой проблемы было предложено воспользоваться таким понятием теории компиляции, как распределение памяти. Напомним, что доступ к объекту по имени осуществляется только в исходном и объектном коде. В исполняемом коде доступ происходит по адресу. По-

этому предлагается ввести явное множество адресов объектов в памяти, и доступ к содержимым ячеек памяти будет «двухэтажным». Вначале по идентификатору мы получаем адрес соответствующего объекта в памяти, а уже по адресу получаем значение.

Замечание. Использование явных адресов вовсе не означает, что их значениями должны быть значения, используемые для адресов в конкретной архитектуре (например, 32-битные целые без знака). Их можно моделировать неинтерпретируемыми символьными константами. Единственное ограничение — все адреса уникальны.

Двухэтапный доступ к значениям объектов не единственное отличие от классической операционной семантики. В каждом конкретном состоянии нам необходимо знать информацию о типах объектов, о внутренней структуре составных объектов и о посланных исключениях. Формально *состояние* — это сопоставление значений следующим метапеременным[5] абстрактной машины.

1. L — метапеременная типа $\mathcal{N} \to \mathcal{L}$, определяющая ячейки памяти для переменных.
2. V — метапеременная типа $\mathcal{L} \to \mathcal{U}$, определяющая значения, хранимые в ячейках.
3. T — метапеременная типа $\mathcal{N} \cup \mathcal{U} \to \mathcal{T}$, определяющая абстрактные имена типов идентификаторов из $\mathcal{N}$ и значений из $\mathcal{U}$.
4. $L2$ — метапеременная типа $\mathcal{U} \times (\mathcal{N} \cup \mathcal{N}at) \to \mathcal{L}$, определяющая ячейки памяти для элементов массивов и полей.
5. $V0$ — метапеременная типа $\mathcal{U}$, хранящая значение последнего вычисленного выражения.
6. E — метапеременная типа $\mathcal{U}$, хранящая значение посланного исключения.

Непосредственно операционная семантика определяется как система правил вывода, интерпретирующая программные конструкции и переводящая входные состояния в выходные. Пример подобной системы можно найти в [20, 3].

1.2.2. Перевод из C#-light в C#-kernel

Заметим, что для верификации обычно используется не операционная, а аксиоматическая семантика Хоара. Однако для целого ряда конструкций языка C#-light невозможно определить логику Хоара в виде

[5]Приставка «мета» используется, чтобы отличить их от исходных переменных.

понятных и компактных правил и аксиом [6, 7]. Так возникает задача сведения языка C#-light к предельно простому языку C#-kernel.

Фундамент для трансляции C#-light в C#-kernel был заложен в работе по верификации языка C. Там составные конструкции языка C-light переводились в последовательности более простых конструкций, но по прежнему в синтаксисе C-light. Специфика языка C# требует переписывания и некоторых простых конструкций в новом синтаксисе, выходящем за рамки C#. Идея в следующем: если интерпретация конструкции в операционной семантике приводит к ряду изменений метапеременных, то конструкцию можно заменить на явную последовательность этих изменений. Так устанавливается связь между метапеременными и метаинструкциями. А именно, для любой метаинструкции вида x `:=` e, x может быть только одной из шести метапеременных.

Например, объявление вида `S x;` переписывается во фрагмент

```
new_instance();
L := upd(L, x, V0);
T := upd(T, L(x), Loc(S));
```

Исполнение метаинструкции `new_instance` приводит к выделению новой ячейки памяти, причем адрес ячейки помещается в метапеременную `V0`. Во второй строке этот адрес ассоциируется с именем `x`. Третья строка говорит о том, что новый адрес есть адрес объекта типа `S`.

Полное описание алгоритмов перевода из C#-light в C#-kernel приводится в [1].

1.2.3. Аксиоматическая семантика языка C#-kernel

Напомним, что классическая логика Хоара — это система вывода специальных формул вида $\{P\}\,S\,\{Q\}$, называемых тройками Хоара, где P и Q — это формулы языка спецификаций (логика первого порядка), а S — программа. Интуитивно истинность тройки Хоара означает, что если истинно *предусловие* P и программа S завершается, то в конце истинно *постусловие* Q. Ключевым моментом является то, что язык программирования и язык спецификаций «пересекаются» по переменным и присваиваниям. Следовательно, истинность формул первого порядка определяется через множества состояний, в которых верны эти формулы. Например, формула $x = 3$ характеризует все состояния, в которых в ячейке памяти x хранится значение 3. А классическая аксиома Хоара для присваивания имеет вид

$$\{P(x \leftarrow e)\}\, x := e\, \{P\} \tag{1}$$

либо эквивалентную форму

$$\{P\}\, x := e\, \{\exists x'.(P(x \leftarrow x') \wedge x = e(x \leftarrow x'))\}\ , \tag{2}$$

где стрелка означает подстановку.

На первый взгляд, использование метапеременных вместо программных переменных нарушает эту картину. Однако изменяются не только переменные, но и языки спецификаций и программирования. В итоге получается классический случай: довольно простой язык программирования, причем в любой программе всего шесть переменных (теперь называемых метапеременными); утверждения над этими метапеременными выразимы в языке спецификаций. Отсюда следует принципиальный вывод: **классическую логику Хоара можно использовать в качестве базы для языка C#-kernel**.

Стоит отметить, что простота структуры аксиоматической семантики дается ценой сложности формул языка спецификаций. Напомним, что первые четыре метапеременные являются отображениями. Например, утверждение $x = 3$ заменяется на $V(L(x)) = 3$. Еще раз отметим, что исходные программные переменные (в данном случае x) в C#-kernel становятся неинтерпретированными строковыми константами, так что уместно брать их в кавычки, но для простоты кавычки будем опускать.

Вывод в классической логике Хоара является неоднозначным процессом, поэтому на практике используют эквивалентную модифицированную систему, в которой на каждом шаге применимо ровно одно правило вывода. Для языка C#-kernel использована система прямого просмотра, когда обрабатывается самая левая конструкция. Недостатком прямого просмотра является необходимость использования более сложной формы (2) для присваивания. С другой стороны, такая система позволяет значительно сократить вывод, отбрасывая заведомо истинные тройки. Также сокращению вывода способствует использование окружения для троек Хоара. В окружении хранится информация о текущем методе, об инициализированных классах и информация о том, было ли на данном пути сгенерировано исключение.

Подробное описание аксиоматической семантики языка C#-kernel можно найти в [21, 22]. В качестве примера рассмотрим правило вывода для метаинструкции `new_instance`.

$$\frac{Env^{-} \vdash \{\exists V0'\ \exists d(newp(d, L, V, L2) \wedge P(V0 \leftarrow V0') \wedge V0 = d)\} A \{Q\}}{Env^{-} \vdash \{P\}\,\texttt{new\_instance();}\ A\,\{Q\}}\ .$$

Здесь A — это последовательность операторов (возможно пустая). Ее использование справа от конкретной конструкции (в данном случае — `new_instance`) — отличительный признак системы прямого просмотра. Формула $newp(d, L, V, L2)$, являющаяся сокращением для утверждения

$$\forall x \forall y(d \notin \{L(x), V(x), L2(x, y)\})\ ,$$

говорит о том, что d — это новая ячейка памяти. Использование знака "$-$" в окружении Env означает предположение об отсутствии исключений на данном пути вывода.

1.2.4. Уточнение «ленивых» условий корректности

Обычное использование аксиоматической семантики для верификации состоит в ее реализации в виде *генератора условий корректности* (УК). Это утверждения языка спецификаций, интерпретируемые в конкретной проблемной области. Истинность этого набора лемм является критерием корректности исходной аннотированной программы. Специфика нашего подхода к языку C# такова, что аксиоматическая семантика C#-kernel порождает неокончательные, «ленивые» УК. Связано это с проблемами динамического связывания и смещения инвариантов циклов.

Динамическое связывание. Концепция динамического (позднего) связывания в ОО языках имеет отношение к концепции виртуальных функций. Объявляя в некотором классе виртуальную функцию, пользователь фактически определяет интерфейс к целому множеству функций в классах-потомках, которые *замещают* (overriding) исходную. Информация о том, какая конкретно функция будет вызвана, в общем случае может быть определена только в момент вызова. С другой стороны, аксиоматическая семантика задает символическое исполнение программ без реального вычисления динамических свойств. Поэтому определить аксиоматическую семантику вызова полиморфной функции в общем случае можно только используя квантификацию по множеству замещающих функций [24, 25]. Разрешение такого квантора, хотя и является конечным перебором для любой конкретной программы, все же усложняет верификацию.

Вместе с тем, использование прямого просмотра предлагает способ частичного решения проблемы перебора. Напомним, что при прямом просмотре информация о значениях объектов (в частности информация о динамическом типе) накапливается в предусловии тройки Хоара. В каком случае предусловие для вызова виртуального метода содержит информацию о динамическом типе объекта? Во-первых, если операция создания объекта предшествует вызову метода на одном линейном участке. Во-вторых, если пользователь сам поместил эту информацию в предусловие, например, определяя предусловие тела функции или инвариант цикла. Зная эту информацию, можно для вызова использовать спецификации конкретной функции. Легко представить и худший случай. Например, если `T` — базовый класс, содержащий виртуальную функцию `f`, то для процедуры

```
void g(T x){ ... x.f(); ... }
```

описать в ее предусловии динамический тип переменной `x` в общем случае невозможно. В итоге, для вызова `x.f();` придется провести перебор по всем потомкам типа `T`.

Заметим, что извлечение информации о динамическом типе из предусловия осуществляется алгоритмами операционной семантики языка C#-light и непосредственное их включение в правила аксиоматики C#-kernel приведет к нежелательной громоздкости. Вместо этого было решено использовать специальный «ленивый» терм $CALL()$, благодаря чему правило для вызова метода (вне зависимости от того, виртуальный это метод или обычный) выглядит просто:

$$\frac{Env \vdash \{CALL(f, x, [args], mvs, \lambda(mvs, P))\}\, A\, \{Q\}}{Env^{-} \vdash \{P\}\, x\,.\,f\texttt{(}args\texttt{);}\ A\, \{Q\}}\ ,$$

где mvs — кортеж метапеременных. Формальный алгоритм уточнения терма $CALL$ описан в [21, 22]. Если f — простой метод либо виртуальный метод, но в предусловии P содержится информация о динамическом типе x, то терм $CALL$ превратится в некоторую конъюнкцию формулы P и спецификаций соответствующего метода с некоторыми подстановками относительно метапеременных. В противном случае добавится квантор по виртуальным методам[6].

Заметим, что для обработки вызовов делегатов языка C# используется похожая формула $DELCALL$ с более сложной семантикой [21, 22].

[6]Напомним, что наш язык спецификаций — это логика высших порядков.

Инварианты циклов. В отличие от денотационной семантики в логике Хоара не используется техника неподвижных точек, поэтому в ней невозможно выводить инварианты циклов. Общепринятый подход состоит в том, что инварианты циклов изначально задаются пользователем как часть спецификаций программы. Однако перевод из C#-light в C#-kernel усложняет ситуацию. Ведь операторы циклов `while`, `do`, `for` и `foreach` переписываются с помощью оператора `goto`. Передача управления в логике Хоара также подразумевает использование инварианта, но это уже инвариант метки, а не цикла. В общем случае некорректно использовать исходный инвариант цикла в качестве инварианта метки. При этом пытаться специфицировать промежуточную C#-kernel программу заведомо бесперспективная задача. Тем более, что при трансляции могут возникать передачи управления «вперед», никак не связанные с исходными циклами.

Решение проблемы инвариантов также основано на использовании «ленивых» термов. Идея в том, что каждый раз, встречая в программе оператор `goto L` или оператор, помеченный меткой `L`, аксиоматическая семантика подставляет в вывод символическую формулу $INV(mvs, L)$, где mvs — кортеж метапеременных. Алгоритм уточнения этой формулы можно найти в [21, 22], неформально же идея в следующем. Среди порожденных УК ищем все формулы вида $A \Longrightarrow INV(mvs, L)$, и берем дизъюнкцию всех A. Очевидно, что эта дизъюнкция будет истинна всякий раз, когда в программе управление попадает на метку `L`, т.е. является инвариантом метки. Среди оставшихся УК любое вхождение формулы вида $INV(\overline{e}, L)$, где $\overline{e}$ — это кортеж термов, заменяем на найденную дизъюнкцию, с подстановкой кортежа $\overline{e}$ вместо mvs. Заметим, что если управление на метку `L` передается только «сверху», то инвариант выводим в самой логике Хоара. Иначе возникает цикл, и чтобы не возникло формулы вида $INV \Longrightarrow INV$, пользователь должен предоставить некоторую формулу, которая «разрежет» тело цикла. Очевидно в качестве такой формулы можно взять исходный инвариант цикла. Таким образом, алгоритм уточнения доопределяет инвариант цикла до инварианта метки.

1.3. Родственные работы

В настоящее время существует большое количество работ по формализации семантики языка Java и систем верификации Java-программ. Язык C# как более новый имеет пока меньшую теоретическую и ин-

струментальную поддержки, но учитывая влияние фирмы Microsoft, сомневаться в его перспективах не приходится. Рассмотрим наиболее интересные работы, связанные с этими языками. Также будет затронут вопрос усложнения УК в реальных системах верификации.

Достойные внимания результаты по формализации семантик языков Java и C# достигнуты в [9, 14, 24, 25, 27, 28]. Отдельно отметим работу [11], выявившую ряд скрытых противоречий в стандарте C# [5]. Как правило, в этих работах либо задается подробная низкоуровневая семантика, слабо пригодная для верификации, либо изначально ставится цель верификации, но в результате поддерживается сильно ограниченное подмножество языка. Из работ, посвященных общим проблемам верификации объектно-ориентированных языков, отметим [19, 26].

В 1997 г. в университете г. Неймеген стартовал проект LOOP [16], который развивается до сих пор. Направлен он на автоматизированную верификацию программ на языке Java. Поддерживается большая часть языка за исключением многопоточной обработки и вложенных классов. Собственно LOOP представляет собой компилятор, написанный на языке O'Caml. На вход ему подается последовательная Java-программа и ее спецификации на языке Java Modelling Language JML. Выходом являются несколько файлов в синтаксисе системы автоматического доказательства теорем PVS, описывающих смысл программы и ее спецификаций. Для задания семантики объектов и классов используется алгебраический подход. Система успешно применялась для верификации программ на языке JavaCard, который используется в так называемых смарт-картах. В качестве недостатка отметим то, что система работает эффективно только для небольших программ.

Другим примером является проект Extended static checking for Java ESC/Java [17], поддерживающий широкое подмножество Java. Ключевой идеей является то, что система нацелена не на доказательство полной функциональной корректности, а на поиск типичных ошибок в программах. Это позволяет повысить мощь автоматического доказателя (ранее использовался Simplify, сейчас разрабатывается другая программа), хотя и ценой пропуска некоторых ошибок. В качестве языка спецификаций используется простой язык контрактов, являющийся подмножеством языка JML. Семантика описывается с помощью исчисления слабейшего предусловия (wp-calculus). Хотя в качестве критики часто указывают на неполноту и противоречивость выбранного подхода, система признана впечатляющей с точки зрения демонстрации потенциа-

ла верификации программ.

Большую известность в последнее время получила система Spec# [8]. Она полностью интегрируется в среду разработки Microsoft Visual Studio и платформу .NET Framework, поэтому обеспечивает полную инфраструктуру, включая библиотеки, инструменты, поддержку проектирования и средства редактирования. Система ориентирована на язык Spec#, который является расширением языка C#. Новые конструкции включают пользовательские спецификации, ненулевые (non-null) типы и некоторые средства высокоуровневой абстракции данных. Спецификации становятся частью исполнения программы и проверяются динамически. Также доступна статическая проверка спецификаций с помощью автоматического доказателя Boogie. Тот факт, что система разрабатывается в Microsoft Research и уже применяется на практике, служит гарантией хороших перспектив Spec#.

Вопрос усложнения УК для нетривиальных языков программирования напрямую влияет на реализуемость системы верификации. Уже в 1979 г. в работе [18] рассматривалась проблема экспоненциального разрастания УК для языка Pascal из-за использования конструкции *upd*. Там же были предложены алгоритмы линеаризации вложенных обращений к этой конструкции, что позволяло заметно упростить УК.

В [10] рассмотрен двухэтапный алгоритм генерации компактных УК в системе ESC/Java. На первом этапе исходный фрагмент транслируется в промежуточную «пассивную» форму, не содержащую присваиваний. На втором этапе используется техника генерации, оптимизированная для пассивной формы. В худшем случае алгоритм порождает УК квадратичного размера по отношению к исходному фрагменту, а на практике зависимость близка к линейной.

Другой подход состоит в предварительном упрощении УК до этапа их доказательства. В [12] рассмотрены стратегии для пропозициональных УК, имеющих дело с цифровыми схемами или алгоритмами синхронизации для параллельных программ. Изучение структурных свойств формул позволяет ввести понятия интерполяции и полярности, которые помогают обнаружить и отбросить доказуемо несущественные части булевских УК.

2. УПРОЩЕНИЕ УСЛОВИЙ КОРРЕКТНОСТИ

Двухуровневая модель доступа к объектам (адрес — значение) позволяет решить все проблемы с составными объектами и множественными ссылками, но приводит к сильному разрастанию УК. Рассмотрим простой пример:

C#-light program	**C#-kernel program**

`{ int x, y, z;`
`  x = 1;`
`  y = 2;`
`  z = 3;`
`  z = 4;`
`}`
	`{ Init(int);`
	`  new_instance();`
	`  L := upd(L, x, V0);`
	`  T := upd(T, L(x), Loc(int));`
	
	`  new_instance();`
	`  L := upd(L, y, V0);`
	`  T := upd(T, L(y), Loc(int));`
	
	`  new_instance();`
	`  L := upd(L, z, V0);`
	`  T := upd(T, L(z), Loc(int));`
	
	`  V := upd(V, L(x), 1);`
	`  V := upd(V, L(y), 2);`
	`  V := upd(V, L(z), 3);`
	`  V := upd(V, L(z), 4);`
	`}`

Исходная программа на C#-light фактически не выходит за рамки языка Pascal и ее можно было бы верифицировать с помощью классической логики Хоара [13]. В тоже время в нашем подходе исходная программа верифицируется опосредованно через верификацию эквивалентной программы на языке C#-kernel в правой колонке. Сравним УК, получаемые в этих двух подходах.

В качестве предусловия в обоих случаях можно взять true.

В качестве постусловия в классическом случае возьмем утверждение $\mathtt{x} = 1 \wedge \mathtt{y} = 2 \wedge \mathtt{z} = 4$. В результате четырехкратного применения правила

(1) получим тождественно истинное УК:

$$\mathsf{true} \Longrightarrow (1 = 1 \wedge 2 = 2 \wedge 3 = 3) \tag{3}$$

В нашей аксиоматике в постусловии необходимо детально указывать доступ к значениям: $V(L(\mathtt{x})) = 1 \wedge V(L(\mathtt{y})) = 2 \wedge V(L(\mathtt{z})) = 4$.

Первое же отличие при выводе состоит в удвоении числа УК. Дело в том, что первым оператором в C#-kernel программе стоит метаинструкция `Init(int)`, осуществляющая статическую инициализацию типа `int`. Если тип еще не инициализирован, то он инициализируется, иначе конструкция `Init` игнорируется[7]. Первое условие, в котором происходит инициализация типа `int`, выглядит так:

$$\left[\begin{array}{ll}
newp(d_4, L_4, V_2, L2) & \wedge \\
newp(d_3, L_3, V_2, L2) & \wedge \\
newp(d_2, L_2, V_2, L2) & \wedge \\
newp(d_1, L_1, V_1, L2) & \wedge \\
newp(e, L_1, V_1, L_2) & \wedge \\
E = \omega & \wedge \\
L_2 = upd(L_1, \mathtt{int}, d_1) & \wedge \\
V_2 = upd(V_1, d_1, e) & \wedge \\
\neg SI(\mathtt{int}, L_1, V_1, T_1, L2) & \wedge \\
V0_1 = d_2 & \wedge \\
L_3 = upd(L_2, \mathtt{x}, V0_1) & \wedge \\
T_2 = upd(T_1, L_3(\mathtt{x}), Loc(\mathtt{int})) & \wedge \\
V0_2 = d_3 & \wedge \\
L_4 = upd(L_3, \mathtt{y}, V0_2) & \wedge \\
T_3 = upd(T_2, L_4(\mathtt{y}), Loc(\mathtt{int})) & \wedge \\
V0 = d_4 & \wedge \\
L = upd(L_4, \mathtt{z}, V0) & \wedge \\
T = upd(T_3, L(\mathtt{z}), Loc(\mathtt{int})) & \wedge \\
V_3 = upd(V_2, L(\mathtt{x}), 1) & \wedge \\
V_4 = upd(V_3, L(\mathtt{y}), 2) & \wedge \\
V_5 = upd(V_4, L(\mathtt{z}), 3) & \wedge \\
V = upd(V_5, L(\mathtt{z}), 4) &
\end{array}\right] \Longrightarrow \left[\begin{array}{ll}
V(L(\mathtt{x})) = 1 & \wedge \\
V(L(\mathtt{y})) = 2 & \wedge \\
V(L(\mathtt{z})) = 4 &
\end{array}\right] \tag{4}$$

Здесь все индексированные переменные связаны кванторами существования, поскольку это новые имена из правила (2). Для простоты кван-

[7]Подробнее проблемы статической инициализации обсуждаются в разд. 3.3.

торы опущены. Отметим отрицание предиката $SI(\texttt{int}, \ldots)$, который используется для указания инициализированности типа.

Во-втором УК предполагается, что тип `int` уже инициализирован, поэтому условие немного проще:

$$\left[\begin{array}{ll}
newp(d_3, L_3, V_1, L2) & \wedge \\
newp(d_2, L_2, V_1, L2) & \wedge \\
newp(d_1, L_1, V_1, L2) & \wedge \\
\boxed{E = \omega} & \wedge \\
\boxed{SI(\texttt{int}, L_1, V_1, T_1, L2)} & \wedge \\
\boxed{V0_1 = d_1} & \wedge \\
L_2 = upd(L_1, \texttt{x}, V0_1) & \wedge \\
\boxed{T_2 = upd(T_1, L_2(\texttt{x}), Loc(\texttt{int}))} & \wedge \\
\boxed{V0_2 = d_2} & \wedge \\
L_3 = upd(L_2, \texttt{y}, V0_2) & \wedge \\
\boxed{T_3 = upd(T_2, L_3(\texttt{y}), Loc(\texttt{int}))} & \wedge \\
\boxed{V0 = d_3} & \wedge \\
L = upd(L_3, \texttt{z}, V0) & \wedge \\
\boxed{T = upd(T_3, L(\texttt{z}), Loc(\texttt{int}))} & \wedge \\
V_2 = upd(V_1, L(\texttt{x}), 1) & \wedge \\
V_3 = upd(V_2, L(\texttt{y}), 2) & \wedge \\
V_4 = upd(V_3, L(\texttt{z}), 3) & \wedge \\
V = upd(V_4, L(\texttt{z}), 4)
\end{array}\right] \Longrightarrow \left[\begin{array}{ll}
V(L(\texttt{x})) = 1 & \wedge \\
V(L(\texttt{y})) = 2 & \wedge \\
V(L(\texttt{z})) = 4 &
\end{array}\right] \quad (5)$$

Замечание. Напомним, что в семантике C#-kernel выводимые УК сопровождаются окружением, в котором они выводились. Но для данного вырожденного примера, где нет ни функций, ни исключений, ни пользовательских типов, окружение не так важно.

При сравнении данных УК с (3) возникает ощущение громоздкости, превышающей все разумные пределы. Причем даже для такого тривиального примера. Очевидно, что при верификации реальных программ пользователь системы столкнется с еще более сложными условиями. Таково следствие детальной модели памяти в C#-light/kernel. Даже использование систем автоматизированного доказательства теорем не является панацеей, поскольку легко достичь предела возможностей системы. В [23] упоминалась программа из 400 строк на языке C, с которой

не справилась система HOL.

Необходимо разработать стратегии упрощения порождаемых УК. Очевидно эти преобразования не должны изменять истинность утверждений. В качестве примера рассмотрим стратегии упрощения УК (5).

Стратегия 1. Рассмотрим конъюнкты, обведенные рамкой. Формула $E = \omega$ говорит о том, что содержимое ячейки E неопределено, т.е. никакое исключение не было сгенерировано на данном пути. В семантике C#-kernel в окружении вывода содержится информация о факте генерации исключений. Для данного примера окружение было опущено, но очевидно, что в нем будет указано об отсутствии исключений. Тем самым, утверждение $E = \omega$ истинно. Аналогично при обработке метаинструкции `Init(int)` в окружении тип `int` будет добавлен ко множеству инициализированных типов. Поэтому утверждение $SI(\texttt{int}, L_1, V_1, T_1, L2)$ является избыточным. Подробнее понятие согласованности окружения с информацией об инициализации типов рассмотрено в [21, 2].

Далее, равенства вида *переменная*$_1$ = *переменная*$_2$ являются просто переименованиями. Можно протянуть первую переменную вместо всех вхождений второй.

Наконец, при доказательстве того, что значения переменных `x`, `y` и `z` есть 1, 2 и 3 соответственно, существенно только наличие соответствующих обновлений метапеременных L и V. Информация же о типах переменных не используется. Более того, консеквент в (5) вообще не содержит вхождений метапеременной T. Следовательно, можно опустить конъюнкты для метапеременных T_2, T_3 и T. Чтобы отбрасывание этой информации было абсолютно корректным, необходимо убедиться в том, что формула

$$\begin{array}{ll} T_2 = upd(T_1, L_2(\texttt{x}), Loc(\texttt{int})) & \wedge \\ T_3 = upd(T_2, L_3(\texttt{y}), Loc(\texttt{int})) & \wedge \\ T = upd(T_3, L(\texttt{z}), Loc(\texttt{int})) & \end{array}$$

не ложна, что достаточно очевидно.

Удаляем соответствующие конъюнкты в УК (5):

$$\left[\begin{array}{ll} newp(d_3, L_3, V_1, L2) & \wedge \\ newp(d_2, L_2, V_1, L2) & \wedge \\ newp(d_1, L_1, V_1, L2) & \wedge \\ L_2 = upd(L_1, \mathtt{x}, d_1) & \wedge \\ L_3 = upd(L_2, \mathtt{y}, d_2) & \wedge \\ L = upd(L_3, \mathtt{z}, d_3) & \wedge \\ V_2 = upd(V_1, L(\mathtt{x}), 1) & \wedge \\ V_3 = upd(V_2, L(\mathtt{y}), 2) & \wedge \\ V_4 = upd(V_3, L(\mathtt{z}), 3) & \wedge \\ V = upd(V_4, L(\mathtt{z}), 4) & \end{array}\right] \Longrightarrow \left[\begin{array}{ll} V(L(\mathtt{x})) = 1 & \wedge \\ V(L(\mathtt{y})) = 2 & \wedge \\ V(L(\mathtt{z})) = 4 & \end{array}\right] \qquad (5')$$

Стратегия 2. В консеквенте программные переменные сравниваются с константами. Можно разбить всю формулу (5′) на три меньших (по числу конъюнктов в консеквенте). Рассмотрим одну из них:

$$\left[\begin{array}{ll} newp(d_3, L_3, V_1, L2) & \wedge \\ newp(d_2, L_2, V_1, L2) & \wedge \\ newp(d_1, L_1, V_1, L2) & \wedge \\ L_2 = upd(L_1, \mathtt{x}, d_1) & \wedge \\ L_3 = upd(L_2, \mathtt{y}, d_2) & \wedge \\ L = upd(L_3, \mathtt{z}, d_3) & \wedge \\ V_2 = upd(V_1, L(\mathtt{x}), 1) & \wedge \\ V_3 = upd(V_2, L(\mathtt{y}), 2) & \wedge \\ V_4 = upd(V_3, L(\mathtt{z}), 3) & \wedge \\ V = upd(V_4, L(\mathtt{z}), 4) & \end{array}\right] \Longrightarrow V(L(\mathtt{z})) = 4 \qquad (5'')$$

Стратегия 3. Тип `int` в C# не является ссылочным типом, а указателей в C#-light нет. Поэтому на объект `z` больше нет никаких ссылок, т.е. присваивания программным переменным `x` и `y` влияют на `z`, только если переменной `z` присваиваются значения выражений над `x` и `y`. Начнем разворачивать антецедент от самого правого конъюнкта. Если в термах *upd* с `z` связываются значения, отличные от выражений над другими программными переменными, то будем протягивать их в консеквент, а все другие присваивания будем игнорировать. Также будем учитывать, что более правый (нижний) конъюнкт имеет дело с более

поздним присваиванием, нежели левый (верхний).

$$\begin{array}{l} \left[\begin{array}{ll} newp(d_3, L_3, V_1, L2) & \wedge \\ newp(d_2, L_2, V_1, L2) & \wedge \\ newp(d_1, L_1, V_1, L2) & \end{array}\right] \\ \Longrightarrow \\ upd(V_4, upd(L_3, \mathtt{z}, d_3)(\mathtt{z}), 4)(upd(L_3, \mathtt{z}, d_3)(\mathtt{z})) = 4 \end{array} \qquad (5''')$$

Если воспользоваться определением функции *upd* из п. 1.1.2, то истинность этого УК очевидна. Хотя формула $(5''')$ по прежнему сложнее, чем (3), она гораздо проще исходного УК (5). Можно также заметить, что доказательство заключения в $(5''')$ не зависит от того, что d_1, d_2 и d_3 — это новые адреса, т.е. посылку тоже можно отбросить.

Таковы наиболее общие стратегии для упрощения УК в нашем методе. При рассмотрении примеров верификации условия будут даны в уже упрощенном виде. Вместе с тем, каждый из примеров предполагает некие специфические стратегии, применимые только к нему. Эти особые стратегии будут поясняться по мере необходимости.

3. ПРИМЕРЫ ВЕРИФИКАЦИИ

Большая часть примеров в данном разделе позаимствована из работы [15]. Естественно, программы на языке Java были переведены в C#. Стоит заметить, что для данных семантических концепций языки Java и C# синтаксически (да и семантически) похожи, поэтому перевод является тривиальным.

Также заметим, что в [15] приводились еще два примера, которые имеют дело с действительно вызовами для верификации. Первый из них относится к проблеме повторного обращения (re-entrance), которая имеет принципиальный характер вообще для ОО-программирования, а не конкретно для Java или C#. Речь идет и потенциальном нарушении инварианта класса в случае, когда из метода этого класса происходит вызов другого (или этого же) метода. Фактически решением этой проблемы является только официальное разрешение такого локального нарушения. В нашем подходе мы не используем концепцию инварианта класса, поэтому эта проблема не актуальна. Второй пример связан с проблемой зацикливания. А именно, счетчик цикла изменяется от `Byte.MIN_VALUE` до `Byte.MAX_VALUE`. Поскольку `Byte.MAX_VALUE + 1 = Byte.MIN_VALUE`, данный цикл работает бесконечно. Однако в нашем методе не используется понятие тотальной корректности, поэтому

проблема незавершимости неформализуема. К тому же для верификации этого примера требуется моделировать и переполнения, что также выходит за рамки нашего подхода.

Помимо примеров из [15] внимание было уделено нововведениям языка C# по сравнению с Java, таким как делегаты и события [29]. Поскольку в C# события реализуются через делегаты, здесь будет рассмотрен один пример для делегатов.

3.1. Множественные ссылки

Как уже говорилось, множественные ссылки или синонимы (aliasing) означают, что доступ к некоторому объекту в памяти происходит через более чем одну ссылку. С точки зрения реализации языка это не представляет проблемы, но в классической логике Хоара это приводит просто к некорректности аксиомы для присваивания (1). Например, если известно, что `x` и `y` ссылаются на один объект, то следующая тройка Хоара ложна:

$$\{x = 1\}\,\texttt{y := 2}\,\{x = 1\}\ .$$

Явное моделирование памяти посредством метапеременных позволяет решить эту проблему.

Пример для множественных ссылок из [15] подходит для C# без всяких синтаксических адаптаций и имеет простой вид:

```
class C {
    C a;
    int i;

    C(){ a = null; i = 1; }
}

class Alias {
    int m(){
        C c = new C();
        c.a = c;
        c.i = 2;
        return c.i + c.a.i;
    }
}
```

Легко заметить, что при возврате значения в методе `Alias.m()` обращение к полю `i` происходит как через объект `c`, так и через его поле `a`, ссылающееся на само `c`. Вообще говоря, в [15] отмечалось, что пример без проблем был верифицирован в системе ESC/Java, но авторы включили его в коллекцию как некий минимум, необходимый для моделирования языка, подобного Java.

Спецификации для конструктора класса `C` и метода `Alias.m()` имеют вид[8]:

Pre(`C.C`) : true
Post(`C.C`) : $V(L(\mathtt{a})) = \mathsf{null} \wedge V(L(\mathtt{i})) = 1$
Pre(`Alias.m`) : true
Post(`Alias.m`) : $V0 = 4$

Верификация конструктора класса `C` не представляет особого интереса, рассмотрим метод `Alias.m()`.

Во-первых, исходная программа транслируется в промежуточную C#-kernel программу:

```
class C {
    public static void SFI() {}
    public void IFI_C(){
        Init(object);
        this.IFI_object();
        new_instance();
        L := upd(L, field_a, V0);
        T := upd(T, L(field_a), Loc(C));
        L2 := upd(L2, <V(this), a>, L(field_a));
        new_instance();
        L := upd(L, field_i, V0);
        T := upd(T, L(field_i), Loc(int));
        L2 := upd(L2, <V(this), i>, L(field_i));
        V := upd(V, L2(this, a), null);
        Init(int);
        V := upd(V, L2(this, i), 0);
    }

    C a;
    int i;

    C(){
```

[8]Как правило, используются адаптации соответствующих спецификаций из [15].

```
        V := upd(V, L2(this, a), null);
        V := upd(V, L2(this, i), 1);
    }
}

class Alias {
    public static void SFI() {}
    public void IFI_Alias(){
        Init(object);
        this.IFI_object();
    }

    int m(){
        new_instance();
        L := upd(L, c, V0);
        T := upd(T, L(c), Loc(C));
        Init(C);
        new_instance();
        L := upd(L, x, V0);
        T := upd(T, L(x), Loc(C));
        new_instance();
        V := upd(V, L(x), V0);
        T := upd(T, V(x), C);
        x.IFI_C();
        x.C();
        V := upd(V, L(c), V(x));
        V := upd(V, L2(c, a), V(c));
        V := upd(V, L2(c, i), 2);
        {
            T := upd(T, y1, int);
            L := upd(L, y1, L2(c, i));
            T := upd(T, y2, int);
            L := upd(L, y2, L2(V2(c, a), i));
            int.+(y1, y2);
            goto __ExitPoint;
        }
        __ExitPoint: ;
    }
}
```

Отметим появление инициализирующих методов. Статических членов в классах `C` и `Alias` нет, поэтому методы `SFI` являются пустыми. В методах `IFI` происходит инициализация типа `object`, являющегося базовым для всех классов в C#. Спецификой алгоритма перевода также является активное использование вспомогательных переменных. Например, в методе `Alias.m()` появляются имена `x`, `y1`, `y2`.

В разд. 2 рассматривались стратегии упрощения УК. Анализ C#-kernel программы показывает, что значительного упрощения можно достичь еще до этапа вывода самих УК. Дело в том, что алгоритмы перевода для отдельных конструкций языка C#-light слабо зависят от контекста, в котором находятся конструкции. Например, если оператор заменяется на последовательность операторов, то эту последовательность в общем случае необходимо обрамлять фигурными скобками. Но довольно часто скобки оказываются избыточными и можно их удалить. В частности, оператор `return` в методе `Alias.m()` находился на верхнем уровне и появившийся на его месте блок можно заменить на последовательность. Более того, этот `return` был последним оператором в теле метода, и моделирование его передачей управления на метку `__ExitPoint` в данном конкретном примере избыточно.

Другой источник избыточности — строгое следование спецификациям C# при моделировании инициализации типов. Но при верификации данного примера совершенно не важно то, что тип `object` является базовым для иерархии типов языка C#. Аналогично из всех свойств целочисленного типа для верификации метода `Alias.m()` достаточно того факта, что $2+2=4$. А то, что, например `Int32.MaxValue` $= 2^{32}-1$, совершенно но важно. Поэтому из C#-kernel программы можно удалить явную инициализацию библиотечных типов, если она не существенна.

Замечание. Полная C#-kernel программа приведена здесь для иллюстрации работы алгоритма перевода в общем случае. Во всех дальнейших примерах промежуточные программы будут оптимизированы с учетом вышеизложенного.

Для вывода УК используется аксиоматическая семантика C#-kernel. Ее особенностью является то, что при обработке вызова функции моделируются две ситуации: нормальное исполнение функции и исполнение, приведшее к генерации исключения. Поэтому в отличие от обычной логики Хоара для линейного участка будет порождено не одно УК, а $n+1$ условие, где n — число вызовов методов, отличных от `SFI` и `IFI`, на данном участке[9]. Семантика вызова инициализирующих методов состоит в текстуальной подстановке их тел в точку вызова. При отсутствии перехвата исключений на самом участке, УК, связанные с «ненормальным» исполнением, окажутся тождественно истинными, поскольку имеют вид $\mathsf{false} \Longrightarrow \Phi$.

Для метода `Alias.m()` после разворачивания вызова `x.IFI_C()` будут порождены 16 УК, 14 из которых являются тавтологиями в силу вышесказанного. Два оставшихся соответствуют нормальному исполнению от начала метода до конца в зависимости от инициализации класса `C`. В их истинности необходимо убедиться для верификации метода. Рассмотрим одно из них (второе выглядит сложнее, но тоже является истинным):

$$Env \vdash \left[\begin{array}{ll} CALL(C, \mathtt{x}, [], [L_{29}, V_{25}, T_{28}, L2, V0, E], & \\ \qquad \lambda(mvs, \Phi)) & \wedge \\ E = \omega & \wedge \\ V_{26} = upd(V_{25}, L_{29}(\mathtt{c}), V_{25}(L_{29}(\mathtt{x}))) & \wedge \\ V_{27} = upd(V_{26}, L2(V_{26}(L_{29}(\mathtt{c})), \mathtt{a}), V_{26}(L_{29}(\mathtt{c}))) & \wedge \\ V = upd(V_{27}, L2(V_{27}(L_{29}(\mathtt{c})), \mathtt{i}), 2) & \wedge \\ T_{30} = upd(T_{28}, \mathtt{y1}, \mathtt{int}) & \wedge \\ L_{31} = upd(L_{29}, \mathtt{y1}, L2(V(L_{29}(\mathtt{c})), \mathtt{i})) & \wedge \\ T = upd(T_{30}, \mathtt{y2}, \mathtt{int}) & \wedge \\ L = upd(L_{31}, \mathtt{y2}, L2(V(L2(V(L_{31}(\mathtt{c})), \mathtt{a})), \mathtt{i})) & \wedge \\ V0 = V(L(\mathtt{y1})) + V(L(\mathtt{y2})) & \end{array}\right] \Rightarrow V0 = 4$$

где $Env = (\mathtt{Alias.m()}, \mathsf{false}, \{\mathtt{Alias}, \mathtt{C}\})$ и

[9]Здесь предполагается, что на участке нет конструкций, удваивающих вывод, таких, как условный оператор или `Init`.

$$\Phi \equiv \left[\begin{array}{ll}
newp(d_{19}, L_{20}, V_{23}, L2_{22}) \wedge newp(d_{14}, L_{15}, V_{23}, L2_{17}) & \wedge \\
newp(d_{10}, L_{15}, V_{11}, L2_{17}) \wedge newp(d_6, L_7, V_{11}, L2_{17}) & \wedge \\
newp(d_2, L_3, V_{11}, L2_{17}) \wedge L_3(\texttt{this}) \notin \{\omega, \mathsf{null}\} & \wedge \\
V_{11}(L_3(\texttt{this})) \neq \omega \wedge T_4(V_{11}(L_3(\texttt{this}))) \subseteq \texttt{Alias} & \wedge \\
SI(T_4(V_{11}(L_3(\texttt{this}))), L_3, V_{11}, T_4, L2_{17}) \wedge E = \omega & \wedge \\
SI(\texttt{Alias}, L_3, V_{11}, T_4, L2_{17}) & \wedge \\
L_7 = upd(L_3, \texttt{c}, d_2) & \wedge \\
T_8 = upd(T_4, L_7(\texttt{c}), Loc(\texttt{C})) & \wedge \\
SI(\texttt{C}, L_7, V_{11}, T_8, L2_{17}) & \wedge \\
L_{15} = upd(L_7, \texttt{x}, d_6) & \wedge \\
T_{12} = upd(T_8, L_{15}(\texttt{x}), Loc(\texttt{C})) & \wedge \\
V_{23} = upd(V_{11}, L_{15}(\texttt{x}), d_{10}) & \wedge \\
T_{16} = upd(T_{12}, V_{23}(L_{15}(\texttt{x})), \texttt{C}) & \wedge \\
L_{20} = upd(L_{15}, \texttt{field_a}, d_{14}) & \wedge \\
T_{21} = upd(T_{16}, L_{20}(\texttt{field_a}), Loc(\texttt{C})) & \wedge \\
L2_{22} = upd(L2_{17}, \langle V_{23}(L_{20}(\texttt{x})), \texttt{a}\rangle, L_{20}(\texttt{field_a})) & \wedge \\
L = upd(L_{20}, \texttt{field_i}, d_{19}) & \wedge \\
T = upd(T_{21}, L(\texttt{field_i}), Loc(\texttt{int})) & \wedge \\
L2 = upd(L2_{22}, \langle V_{23}(L(\texttt{x})), \texttt{i}\rangle, L(\texttt{field_i})) & \wedge \\
V_{24} = upd(V_{23}, L2(V_{23}(L(\texttt{x})), \texttt{a}), \mathsf{null}) & \wedge \\
V = upd(V_{24}, L2(V_{24}(L(\texttt{x})), \texttt{i}), 0) &
\end{array}\right]$$

Как и в разд. 2, кванторы по индексным переменным для простоты опущены. В окружении *Env* говорится о том, что вывод проводился для метода `Alias.m()`, что исключений не было (false), а типы `Alias` и `C` инициализированы.

При доказательстве данного УК можно воспользоваться тем фактом, что в программе не используются начальные значения полей класса `C` — ни значения по умолчанию, ни значения из конструктора класса. Поэтому можно полностью отбросить вызов конструктора, содержащий громоздкую формулу Φ. Применяя также стратегии из разд. 2, получим

следующую формулу[10]:

$$\exists V'\ \exists V''\ \exists V'''\ \exists L'\ \exists L''.$$
$$\bigwedge \left[\begin{array}{lcl} V'' & = & upd(V', L'(\mathtt{c}), V'(L'(\mathtt{x}))) \\ V''' & = & upd(V'', L2(V''(L'(\mathtt{c})), \mathtt{a}), V''(L'(\mathtt{c}))) \\ V & = & upd(V''', L2(V'''(L'(\mathtt{c})), \mathtt{i}), 2) \\ L'' & = & upd(L', \mathtt{y1}, L2(V(L'(\mathtt{c})), \mathtt{i})) \\ L & = & upd(L'', \mathtt{y2}, L2(V(L2(V(L''(\mathtt{c})), \mathtt{a})), \mathtt{i})) \\ V0 & = & V(L(\mathtt{y1})) + V(L(\mathtt{y2})) \end{array}\right] \Rightarrow V0 = 4\ ,$$

истинность которой непосредственно следует из свойств функции *upd*.

3.2. Прерывание цикла

Выход из цикла, не предусмотренный условием цикла, как правило, осуществляется с помощью оператора `break`, реже — с помощью `goto`. Передача управления представляет проблему для верификации, поскольку усложняет анализ потока управления.

Исторически в среде программистов-теоретиков сложилось негативное мнение об операторе `goto`. Однако для логики Хоара этот оператор гораздо лучше, чем такие операторы, как `break`, `continue` или `return`. Как уже упоминалось, аксиоматическая семантика `goto` определяется с помощью понятия инварианта метки, на которую передается управление, и самый популярный вариант тройки Хоара для этого оператора выглядит как

$$\{Inv(\mathtt{l})\}\ \mathtt{goto\ l}\ \{\mathsf{false}\}\ .$$

Данная аксиома означает, что при передаче управления на метку `l` необходимо убедиться в истинности соответствующего инварианта. Ложь в качестве постусловия означает, что оператор, непосредственно следующий за `goto l`, недостижим, так как управление передается в другую точку программы.

Однако для оператора `break` нет никакой метки, поэтому непонятно, что можно взять в качестве предусловия. Очевидно, что семантика этого оператора зависит от того, в каком операторе цикла он находится. Поэтому в классической логике Хоара его семантику можно было бы задать только одновременно с правилом для соответствующего цикла, что приведет к неимоверному усложнению семантики обоих операторов.

[10] Для ясности переименуем индексные переменные и вернем кванторы.

В настоящее время единственная достаточно простая аксиоматизация представлена в [14]. Она основана на том факте, что `break` передает управление только вперед, поэтому его семантику можно смоделировать как протягивание некоторого исключения до нужной точки программы. Однако недостатком является усложнение понятия частичной корректности.

В нашем методе этот оператор заменяется на передачу управления на новую метку, помечающую первый после цикла оператор. Это позволяет использовать отработанную семантику оператора `goto`.

Исходная C#-light программа имеет вид

```
class C {
    int[] ia;

    void NegateFirst(){
        for (int i = 0; i < ia.Length; i++){
            if (ia[i] < 0){
                ia[i] = -ia[i];
                break;
            }
        }
    }
}
```

При достижении первого отрицательного элемента массива его знак меняется, и происходит выход из цикла.

В работе [15] не совсем четко представлена проблема, которую вызвала соответствующая Java-программа в системе ESC/Java. Фактически же применение безусловного выхода из цикла в сочетании с возможным изменением массива приводит к усложнению спецификаций. Пред/постусловие для метода `NegateFirst` и инвариант цикла `for` имеют следующий вид:

$$\begin{aligned}
\mathrm{Pre}(\texttt{NF()}) \;:\; & \exists old : \texttt{int[]}.\ V(L2(V(L(\texttt{this})), \texttt{ia})) \neq \mathsf{null} \wedge \\
& \qquad V(L2(V(L(\texttt{this})), \texttt{ia})) = old \\
\mathrm{Post}(\texttt{NF()}) \;:\; & \forall i.\ (0 \leq i \leq V(L2(V(L(\texttt{this})), \texttt{ia})).\texttt{Length} \Longrightarrow \\
& \qquad ((old[i] < 0 \wedge (\forall j.\ 0 \leq j < i \Rightarrow old[j] \geq 0)) \Rightarrow \\
& \qquad\qquad V(L2(V(L(\texttt{this})), \texttt{ia}))[i] = -old[i]) \wedge \\
& \qquad old[i] \geq 0 \Rightarrow V(L2(V(L(\texttt{this})), \texttt{ia}))[i] = old[i])
\end{aligned}$$

$$\begin{aligned}\text{Inv(for)} \quad : \quad & 0 \le V(L(\texttt{i})) \le V(L2(V(L(\texttt{this})), \texttt{ia})).\texttt{Length} \wedge \\ & (\forall j.\ 0 \le j < V(L(\texttt{i})) \Rightarrow \\ & \quad (V(L2(V(L(\texttt{this})), \texttt{ia}))[j] \ge 0 \wedge \\ & \quad\ V(L2(V(L(\texttt{this})), \texttt{ia})) = old[j])) \ .\end{aligned}$$

Для верификации нам необходимо показать, что первый отрицательный элемент массива меняет знак, а все остальные элементы не меняются. Поэтому содержимое массива запоминается во вспомогательной переменной-массиве *old*.

Результат трансляции (после оптимизаций) имеет вид:

```
class C {
    public void IFI(){
        new_instance();
        L := upd(L, field_ia, V0);
        T := upd(T, L(field_ia), Loc(int[]));
        L2 := upd(L2, <V(this), ia>, L(field_ia));
        V := upd(V, L2(this, ia), null);
    }

    int[] ia;

    void NegateFirst(){
        new_instance();
        L := upd(L, i, V0);
        T := upd(T, L(i), Loc(int));
        V := upd(V, L(i), 0);
        new_instance();
        L := upd(L, b0, V0);
        T := upd(T, L(b0), Loc(bool));
        new_instance();
        L := upd(L, x1, V0);
        T := upd(T, L(x1), Loc(int[]));
        L1:;
        V0 := V2(this, ia);
        V := upd(V, L(x1), V0);
        x1.get_Length();
        V := upd(V, L(x0), V0);
        int.<(i, x0);
        V := upd(V, L(b0), V0);
```

```
        if (b0){
            new_instance();
            L := upd(L, b1, V0);
            T := upd(T, L(b1), Loc(bool));
            new_instance();
            L := upd(L, x3, V0);
            T := upd(T, L(x3), Loc(int));
            V0 := V2(V2(this, ia), i);
            V := upd(V, L(x3), V0);
            new_instance();
            L := upd(L, x4, V0);
            T := upd(T, L(x4), Loc(int));
            V := upd(V, L(x4), 0);
            int.<(x3, x4);
            V := upd(V, L(b1), V0);
            if (b1){
                new_instance();
                L := upd(L, x4, V0);
                T := upd(T, L(x4), Loc(int));
                V := upd(V, L(x4), V2(V2(this, ia), i));
                int.-(x4);
                V := upd(V, L2(V2(this, ia), i), V0);
                goto L0;
            }
            new_instance();
            L := upd(L, x5, V0);
            T := upd(T, L(x5), Loc(int));
            V := upd(V, L(x5), V(i));
            int.++(i);
            V := upd(V, L(i), V0);
            V0 := V(x5);
            goto L1;
        }
        L0:;
    }
}
```

Обратим внимание на то, что оператор `break` моделируется передачей управления на метку `L0` и исходный цикл моделируется обратной пе-

редачей управления на метку `L1`. Для упрощения семантики метка в C#-kernel всегда помечает пустой оператор, поэтому используется запись `L1:;`.

Условия корректности для данного примера являются достаточно громоздкими. Рассмотрим одно из них, соответствующее исполнению от начала метода `C.NegateFirst()` до точки входа в цикл.

$$Env \vdash \left[\begin{array}{l} CALL(\texttt{<}, \texttt{int}, [\texttt{i}, \texttt{x0}], [L, V''', T, L2, V0, E], \lambda(mvs, \Phi)) \wedge \\ E = \omega \wedge \\ V = upd(V''', L(\texttt{b0}), V0) \end{array}\right] \Rightarrow \text{Inv}(\texttt{for})$$

Здесь $Env \equiv (\texttt{NegateFirst}, \mathsf{false}, \{\texttt{C}\})$ и

$$\Phi \equiv \left[\begin{array}{l} CALL(\ \texttt{get\_Length}, \texttt{x1}, [], [L, V'', T, L2, V0, E], \\ \qquad \lambda(mvs, (\ INV([L, V', T, L2, V0', E], \texttt{L1}) \wedge \\ \qquad\qquad V0 = V'(L2(V'(L(\texttt{this})), \texttt{ia})) \wedge \\ \qquad\qquad V = upd(V', L(\texttt{x1}), V0))) \\ ) \wedge \\ E = \omega \wedge \\ V = upd(V'', L(\texttt{x0}), V0) \end{array}\right]$$

Для унификации семантики C#-kernel использование стандартных операций приводится к синтаксису вызова метода. Поэтому возникают два терма $CALL$ для отношения `<` и свойства `Length`, принадлежащего любому массиву. Эти термы уточняются легко и фактически они выражают утверждение о том, что значение счетчика `i` строго меньше длины массива, которая хранится во вспомогательной переменной `x0`. При их уточнении наружу просачивается терм INV для метки `L1`. При уточнении его значением становится дизъюнкция двух утверждений. Первое — это предусловие метода, в котором с переменной `i` связано начальное значение 0. Второе — это исходный инвариант цикла, протянутый (как предусловие) через тело цикла. Заметим, что достижение конца тела цикла означает, что при текущем `i` элементы `ia[0]`, ..., `ia[i-1]` неотрицательны, т.е. не было выхода по `break`. Таким образом в посылке утверждается, что либо массив имеет нулевую длину, либо при увеличении счетчика на единицу инвариант сохраняется. Отсюда по индукции следует заключение Inv(`for`).

3.3. Статическая инициализация

Напомним, что статическим полем/методом класса называется поле/метод, который принадлежит всему классу, а не конкретному объекту класса. В частности его можно использовать, даже если ни одного объекта класса не создано. Тем самым инициализация статических членов класса (т.е. инициализация класса) отделяется от инициализации нестатических. В C# (как и в Java) статическая инициализация является «ленивой» и производится не в момент загрузки программы в память, а когда это необходимо. Например, в Java статическая инициализация происходит непосредственно при первом создании объекта класса или при первом обращении к статическому члену/методу (так называемое *активное использование*). Проблемой здесь может быть только то, что если инициализация провоцирует некие нетривиальные процессы в программе, то пользователь должен четко представлять себе, где они произойдут. В частности, если все использования класса в программе являются пассивными, то инициализации вообще не будет.

Классическая логика Хоара задает семантику конструкции вне зависимости от контекста этой конструкции. Рассмотрим тройку Хоара вида

$$\{NewObj(\texttt{C}, c) \Rightarrow Q(x \leftarrow c)\}\,\texttt{C x = new C;}\,\{Q\}\ , \tag{6}$$

где $NewObj(\texttt{C}, c)$ означает, что c — новый объект типа `C`. Очевидно, она соответствует интуитивной семантике операции `new`, но только для случая уже инициализированного класса. Чтобы сделать семантику полной, придется использовать правило вида:

$$\frac{\begin{array}{c} Initialized(\texttt{C}) \wedge NewObj(\texttt{C}, c) \wedge P \Rightarrow Q(x \leftarrow c) \\ \{\neg Initialized(\texttt{C}) \wedge NewObj(\texttt{C}, c) \wedge P\}\,\texttt{Init(C)}\,\{Q(x \leftarrow c)\} \end{array}}{\{P\}\,\texttt{C x = new C;}\,\{Q\}}\ , \tag{7}$$

где `Init` — конструкция, инициализирующая класс. Сразу виден недостаток — число выводимых УК удваивается.

Язык C# усугубляет проблему, так как в нем инициализация класса происходит в момент первого активного использования только при наличии в классе статического конструктора. В остальных случаях инициализация произойдет в некий нефиксированный момент, предшествующий активному использованию [5, §17.4.5.1]. Тем самым возникает неоднозначность в порядке инициализаций классов [11].

Данная проблема активно проявляется в следующей программе:

```
class C {
    static bool result1, result2, result3, result4;

    static void m(){
        result1 = C1.b1;
        result2 = C2.b2;
        result3 = C1.d1;
        result4 = C2.d2;
    }
}

class C1 {
    public static bool b1 = C2.d2;
    public static bool d1 = true;
}

class C2 {
    public static bool d2 = true;
    public static bool b2 = C1.d1;
}
```

В языке Java[11] ситуация вполне однозначна, хотя может стать неприятным сюрпризом. Первое присваивание в теле метода `m()` запустит процесс инициализации класса `C1`, который в свою очередь спровоцирует инициализацию класса `C2`. Поскольку при инициализации класса `C2` класс `C1` остается «недоинициализированным», будет использовано значение по умолчанию. В результате поле `C2.b2` получит значение false, а все остальные поля — значение true. Если же поменять местами первые две строки метода `m()`, то первым начнет инициализироваться класс `C2` и в результате все поля получат значение true.

В языке C# вообще не ясно, какой класс будет проинициализирован первым. Конечно, компиляция в Visual Studio 2005 дает первый вариант, но еще раз подчеркнем, что с точки зрения стандарта это не является гарантией. Поэтому в общем случае верифицировать эту программу невозможно, но можно попробовать воспользоваться упомянутыми выше статическими конструкторами. Добавим объявления

```
static C1() {}
```

[11] В [15] эта программа не была верифицирована, поскольку система ESC/Java не способна моделировать статическую инициализацию.

и

```
static C2() {}
```

в классы `C1` и `C2` соответственно. В результате мы получим поведение, гарантированно соответствующее первому случаю.

Спецификации метода `C.m()` выглядят так:

$\mathrm{Pre}(\texttt{C.m})$: `true`

$\mathrm{Post}(\texttt{C.m})$: $V(L(\texttt{result1})) \wedge \neg V(L(\texttt{result2})) \wedge$
$V(L(\texttt{result3})) \wedge V(L(\texttt{result4}))$

В [15] в качестве предусловия предлагалось утверждение о том, что все три класса проинициализированы, но это верно только для класса `C`. Удобство нашей логики Хоара для C#-kernel в том, что пользователю нет необходимости указывать инициализацию класса, методы которого верифицируются. Это делается автоматически генератором УК.

В результате трансляции в C#-kernel была получена следующая программа:

```
class C {
    public static void SFI(){
        new_instance();
        L := upd(L, field_result1, V0);
        T := upd(T, L(field_result1), Loc(bool));
        L2 := upd(L2, <V(C), result1>, L(field_result1));
        new_instance();
        L := upd(L, field_result2, V0);
        T := upd(T, L(field_result2), Loc(bool));
        L2 := upd(L2, <V(C), result2>, L(field_result2));
        new_instance();
        L := upd(L, field_result3, V0);
        T := upd(T, L(field_result3), Loc(bool));
        L2 := upd(L2, <V(C), result3>, L(field_result3));
        new_instance();
        L := upd(L, field_result4, V0);
        T := upd(T, L(field_result4), Loc(bool));
        L2 := upd(L2, <V(C), result4>, L(field_result4));
        V := upd(V, L2(C, result1), false);
        V := upd(V, L2(C, result2), false);
        V := upd(V, L2(C, result3), false);
        V := upd(V, L2(C, result4), false);
    }
```

```
    static bool result1;
    static bool result2;
    static bool result3;
    static bool result4;

    static void m(){
        Init(C1);
        V := upd(V, L2(C, result1), V2(C1, b1));
        Init(C2);
        V := upd(V, L2(C, result2), V2(C2, b2));
        V := upd(V, L2(C, result3), V2(C1, d1));
        V := upd(V, L2(C, result4), V2(C2, d2));
    }
}

class C1 {
    public static void SFI(){
        new_instance();
        L := upd(L, field_b1, V0);
        T := upd(T, L(field_b1), Loc(bool));
        L2 := upd(L2, <V(C1), b1>, L(field_b1));
        new_instance();
        L := upd(L, field_d1, V0);
        T := upd(T, L(field_d1), Loc(bool));
        L2 := upd(L2, <V(C1), d1>, L(field_d1));
        V := upd(V, L2(C1, b1), false);
        V := upd(V, L2(C1, d1), false);
        Init(C2);
        V := upd(V, L2(C1, b1), V2(C2, d2));
        V := upd(V, L2(C1, d1), true);
    }

    public static bool b1;
    public static bool d1;
}

class C2 {
```

```
    public static void SFI(){
        new_instance();
        L := upd(L, field_d2, V0);
        T := upd(T, L(field_d2), Loc(bool));
        L2 := upd(L2, <V(C1), d2>, L(field_d2));
        new_instance();
        L := upd(L, field_b2, V0);
        T := upd(T, L(field_b2), Loc(bool));
        L2 := upd(L2, <V(C1), b2>, L(field_b2));
        V := upd(V, L2(C2, d2), false);
        V := upd(V, L2(C2, b2), false);
        V := upd(V, L2(C2, d2), true);
        Init(C1);
        V := upd(V, L2(C2, b2), V2(C1, d1));
    }

    public static bool d2;
    public static bool b2;
}
```

В отличие от предыдущих примеров здесь по существу инициализирующие методы `SFI`. Нестатических полей в классах нет, поэтому методы `IFI` можно опустить. При наличии методов `SFI` сами статические поля классов объявляются уже без инициализаторов. Оператор `new` заменяется на метаинструкцию `new_instance`.

Значительное удобство аксиоматики языка C#-kernel связано с прямым просмотром. Благодаря этому, удвоение вывода, как в правиле (7), происходит только при первом активном использовании класса. После этого применяется только правило (6).

Текстуальная подстановка методов `SFI` в метод `m()` приводит к силь-

ному разрастанию получаемых УК. Рассмотрим одно из них:

$$
\begin{array}{l}
Env \vdash \\
\left[\begin{array}{ll}
newp(d_6, L_6, V_5, L2_4) \wedge newp(d_5, L_5, V_5, L2_3) & \wedge \\
newp(d_4, L_4, V_4, L2_3) \wedge newp(e_2, L_4, V_4, L2_3) & \wedge \\
newp(d_3, L_3, V_2, L2_2) \wedge newp(d_2, L_2, V_2, L2_1) & \wedge \\
newp(d_1, L_1, V_1, L2_0) \wedge newp(e_1, L_1, V_1, L2_0) & \wedge \\
L_2 = upd(L_1, \texttt{C1}, d_1) & \wedge \\
V_2 = upd(V_1, d_1, e_1) & \wedge \\
\neg SI(\texttt{C1}, L_1, V_1, T_0, L2_0) \wedge \neg SI(\texttt{C2}, L_4, V_4, T, L2_3) & \wedge \\
L_3 = upd(L_2, \texttt{field\_b1}, d_2) & \wedge \\
L2_2 = upd(L2_1, \langle V_2(\texttt{C1}), \texttt{b1}\rangle, L_3(\texttt{field\_b1})) & \wedge \\
L_4 = upd(L_3, \texttt{field\_d1}, d_3) & \wedge \\
L2_3 = upd(L2_2, \langle V_2(\texttt{C1}), \texttt{d1}\rangle, L_4(\texttt{field\_d1})) & \wedge \\
V_3 = upd(V_2, L2_3(\texttt{C1}, \texttt{b1}), \mathsf{false}) & \wedge \\
V_4 = upd(V_3, L2_3(\texttt{C1}, \texttt{d1}), \mathsf{false}) & \wedge \\
L_5 = upd(L_4, \texttt{C2}, d_4) & \wedge \\
V_5 = upd(V_4, d_4, e_2) & \wedge \\
L_6 = upd(L_5, \texttt{field\_d2}, d_5) & \wedge \\
L2_4 = upd(L2_3, \langle V_5(\texttt{C1}), \texttt{d2}\rangle, L_6(\texttt{field\_d2})) & \wedge \\
L = upd(L_6, \texttt{field\_b2}, d_6) & \wedge \\
L2 = upd(L2_4, \langle V_5(\texttt{C1}), \texttt{b2}\rangle, L(\texttt{field\_b2})) & \wedge \\
V_6 = upd(V_5, L2(\texttt{C2}, \texttt{d2}), \mathsf{false}) & \wedge \\
V_7 = upd(V_6, L2(\texttt{C2}, \texttt{b2}), \mathsf{false}) & \wedge \\
V_8 = upd(V_7, L2(\texttt{C2}, \texttt{d2}), \mathsf{true}) & \wedge \\
V_9 = upd(V_8, L2(\texttt{C2}, \texttt{b2}), V_8(L2(\texttt{C1}, \texttt{d1}))) & \wedge \\
V_{10} = upd(V_9, L2(\texttt{C1}, \texttt{b1}), V_9(L2(\texttt{C2}, \texttt{d2}))) & \wedge \\
V_{11} = upd(V_{10}, L2(\texttt{C1}, \texttt{d1}), \mathsf{true}) & \wedge \\
V_{12} = upd(V_{11}, L2(\texttt{C}, \texttt{result1}), V_{11}(L2(\texttt{C1}, \texttt{b1}))) & \wedge \\
V_{13} = upd(V_{12}, L2(\texttt{C}, \texttt{result2}), V_{12}(L2(\texttt{C2}, \texttt{b2}))) & \wedge \\
V_{14} = upd(V_{13}, L2(\texttt{C}, \texttt{result3}), V_{13}(L2(\texttt{C1}, \texttt{d1}))) & \wedge \\
V = upd(V_{14}, L2(\texttt{C}, \texttt{result4}), V_{14}(L2(\texttt{C2}, \texttt{d2}))) &
\end{array}\right] \Rightarrow \mathrm{Post}(\texttt{C.m})
\end{array}
$$

Здесь $Env \equiv (\texttt{m}, \mathsf{false}, \{\texttt{C}, \texttt{C1}, \texttt{C2}\})$. Чтобы доказать это УК, достаточно заметить, что последовательность обновлений метапеременных $\texttt{V}_i$ в точности соответствует последовательности инициализаций полей для первого сценария поведения.

Замечание. Необходимо отметить, что концепция статической инициализации как таковая не является корнем проблем верификации данно-

го примера. Проблема только в том, что два объекта взаимно ссылаются друг на друга при конструировании. Не случайно в [15] этот пример был назван «нездоровым» (sick). Также проблема зависимости от порядка вычислений не является присущей только ОО языкам. Например, если `x` и `y` — две переменные, равные нулю, то в языке C выражение

```
(x = y + 1) < (y = x + 1)
```

может быть как истинным, так и ложным, в зависимости от реализации и настроек конкретного компилятора.

3.4. Замещение и динамические типы

Как отмечалось ранее, феномен *позднего связывания* означает, что при замещении виртуального метода только исходя из динамического типа объекта можно определить, какая реализация будет вызвана. Рассмотрим следующую программу:

```
class C {
    virtual void m() { m(); }
}

class D : C {
    override void m() { throw new System.Exception(); }

    void test() { base.m(); }
}
```

На первый взгляд метод `test()` будет исполняться бесконечно. В реальности же исполнение конечно: метод `test()` вызывает метод `m()` из класса `C`, который, в свою очередь, вызовет метод `m()` из класса `D`, поскольку динамический тип указателя `this` есть `D`. Стоит отметить, что в языке Java поведение программы может быть более наглядным, поскольку в синтаксисе Java есть средство указания исключений, выбрасываемых из функций. В частности, в [15] все три метода сопровождались записью `throws Exception`.

В [15] программа была верифицирована, но при верификации метода `test()` пришлось явно использовать реализации метода `m()`, поскольку семантика наследования в языке JML не позволяет записать пригодные для доказательства спецификации обоих методов `m()`. Наш метод позволяет задать спецификации, хотя и довольно сложные. Вообще, этот пример заслуживает более подробного изложения.

Вначале введем некоторые обозначения. Для кортежа x терм $x[i]$ будет обозначать i-й элемент x, терм $subtuple(x, i, j)$ будет обозначать отрезок $x[i..j]$ и функция $len(x)$ возвращает длину x.

Пусть a обозначает кортеж параметров спецификации для `C.m()`. Мы предполагаем, что элемент $a[1]$ (обозначаемый как as $TOT(a)$) хранит тип переменной `this`, а кортеж $subtuple(a, 2, 7)$ (обозначаемый как $mvs_0(a)$) хранит начальные значения метапеременных из mvs. Остающиеся параметры (т.е. $subtuple(a, 8, len(a))$) обозначаются как $rest(a)$. Для краткости в спецификациях будем использовать имя SE вместо `System.Exception`.

Теперь можно описать спецификации методов программы:

$$\begin{array}{lll}
\mathrm{Pre}(\texttt{D.m}) & : & \mathsf{true} \\
\mathrm{Post}(\texttt{D.m}) & : & T(E) = SE
\end{array}$$

$$\begin{array}{lll}
\mathrm{Pre}(\texttt{D.test}) & : & T(V(\texttt{this})) = D \\
\mathrm{Post}(\texttt{D.test}) & : & T(E) = SE
\end{array}$$

$$\begin{array}{lll}
\mathrm{Pre}(\texttt{C.m})(mvs, a) & : & prec(C.m, mvs) \wedge \\
& & T(V(\texttt{this})) = TOT(a) \wedge mvs = mvs_0(a) \wedge \\
& & (TOT(a) \neq \texttt{C} \Rightarrow \\
& & \quad \forall f \; \forall mvs' \; ((invoker(f, \texttt{m}, \texttt{this}, [], mvs) \wedge \\
& & \qquad\qquad subst(mvs', mvs, f, \texttt{this}, [\,]) \Rightarrow \\
& & \qquad\quad pre(f)(mvs', rest(a)))) \wedge \\
& & (TOT(a) = \texttt{C} \Rightarrow \mathsf{true})
\end{array}$$

$$\begin{array}{lll}
\mathrm{Post}(\texttt{C.m})(mvs, a) & : & (TOT(a) \neq \texttt{C} \Rightarrow \\
& & \quad \forall f \; (invoker(f, \texttt{m}, \texttt{this}, [], mvs_0(a)) \Rightarrow \\
& & \qquad post(f)(mvs, rest(a)))) \wedge \\
& & (TOT(a) = \texttt{C} \Rightarrow \mathsf{false}) \; .
\end{array}$$

Постусловие метода `D.Test()` говорит о том, что в метапеременной E содержится непойманное исключение типа SE. Спецификация виртуального метода `C.m()` описывает разбор случаев. Если тип переменной `this` есть `C`, то вызов `m()` есть на самом деле рекурсивный вызов самого `C.m()`, что ведет к бесконечному циклу ($TOT(a) = \texttt{C} \Rightarrow \mathsf{false}$). В противном случае вызов `m()` — это вызов некоторой реализации `C.m()` в некотором классе-наследнике. Совокупность всех этих реализаций квантифицирована переменной f, которая удовлетворяет предикату $invoker$. Этот предикат является логическим представлением алгоритма позднего связывания. Результат вызова удовлетворяет постусловию факти-

ческого f.

Оптимизированный результат трансляции в C#-kernel имеет вид:

```
class C {
    virtual void m() { this.m(); }
}

class D : C {
    public void IFI_D() {
        Init(C);
        this.IFI_C();
    }

    override void m() {
        Init(System_Exception);
        new_instance();
        L := upd(L, x, V0);
        T := upd(T, L(x), Loc(System_Exception));
        new_instance();
        V := upd(V, L(x), V0);
        T := upd(T, V(x), System_Exception);
        x.IFI_System_Exception();
        x.System_Exception();
        E := V(x);
    }

    void test() { base.m(); }
}
```

Поскольку в данном примере есть наследование, то в классе `D` нельзя отбросить инициализирующий метод `IFI`. Также отметим, что при переводе в C#-kernel исчезают пространства имен, поэтому полное имя `System.Exception` заменяется на имя `System_Exception` глобального уровня.

Рассмотрим, например, верификацию метода `D.test()` Для него порождаются два УК, соответствующие выполнению вызова `base.m()` с исключением и без исключения. Условие без исключения оказывается тавтологией, так как имеет вид $\mathsf{false} \Rightarrow \Phi$. Рассмотрим УК с исключением:

$$Env \vdash CALL(\mathtt{m}, \mathsf{base}, [\,], mvs, \lambda(mvs, P)) \wedge E \neq \omega \Rightarrow T(E) = SE \ , \quad (8)$$

где $Env = (\texttt{D.test}, \texttt{true}, \{\texttt{D}\})$ и

$$P \equiv prec(\texttt{D.test}, mvs) \wedge T(V(\texttt{this})) = \texttt{D} \ .$$

Формула $prec()$ автоматически добавляется генератором УК для любого метода и налагает некоторые стандартные условия на класс метода и указатель `this`. На истинность (8) она не влияет, поэтому детально не рассмотрена.

Согласно алгоритма уточнения [21, 2] терм CALL в (8) превращается в

$$\begin{array}{lll} \exists\, mvs_1\ \exists\, f\ \exists\, mvs_2\ \exists\, a & (& T_1(V_1(\texttt{this})) = \texttt{D} \wedge \\ & & prec(\texttt{D.test}, mvs_1) \wedge \\ & & E_1 = \omega \wedge \\ & & SI(D, L_1, V_1, T_1, L2_1) \wedge \\ & & invoker(f, \texttt{m}, \texttt{base}, [\,], mvs_1) \wedge \\ & & subst(mvs_2, mvs_1, f, \texttt{base}, [\,]) \wedge \\ & & pre(f)(mvs_2, a) \wedge post(f)(mvs, a) \\ &) \ . & \end{array} \tag{9}$$

Упомянутый выше предикат $invoker(f, \ldots)$ определяется на основе операционной семантики. Окружение неявно хранит всю программу, поэтому по определению $invoker$ имеем $f = \texttt{C.m}$ в Env. Тогда параметрическое предусловие $pre(f)(mvs_2, a)$ превратится в

$$\begin{array}{l} T_2(V_2(\texttt{this})) = TOT(a) \wedge \\ mvs_2 = mvs_0(a) \wedge \\ (TOT(a) \neq \texttt{C} \Rightarrow \\ \quad \forall\, g\ \forall\, mvs'\ ((invoker(g,\ \texttt{m},\ \texttt{this},\ [],\ mvs_2) \wedge \\ \qquad\qquad subst(mvs', mvs_2, g, \texttt{this}, [])) \Rightarrow \\ \qquad\qquad\quad pre(g)(mvs',\ rest(a))) \wedge \\ (TOT(a) = \texttt{C} \Rightarrow true)\ . \end{array} \tag{10}$$

Предикат $subst(w, \ldots)$ проверяет, является ли кортеж w результатом подстановки аргументов в функциональный член. По определению формулу $subst(mvs_2, mvs_1, f, \texttt{base}, [])$ можно переписать как

$$\begin{array}{lll} \exists\, d\ (& newp(d, L_1, V_1, L2_1) \wedge \\ & L_2 = upd(L_1, \texttt{this}, d) \wedge \\ & V_2 = upd(V_1, d, V_1(\texttt{this})) \wedge \\ & T_2 = upd(upd(T_1, \texttt{this}, \texttt{C}), d, Loc(\texttt{C}))\ . \end{array}$$

Условие $T_1(V_1(\texttt{this})) = \texttt{D}$ влечет $T_2(V_2(\texttt{this})) \neq \texttt{C}$. Тогда (10) превратится в

$$\begin{array}{l} TOT(a) = \texttt{D} \wedge mvs_2 = mvs_0(a) \wedge \\ \forall\, g\ \forall\, mvs'\ ((invoker(g,\ \texttt{m},\ \texttt{this},\ [\,],\ mvs_2) \wedge \\ \qquad subst(mvs', mvs_2, g, \texttt{this}, [\,])) \Rightarrow \\ \qquad\quad pre(g)(mvs',\ rest(a)))\ . \end{array} \tag{11}$$

По определению *invoker* имеем $g = \texttt{D.m}$ в Env. Тогда (11) превращается в

$$\begin{array}{l} TOT(a) = D \wedge mvs_2 = mvs_0(a) \wedge \\ \forall\, mvs'(subst(mvs', mvs_2, \texttt{D.m}, \texttt{this}, [\,]) \Rightarrow prec(\texttt{D.m}, mvs'))\ . \end{array}$$

Аналогично постусловие $post(f)(mvs, a)$ переписывается в

$$\forall\, h\ (invoker(h, \texttt{m},\ \texttt{this}, [], mvs_2) \Rightarrow post(h)(mvs, []))\ . \tag{12}$$

По определению *invoker* имеем $h = \texttt{D.m}$ в Env. Следовательно постусловие (12) превратится в $T(E) = SE$. Тогда (9) обратится в

$$\begin{array}{ll} \exists\, mvs_1\ \exists\, mvs_2\ \exists\, a & (T_1(V_1(\texttt{this})) = \texttt{D} \wedge prec(\texttt{D.test}, mvs_1) \wedge \\ & \quad prec(\texttt{C.m}, mvs_2) \wedge TOT(a) = \texttt{D} \wedge \\ & \quad \forall\, mvs'(subst(mvs', mvs_2, \texttt{D.m}, \texttt{this}, [\,]) \Rightarrow \\ & \qquad prec(\texttt{D.m}, mvs')) \wedge \\ & \quad mvs_2 = mvs_0(a) \wedge T(E) = SE)\ . \end{array}$$

Это влечет $T(E) = SE$ и, следовательно, УК (8) истинно.

Замечание. Сложность верификации тривиального метода `D.test()`, как легко заметить, была обусловлена использованием параметрических метаспецификаций для виртуального метода `C.m()`. Но любая попытка использовать для виртуального метода не параметрические, а конкретные спецификации столкнется с другими проблемами. С одной стороны, если виртуальный метод задает интерфейс ко всем методам-реализациям, то можно задать только спецификации базового виртуального метода, и считать, что все методы-реализации автоматически удовлетворяют этим спецификациям. Тут возможны две стратегии.

1. Пользователь может вначале задать базовый виртуальный метод и его спецификации, а исходя из них создавать классы-потомки и их методы. Но тем самым, разнообразие методов-реализаций будет жестко

ограничено базовыми спецификациями, и если новый метод не вкладывается в них, то возможно придется расширять сами базовые спецификации, а значит заново проверять их для остальных методов. В противном случае базовые спецификации могут носить настолько общий характер, что из них невозможно будет вывести свойства вызовов конкретных методов.

2. Пользователь может задать базовые спецификации, исходя из свойств уже существующего множества реализаций, просто как дизъюнкцию этих свойств. Тогда при добавлении нового метода-реализации его свойства просто добавятся как новый дизъюнкт. Однако эта дизъюнкция фактически будет работать как квантор высшего порядка по предикатам, что усложнит верификацию.

3.5. Делегаты

Делегат в языке C# — это фактически указатель на функцию, реализованный в виде класса. Тем самым, этот механизм предполагает большую безопасность по сравнению с языком C. Также в языке C# делегат может указывать не на одну функцию, а на целый список функций, и при вызове делегата будут вызваны все функции из списка. Хотя, как правило, список из более чем одной функции используют редко.

Рассмотрим следующую программу:

```
class Minimum {
    public delegate bool Order(int e1, int e2);

    static public int Find(int[] arr, Order ord) {
        int min = arr[0];
        int xx = 0;

        while(xx < arr.Length){
            if(ord(arr[xx], min)) min = arr[xx];
            xx++;
        }

        return min;
    }
}
```

```
class C {
    static bool LessThan(int e1, int e2) {
        return e1 < e2;
    }

    static void Main(string[] args) {
        Minimum.Order order = new Minimum.Order(LessThan);

        int[] arr = new int[] {3, 5, 1, 7, 4};

        Minimum.Find(arr, order);

    }
}
```

В классе `Mininum` объявлен метод `Find()`, который должен выдать минимальный элемент в целочисленном массиве, передаваемом в качестве параметра. Минимальность означает использование некоторого отношения порядка. Если бы мы сразу взяли в качестве отношения стандартное "$<$", то для любого другого отношения пришлось бы реализовать свою версию `Find()`. Механизм делегатов позволяет задать этот метод как некий шаблон. Поэтому вторым параметром является делегат `ord` типа `Order`, через который может передаваться произвольная функция от двух целочисленных аргументов, возвращающая булевское значение. В этом есть определенный риск, поскольку нет никакого контроля того, что передаются только отношения порядка. Далее в классе `C` в качестве порядка взято «хорошее» отношение "$<$". В результате будет выдана единица.

Спецификации всех методов программы и инварианта цикла в методе `Minimum.Find()` имеют следующий вид:

$$\begin{aligned}
\text{Pre}(\texttt{Find}) \quad : \quad & 0 \le V(L2(V(L(\texttt{arr})), \texttt{Length})) \wedge \\
& \exists (somef : \texttt{Order}).\ \lambda e_1 e_2.\ V(L(\texttt{ord})) = \lambda e_1 e_2 . somef
\end{aligned}$$

$$\begin{aligned}
\text{Post}(\texttt{Find}) \quad : \quad & \exists j\ (\ 0 \le j \le V(L2(V(L(\texttt{arr})), \texttt{Length})) \wedge \\
& \quad V0 = V(L2(V(L(\texttt{arr})), j)) \wedge \\
& \quad \forall i\ (i \ne j \Rightarrow somef(V0, V(L2(V(L(\texttt{arr})), i)))))
\end{aligned}$$

$$\begin{aligned}
\text{Inv}(\texttt{while}) \quad : \quad & V(L(\texttt{xx})) \le V(L2(V(L(\texttt{arr})), \texttt{Length})) \wedge \\
& \forall k \; (\; k \le V(L(\texttt{xx})) \Rightarrow \\
& \qquad (somef(V(L(\texttt{min})), V(L2(V(L(\texttt{arr})), k))) \vee \\
& \qquad \; V(L(\texttt{min})) = V(L2(V(L(\texttt{arr})), k))))
\end{aligned}$$

$$\begin{aligned}
&\text{Pre}(\texttt{LessThan}) \; : \; \mathsf{true} \\
&\text{Post}(\texttt{LessThan}) : \; V(L(e_1)) < V(L(e_2)) \Longrightarrow V0 = \mathsf{true}
\end{aligned}$$

$$\begin{aligned}
&\text{Pre}(\texttt{Main}) \quad : \; \mathsf{true} \\
&\text{Post}(\texttt{Main}) \quad : \; V0 = 1
\end{aligned}$$

При задании спецификаций метода `Minimum.Find()` возникает та же проблема, что и для виртуальных методов. Указать, что с помощью делегата `ord` передается только отношение частичного порядка, значит ограничить применимость метода `Minimum.Find()` только данным конкретным примером. Если мы хотим специфицировать класс `Minimum` как библиотечный, то можно задать только спецификации очень общего вида. В частности, указано, что передается некая логическая функция $somef$ без всяких предположений о характере этой функции.

В нашем подходе при наличии метода `Main()` формируется специальная тройка Хоара, соответствующая вызову этого метода. При этом необходимо задать пред- и постусловие для всей программы. В данном примере в качестве этих спецификаций можно взять спецификации самого `Main()`.

При переводе получаем следующую программу:

```
class Minimum {

    public delegate bool Order(int e1, int e2);

    static public int Find(int[] arr, Order ord) {
        new_instance();
        L := upd(L, min, V0);
        T := upd(T, L(min), Loc(int));
        V := upd(V, L(min), V2(arr, 0));
        new_instance();
        L := upd(L, xx, V0);
        T := upd(T, L(xx), Loc(int));
        V := upd(V, L(xx), 0);
        L:;
```

```
        new_instance();
        L := upd(L, b0, V0);
        T := upd(T, L(b0), Loc(bool));
        new_instance();
        L := upd(L, get_Length, V0);
        T := upd(T, L(get_Length), Loc(int));
        arr.get_Length();
        V := upd(V, L(arr_Length), V0);
        int.<(xx, arr_Length);
        V := upd(V, L(b0), V0);
        if (b0) {
            new_instance();
            L := upd(L, b, V0);
            T := upd(T, L(b), Loc(bool));
            new_instance();
            L := upd(L, x0, V0);
            T := upd(T, L(x0), Loc(int));
            V := upd(V, L(x0), V2(arr, xx));
            ord(x0, min);
            V := upd(V, L(b), V0);
            if (b) V := upd(V, L(min), V2(arr, xx));
            int.++(xx);
            V := upd(V, L(xx), V0));
            goto L;
        }
        V0 := V(min);
    }
}

class C {
    static bool LessThan(int e1, int e2) { int.<(e1, e2); }

    static void Main(string[] args) {
        Init(Minimum.Order);
        new_instance();
        L := upd(L, order, V0);
        T := upd(T, L(order), Loc(Minimum.Order))
        V := upd(V, L(order), add(V(order), <V(C), LessThan>));
```

```
V := upd(V, L(order), V(x1));
Init(int[5]);
new_instance();
L := upd(L, arr, V0);
T := upd(T, L(arr), Loc(int[5]))
new_instance(x);
L := upd(L, x2, V0);
T := upd(T, L(x2), Loc(int[5]));
new_instance();
L := upd(L, elem0, V0);
T := upd(T, L(elem0), Loc(int));
L2 := upd(L2, <V(x2), 0>, L(elem0));
V := upd(V, L2(x2, 0), 0);
new_instance();
L := upd(L, elem1, V0);
T := upd(T, L(elem1), Loc(int));
L2 := upd(L2, <V(x2), 1>, L(elem1));
V := upd(V, L2(x2, 1), 0);
new_instance();
L := upd(L, elem2, V0);
T := upd(T, L(elem2), Loc(int));
L2 := upd(L2, <V(x2), 2>, L(elem2));
V := upd(V, L2(x2, 2), 0);
new_instance();
L := upd(L, elem3, V0);
T := upd(T, L(elem3), Loc(int));
L2 := upd(L2, <V(x2), 3>, L(elem3));
V := upd(V, L2(x2, 3), 0);
new_instance();
L := upd(L, elem4, V0);
T := upd(T, L(elem4), Loc(int));
L2 := upd(L2, <V(x2), 4>, L(elem4));
V := upd(V, L2(x2, 4), 0);
V := upd(V, L2(x2, 0), 3);
V := upd(V, L2(x2, 1), 5);
V := upd(V, L2(x2, 2), 1);
V := upd(V, L2(x2, 3), 7);
V := upd(V, L2(x2, 4), 4);
```

```
        V := upd(V, L(arr), V(x2));
        Minimum.Find(arr, order);
    }
}
```

Для данной программы генератор порождает более сотни УК. Рассмотрим одно условие, в котором есть вызов делегата `ord`. Линейный участок, которому соответствует это УК, есть тело цикла в методе `Minimum.Find()`. После применения стратегии упрощения 1 оно имеет вид

$$Env \vdash \left[\begin{array}{l} CALL(\texttt{++}, \texttt{int}, [\texttt{xx}], [L, V_4, T, L2, V0, E], \lambda(mvs, \Phi)) \wedge \\ V = upd(V_4, L(\texttt{xx}), V0) \end{array}\right] \Rightarrow Inv(\texttt{L}) \quad (13)$$

где $Env \equiv (\texttt{Find}, \mathsf{false}, \{\texttt{Minimum}, \texttt{int}, \texttt{bool}\})$ и

$$\Phi \equiv \left[\begin{array}{ll} DELCALL(V_2(L(\texttt{ord})), [\texttt{x0}, \texttt{min}], [L, V_2, T, L2, V0, E], & \\ \qquad \lambda(mvs, \Phi_2)) & \wedge \\ V_3 = upd(V_2, L(\texttt{b}), V0) & \wedge \\ V_3(L(\texttt{b})) = \mathsf{true} & \wedge \\ V_4 = upd(V_3, L(\texttt{min}), V2(\texttt{arr}, \texttt{xx})) & \end{array}\right]$$

$$\Phi_2 \equiv \left[\begin{array}{ll} newp(d_2, L_2, V_1, L2) & \wedge \\ newp(d_1, L_1, V_1, L2) & \wedge \\ V_1(L_1(\texttt{xx})) \leq V_1(L2(V_1(L_1(\texttt{arr})), \texttt{Length})) & \wedge \\ \forall k.(k \leq V_1(L_1(\texttt{xx})) \Rightarrow & \\ \qquad somef(V_1(L_1(\texttt{min})), V_1(L2(V_1(L_1(\texttt{arr})), k))) \vee & \\ \qquad V_1(L_1(\texttt{min})) = V_1(L2(V_1(L_1(\texttt{arr})), k)) & \\ ) & \wedge \\ L_2 = upd(L_1, \texttt{b}, d_1) & \wedge \\ L = upd(L_2, \texttt{x0}, d_2) & \wedge \\ V_2 = upd(V_1, L(\texttt{x0}), V2(\texttt{arr}, \texttt{xx})) & \end{array}\right]$$

В нем присутствуют все виды «ленивых» термов: *CALL*, *DELCALL* и *INV*. Детальное уточнение и доказательство являются достаточно громоздкими, поэтому поясним неформально. Объемлющий терм *CALL* связан с инкрементом `xx++` в исходной программе. Для инкремента в качестве постусловия достаточно взять равенство $V0 = V(L(arg)) + 1$. Заметим, что сам побочный эффект увеличения `xx` на единицу записан

во втором конъюнкте в (13). При уточнении терма $CALL$ имя V заменяется на V_4 и Φ «просачивается» наружу. В примере делегат указывает на одну функцию, поэтому уточнение $DELCALL$ аналогично уточнению для $CALL$ с той лишь разницей, что нет объекта. Следовательно к формуле Φ_2 будут добавлены спецификации делегата с некоторой адаптацией метапеременных. Наконец можно выяснить, что $INV(\texttt{L}) = \mathrm{Inv}(\texttt{while})(V0 \leftarrow V(L(\texttt{min})))$. В итоге утверждение становится эквивалентным утверждению о том, что при увеличении счетчика `xx` на единицу инвариант цикла не изменится. Это можно доказать по индукции.

ЗАКЛЮЧЕНИЕ

В работе был рассмотрен трехуровневый подход к верификации программ на языке C#-light и его применение для ряда программ, справедливо считающихся «вызовами» для верификации в ОО языках. Этот трехуровневый подход является объектным расширением нашего двухуровневого подхода к верификации программ на языке C [20, 4]. Достоинства языка C#-light и трехуровневого подхода заключаются в следующем.

- Язык C#-light поддерживает большинство последовательных конструкций языка C#.
- Для верификации используется простая логика Хоаровского типа. Простота является следствием трансляции сложных конструкций языка C#-light в промежуточный язык C#-kernel и откладыванием обработки ряда динамических аспектов поведения до этапа уточнения.
- Использование правил прямого просмотра в аксиоматической семантике языка C#-kernel позволяет проводить детерминированный вывод «ленивых» условий корректности и избежать увеличения числа этих условий, характерного для других подходов.

Вследствие детализированной модели памяти мы столкнулись с проблемой усложнения условий корректности, что становится препятствием при их доказательстве. Поэтому был предложен ряд стратегий упрощения как промежуточных программ на языке C#-kernel, так и условий корректности.

В настоящее время разрабатывается экспериментальный прототип системы верификации программ на языке C#-light. Он будет включать транслятор из языка C#-light в язык C#-kernel, генератор «ленивых»

УК и модуль уточнения «ленивых» термов. Предполагается также реализовать статический анализатор, который позволит упростить проверку применимости тех или иных стратегий упрощения.

Тем не менее, верификация больших программ на языке C#-light попрежнему является вызовом. Комбинация нашего подхода с модульным подходом [19] и/или методом расширенного статического анализа [17] может стать важным шагом на пути решения этой проблемы.

СПИСОК ЛИТЕРАТУРЫ

1. **Дубрановский И.В.** Верификация C# программ: трансляция из C#-light в C#-kernel. — Новосибирск, 2006. — 56 с. — (Препр. / РАН. Сиб. Отд-ние. ИСИ; N 140).
2. **Непомнящий В.А., Ануреев И.С., Дубрановский И.В., Промский А.В.** На пути к верификации C#-программ: трехуровневый подход // Программирование. — 2006. — N 4. — С. 4–20.
3. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации C программ. Язык C-light и его формальная семантика // Программирование. — 2002. — N 6. — С. 1–13.
4. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации C программ. Аксиоматическая семантика языка C-kernel // Программирование. — 2003. — N 6. — С. 1–16.
5. C# Language Specification. Standard ECMA-334. (2001) `http://www.ecma-international.org/`.
6. **Apt K.R.** Ten years of Hoare's logic: A survey — Part I // ACM Trans. Progr. Lang. and Systems. — 1981. — Vol. 3, N 4. — P. 431–483.
7. **Apt K.R., Olderog E.R.** Verification of sequential and concurrent programs. — Berlin etc.: Springer, 1991. — 450 p.
8. **Barnett M., Leino K.R.M., Schulte W.** The Spec# programming system: An overview // Proc. CASSIS 2004. — Lect. Notes Comput. Sci. — 2004. — Vol. 3362. — P. 49–69.
9. **Börger E., Fruja N.G., Gervasi V., Stärk R.** A High-Level Modular Definition of Semantics of C# // Theoretical Computer Sci. — 2004. — N 336(2/3).
10. **Flanagan C., Saxe J.B.** Avoiding Exponential Explosion: Generating Compact Verification Conditions // POPL 2001: Proc. / The 28th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages. — ACM Press, January 2001. — P. 193–205.
11. **Fruja N.G.** Specification and Implementation Problems for C# // Proc. ASM 2004. — Lect. Notes Comput. Sci. — 2004. — Vol. 3052. — P. 127–143.
12. **Gribomont P.E.** Simplification of Boolean verification conditions // Theoretical Computer Sci. — 2000. — Vol. 239, N 1. — P. 165–185.
13. **Hoare C.A.R.** An axiomatic basis for computer programming // Commun ACM. — 1969. — Vol. 12, N 1. — P. 576–580.
14. **Huisman M., Jacobs B.** Java Program Verification via a Hoare Logic with Abrupt Termination // Proc. FASE 2000. — Lect. Notes Comput. Sci. — 2000. — Vol. 1783. — P. 284–303.

15. **Jacobs B., Kiniry J.L., Warnier M.** Java Program Verification Challenges // Proc. FMCO 2002. — Lect. Notes Comput. Sci. — 2003. — Vol. 2852. — P. 202–219.
16. **Jacobs B., Poll E.** Java Program Verification at Nijmegen: Development and Perspective // Lect. Notes Comput. Sci. — 2004. — Vol. 3233. — P. 134–153.
17. **Leino K.R.M.** Extended Static Checking: a Ten-Year Perspective // Lect. Notes Comput. Sci. — 2001. — Vol. 2000. — P. 157–175.
18. **Luckham D.C., Suzuki N.** Verification of array, record and pointer operations in Pascal // ACM Trans. Progr. Lang., and Systems. — 1979. — Vol. 1, N 2. — P. 226–244.
19. **Müller P.** Modular Specification and Verification of Object-Oriented Programs // Lect. Notes Comput. Sci. — 2002. — Vol. 2262.
20. **Nepomniaschy V.A., Anureev I.S., Promsky A.V.** Verification-Oriented Language C-light and its Structural Operational Semantics // Proc. PSI 2003. — Lect. Notes Comput. Sci. — 2003. — Vol. 2890. — P. 103–111.
21. **Nepomniaschy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V.** Towards C# program verification: A three-level approach. — Novosibirsk, 2005. — (Prepr. / IIS SB RAS; N 128).
22. **Nepomniaschy V.A., Anureev I.S., Promsky A.V.** Towards C# program verification: C#-kernel and its axiomatic semantics // Proc. Workshop on Concurrency, Specification and Programming (CS&P'2006). — Humboldt University, Berlin, 2006. — Vol. 2: Specification. — P. 195-206.
23. **Norrish M.** C formalised in HOL: Thes. doct. phylosophy (computer sci.). — Cambridge, 1998. — 150 p.
24. **Oheimb D.v.** Hoare Logic for Java in Isabelle/HOL // Concurrecy and Computation. — 2001. — Vol. 13.
25. **Oheimb D.v., Nipkow T.** Hoare Logic for NanoJava: Auxiliary Variables, Side Effects, and Virtual Methods Revisited // Proc. FME 2002. — Lect. Notes Comput. Sci. — 2002. — Vol. 2391. — P. 89–105.
26. **Pierik C., de Boer F.S.** A Syntax-Directed Hoare Logic for Object-Oriented Programming Concepts // Lect. Notes Comput. Sci. — 2003. — Vol. 2884. — P. 64–78.
27. **Poetzsch-Heffter A., Muller P.** A Programming Logic for Sequential Java // Proc. ESOP'99. — Lect. Notes Comput. Sci. — 1999. — Vol. 1576. — P. 162–176.
28. **Reus B., Wirsing M., Hennicker R.** A Hoare Calculus for Verifying Java Realizations of OCL-constrained Design Models // Proc. FASE 2001. — Lect. Notes Comput. Sci. — 2001. — Vol. 2029. — P. 300–317.
29. **Wiener R.** Delegates and Events in C# // J. of Object Technology. — 2004. — Vol. 3, N 5. — P. 78–85.
`http://www.jot.fm/issues/issue_2004_05/column8`

А.В. Промский

ПРИМЕНЕНИЕ ТРЕХУРОВНЕВОГО ПОДХОДА К ВЕРИФИКАЦИИ ПРОГРАММ НА ЯЗЫКЕ C#-LIGHT

Препринт
139

Рукопись поступила в редакцию 11.12.2006
Рецензент И.С. Ануреев
Редактор З. В. Скок

Подписано в печать 28.12.2006
Формат бумаги 60×84 1/16 Объем 3,1 уч.-изд.л., 3,4 п.л.
Тираж 60 экз.

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