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

И.С. Ануреев

КОНЦЕПТУАЛЬНЫЙ БАЗИС ТРЕХУРОВНЕВОГО МЕТОДА ВЕРИФИКАЦИИ C# ПРОГРАММ

Препринт
170

Новосибирск 2013

В работе описан концептуальный базис улучшенного трехуровневого метода верификации C# программ. Язык C#-light расширен на небезопасный код. Определена формальная операционная семантика языков C#-light и C#-kernel на основе предметно-ориентированных систем переходов. Предложен подход к генерации условий корректности, базирующийся на логике безопасности языка C#-kernel. Новая концепция языка C#-kernel рассматривает его как объединение языка C#-light с выделенными элементами языка Atoment. Трансляция C#-light программы в C#-kernel сведена к преобразованию состояния программы на основе метода атрибутных аннотаций и предвычислению константных выражений языка C#.

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

I.S. Anureev

CONCEPTUAL BASIS OF THREE-LEVEL METHOD OF C# PROGRAM VERIFICATION

Preprint
170

Novosibirsk 2013

In this paper a conceptual basis of the improved three-level method of C# program verification is described. The C#-light language is expanded upon unsafe code. Formal operational semantics of the languages C#-light and C#-kernel, based on domain-specific transition systems, is defined. The approach to verification conditions generation, based on safety logic of C#-kernel, is suggested. A new conception of the C#-kernel language considers it as a union of the C#-light language with the selected elements of the Atoment language. Translation of a C#-light program into C#-kernel is reduced to transformation of the program state based on the attributive annotation method and precomputation of C# constant expressions.

1. ВВЕДЕНИЕ

Целью работы является уточнение и формализация трехуровневого метода верификации C# программ [5, 8] на базе предметно-ориентированных систем переходов (ПОСП) [1] и языка описания таких систем Atoment. Для краткости будем называть в этой работе существующий вариант метода 3-методом, а его уточненный вариант — 3M-методом.

3-метод применяется к языку C#-light (подмножеству языка C# [7]) и состоит из трех этапов. На первом этапе C#-light программа транслируется в программу на языке C#-kernel. На втором этапе для C#-kernel программы генерируются ленивые условия корректности. Генерация условий корректности выполняется на базе аксиоматической семантики (логики Хоара) языка C#-kernel. Условия называются ленивыми, т.к. включают интерпретированные функции, интерпретация которых задается в терминах операционной семантики языка C#-light, и символические (ленивые) инварианты меток помеченных операторов. Эти инварианты приписываются новым меткам, которые появляются в программе на языке C#-kernel на этапе трансляции. На третьем этапе уточняются ленивые условия корректности. Разрешаются ленивые инварианты меток. Интерпретированные функции рассматриваются как неинтерпретированные функциональные символы, и для них задаются аксиомы, описывающие в декларативном виде соответствующие алгоритмы операционной семантики. Как правило, такая аксиоматизация является неполной из-за сложности алгоритмов операционной семантики.

В отличие от двухуровневого метода верификации C программ [3, 4], где также выделяются языки C-light и C-kernel, язык C#-kernel не является подмножеством языка C#-light. Он включает дополнительные конструкции, называемые метаинструкциями. Они позволяют напрямую обращаться к абстрактной машине языка C#-light, описывающей операционную семантику этого языка.

3-метод предлагает общую схему верификации C# программ. Реализация его на практике требует дальнейшего уточнения этой схемы. В этой статье решаются две задачи. Во-первых, предлагаются средства уточнения схемы. Во-вторых, предлагается существенная ревизия самого метода.

Опишем изменения, которые внесены в 3-метод.

Во-первых, 3M-метод оперирует моделями C#-light и C#-kernel программ, представленными на языке Atoment, а не самими программами.

Во-вторых, для языка C#-light определена формальная операционная семантика, базирующаяся на ПОСП. Тем самым, мы получаем точное математическое определение языка C#-light. Кроме того, операционная семантика языка C#-light в комбинации с операционной семантикой метаинструкций языка C#-kernel может быть использована для доказательства корректности аксиоматической семантики языка C#-kernel.

В-третьих, аксиоматическая семантика языка C#-kernel заменена логикой безопасности [6] (расширением логики Хоара для проверки свойств безопасности). Эта логика описывается с помощью дедуктивных ПОСП [1].

В-четвертых, изменяется промежуточный язык C#-kernel. Поскольку логика безопасности позволяет работать с побочными эффектами и конструкциями с переменным числом параметров напрямую, то не требуется предварительных преобразований C#-light конструкций, необходимых для логики Хоара. Поэтому новая версия языка C#-kernel содержит все C#-light конструкции (некоторые из них в нормализованном виде, например, константное выражение `A` в операторе `goto case A` должно быть вычислено) и дополнительно элементы языка Atoment (в частности, элементы, моделирующие метаинструкции).

В-пятых, использование логики безопасности позволило снять ряд ограничений на язык C#-light, тем самым расширив этот язык. Так, `3M`-метод может применяться к программам с небезопасным кодом.

Третий этап `3`-метода модифицируется следующим образом. Разрешение ленивых инвариантов меток не требуется, поскольку новых меток при трансляции C#-light программы в C#-kernel не появляется. Ленивые термы частично разрешаются на этапе дедуктивного вывода условий корректности за счет встраивания алгоритмов операционной семантики в этот вывод. С этой целью логика безопасности реализуется как комбинация операционных и дедуктивных ПОСП, где операционные ПОСП определяют семантику интерпретированных символов.

Работа имеет следующую структуру. В разд. 2 даны краткие спецификации языков C#-light и C#-kernel. Подробные спецификации этих языков и других составляющих `3`-метода могут быть найдены в [5]. В разд. 3 описываются модели языков C#-light и C#-kernel. В разд. 4 представлена операционная семантика этих языков. В разд. 5 определяется логика безопасности для языка C#-kernel. В разд. 6 рассмотрено расширение C#-light на небезопасный код [7]. Пример, иллюстрирую-

щий применение ЗМ-метода, представлен в разд. 7. В заключение подытоживаются результаты работы, и даются будущие планы.

Работа выполнена при финансовой поддержке гранта РФФИ №11-01-00028-а.

2. ЯЗЫКИ C#-LIGHT И C#-KERNEL

В этом разделе кратко описываются языки C#-light и C#-kernel. Полное описание этих языков может быть найдено в [5].

2.1. Язык C#-light

Язык C#-light поддерживает большинство конструкций последовательного подмножества языка C# 1.0, за исключением атрибутов, деструкторов, оператора `using`, конструкций `checked` и `unchecked`, небезопасного кода и директив препроцессора. Таким образом, язык C#-light является достаточно представительным подмножеством языка C#, включающим, в частности, свойства, события, делегаты и индексаторы. В языке C#-light явным образом зафиксирован способ аннотирования программ посредством комментариев вида `/// <a> R </a>`, где `R` — формула языка аннотаций.

2.2. Язык C#-kernel

Язык C#-kernel строится на основе некоторого подмножества M языка C#-light, удовлетворяющего следующим ограничениям:

- M не содержит пространств имен и `using`-директив;
- M не содержит событий;
- M не содержит операторов перехода `break`, `continue`, `return`, `goto case`, `goto default` и `throw`, оператора `try`, оператора выбора `switch`, операторов циклов, операторов-объявлений;
- M содержит только операторы `if`, в которых имеется ветвь `else`, а условное выражение является переменной типа `boolean`;
- все метки, имена локальных переменных и имена локальных констант должны быть уникальны в программах из M;
- множества меток, имен типов, имен локальных переменных и имен локальных констант не пересекаются.

Кроме того, язык C#-kernel содержит метаинструкции и имеет ограничения на операторы-выражения и объявления классов и структур.

Метаинструкции используются для работы с метапеременными, описывающими состояние абстрактной машины языка C#-light. Абстрактная машина включает 6 метапеременных:

- метапеременная `L` сопоставляет локальным переменным программы ячейки памяти;
- метапеременная `L` сопоставляет ячейкам памяти хранящиеся в них значения;
- метапеременная `T` сопоставляет локальным переменным и ячейкам памяти абстрактные имена типов;
- метапеременная `L2` сопоставляет ячейке памяти, соответствующей структуре, и имени поля этой структуры, ячейку памяти, соответствующую полю структуры, а ячейки памяти, соответствующей массиву, и индексу, — ячейку памяти для элемента массива с этим индексом;
- метапеременная `V0` содержит значение последнего вычисленного выражения;
- метапеременная `E` содержит значение последнего порожденного исключения.

В C#-kernel существует пять метаинструкций:

- `A := B` присваивает метапеременной `A` значение выражения `B` языка аннотаций;
- `new_instance()` выделяет новую ячейку памяти и помещает ее в `V0`;
- `Init(A)` выполняет статическую инициализацию класса или структуры `A`, если тип `A` не инициализирован;
- `catch(A, B)` возвращает `true`, если `E` содержит значение типа `A`, а иначе возвращает `false`. Кроме того, в первом случае объект-исключение, находящийся в `E`, становится значением переменной `B`, а неопределенное значение ω становится значением метапеременной `E`, чтобы показать, что исключение перехвачено. Эта метаинструкция может использоваться только как условное выражение в операторе `if`;
- `catch(A)` возвращает `true`, если выполнено `E` $\neq \omega$, иначе возвращает `false`. Кроме того, в первом случае объект-исключение, находящийся в `E`, становится значением переменной `A`, а ω становится значением метапеременной `E`. Эта метаинструкция используется как условное выражение в операторе `if`, моделируя общую `catch`-секцию оператора `try`.

3. МОДЕЛИ ЯЗЫКОВ C#-LIGHT И C#-KERNEL

В соответствии с методологией применения ПОСП к разработке различных видов семантики языков программирования (операционной, аксиоматической, трансформационной), семантика задается не для самих этих языков, а для их представлений (моделей) на языке описания ПОСП Atoment. Модель языка программирования включает описание сортов, состояний и конструкций языка. Сорта используются для категоризации элементов языка. В 3-методе используются аннотированные программы языков C#-light и C#-kernel, поэтому в модель включено также описание конструкций языка аннотаций.

3.1. Сорта

В дополнение к предопределенным сортам языка Atoment модель языков C#-light и C#-kernel включает следующие базовые сорта.

Элементами сорта `identifier` являются идентификаторы языка C#. Сорт `identifier` является подсортом предопределенного сорта `atom` языка Atoment.

Элементами сорта `literal` являются представления всех литералов языка C#-kernel за исключением литералов `true` и `false`. Эти литералы являются элементами предопределенного сорта `bool`. Они имеют вид `{literal A}`, где `A` — элемент. Например, литералы `2`, `2u` и `2.3` представляются как `{literal 2}`, `{literal 2u}` и `{literal 2.3}` и `{literal true}`, где `A` принадлежит предопределенному сорту `int`, `atom` и `real`, соответственно.

Сорт `literal` разбивается на попарно непересекающиеся подсорта, соответствующие типам языка C#, значения которых представляются литералами. Имена этих подсортов образуются как имена соответствующих типов с постфиксом C#. Например, подсорт `intC#` соответствует типу `int` языка C#.

Элементами сорта `namespace` являются пространства имен, используемые в C#-light программе. Элементами сорта `type` являются типы, используемые в C#-light программе. Элементами сорта `funMem` являются функциональные члены, используемые в C#-light программе. Элементами сорта `value` являются значения value-типов языка C# за исключением литералов. Элементами сорта `object` являются значения reference-типов языка C#.

Элементы сорта $\delta \in \{$`namespace`,`type`,`funMem`,`value`,`object`$\}$ опреде-

ляются конструктором со спецификацией (fun {δ A} par (A nat0)). Такой сорт называется нумерованным.

3.2. Состояния

Состояние в 3-методе определялось как отображение метапеременных в их значения. В языке Atoment состояние определяется как отображение параметрических переменных в их значения. Таким образом, в 3M-методе метапеременные являются частным случаем параметрических переменных языка Atoment. Например, метапеременной $V0$ соответствует предопределенная переменная (val) языка Atoment. . Опишем параметрические переменные, используемые для описания состояния C#-light и C#-kernel программ.

Переменная со спецификацией (var (value of A) par (A identifier) sort stack) принимает значение U тогда и только тогда, когда U — значение переменной A программы.

Переменная со спецификацией (var (type of A) par (A identifier) sort (stack type)) принимает значение U тогда и только тогда, когда U — тип переменной A программы.

Переменная со спецификацией (var (A is variable) par (A identifier) sort (stack bool)) принимает значение true тогда и только тогда, когда A — локальная переменная.

Переменная со спецификацией (var (locVarDecType) sort (stack type)) хранит типы, объявляемые в декларациях локальных переменных, и используется для получения информации о типе локальной переменной в деклараторах локальных переменных.

Переменная со спецификацией (var (A of B) par (A identifier) (B *) sort *) принимает значение U тогда и только тогда, когда U — значение поля A объекта или типа B.

Переменная со спецификацией (var (A of B) par (A intC#) (B *) sort *) принимает значение U тогда и только тогда, когда U — значение элемента массива B с индексом A.

Переменная со спецификацией (var (catchedException) sort stack) хранит исключения, обработанные элементами catch операторов try, и используется для порождения выражения перехода оператором throw без аргументов.

Переменная со спецификацией (var (entryPoint) sort funMem) принимает значение U тогда и только тогда, когда U — метод, который является точкой входа в программу.

Переменная со спецификацией `(var (progBody) sort exp)` принимает значение `U` тогда и только тогда, когда U — тело программы.

Семейство переменных со спецификациями `(var (name of A) par (A` δ`) sort identifier)`, где $\delta \in$ {`namespace`,`type`,`funMem`}, принимает значение `U` тогда и только тогда, когда U — имя элемента `A`.

Семейство переменных со спецификациями `(var (declaration of A) par (A` δ`) sort exp)`, где $\delta \in$ {`namespace`,`type`,`funMem`}, принимает значение `U` тогда и только тогда, когда U — декларация элемента `A`.

Семейство переменных со спецификациями `(var (containingElement of A) par (A` δ`) sort *)`, где $\delta \in$ {`namespace`, `type`,`funMem`}, принимает значение `U` тогда и только тогда, когда U — элемент, декларация которого содержит декларацию элемента `A` на верхнем уровне.

Семейство переменных со спецификациями `(var (subElements of A) par (A` δ`) sort exp)`, где $\delta \in$ {`namespace`,`type`,`funMem`}, принимает значение `U` тогда и только тогда, когда U — элемент, декларация которого содержится в декларации элемента `A` на верхнем уровне.

Переменная со спецификацией `(var (A is initialized) par (A type) sort bool)` принимает значение `true` тогда и только тогда, когда тип `A` инициализирован.

Переменная со спецификацией `(var (A is literalType) par (A type) sort bool)` принимает значение `true` тогда и только тогда, когда `A` — тип, элементы которого представляются литералами языка C#.

Переменная со спецификацией `(var (A is valueType) par (A type) sort bool)` принимает значение `true` тогда и только тогда, когда `A` — value-тип, элементы которого не представляются литералами языка C#.

Переменная со спецификацией `(var (A is referenceType) par (A type) sort bool)` принимает значение `true` тогда и только тогда, когда `A` — reference-тип.

3.3. Язык аннотаций

Выражения языка аннотаций представляются выражениями языка Atoment. Язык Atoment имеет достаточный набор предопределенных функций для представления формул.

Логические константы представляются элементами `true` и `false` сорта `bool`.

Пропозициональные связки представляются функциями со спецификациями:

- `(fun (A and B) par (A bool) (B bool) sort bool)`,
- `(fun (A or B) par (A bool) (B bool) sort bool)`,
- `(fun (A => B) par (A bool) (B bool) sort bool)`,
- `(fun (A <=> B) par (A bool) (B bool) sort bool)`,
- `(fun (not A) par (A bool) sort bool)`.

Кванторы представляются функциями со спецификациями:

- `(fun (forall A B) par (A atom) (B !!) sort bool)`,
- `(fun (forall A B) par (A atom) (B !!) sort bool)`.

Элементы сорта `!!` определяются конструктором со спецификацией `(fun {!! A} par (A *))`. Они играют роль quote-выражений, которые не вычисляются, а подставляются как есть.

В отличие от 3-метода в 3M-методе λ-выражения не используются. Поэтому не требуется средств для их представления, хотя, для того чтобы охватить и λ-выражения, в язык Atoment достаточно добавить предопределенную функцию со спецификацией `(fun (lambda A B) par (- A exp) (- B *) sort *)`, где выражение `A` представляет список переменных λ-выражения, а B представляет само λ-выражение.

3.4. Конструкции языка C#-light

Проиллюстрируем принципы построения модели на примере нескольких конструкций языка C#-light. Для остальных конструкций модели строятся аналогичным образом.

Литералы языка C# представляются элементами сортов `literal` и `bool`.

Переменные представляются элементами сорта `identifier`.

Оператор `while (A) B` представляется элементом `(while` A' B'`)`, где A' и B' — модели условия A и тела B этого оператора.

Выражение `A + B` представляется элементом `(`A' `+` B'`)`, где A' и B' — модели подвыражений A и B.

Выражение `A.B(`C_1`, ..., `C_n`)` представляется элементом `(call` A' `on` B' `args` C'_1 `...` C'_n`)`, где A', B', C'_1, ..., C'_n — модели соответствующих конструкций языка C#-light.

Таким образом, трансляция из C#-light в Atoment определяется так, что существует взаимно-однозначное соответствие между конструкциями языка C#-light (C#-kernel) и их моделями на языке Atoment. Как следствие, описав операционную семантику моделей программ языка

C#-light (C#-kernel), тем самым мы опишем операционную семантику самого языка C#-light (C#-kernel). Более точно, операционная семантика программы на языке C#-light (C#-kernel) является комбинацией денотационной семантики, отображающей эту программу в ее модель на языке Atoment, и операционной семантики полученной модели.

3.5. Дополнительные элементы языка C#-kernel

Метаинструкции, с одной стороны, представляют базис (минимальный набор), к которым сводятся конструкции языка C#-light при трансляции в C#-kernel. С другой стороны, они выбраны таким образом, чтобы для них правила аксиоматической семантики можно было бы задать относительно просто.

Поскольку в 3M-методе конструкции языка C#-light остаются такими как есть, язык C#-kernel становится надъязыком языка C#-light, а именно, он содержит все конструкции языка C#-light и дополнительно метаинструкции. Метаинструкции в этом случае представляют собой удобное средство упрощения правил операционной семантики.

Моделью метаинструкции присваивания `A := B;` является предопределенный элемент `(A′ ::= B′)`. Параметрическая переменная `A′` языка Atoment представляет соответствующую метапеременную `A`, элемент `B′` — выражение `B`.

Метаинструкция выделения памяти `new_instance()` моделируется предопределенным элементом `(newElemOfSort object)`.

Моделью метаинструкции `Init(C)` является элемент `(initType C′)`, где элемент `C′` — модель типа `C`.

Метаинструкции `catch(A, B)` и `catch(A)` не нужны, т.к. C#-kernel содержит оператор `try`.

Элемент `(gotoStop A)` обрабатывает выражения перехода вида `(jump goto B)`, где `B` — метка, в случае, когда точка назначения выражения перехода находится внутри блока.

Элемент `(switchJumpStop A)` обрабатывает выражения перехода вида `(jump switch B)`, `(jump goto default)`, `(jump goto case C)` и `(jump goto D)` в случае, когда точка назначения выражения перехода находится внутри блока оператора `switch`.

Элемент `(deleteVarScope A)` уменьшает на единицу стеки `(value of B)`, `(type of B)` и `(B is variable)` для каждой переменной `B`, входящей в список переменных `A`. Он используется при описании семантики блока.

Элемент `(initiateObject A type B beforeConstructorCall)` выполняет инициализацию полей объекта `A` типа `B`. Он соответствует методу `IFI` для этого типа, декларация которого добавлялась к типу на этапе перевода из C#-light в C#-kernel в `3`-методе.

Элемент `(initiateType A beforeStaticConstructorCall)` выполняет инициализацию статических полей типа `A`. Он соответствует статическому методу `SFI` для этого типа, декларация которого добавлялась к типу на этапе перевода из C#-light в C#-kernel в `3`-методе.

Элемент `(breakStop)` является обработчиком выражения перехода `(jump break)`.

Элемент `(SpecifyFunMemPars A B)` определяет параметры функционального члена `A` как локальные переменные и присваивает из значения из списка значений аргументов `B`.

Элемент `(SpecifyThis type A val B)` определяет `this` как локальную переменную типа `A` со значением `B`.

Элемент `(preprocess A)` выполняет преобразования тела `A` программы. Эти преобразования базируются на методе атрибутных аннотаций [2], особенность которого состоит в том, что преобразования не меняют операционную семантику программы, а только добавляют дополнительную информацию в нее. Этот элемент выполняет следующие преобразования:

- инициализирует переменную `(progBody)` значением `(A)`;
- вычисляет константные выражения `A` в элементах `(case A)` и `(goto case A)` операторов `switch`. Эти выражения в результате принимают вид `(case B)` и `(goto case B)`, где `B` — значение выражения `A`;
- порождает функциональные члены (элементы сорта `funMem`), типы (элементы сорта `type`) и пространства имен (элементы сорта `namespace`) для всех таких элементов, используемых или декларируемых в `A`. Типы при этом преобразовании не инициализируются;
- выполняет алгоритмы времени компиляции в соответствие со спецификацией [7], которые заменяют простые и квалифицированные имена соответствующими элементами из предыдущего пункта;
- доступ к свойствам и индексаторам, а также вызовы операций преобразуются в вызовы методов в соответствие с правилами для зарезервированных имен членов. Например, присваивание `((A .`

B) = C) свойству B объекта A значения C заменяется вызовом (call set_B on A args C) метода set_B;

- означивает переменные (entryPoint), (name of (** δ)), ((** type) is literalType), ((** type) is valueType), ((** type) is referenceType), (declaration of (** δ)), (containingElement of (** δ)) и (subElements of (** δ)) для $\delta \in$ {namespace,type,funMem}.

3.6. Функции

В этом разделе описываются функции, добавляемые в язык Atoment, которые используются при формализации 3M-метода.

Функция со спецификацией (fun (varList of A) par (- A exp) sort exp) принимает значение U тогда и только тогда, когда U — список переменных, определяемых в декларациях, которые входят в выражение A языка Atoment на верхнем уровне. Эта функция определяется с помощью правила перехода. Ее определение здесь не приводится.

Функция со спецификацией (fun (A <=type B) par (A type) (B type) sort bool) принимает значение true тогда и только тогда, когда B — базовый тип A, или B = A.

Функция со спецификацией (fun (cast A B) par (A *) (B type) sort *) принимает значение U тогда и только тогда, когда U — результат приведения элемента A к типу B.

Функция со спецификацией (fun (defaultValue of A) par (A type)) принимает значение U тогда и только тогда, когда U — значение по умолчанию для типа A.

Семейство предопределенных функций языка Atoment со спецификациями (fun (A isOf δ) par (A *)) принимает значение true тогда и только тогда, когда A — элемент сорта δ.

Семейство функций со спецификациями (fun (A $\odot$C# B) par (A δ_1) (B δ_2) sort intC#) принимает значение U тогда и только тогда, когда U — результат выполнения операции $\odot$ языка C# для аргументов A и B сортов δ_1 и δ_2. Например, функция из этого семейства со спецификацией (fun (A +C# B) par (A intC#) (B intC#) sort intC#) принимает значение U тогда и только тогда, когда U — результат сложения целых чисел A и B.

Функция со спецификацией (fun (resolveName A) par (A *) sort *) принимает значение U тогда и только тогда, когда U — элемент одного из сортов funMem, type или namespace, соответствующий простому или

квалифицированному имени A.

Функция со спецификацией (fun (resolveConstructor A arg B) par (A type) (B exp) sort funMem) принимает значение U тогда и только тогда, когда U — динамический конструктор типа A относительно списка аргументов B.

Функция со спецификацией (fun (resolveConstructor A) par (A type) sort funMem) принимает значение U тогда и только тогда, когда U — статический конструктор типа A.

4. ОПЕРАЦИОННАЯ СЕМАНТИКА ЯЗЫКОВ C#-LIGHT И C#-KERNEL

Операционная семантика этих языков описывается правилами ПОСП. В разд. 4.1 описан механизм передачи управления в языке C#. В разд. 4.2 представлены правила для C#-light конструкций. В разд. 4.3 представлены правила для дополнительных элементов языка C#-kernel, отсутствующих в языке C#-light.

4.1. Механизм передачи управления

Механизм передачи управления (в частности, просачивания исключений) в языке C# описывается с помощью концепции выражения перехода. Выражение перехода — это выражение вида (jump A), где A — последовательность элементов, кодирующая информацию передаваемую оператором перехода (jump statement), который породил это выражение перехода. Например, для оператора (goto A) порождаемое выражение перехода будет иметь вид (jump goto A), а для оператора (throw A) — (jump A′), где A′ — результат вычисления выражения A.

Правила для выражения перехода описывают случаи, когда отношение перехода обрабатывается элементами языков C#-light и C#-kernel, и случаи просачивания (propagation) выражения перехода. Элементы, которые обрабатывают выражение перехода, называются обработчиками отношения перехода. Правила для выражения перехода имеют вид
(if (jump goto default) (default) then)

(if (jump goto case A) (case A) par A then)

(if (jump goto switch A) (case A) par A then)

```
(if (jump goto default) (switchJumpStop U (default) V)
 par (!s U) (!s V) then V (switchJumpStop U (default) V) )

(if (jump goto case A) (switchJumpStop U (case A) V)
 par A (!s U) (!s V) then V (switchJumpStop U (case A) V) )

(if (jump switch A) (switchJumpStop U (default) V)
 par A (!s U) (!s V) then V (switchJumpStop U (default) V) )

(if (jump switch A) (switchJumpStop B) par A (!s B) then)

(if (jump goto A) (gotoStop U (label A) V) par A (!s U) (!s V)
 then V (gotoStop U (label A) V))

(if (jump break) (breakStop) var (!s A) then)

(if (jump continue) (while A thereWasIteration) var (!s A)
 then (while A thereWasIteration) )

(if (jump goto A) (label A) par A B then)

(if (jump exc E) (catch A B C) par A B C E then (cases
 (if ((type of E) <=type A) then
  (add (catchedException)) ((catchedException) ::= E)
  (add (value of B)) ((value of B) ::= E)
  (add (type of B)) ((type of B) ::= A)
  C
  (finally (delete (value of B)) (delete (type of B))
   (delete (catchedException)) ))
 (else (jump exc E)) ))

(if (jump exc E) (catch A C) par A C then (cases
  (if ((type of E) <=type A) then
   (add (catchedException)) ((catchedException) ::= E)
   C (finally (delete (catchedException)) ))
  (else (jump exc E)) ))

(if (jump exc E) (catch C) par C then
```

```
 (add (catchedException)) ((catchedException) ::= E)
 C (finally (delete (catchedException))) )

(if (jump E) (finally A) par (!s E) (!s A) then A (jump E))

(if (jump E) A par (!s E) A then (jump E))
```

Для простоты рассмотрены только те обработчики выражения перехода элементами, для которых в разд. 4 и 4.3 заданы правила операционной семантики. Например, оператор цикла `do-while` не рассмотрен.

4.2. Правила операционной семантики языка C#-light

Задачу разработки операционной семантики модели языка C#-light мы проиллюстрируем на примере нескольких конструкций этого языка. Для остальных конструкций правила операционной семантики описываются аналогичным образом.

4.2.1. Правило для программы

Правило для программы имеет вид

```
(if (prog A) var (!s A) then ((count) ::= 1) (preprocess A)
 (call (!* entryPoint) on type
  (!* containingElement of (entryPoint)) args ))
```

Для простоты мы ограничились случаем, когда функция `Main` не имеет аргументов и не возвращает значения. Общий случай получается некоторым усложнением этого правила.

4.2.2. Блок

Правило для блока имеет вид

```
(if (block A) par (!s A) then A (gotoStop A)
 (finally (deleteVarScope (!* (varList of (A))))) )
```

Напомним, что согласно определению правила перехода, перед сопоставлением элемента с образцом правила каждый подэлемент этого элемента вида `(!* A)` заменяется значением элемента `A`.

4.2.3. Оператор `if`

Правила для условного оператора `if` имеют вид

```
(if (if A then B else C) par A B C then B (cases
```

```
 (if ((val) = true) then B) (else C) ))

(if (if A then B) par A B then B (cases
 (if ((val) = true) then B) (else) ))
```

4.2.4. Оператор `switch`

Правила для оператора `switch` имеют вид

```
(if (switch A (block B)) par A (!s B) hvar WA then
 A (jump switch (!* (val))) B (switchJumpStop A) (breakStop)
 (finally (deleteVarScope (!* (varList of (A))))) )
```

4.2.5. Цикл `while`

Правило для оператора цикла `while` имеет вид

```
(if (while A B) var A B then A (cases
 (if ((val) = true) then B (while A B thereWasIteration)
  (breakStop) )
 (else) ))
```

4.2.6. Оператор-метка

Правило для оператора-метки имеет вид

```
(if (label A) par A then)
```

4.2.7. Оператор `try`

Правило для оператора `try` имеет вид

```
(if (try A) par (!s A) then A)
```

Правила для элементов `(catch A)` и `(finally A)` имеют вид

```
(if (catch A) par (!s A) then)

(if (finally A) par (!s A) then A)
```

4.2.8. Операторы передачи управления

Правила для операторов передачи управления имеют вид

```
(if (goto default) then (jump goto default))

(if (goto case A) par A then A (jump goto case A))
```

```
(if (goto A) par A then (jump goto A))

(if (break) then (jump break))

(if (continue) then (jump continue))

(if (return) then (jump return))

(if (return A) par A then A (jump return (!* (val))))

(if (throw A) par A hvar W then A (cases
 (if ((val) = null) then
  (new (!* resolveName (System . NullReferenceException)))
  (jump exc (!* (val))))
 (else (jump exc (!* (val)))) ))

(if (throw) then (jump exc (!* (catchedException))))
```

4.2.9. Декларации локальных переменных

Правило для декларации локальной переменной имеет вид

```
(if (locVarDec A B) par A (!s B) then (add (locVarDecType))
 ((locVarDecType) ::= A) B
 (finally (delete (locVarDecType))) )
```

Правила для декларатора локальной переменной имеют вид

```
(if (locVarDec-r A) par A then ((A is variable) ::= true)
 ((type of A) ::= (locVarDecType))
 ((value of A) ::= (defaultValue of (locVarDecType))) )

(if (locVarDec-r A = B) par A B then
 ((A is variable) ::= true)
 ((type of A) ::= (locVarDecType))
 ((value of A) ::= (defaultValue of (locVarDecType)))
 B ((value of A) ::= (val)) )
```

4.2.10. Оператор выражения

Правило для оператора выражения имеет вид

```
(if (A ;) par A then A)
```

4.2.11. Выражения

Расмотрим типовые схемы определения правил операционной семантики на примере нескольких видов выражений. Для остальных выражений правила определяются по одной из этих схем.

Правило для операции сложения целых чисел имеет вид

```
(if (A + B) par A B where ((A isOf intC#) and (B isOf intC#))
 hvar WA then A (WA ::= (val)) B ((val) ::= (WA +C# (val))) )
```

Правила для литералов языка C# имеют вид

```
(if true then ((val) ::= true))

(if false then ((val) ::= false))

(if A par A where (A isOf literal) then ((val) ::= A))
```

Правило для доступа к значению локальной переменной имеет вид

```
(if A par A where (A is variable) then
 ((val) ::= (value of x)) )
```

Правило для вызова функционального члена `(call ...)` на экземпляре типа имеет вид

```
(if (call A on B args C) par A B (!s C) hvar WA WB WC then
 B (WB ::= (val)) (evaluateArgs C) (WC ::= (val))
 (findFunMem A on WB args (!* WC)) (WA ::= (val))
 (block (specifyFunMemPars WA WC)
  (specifyThis type (!* (containingElement of WA)) val WB)
  (insert body of (!* WA)) ))
```

Правило для вызова функционального члена `(call ...)` на типе имеет вид

```
(if (call A on type B args C) par A B (!s C) hvar WA WC then
 (evaluateArgs C) (WC ::= (val))
 (findFunMem A on B args (!* WC)) (WA ::= (val)) (initType B)
 (block (specifyFunMemPars WA WC)
  (insert body of (!* WA)) ))
```

Правила для операции `(new ...)` создания элемента некоторого типа имеют вид

```
(if (new A B) par A (!s B) hvar W then
 (InitType A) (cases
  (if (A is literalType) then
```

```
   ((val) ::= (default value of A)))
  (if (A is valueType) then (newElemOfSort value)
   (W ::= (val)) ((type of W) ::= A)
   (initiateObject W type A args B) ((val) ::= W) )
  (if (A is referenceType) then (newElemOfSort object)
   (W ::= (val)) ((type of W) ::= A)
   (initiateObject W type A args B) ((val) ::= W) )))
```

4.3. Операционная семантика дополнительных элементов языка C#-kernel

Правило для элемента `(initType A)` имеет вид

```
(if (initType A) par A then (cases
 (if (A is initialized) then)
 (else ((A is initialized) ::= true)
  (initType A beforeStaticConstructorCall)
  (call (resolveConstructor A) on A args) )))
```

Правило для элемента `(specifyAsLocalVariable A type B val C)` имеет вид

```
(if (specifyAsLocalVariable A type B val C) par A B C then
 (add (type of A)) (add (value of A)) (add (A is variable))
 ((type of this) ::= B) ((value of this) ::= C)
 ((A is variable) ::= true) )
```

Правило для элемента `(specifyThis type A val B)` имеет вид

```
(if (specifyThis type A val B) par A B then
 (add (type of this)) (add (value of this))
 (add (this is variable))
 ((type of this) ::= A) ((value of this) ::= B)
 ((this is variable) ::= true) )
```

Правила для элемента `(deleteVarScope` x`)` имеют вид

```
(if (deleteVarScope (A B)) par A (!s B) then
 (delete (value of A)) (delete (type of A))
 (delete (A is variable)) (deleteVarScope (B)) )

(if (deleteVarScope ()) then)
```

Правило для элемента `(gotoStop A)` имеет вид

```
(if (gotoStop A) par (!s A) then)
```

Правило для элемента `(breakStop)` имеет вид

```
(if (breakStop) then)
```

Правило для элемента `(initiateObject A type B args C)` имеет вид

```
(if (initiateObject A type B args C) par A B C then
 (initiateObject A type B beforeConstructorCall)
 (call (resolveConstructor A args C) on A args C) )
```

Правило для элемента `(while A B thereWasIteration)` имеет вид

```
(if (while A B thereWasIteration) var A B then A (cases
 (if ((val) = true) then B (while A B thereWasIteration))
 (else) ))
```

Правило для элемента `(switchJumpStop A)` имеет вид

```
(if (switchJumpStop A) par A then)
```

Правило для элемента `(insert body of A)` имеет вид

```
(if (insert body of A) par A then (!** (body of A)))
```

Чтобы упростить изложение, мы не приводим правил операционной семантики для некоторых дополнительных элементов из-за их громоздкости, например для элемента `(initiateObject A type B beforeConstructorCall)`. Разработка этих правил является рутинной процедурой.

5. ЛОГИКА БЕЗОПАСНОСТИ ЯЗЫКА C#-KERNEL

Логика безопасности языка C#-kernel описывается правилами дедуктивных ПОСП. В разд. 5.1 специфицируются версионные переменные логики безопасности языка C#-kernel. В разд. 5.2 перечислены функции, используемые в правилах логики безопасности языка C#-kernel. В разд. 5.3 описаны правила логики безопасности для выражения перехода. В разд. 5.4 представлены правила логики безопасности для C#-light конструкций. В разд. 5.5 представлены правила логики безопасности для дополнительных элементов языка C#-kernel, отсутствующих в языке C#-light.

5.1. Версионные переменные

В качестве версионных переменных логика безопасности языка C#-kernel использует переменные, определенные в разд. 3.2. Дополнительно она использует следующие переменные.

Переменная со спецификацией `(var (exc) sort object)` принимает значение `U` тогда и только тогда, когда U — порожденное исключение. Она используется в логике безопасности языка C#-kernel. Свойство

`U = ()` означает, что никакого исключения не порождено. При описании операционных семантик языков C#-light и C#-kernel эта переменная не используется, т.к. в этом случае `U` хранится в выражении перехода вида `(jump exc U)`.

Переменная со спецификацией `(var (assume of A) par (A funMem) sort *)` принимает значение `U` тогда и только тогда, когда U — предусловие функционального члена `A`. Свойство `U = ()` означает, что `A` не имеет предусловия.

Переменная со спецификацией `(var (assert of A) par (A funMem) sort *)` принимает значение `U` тогда и только тогда, когда U — постусловие функционального члена `A`. Свойство `U = ()` означает, что `A` не имеет постусловия.

Переменная `((** type) is initialized)` используется в логике безопасности языка C#-kernel в двух контекстах. В первом контексте она является обычной переменной и хранит информацию об инициализированных типах, используемую для оптимизации вывода условий корректности. Во втором контексте она является версионной переменной логики безопасности.

5.2. Функции

Функция со спецификацией `(fun (pre of A) par (A funMem) sort *)` возвращает `U` тогда и только тогда, когда `U` имеет вид

```
(((value of this) != null) and ((value of this) != ()) and
 ((type of (value of this)) <=type
  (containingElement of A)) and
 ((type of (value of this)) is initialized) and
 ((exc) = ()) and ((containingElement of A) is initialized) )
```

Она используется для подстановки при вызове функционального члена предусловия, истинного для всех функциональных членов.

5.3. Механизм передачи управления

Правила логики безопасности для выражения перехода имеют вид

```
(if (jump goto default) (default) then)

(if (jump goto case A) (case A) par A then)

(if (jump goto switch A) (case B) par A B then
```

```
 (casesD (if (A = B) then) (else (jump goto switch A))) )

(if (jump goto default) (switchJumpStop U (default) V)
 par (!s U) (!s V) then V (switchJumpStop U (default) V) )

(if (jump goto case A) (switchJumpStop U (case A) V)
 par A (!s U) (!s V) then V (switchJumpStop U (case A) V) )

(if (jump switch A) (switchJumpStop U (default) V)
 par A (!s U) (!s V) then V (switchJumpStop U (default) V) )

(if (jump switch A) (switchJumpStop B) par A (!s B) then)

(if (jump goto A) (gotoStop U (label A) V) par A (!s U) (!s V)
 then V (gotoStop U (label A) V))

(if (jump break) (breakStop) var (!s A) then)

(if (jump continue) (while A thereWasIteration) var (!s A)
 then (while A thereWasIteration) )

(if (jump goto A) (label A) par A B then)

(if (jump exc E) (catch A B C) par A B C E then (casesD
 (if ((type of E) <=type A) then
  (addD (catchedException)) ((catchedException) ::=D E)
  (add (value of B)) ((value of B) ::=D E)
  (addD (type of B)) ((type of B) ::=D A) C
  (finally (deleteD (value of B)) (deleteD (type of B))
   (deleteD (catchedException)) ))
 (else (jump exc E)) ))

(if (jump exc E) (catch A C) par A C then (casesD
  (if ((type of E) <=type A) then
   (addD (catchedException)) ((catchedException) ::=D E)
   C (finally (deleteD (catchedException)) ))
  (else (jump exc E)) ))
```

```
(if (jump exc E) (catch C) par C then
 (addD (catchedException)) ((catchedException) ::=D E)
 C (finally (deleteD (catchedException))) )

(if (jump E) (finally A) par (!s E) (!s A) then A (jump E))

(if (jump E) A par (!s E) A then (jump E))
```

5.4. Логика безопасности C#-light конструкций

5.4.1. Правило для программы

Правило для программы имеет вид

```
(if (prog A assume B assert C) var (!s A) B C then
 ((count) ::= 1) (initVer) (preprocess A)
 (setPre B) ((catchedException) ::= ()) (assumeD ((exc) = ()))
 (call (!* entryPoint) on type
  (!* containingElement of (entryPoint)) args )
 (assertD C) )
```

Для простоты, как и в случае операционной семантики, мы ограничиваемся случаем, когда функция `Main` не имеет аргументов и не возвращает значения.

Предопределенный элемент `(initVer)` языка Atoment устанавливает значения версий всех версионных переменных равными значению переменной `(count)`.

По сравнению с операционной семантикой в случае логики безопасности элемент `(preprocess A)` дополнительно означивает переменные `(assume of (** funMem))` и `(assert of (** funMem))`.

5.4.2. Блок

Правило для блока имеет вид

```
(if (block A) par (!s A) then A (gotoStop A)
 (finally (deleteVarScope (!* (varList of (A))))) )
```

5.4.3. Оператор `if`

Правила для оператора `if` имеют вид

```
(if (if A then B else C) par A B C then B (casesD
 (if ((val) = true) then B) (else C) ))
```

```
(if (if A then B) par A B then B (casesD
 (if ((val) = true) then B) (else) ))
```

5.4.4. Оператор `switch`

Правила для оператора `switch` имеют вид

```
(if (switch A (block B)) par A (!s B) hvar WA then
 A (jump switch (val)) B (switchJumpStop A) (breakStop)
 (finally (deleteVarScope (!* (varList of (A))))) )
```

5.4.5. Оператор `while`

Правило для оператора цикла `while` имеет вид

```
(if (while A B) var A B then A (casesD
 (if ((val) = true) then B (while A B thereWasIteration)
  (breakStop) )
 (else) ))
```

5.4.6. Оператор-метка

Правило для оператора-метки имеет вид

```
(if (label A) par A then)
```

5.4.7. Оператор `try`

Правило для оператора `try` имеет вид

```
(if (try A) par (!s A) then A)
```

Правила для элементов `(catch A)` и `(finally A)` имеют вид

```
(if (catch A) par (!s A) then)

(if (finally A) par (!s A) then A)
```

5.4.8. Операторы передачи управления

Правила для операторов передачи управления имеют вид

```
(if (goto default) then (jump goto default))

(if (goto case A) par A then A (jump goto case A))
```

```
(if (goto A) par A then (jump goto A))

(if (break) then (jump break))

(if (continue) then (jump continue))

(if (return) then (jump return))

(if (return A) par A then A (jump return (val)))

(if (throw A) par A hvar W then A (casesD
 (if ((val) = null) then
  (new (!* resolveName (System . NullReferenceException)))
  (jump exc (val)) )
 (else (jump exc (val))) ))

(if (throw) then (jump exc (catchedException)))
```

5.4.9. Декларации локальных переменных

Правило для декларации локальной переменной имеет вид

```
(if (locVarDec A B) par A (!s B) then (addD (locVarDecType))
 ((locVarDecType) ::=D A) B
 (finally (deleteD (locVarDecType))) )
```

Правила для декларатора локальной переменной имеют вид

```
(if (locVarDec-r A) par A then ((A is variable) ::=D true)
 ((type of A) ::=D (locVarDecType))
 ((value of A) ::=D (defaultValue of (locVarDecType))) )

(if (locVarDec-r A = B) par A B then
 ((A is variable) ::=D true)
 ((type of A) ::=D (locVarDecType))
 ((value of A) ::=D (defaultValue of (locVarDecType)))
 B ((value of A) ::=D (val)) )
```

5.4.10. Оператор выражения

Правило для оператора выражения имеет вид

```
(if (A ;) par A then A)
```

5.4.11. Выражения

Расмотрим типовые схемы определения правил логики безопасности на примере нескольких видов выражений. Для остальных выражений правила определяются по одной из этих схем.

Правило для операции сложения целых чисел имеет вид

```
(if (A + B) par A B whereD ((A isOf intC#) and (B isOf intC#))
 hvar WA then A (WA ::=D (val)) B
  ((val) ::=D (WA +C# (val))) )
```

Правила для литералов языка C# имеют вид

```
(if true then ((val) ::=D true))

(if false then ((val) ::=D false))

(if A par A where (A isOf literal) then ((val) ::=D A))
```

Правило для доступа к значению локальной переменной имеет вид

```
(if A par A where (A is variable) then
 ((val) ::=D (value of x)) )
```

Правило для вызова функционального члена `(call ...)` на экземпляре типа имеет вид

```
(if (call A on B args C) par A B (!s C) hvar WA WB WC then
 B (WB ::= (val)) (evaluateArgs C) (WC ::= (val))
 (findFunMem A on WB args (!* WC)) (WA ::= (val))
 (block (specifyFunMemPars WA WC)
  (specifyThis type (!* (containingElement of WA)) val WB)
  (cases
 (if ((assert of WA) != ()) then
  (assumeD ((!* assume of WA) and (!* pre of WA)))
  (modifyD (!* (assert of WA))) )
 (else (insert body of (!* WA))) )))
```

Правило для вызова функционального члена `(call ...)` на типе имеет вид

```
(if (call A on type B args C) par A B (!s C) hvar WA WC then
 (evaluateArgs C) (WC ::= (val))
 (findFunMem A on B args (!* WC)) (WA ::= (val)) (initType B)
 (block (specifyFunMemPars WA WC)
  (cases
   (if ((assert of WA) != ()) then
    (assumeD ((!* assume of WA) and (!* pre of WA)))
```

```
  (modifyD (!* (assert of WA))) )
 (else (insert body of (!* WA))) )))
```

Элемент `(findFunMem A B C)` в случае логики безопасности реализует эмпирический алгоритм нахождения функционального члена, который вызывается. Этот алгоритм комбинирует алгоритмы операционной семантики с логическим выводом, информация для которого извлекается из предусловия `(pre)`. Если алгоритм терпит неудачу, он возвращает элемент `(no function member was found)`.

Правила для операции `(new ...)` создания элемента некоторого типа имеют вид

```
(if (new A B) par A (!s B) hvar W then
 (InitType A)
 (cases
  (if (A is literalType) then
   ((val) ::=D (default value of A)) )
  (if (A is valueType) then (newElemOfSort value)
   (W ::=D (val)) ((type of W) ::=D A)
   (initiateObject W type A args B) ((val) ::=D W) )
  (if (A is referenceType) then (newElemOfSort object)
   (W ::=D (val)) ((type of W) ::=D A)
   (initiateObject W type A args B) ((val) ::=D W) )))
```

5.5. Логика безопасности дополнительных элементов C#-kernel

Правило для элемента `(initType A)` имеет вид

```
(if (initType A) par A then (cases
 (if (A is initialized) then)
 (else ((A is initialized) ::= true)
  ((A is initialized) ::=D true)
  (initType A beforeStaticConstructorCall)
  (call (resolveConstructor A) on A args) )))
```

Правило для элемента `(specifyAsLocalVariable A type B val C)` имеет вид

```
(if (specifyAsLocalVariable A type B val C) par A B C then
 (addD (type of A)) (addD (value of A)) (addD (A is variable))
 ((type of A) ::=D B) ((value of A) ::=D C)
 ((A is variable) ::=D true) )
```

Правило для элемента `(specifyThis type A val B)` имеет вид

```
(if (specifyThis type A val B) par A B then
 (addD (type of this)) (addD (value of this))
 (addD (this is variable))
 ((type of this) ::=D A) ((value of this) ::=D B)
 ((this is variable) ::=D true) )
```

Правила для элемента `(deleteVarScope A)` имеют вид

```
(if (deleteVarScope (A B)) par A (!s B) then
 (deleteD (value of A)) (deleteD (type of A))
 (deleteD (A is variable)) (deleteVarScope (B)) )

(if (deleteVarScope ()) then)
```

Правило для элемента `(while A B thereWasIteration)` имеет вид

```
(if (while A B thereWasIteration) var A B then A (casesD
 (if ((val) = true) then B (while A B thereWasIteration))
 (else) ))
```

Правило для элемента `(gotoStop A)` имеет вид

```
(if (gotoStop A) par (!s A) then)
```

Правило для элемента `(breakStop)` имеет вид

```
(if (breakStop) then)
```

Правило для элемента `(switchJumpStop A)` имеет вид

```
(if (switchJumpStop A) par A then)
```

Правило для элемента `(insert body of A)` имеет вид

```
(if (insert body of A) par A then (!** (body of A)))
```

Правило для элемента `(initiateObject A type B args C)` имеет вид

```
(if (initiateObject A type B args C) par A B C then
 (initiateObject A type B beforeConstructorCall)
 (call (resolveConstructor B args C) on A args C) )
```

6. РАСШИРЕНИЕ ЯЗЫКА C#-LIGHT НА НЕБЕЗОПАСНЫЙ КОД

Язык C#-light расширен на небезопасный код, и для расширенного языка разработана операционная семантика на базе ОС-систем.

Набор сортов, описывающих состояние программы с небезопасным кодом, расширяется сортом `loc`. Элементами этого сорта являются ячейки памяти, выделяемые при работе C# программы. Элементы сорта `loc` определяются конструктором со спецификацией `(fun {loc A} par (A nat0))`.

Набор переменных, описывающих состояние программы с небезопасным кодом, расширяется следующими переменными.

Переменная со спецификацией `(var (loc of A) par (A identifier) sort loc)` принимает значение `U` тогда и только тогда, когда `U` — ячейка памяти, выделяемая для переменной `A` программы.

Переменная со спецификацией `(var (value of A) par (A loc) sort *)` принимает значение `U` тогда и только тогда, когда `U` — значение ячейки памяти `A`.

Переменная со спецификацией `(var (type of A) par (A loc) sort type)` принимает значение `U` тогда и только тогда, когда `U` — тип ячейки памяти `A`.

Переменная со спецификацией `(var (loc of A of B) par (A identifier) (B *) sort loc)` принимает значение `U` тогда и только тогда, когда `U` — ячейка памяти, выделенная под поле `A` структуры `B`.

Переменная со спецификацией `(var (loc of A of B) par (A intC#) (B *) sort loc)` принимает значение `U` тогда и только тогда, когда `U` — ячейка памяти, выделенная под элемент массива `B` с индексом `A`.

Правила операционной семантики и логики безопасности для конструкций языков C#-light и C#-kernel, попадающих в участки небезопасного кода, определяются аналогично правилам для языка C, описанным в работе [6].

7. ПРИМЕР ВЕРИФИКАЦИИ

В качестве примера верифицируемой программы возьмем пример из [5], который, в свою очередь, является адаптированным для языка C# вариантом Java программы из коллекции трудных для верификации примеров [9]. Этот пример иллюстрирует динамическое связывание, благодаря которому не происходит зацикливание при вызове метода `C.m`, хотя согласно его определению этот метод вызывает сам себя. В [5] с помощью `3`-метода, использующего аксиоматическую семантику языка C#-kernel, была доказана частичная корректность этой программы, т.е. что завершение этой программы порождает исключение типа `System.Exception`. Здесь мы применяем `3M`-метод, использующий логику безопасности языка C#-kernel, и доказываем тотальную корректность данной программы.

Исходная программа на языке C#-light имеет вид

```
class C {
```

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

class D:C {

 override void m() {
 throw new System.Exception(); }

 void Test() {base.m();}

 static void Main() {D x = new D; x.Test();} }
```

Результатом трансляции этой программы в модель на языке Atoment является элемент

```
(prog

 (class C body (

  (methodDeclaration modifiers (virtual) void m ()
   body (block ((mCallCount) ::=D (mCallCount) + 1)
   (assertD ((mCallCount) <= 1)) (call m ()) ))))

 (class D baseClass C body (

  (methodDeclaration modifiers (override) void m ()
   body (block (throw (new (System . Exception) ())))
   assert ((! type of (! exc)) = (System . Exception)) ))

  (methodDeclaration void Test ()
   body (block ((mCallCount) ::=D 0) (call m base ())) )

 (methodDeclaration modifiers (static) void Main ()
   body (block (locVarDec D (locVarDec-r x =  (new D)))
    (call Test x ()) )))

assume true
assert ((! type of (! exc)) = (System . Exception)) )
```

На первом шаге трансляции конструкции языка C#-light транслируются в соответствующие их модели. На втором шаге в результирующую

программу на языке C#-kernel добавляются пред- и постусловия с помощью ключевых слов `assume` и `assert` и новые элементы (они заключены в рамки), используемые для доказательства завершимости выполнения программы. Выполнение свойства безопасности `((mCallCount) <= 1)` гарантирует завершимость выполнения программы, т.к. переменная `(mCallCount)`, обозначающая число срабатываний метода `m` класса `C`, получает начальное значение `0`, и ее значение меняется только элементом `((mCallCount) ::= (mCallCount) + 1)`, т. е. может только возрастать. Для методов классов `C` и `D` нет спецификаций, поэтому при применении правил логики безопасности при вызове этих методов будут подставляться их тела.

Покажем, что выполнение программы безопасно относительно заданных свойств безопасности. В нашем случае эти свойства означают, что программа завершает свою работу и возвращает в качестве результата исключение типа `System.Exception`.

Пусть `p0` — верифицируемая программа, и `s0` — начальное состояние программы. Требуется проверить безопасность конфигурации `(p0, s0)`. Пусть `ProgBody` и `ProgPost` обозначает тело (последовательность деклараций) и постусловие программы `p0`, соответственно.

Применяя правило для программы, преобразуем `(p0, s0)` в `(p1, s1)`, где `s1 = s0`, и `p1` имеет вид

```
((count) ::= 1) (initVer) (preprocess ProgBody)
(setPre true) ((catchedException) ::= ())
(assumeD ((exc) = ()))
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Применяя правило для предопределенного элемента `(. ::= .)` языка Atoment, преобразуем `(p1, s1)` в `(p2, s2)`, где `p2` имеет вид

```
(initVer) (preprocess ProgBody)
(setPre true) ((catchedException) ::= ())
(assumeD ((exc) = ()))
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Состояние `s2` отличается от `s1` только на `(count)`, и `s2(count) = 1`.

Применяя правило для предопределенного элемента `(initVer)` языка Atoment, преобразуем `(p2, s2)` в `(p3, s3)`, где `p3` имеет вид

```
(preprocess ProgBody)
(setPre true) ((catchedException) ::= ())
(assumeD ((exc) = ()))
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Состояние `s3` отличается от `s2` только на `(ver (** verVar))`, и `s2(ver)(A) = s1(count) = 1` для любой версионной переменной `A`.

Применяя правило для элемента `(preprocess)`, преобразуем `(p3, s3)` в `(p4, s4)`, где `p4` имеет вид

```
(setPre true) ((catchedException) ::= ())
(assumeD ((exc) = ()))
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Состояние `s4` отличается от `s3` только на следующих аргументах:

- `s4(progBody)` возвращает тело программы;
- `s4(count) = 12`. Элемент `preprocess` порождает функциональные члены `{funMem` i`}` для $2 \leq i \leq 7$, соответствующие методам `C.m`, `D.m`, `D.Test`, `D.Main`, `C.C`, `D.D`; типы `{type` j`}` для $8 \leq j \leq 11$, соответствующие классам `C`, `D`, `System.Exception`, `System.NullReferenceException`; пространство имен `{namespace 12}`, соответствующее пространству имен `System`. Для простоты далее будем ссылаться на порожденные элементы по их символическим именам, например, `{C.m}` для `{funMem 2}`;
- `s4(entryPoint) = {D.Main}`;
- `s4(containingElement of {C.m})` возвращает `{C}`, ... ;
- `s4({C} is referenceType) = true`, ... ;
- `s4(name of {C.m}) = m`, ... ;
- `s4(declaration of {C.m})` возвращает декларацию метода `C.m`, ... ;
- `s4(subElements of {D}) = ({D.m} {D.Test} {D.Main})`, ...

Применяя правило для предопределенного элемента `(setPre .)` языка Atoment, преобразуем `(p4, s4)` в `(p5, s5)`, где `p5` имеет вид

```
((catchedException) ::= ()) (assumeD ((exc) = ()))
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Состояние s5 отличается от s4 только на (pre), и s5(pre) = true.

Применяя правило для элемента (. ::= .), преобразуем (p5, s5) в (p6, s6), где p6 имеет вид

```
(assumeD ((exc) = ()))
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Состояние s6 отличается от s5 только на (catchedException), и s6(catchedException) = ().

Применяя правило для предопределенного элемента (assumeD .) языка Atoment, преобразуем (p6, s6) в (p7, s7), где p7 имеет вид

```
(call (!* entryPoint) on type
 (!* containingElement of (entryPoint)) args)
(assertD ProgPost)
```

Состояние s7 отличается от s6 только на (pre), и s7(pre) = (true and ((!1 exc) = ())).

Применяя правило для элемента (call . on type . args .), преобразуем (p7, s7) в (p8, s8), где p8 имеет вид

```
(evaluateArgs) ((!h 9) ::= (val))
(findFunMem {D.Main} {D} (!h 9)) ((!h 8) ::= (val))
(initType {D})
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s8 отличается от s7 только на (count), и s8(count) = 14.

Применяя правило для элемента (evaluateArgs), преобразуем (p8, s8) в (p9, s9), где p9 имеет вид

```
((!h 9) ::= (val))
(findFunMem {D.Main} {D} (!h 9)) ((!h 8) ::= (val))
(initType {D})
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
```

```
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s9 отличается от s8 на (val), и s9(val) = ().

Применяя правило для элемента (. ::= .), преобразуем (p9, s9) в (p10, s10), где p10 имеет вид

```
(findFunMem {D.Main} {D} (!h 9)) ((!h 8) ::= (val))
(initType {D})
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s10 отличается от s9 только на (!h 9), и s10(!h 9) = ().

Применяя правило для элемента (findFunMem), преобразуем (p10, s10) в (p11, s11), где p11 имеет вид

```
((!h 8) ::= (val))
(initType {D})
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s11 отличается от s10 только на (val), и s11(val) = {D.Main}.

Применяя правило для элемента (. ::= .), преобразуем (p11, s11) в (p12, s12), где p12 имеет вид

```
(initType {D})
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
```

```
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние `s12` отличается от `s11` только на `(!h 8)`, и `s12(!h 10) = {D.Main}`.

Применяя правило для элемента `(initType .)`, преобразуем `(p12, s12)` в `(p13, s13)`, где `p13` имеет вид

```
(cases
 (if ({D} is initialized) then)
 (else (({D} is initialized) ::= true)
  (({D} is initialized) ::=D true)
  (initType {D} beforeStaticConstructorCall)
  (call (resolveConstructor {D}) on A args) ))
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние `s13` совпадает с `s12`.

Применяя правило для предопределенного элемента `(cases ...)` языка Atoment, преобразуем `(p13, s13)` в `(p14, s14)`, где `p14` имеет вид

```
(({D} is initialized) ::= true)
(({D} is initialized) ::=D true)
(initType {D} beforeStaticConstructorCall)
(call (resolveConstructor {D}) on A args)
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
```

```
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```
Состояние s14 совпадает с s13.

Применяя правило для элемента (. ::= .), преобразуем (p14, s14) в (p15, s15), где p15 имеет вид

```
(({D} is initialized) ::=D true)
(initType {D} beforeStaticConstructorCall)
(call (resolveConstructor {D}) on A args)
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s15 отличается от s14 только на (D is initialized), и s15(D is initialized) = true.

Применяя правило для элемента (. ::=D .), преобразуем (p15, s15) в (p16, s16), где p16 имеет вид

```
(initType {D} beforeStaticConstructorCall)
(call (resolveConstructor {D}) on A args)
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s16 отличается от s15 на следующих аргументах:

- s16(count) = 15;
- s16(ver (** verVar))((** type) is initialized) = 15;
- s16(pre) = (s15(pre) and
  ((!15 (** type) is initialized) =
  (upd (!1 (** type) is initialized) (D) true))).

Применяя правила для элементов (initType . beforeStaticConstructorCall) и (call . on . args .), преобразуем

(p16, s16) в (p17, s17), где p17 имеет вид

```
(block
 (SpecifyFunMemPars (!h 8) (!h 9))
 (cases
  (if ((assert of (!h 8)) != ()) then
   (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
   (modifyD (!* (assert of (!h 8)))) )
  (else (insert body of (!* (!h 8)))) ))
(assertD ProgPost)
```

Состояние s17 совпадает с s16.

Применяя правило для элемента (block .), преобразуем (p17, s17) в (p18, s18), где p18 имеет вид

```
(SpecifyFunMemPars (!h 8) (!h 9))
(cases
 (if ((assert of (!h 8)) != ()) then
  (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
  (modifyD (!* (assert of (!h 8)))) )
 (else (insert body of (!* (!h 8)))) )
(assertD ProgPost)
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
```

Состояние s18 совпадает с s17.

Применяя правило для элемента (SpecifyFunMemPars . .), преобразуем (p18, s18) в (p19, s19), где p19 имеет вид

```
(cases
 (if ((assert of (!h 8)) != ()) then
  (assumeD ((!* assume of (!h 8)) and (!* pre of (!h 8))))
  (modifyD (!* (assert of (!h 8)))) )
 (else (insert body of (!* (!h 8)))) )
(assertD ProgPost)
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
```

Состояние s19 совпадает с s18.

Применяя правило для элемента (cases ...), преобразуем (p19, s19) в (p20, s20), где p20 имеет вид

```
(insert body of (!* (!h 8)))
(assertD ProgPost)
(gotoStop ...)
```

```
(finally (deleteVarScope (!* (varList of (...)))))
```
Состояние s20 совпадает с s19.

Применяя правило для элемента (insert body of .), преобразуем (p20, s20) в (p21, s21), где p21 имеет вид
```
(block (locVarDec {D} (locVarDec-r x = (new {D})))
 (call Test x ()) )
(assertD ProgPost)
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
```
Состояние s21 совпадает с s20.

Применяя правило для элемента (block .), преобразуем (p21, s21) в (p22, s22), где p22 имеет вид
```
(locVarDec {D} (locVarDec-r x =  (new {D})))
(call Test x ())
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
(assertD ProgPost)
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
```
Состояние s22 совпадает с s21.

Применяя правило для элемента (locVarDec ...), преобразуем (p22, s22) в (p23, s23), где p23 имеет вид
```
(addD (locVarDecType))
((locVarDecType) ::=D {D}) (locVarDec-r x =  (new {D}))
(finally (deleteD (locVarDecType)))
(call Test x ())
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
(assertD ProgPost)
(gotoStop ...)
(finally (deleteVarScope (!* (varList of (...)))))
```
Состояние s23 совпадает с s22.

Мы описали несколько шагов вывода условий корректности посредством применения правил логики безопасности. Остальные шаги строятся аналогичным образом. Полученные в результате условия корректности истинны.

8. ЗАКЛЮЧЕНИЕ

В работе описан концептуальный базис улучшенного трехуровневого метода верификации C# программ на основе ПОСП и языка описания ПОСП Atoment. Предложены схемы описания моделей, операционной семантики и логики безопасности для языков C#-light и C#-kernel. Схемы проиллюстрированы на представительном наборе конструкций этих языков. Язык C#-light расширен на небезопасный код, а язык C#-kernel — конструкции языка C#-light и элементы языка Atoment. Представлена новая концепция трансляции C#-light программы в C#-kernel, базирующаяся на реализации метода атрибутных аннотаций через параметрические переменные и нумерацию функциональных членов и типов, используемых в этой программе.

В дальнейшем на основе предложенных концептуальных схем планируется разработать полные операционную семантику и логику безопасности для языков C#-light и C#-kernel. Также предполагается исследовать возможность расширения языка C#-light на неохваченные конструкции языка C#.

СПИСОК ЛИТЕРАТУРЫ

1. Ануреев И.С. Предметно-ориентированные системы переходов: объектная модель и язык // Системная информатика. 2013. N 1. С. 1–34.
2. Атучин М.М., Ануреев И.С. Атрибутные аннотации и их применение в дедуктивной верификации C-программ // Моделирование и анализ информационных систем. 2011. Т. 20. N 4. С. 21–33.
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. 19–30.
4. Непомнящий В.А., Ануреев И.С., Промский А.В. На пути к верификации C программ. Аксиоматическая семантика языка C-kernel // Программирование. 2003. N 6. С. 65–80.
5. Непомнящий В.А., Ануреев И.С., Промский А.В., Дубрановский И.В. На пути к верификации C# программ: трехуровневый подход // Программирование. 2006. Т.32. N 4. С. 4–20.
6. Anureev I.S., Maryasov I.V., Nepomniaschy V.A. Two-level mixed verification method of C-light programs in terms of safety logic // Bulletin of the Novosibirsk Computing Center, Series Computer Science. 2012. Vol. 34. P. 23–42.
7. C# Language Specification. Standard ECMA-334. 2001. http://www.ecma-international.org/.
8. Nepomniashy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V. A three-level approach to C# program verification // Bulletin of the Novosibirsk Computing Center, Series Computer Science. 2004. Vol. 20. P. 61–85.
9. Jacobs B., Kiniry J.L., Warnier M. Java Program Verification Challenges // Proc. FMCO. Lecture Notes in Comput. Sci. 2003. Vol. 2852. P. 202–219.

И.С. Ануреев

КОНЦЕПТУАЛЬНЫЙ БАЗИС ТРЕХУРОВНЕВОГО МЕТОДА ВЕРИФИКАЦИИ C# ПРОГРАММ

Препринт
170

Рукопись поступила в редакцию 6.11.2013
Рецензент А.В. Промский
Редактор Т.М. Бульонкова

Подписано в печать 6.12.2013
Формат бумаги 60×84 1/16 Объем 2,75 уч.-изд.л., 2,5 п.л.
Тираж 60 экз.

Центр оперативной печати "Оригинал 2"
г.Бердск, ул. Островского, 55, оф. 02, тел. (383) 214 45 35