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

И. В. Дубрановский

НА ПУТИ К ВЕРИФИКАЦИИ C#-ПРОГРАММ: АЛГОРИТМЫ ПЕРЕВОДА ИЗ C#-LIGHT В C#-KERNEL

Препринт
140

Новосибирск 2006

В работе дается краткое описание входного языка C#-light системы верификации C#-программ и внутреннего языка C#-kernel, используемого в трехуровневой схеме верификации. Рассматриваются алгоритмы перевода из C#-light в C#-kernel.

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

Igor V. Dubranovsky

TOWARDS C#-PROGRAM VERIFICATION: ALGORITHMS OF TRANSLATION FROM C#-LIGHT INTO C#-KERNEL

Preprint
140

Novosibirsk 2006

This paper presents C#-light, an input language of the C#-program verification system, and C#-kernel, an internal language that is used in the three-level scheme of verification. The algorithms for translation from C#-light into C#-kernel are considered.

1. ВВЕДЕНИЕ

В настоящее время большой интерес представляет верификация программ, написанных на таких широко распространенных объектно-ориентированных языках программирования, как C++, C#, Java. Существенной предпосылкой к тому чтобы язык программирования подходил для верификации, служит компактная обозримая формальная семантика. Наиболее широко используемым подходом к формализации семантики является операционный подход, оперирующий такими понятиями, как система перехода и абстрактная машина. Так, формальная операционная семантика была разработана для Java [5]. Однако процесс верификации в терминах операционной семантики, как правило, намного более трудоемкий по сравнению с верификацией в аксиоматической семантике, основанной на логике Хоара.

Сложности разработки компактной обозримой аксиоматической семантики объектно-ориентированных языков программирования связаны с такими концепциями, как перегрузка, динамическое связывание методов, обработка исключений, статическая инициализация классов. Аксиоматическая семантика была предложена в [7, 10] для различных последовательных подмножеств Java. Однако компактная обозримая аксиоматическая семантика была разработана только для отдельных, более сложных конструкций Java, и в случае широкого последовательного подмножества Java [10] семантика оказалась громоздкой и неудобной для практического использования.

В этой работе детально представлен этап перевода нового трехуровневого подхода [2, 8], совмещающего операционную и аксиоматическую семантики. На первом этапе такого подхода C#-light переводится в промежуточный язык C#-kernel. Во-первых, для того, чтобы элиминировать сложные для аксиоматической семантики конструкции, такие как оператор `try`. Во-вторых, чтобы построить аксиоматическую семантику в более компактной обозримой форме. На втором этапе с помощью обратных правил аксиоматической семантики C#-kernel генерируются ленивые условия корректности. Эти условия корректности являются “ленивыми”, так как могут содержать особые функциональные символы, уточняемые на третьем этапе.

Этап перевода требует аккуратного описания и формального обоснования. Подобная проблема решается в [4] для языка C. Но там перевод значительно менее трудоемкий и носит характер локальных трансформаций

фрагментов программы, в то время как в данной работе часть трансформаций имеет более сложный вид. Это связано с объектной ориентированностью языка C# и с механизмом обработки исключений, которого нет в C.

Данная работа отражает изменения языка аннотаций и модели памяти языков C#-light и C#-kernel, связанные с оптимизацией аксиоматической семантики C#-kernel, и является следующей версией работы [1].

Работа состоит из семи разделов. Язык аннотаций описан в разд. 0. Краткое описание языков C#-light и C#-kernel приводится в разд. 0. Перевод из C#-light в C#-kernel рассматривается в разд. 0, 0 и 0. Результаты и перспективы дальнейшей работы обсуждаются в разд. 0.

Эта работа частично поддержана грантом РФФИ 04-01-00114а и Лаврентьевским молодежным грантом СО РАН № 14.

2. ЯЗЫК АННОТАЦИЙ

В этом разделе мы дадим определение основных конструкций языка аннотаций, используемого для описания предусловий, постусловий, инвариантов, а также для работы с метапеременными абстрактной машины.

2.1. Типы

Допустимые типы языка аннотаций включают базовые типы U, N, T, функции $\mathsf{T}_1 \to \mathsf{T}$ и декартовы произведения $\mathsf{T}_1 \times \mathsf{T}_2$, где

- U — универсум, включающий, по крайней мере, все C# литералы, множество L ячеек, множество Nat натуральных чисел (с нулем) и неопределенное значение ω,
- N — множество C# идентификаторов,
- T — множество абстрактных имен типов.

Экземпляры классов и структур отождествляются с ячейками, в которых они хранятся. Обозначим $Loc(\mathsf{T})$ множество ячеек, сопоставленных C# переменным типа T.

2.2. Выражения

Выражения языка аннотаций определяются индуктивно:

– переменные и константы типа T являются выражениями типа T;

– если s_1, …, s_n — выражения типов T_1, …, T_n соответственно и s — выражение типа $T_1 \times \ldots \times T_n \to T$, то s(s1, …, sn) — выражение типа T;
– если s_1, …, s_n — выражения типов T_1, …, T_n соответственно то <s1, …, sn> — выражение типа $T_1 \times \ldots \times T_n$;
– если x — переменная типа T и s — выражение типа T_1, то λ(x, s) — выражение типа $T \to T_1$, называемое λ-выражением. Оно определяет функцию, значением которой для заданного аргумента a является значение выражения s_1, полученного из s заменой переменной x на a.

Полагаем, что специальная функция upd имеет следующую фиксированную интерпретацию: если s — выражение типа $T \to T'$, и выражения e_1, e_2 имеют типы T и T' соответственно, то терм upd(s, e1, e2) является выражением s' типа $T \to T'$, которое совпадает с s на всех аргументах, кроме, возможно, e_1 и $s'(e_1) = e_2$.

Логические выражения или просто *аннотации* строятся из выражений типа bool с помощью логических связок $\wedge$, $\vee$, $\neg$, $\Rightarrow$ и кванторов $\exists$, $\forall$ обычным образом.

Следует отметить, что операции C#-light записываются в префиксной форме. Например, выражения a[3] и x+3 запишутся как [](a, 3) и +(x, 3) соответственно. Для удобства чтения в некоторых случаях мы будем использовать обычный синтаксис.

2.3. Метапеременные

В описываемом трехуровневом подходе к верификации программные переменные суть константы языка аннотаций, а их связь со значениями моделируется следующими фиксированными переменными языка аннотаций, называемыми метапеременными:

1) L — метапеременная типа $N \to L$, определяющая ячейки памяти для переменных;
2) V — метапеременная типа $L \to U$, определяющая значения, хранимые в ячейках;
3) T — метапеременная типа $N \cup U \to T$, определяющая абстрактные имена типов идентификаторов из N и значений из U;
4) L2 — метапеременная типа $U \times (N \cup Nat) \to L$, определяющая ячейки памяти для элементов полей и массивов;
5) V0 — метапеременная типа U, которая содержит значение последнего вычисленного выражения;

6) `E` — метапеременная типа *U*, которая содержит значение посланного исключения.

Например, тот факт, что переменная `x` в программе равна 3, в нашем подходе записывается в виде `eq(V(L(x)), 3)`, где `x` — уже символическая константа.

Введем сокращения `V(x)`, `L2(x, y)` и `V2(x, y)`, где `x` — имя и `y` — индекс, для выражений `V(L(x))`, `L2(V(x), y)` и `V(L2(x, y))`.

3. КРАТКОЕ ОПИСАНИЕ ЯЗЫКОВ C#-LIGHT И C#-KERNEL

Язык C#-light является подмножеством языка C#. Это язык последовательных программ, т.е. в нем запрещено использование оператора `lock` и классов, связанных с созданием и управлением потоками. В C#-light не поддерживаются атрибуты и деструкторы. Еще одним ограничением является отсутствие оператора использования ресурса `using`. Также не поддерживаются операторы и выражения `checked` и `unchecked`. Следствием этого ограничения является отсутствие в C#-light возможности выбора контекста вычисления выражений (`checked` или `unchecked`). По определению, в C#-light всегда используется проверяемый (checked) контекст вычислений. Наконец, в C#-light запрещено использование небезопасного кода и директив препроцессора.

C#-kernel — это объектно-ориентированный язык, строящийся на базе подмножества `S` языка C#-light, которое определяется следующими ограничениями:

- `S` не содержит пространств имен и using-директив;
- `S` не содержит событий;
- `S` не содержит следующих операторов: операторы перехода `break`, `continue`, `return`, `goto case`, `goto default` и `throw`, оператор `try`, оператор выбора `switch`, операторы циклов, операторы-объявления;
- `S` содержит только те операторы `if`, в которых имеется ветвь `else`, а условное выражение является переменной типа boolean;
- вызов статических функциональных членов может производиться только в тех местах программ, в которых выполнена статическая инициализация соответствующих классов или структур;
- все метки, имена локальных переменных и имена локальных констант должны быть уникальны в программах;

- множества меток, имен типов, имен локальных переменных и имен локальных констант не пересекаются.

Подмножество `S` расширяется метаинструкциями, а на операторы-выражения и объявления классов и структур накладываются ограничения.

3.1. Метаинструкции

Метаинструкции используются для работы с метапеременными. В C#-kernel существует пять метаинструкций:

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

3.2. Операторы-выражения

Оператор-выражение в C#-kernel — это *нормализованное выражение* или метаинструкция, за которой следует точка с запятой. *Нормализованные выражения* определяются с помощью следующих ограничений, накладываемых на выражения языка C#-light:

- нормализованное выражение имеет вид `x.y(`z_1`, …, `z_n`)` или `y(`z_1`, …, `z_n`)`, где `x` — имя переменной или типа, `y` — имя метода, делегата, конструктора или операции, в качестве z_1, …, z_n допускаются имена переменных (возможно с модификаторами `ref` и `out`), литералы и метапеременные `E` и `V0`;
- в нормализованных выражениях функциональные члены могут быть вызваны только в нормальной форме [6];

- в нормализованных выражениях логические операции `||` и `&&`, условная операция `?:`, операция `new` и все операции присваивания запрещены.

3.3. Декларации классов и структур

В объявлениях полей и констант в классах и структурах запрещено использование инициализаторов. Вместо этого, для каждой декларации класса и структуры резервируются два инициализирующих метода `SFI` и `IFI`, в которых выполняется инициализация статических и нестатических полей соответственно. Инициализация констант и статических полей выполняется в методе

```
public static void SFI() {…}
```

где сначала каждое статическое поле и константа инициализируется значением по умолчанию для типа этого поля или константы, а затем следует обязательная инициализация констант и дополнительная инициализация полей. Инициализация нестатических полей класса `C` выполняется в методе

```
public void IFI_C() {…}
```

Эти методы расширяют контекст прямого присваивания в `readonly`-поле.

4. ПЕРЕВОД ИЗ C#-LIGHT В C#-KERNEL

Перевод из C#-light в C#-kernel имеет вид последовательного применения трансформаций. Правила перевода ряда конструкций определяются наборами трансформаций, недетерминировано применяемых к программе. Остальные правила имеют императивный вид и при их определении трансформации используются в качестве элементарных действий. Каждая трансформация указывает фрагмент программы, который необходимо преобразовать, как преобразовывается этот фрагмент и условие применимости трансформации.

Все алгоритмы для удобства разделены на два класса: нормализация и элиминация.

Введем несколько соглашений и обозначений. Для обозначения выражений произвольного вида используются символы `e`, `f`, `g`, e_1, f_1, g_1, и т.д. Для обозначения boolean-выражений используется символ `b`. Пустая строка обозначается через ε.

Переменные обозначаются символами `x`, `y`, `z`, `a`, x_1, y_1, z_1, a_1, x_2, y_2 и т.д. Символ `S` обычно используется для обозначения операторов. Символами `A`,

B, C, D обычно обозначаются произвольные фрагменты кода. Если e — выражение, то через T_e обозначается тип выражения e.

Из множества имен функций и предикатов языка аннотаций мы выделяем следующие имена, каждое из которых имеет фиксированную интерпретацию.

1. γ — функция округления. Если T_1, T_2 — числовые типы, а выражение e имеет тип T_1, то $\gamma(T_1, T_2, e)$ имеет тип T_2 и возвращает округленное по правилам округления числовых типов [6] значение выражения e. Отметим, что в случаях, предусмотренных спецификацией [6], γ имеет побочный эффект в виде записи объекта исключения в ячейку в метапеременную E. Кроме того, если существует неявное числовое преобразование из T_1 в T_2, то $\gamma(T_1, T_2, e) = e$.
2. add — функция добавления метода в делегат. Если s — выражение типа $Nat \to U \times N$, а m — выражение типа $U \times N$, то add(s, m) является выражением типа $Nat \to U \times N$ и add(s, m) = upd(s, n, m), где $n \in Nat$, n = max(Dom(s)) + 1; иначе, add(s, m) = ω.
3. can_cast — предикат существования неявного преобразования ссылочных типов. Если T_1, T_2 — ссылочные типы, такие что существует неявное ссылочное преобразование из T_1 в T_2 [6], то выражение can_cast(T_1, T_2) истинно; иначе это выражение ложно.

В дополнение к метапеременным языка аннотаций, определим метапеременные L0 и UT.

1. L0 — метапеременная типа L, которая содержит ячейку, хранящую значение последнего вычисленного выражения, если оно классифицируется как переменная; иначе значение метапеременной L0 не определено.
2. UT — метапеременная типа $T \to T$, определяющая для каждого типа перечисления его базовый тип.

5. НОРМАЛИЗАЦИЯ

В этом разделе мы детально рассмотрим алгоритмы нормализации. Они подготавливают конструкции преобразуемого языка к применению алгоритмов элиминации. Некоторые из них приводят конструкции к форме, которая допустима в C#-kernel.

5.1. Декларация класса и структуры

Для каждого свойства, индексатора и события с пользовательскими функциями доступа (accessors) в объявление класса или структуры добавляются соответствующие методы, по одному на каждую из функций доступа. Имена и сигнатуры методов строятся по правилам для зарезервированных имен членов (reserved member names) [6]. Тела методов дублируют тела соответствующих функций доступа. Добавление таких методов в декларации классов и структур позволяет без труда преобразовать доступ к свойствам, индексаторам и событиям в функциональную форму.

Объявления событий без пользовательских функций доступа преобразуются в объявления делегатов путем удаления ключевого слова `event`. Это законно, так как такие события имеют такую же семантику, как делегаты.

Правило NMEMBER1. Для свойства `P` вида

property-modifiers$_{opt}$ *type* `P {`*accessor-declarations*`}`

объявленного в классе или структуре `T`, если свойство имеет *get-accessor* вида `get` *accessor-body*, то в `T` добавляется метод

property-modifiers$_{opt}$ *type* `get_P()` *accessor-body*

Правило NMEMBER2. Для свойства `P` вида

property-modifiers$_{opt}$ *type* `P {`*accessor-declarations*`}`

объявленного в классе или структуре `T`, если свойство имеет *set-accessor* вида `set` *accessor-body*, то в `T` добавляется метод

property-modifiers$_{opt}$ `void set_P(`*type* `value)` *accessor-body*

Правило NMEMBER3. Для индексатора вида

indexer-modifiers$_{opt}$ *type* `this[`*formal-parameter-list*`]`
`{`*accessor-declarations*`}`

объявленного в классе или структуре `T`, если индексатор имеет *get-accessor* вида `get` *accessor-body*, то в `T` добавляется метод

indexer-modifiers$_{opt}$ *type* `get_Item(`*formal-parameter-list*`)`
accessor-body

Правило NMEMBER4. Для индексатора вида

indexer-modifiers$_{opt}$ *type* `this[`*formal-parameter-list*`]`
`{`*accessor-declarations*`}`

объявленного в классе или структуре `T`, если индексатор имеет *set-accessor* вида `set` *accessor-body*, то в `T` добавляется метод

indexer-modifiers$_{opt}$ `void set_Item(`
formal-parameter-list`,`
type `value)`
accessor-body

Правило NMEMBER5. Для события `Evt` вида

event-modifiers$_{opt}$ `event` *type* `Evt {`*event-accessor-declarations*`}`

объявленного в классе или структуре `T`, если событие имеет *add-accessor-declaration* вида `add` *block*, то в `T` добавляется метод

event-modifiers$_{opt}$ `void add_Evt(`*type* `value)` *block*

Правило NMEMBER6. Для события `Evt` вида

event-modifiers$_{opt}$ `event` *type* `Evt {`*event-accessor-declarations*`}`

объявленного в классе или структуре `T`, если событие имеет *remove-accessor-declaration* вида `remove` *block*, то в `T` добавляется метод

event-modifiers$_{opt}$ `void remove_Evt(`*type* `value)` *block*

Правило NMEMBER7. Фрагмент вида

event-modifiers$_{opt}$ `event` *type variable-declarator*`,` *variable-declarators*`;`

заменяется фрагментом

event-modifiers$_{opt}$ `event` *type variable-declarator*`;`
event-modifiers$_{opt}$ `event` *type variable-declarators*`;`

Правило NMEMBER8. Фрагмент вида

event-modifiers$_{opt}$ `event` *type variable-declarator*`;`

заменяется фрагментом

event-modifiers$_{opt}$ *type variable-declarator*`;`

Правило NSTATINIT. В объявление класса или структуры добавляется метод

```
public static void SFI() {
    constant-default-initialization
    static-field-default-initialization
```

constant-initialization
static-field-initialization }

если его еще не было.

Правило NINSTINIT. В объявление класса `C` добавляется метод

```
public void IFI_C() {
```
instance-field-default-initialization
instance-field-initialization }

если его еще не было.

Таким образом, с помощью этих преобразований декларации классов и структур приводятся к форме, допустимой в C#-kernel. Рассмотрим теперь нормализацию декларации локальной переменной и константы.

5.2. Декларация локальной переменной и константы

Операторы-объявления локальной переменной и константы нормализуются следующим образом. Объявления списков переменных или констант преобразуются в список объявлений. Объявления переменных с инициализаторами расклеиваются на объявления переменных без инициализаторов и операторы-выражения, инициализирующие эти переменные.

Правило NDECL1. Фрагмент вида

type variable-declarator, *variable-declarators*;

заменяется фрагментом

type variable-declarator;
type variable-declarators;

Правило NDECL2. Фрагмент вида

`const` *type constant-declarator*, *constant-declarators*;

заменяется фрагментом

`const` *type constant-declarator*;
`const` *type constant-declarators*;

Правило NDECL3. Фрагмент вида

type identifier = *expression*;

где *expression* — выражение, не являющееся инициализатором массива, заменяется фрагментом

type identifier;
identifier = *expression*;

Правило NDECL4. Фрагмент вида

array-type identifier = *array-initializer*;

заменяется фрагментом

array-type identifier = new *array-type array-initializer*;

Итак, эти преобразования позволяют нам в дальнейшем применять алгоритмы элиминации и удалять операторы-объявления локальной переменной и константы, заменяя их набором метаинструкций.

5.3. Функциональные члены с переменным числом аргументов

Неформально, в выражениях вызовы функциональных членов в расширенной форме заменяются вызовами этих членов в нормальной форме.[1] Такая нормализация упрощает правила аксиоматической семантики C#-kernel для вызова функциональных членов.

Правило NFVNA1. Фрагмент вида

$P(A_1, A_2, \ldots A_n)$

являющийся выражением вызова (т.е. первичное выражение P классифицируется как группа методов или значение типа делегат), в котором P применимо в расширенной форме [6] к списку аргументов A_1, A_2, …, A_n, заменяется фрагментом

$P(A_1, A_2, \ldots A_k, \text{new } T_A \{A_{k+1}, \ldots A_n\})$

где k — число фиксированных параметров в объявлении функционального члена или делегата, заданного выражением P, T_A — тип параметра-массива.

Правило NFVNA2. Фрагмент вида

$\text{new } T(A_1, A_2, \ldots A_n)$

[1] Определение понятий *расширенная форма* и *нормальная форма* вызова функционального члена можно найти в спецификации C# [6].

являющийся выражением создания объекта, в котором выбранный нестатический конструктор применим в расширенной форме [6] к списку аргументов A_1, A_2, …, A_n, заменяется фрагментом

```
new T(A1, A2, … Ak, new TA {Ak+1, … An})
```

где k — число фиксированных параметров в объявлении выбранного нестатического конструктора, T_A — тип параметра-массива.

Правило NFVNA3. Фрагмент вида

```
e[e1, e2, … en]
```

являющийся выражением доступа к индексатору, в котором выбранный индексатор применим в расширенной форме [6] к списку аргументов e_1, e_2, …, e_n, заменяется фрагментом

```
e[e1, e2, … ek, new TA {ek+1, … en}]
```

где k — число фиксированных параметров в объявлении выбранного индексатора, T_A — тип параметра-массива.

Одним из наиболее сложных преобразований является нормализация операторов-выражений.

5.4. Оператор-выражение

Для нормализации операторов-выражений и элиминации операторов-объявлений (разд. 0) языка C#-light используется преобразование `Norm`. Это преобразование определяется с помощью правил вывода выражений C#-light. Для каждого типа выражения `Norm` указывает, каким образом данное выражение преобразуется в набор нормализованных операторов-выражений. `Norm` является рекурсивным преобразованием, так как грамматика выражений C# — рекурсивна.

Преобразование `Norm` задано в виде набора правил. Применение практически любого правила создает новые локальные переменные. Будем предполагать, что такие переменные являются уникальными в контексте всей программы.

Правило NES. Фрагмент вида

statement-expression **;**

заменяется фрагментом

```
Norm[statement-expression];
```

Определим `Norm` для каждого типа выражений.

5.4.1. Вспомогательные преобразования

При определении `Norm` используются вспомогательные преобразования.

`Mod`. Пусть `g` — выражение. Тогда

```
Mod[g] ::= ε
Mod[ref g] ::= ref
Mod[out g] ::= out
```

Здесь и далее через ε обозначается пустая строка.

`Arg`. Пусть `e` — выражение, `Q` — его тип. Тогда

```
Arg[e, Q] ::= (Q) e
Arg[ref e, Q] ::= e
Arg[out e, Q] ::= e
```

`Var`. Пусть `e` — выражение. Если `e` классифицируется как переменная, то `Var[e] ::= true`. Иначе, `Var[e] ::= false`.

`GetVal`. Пусть `x` — локальная переменная. Тогда

```
GetVal[x] ::= V := upd(V, L(x), V0)
```

`GetLoc`. Пусть `x` — локальная переменная. Тогда

```
GetLoc[x] ::= L := upd(L, x, L0)
```

`GetLocOrVal`. Пусть `x` — локальная переменная. Тогда

```
GetLocOrVal[x, true] ::= GetLoc[x]
GetLocOrVal[x, false] ::= GetVal[x]
```

`PutVal`. Пусть `x` — локальная переменная или `this`. Тогда

```
PutVal[x] ::= V0 := V(x)
```

`WriteMember`. Пусть `e` — тип или переменная типа, в котором объявлено поле `x`. Тогда

```
WriteMember[e, x] ::= L0 := L2(e, x); V0 := V2(e, x)
```

`DefaultValue`. Это рекурсивное преобразование используется для инициализации объекта произвольного типа значением по умолчанию.

Пусть e — выражение, классифицируемое как переменная. Если e имеет тип sbyte, byte, short, ushort, int, uint, long, ulong или тип перечисления, то

```
DefaultValue[e] ::= Norm[e = 0]
```

Если e имеет тип char, то

```
DefaultValue[e] ::= Norm[e = '\x0000']
```

Если e имеет тип float, то

```
DefaultValue[e] ::= Norm[e = 0.0f]
```

Если e имеет тип double, то

```
DefaultValue[e] ::= Norm[e = 0.0d]
```

Если e имеет тип decimal, то

```
DefaultValue[e] ::= Norm[e = 0.0m]
```

Если e имеет тип bool, то

```
DefaultValue[e] ::= Norm[e = false]
```

Если e имеет ссылочный тип, то

```
DefaultValue[e] ::= Norm[e = null]
```

Если e имеет тип структуры с полями a_1, a_2, … a_n, то

```
DefaultValue[e] ::=
    DefaultValue[e.a1]; … DefaultValue[e.an]
```

MGType. Пусть e — выражение, классифицируемое как группа методов, которое порождает статический метод, объявленный в типе Q. Тогда

```
MGType[e] ::= Q
```

MGMethod. Пусть e — выражение, классифицируемое как группа методов, которое порождает статический или нестатический метод M. Тогда

```
MGMethod[e] ::= M
```

NormArg. Пусть e — выражение, Q — тип. Тогда

```
NormArg[e, Q] ::= Norm[(Q) e]
```

Пусть e — выражение, классифицируемое как переменная, T_e — тип выражения e.

```
NormArg[ref e, Q] ::= Norm[e]
NormArg[out e, Q] ::= Norm[e]
```

CopyValue. Определим сначала вспомогательное преобразование Rec. Пусть x — переменная типа Q, где Q — структура, рекурсивно содержащая вложенные структуры. Тогда

$$\texttt{Rec[x.}a_1\texttt{.}\ldots\texttt{.}a_{n-1}\texttt{.}a_n\texttt{]} ::= \texttt{V2(Rec[x.}a_1\texttt{.}\ldots\texttt{.}a_{n-1}\texttt{],}a_n\texttt{)},\ n>0$$
```
Rec[x] ::= V(x)
```

Теперь определим преобразование CopyValue. Пусть x, y — переменные ссылочного, перечислимого или предопределенного типа. Тогда

```
CopyValue[x, y] ::= V := upd(V, L(x), V(y))
```

Допустим $\texttt{x.}a_1\texttt{.}\ldots\texttt{.}a_{n-1}\texttt{.}a_n$ — поле ссылочного, перечислимого или предопределенного типа. Тогда

$$\texttt{CopyValue[x.}a_1\texttt{.}\ldots\texttt{.}a_{n-1}\texttt{.}a_n\texttt{, y.}a_1\texttt{.}\ldots\texttt{.}a_{n-1}\texttt{.}a_n\texttt{] ::=}$$
```
    V := upd(V, L2(Rec[x.a1.….an-1], an),
                V2(Rec[y.a1.….an-1], an))
```

Пусть x, y — переменные типа Q, где Q — структура с полями f_1, …, f_n. Тогда

$$\texttt{CopyValue[x, y] ::=}$$
$$\texttt{CopyValue[x.}f_1\texttt{, y.}f_1\texttt{]; … CopyValue[x.}f_n\texttt{, y.}f_n\texttt{]}$$

5.4.2. Первичные выражения

Литерал. Пусть I — литерал. Тогда

```
Norm[I] ::= V0 := I
```

Простое имя. Пусть I — идентификатор. Если I является локальной переменной или параметром, то

```
Norm[I] ::= L0 := L(I); V0 := V(I)
```

Если I является нестатическим членом (кроме нестатического метода), то

```
Norm[I] ::= this.I
```

Если I является статическим членом (кроме статического метода), объявленным в типе T, то

```
Norm[I] ::= Norm[T.I]
```

Выражение в скобках. Пусть e — выражение. Тогда

```
Norm[(e)] ::= Norm[e]
```

Доступ к члену. Пусть e — предопределенный тип или первичное выражение, классифицируемое как тип. Если I — статическое свойство, то

```
Norm[e.I] ::= Init(e); e.get_I()
```

Иначе:

```
Norm[e.I] ::= WriteMember[e, I]
```

Пусть выражение e классифицируется как доступ к свойству, доступ к индексатору, переменная или значение, T_e — тип выражения e. Если I — свойство, то

```
Norm[e.I] ::= Norm[Te x]; Norm[e]; GetVal[x]; x.get_I()
```

Иначе:

```
Norm[e.I] ::= Norm[Te x]; Norm[e]; GetVal[x];
              WriteMember[x, I]
```

Выражение вызова. Пусть первичное выражение P в выражении вызова является группой методов, состоящей из статического метода, а T_1, …, T_n — типы формальных параметров метода. Тогда

```
Norm[P(A1, … An)] ::=
    Norm[T1 x1];
    NormArg[A1, T1]; GetLocOrVal[x1, Var[Arg[A1, T1]]];
    …
    Norm[Tn xn];
    NormArg[An, Tn]; GetLocOrVal[xn, Var[Arg[An, Tn]]];
    P(Mod[A1] x1, … , Mod[An] xn)

Norm[base(A1, … An)] ::=
    Norm[T1 x1];
    NormArg[A1, T1]; GetLocOrVal[x1, Var[Arg[A1, T1]]];
    …
    Norm[Tn xn];
```

```
    NormArg[A_n, T_n]; GetLocOrVal[x_n, Var[Arg[A_n, T_n]]];
    base(Mod[A_1] x_1, … , Mod[A_n] x_n)
```

Пусть первичное выражение `P` в выражении вызова является группой методов, состоящей из нестатического метода и нестатического выражения `e`, имеющего тип T_e, а T_1, …, T_n — типы формальных параметров нестатического метода. Тогда

```
Norm[P(A_1, … A_n)] ::=
    Norm[T_e x_0]; Norm[e];  GetLocOrVal[x_0, Var[e]];
    Norm[T_1 x_1];
    NormArg[A_1, T_1]; GetLocOrVal[x_1, Var[Arg[A_1, T_1]]];
    …
    Norm[T_n x_n];
    NormArg[A_n, T_n]; GetLocOrVal[x_n, Var[Arg[A_n, T_n]]];
    x_0.MGMethod[P](Mod[A_1] x_1, … , Mod[A_n] x_n)
```

Пусть первичное выражение `P` в выражении вызова имеет тип делегата, а T_1, …, T_n — типы формальных параметров делегата. Тогда

```
Norm[P(A_1, … A_n)] ::=
    Norm[T_P x];       Norm[P];     GetVal[x];
    Norm[T_1 x_1];
    NormArg[A_1, T_1]; GetLocOrVal[x_1, Var[Arg[A_1, T_1]]];
    …
    Norm[T_n x_n];
    NormArg[A_n, T_n]; GetLocOrVal[x_n, Var[Arg[A_n, T_n]]];
    x(Mod[A_1] x_1, … , Mod[A_n] x_n)
```

Доступ к элементу. Пусть `e` — выражение, имеющее тип массива, T_1, …, $T_n \in$ {`int`, `uint`, `long`, `ulong`} — типы, к которым неявно могут быть преобразованы типы выражений e_1, e_2, …, e_n. Тогда

```
Norm[e[e_1, … e_n]] ::=
    Norm[T_e x];    Norm[e];           GetVal[x];
    Norm[T_1 x_1];  Norm[(T_1) e_1];   GetVal[x_1];
    …
    Norm[T_n x_n];  Norm[(T_n) e_n];   GetVal[x_n];
    WriteMember[x, <x_1, … x_n>]
```

Пусть `e` — выражение, имеющее тип класса, структуры или интерфейса, который реализует применимый индексатор, а T_1, …, T_n — типы формальных параметров индексатора. Тогда

```
Norm[e[e1, … en]] ::=
    Norm[Te x];    Norm[e];          GetVal[x];
    Norm[T1 x1];   Norm[(T1) e1];    GetVal[x1];
    …
    Norm[Tn xn];   Norm[(Tn) en];    GetVal[xn];
    x.get_Item(x1, … xn)
```

Base-доступ. Пусть `I` — идентификатор, e_1, …, e_n — выражения. Тогда

```
Norm[base[e1, … en]] ::=
    Norm[T1 x1];   Norm[(T1) e1];    GetVal[x1];
    …
    Norm[Tn xn];   Norm[(Tn) en];    GetVal[xn];
    base.get_Item(x1, … xn)
```

Если `I` — свойство, то

```
Norm[base.I] ::= base.get_I()
```

Иначе:

```
Norm[base.I] ::= base.I
```

Операции постфиксного инкремента и декремента. Пусть *op* ∈ {++, --}, `e` — выражение, классифицируемое как переменная. Тогда

```
Norm[e op] ::=
    Norm[Te x];    Norm[e];     GetLoc[x];
    Norm[Te y];    Te.op(x);    GetVal[y];
    PutVal[x];     V := upd(V, L(x), V(y))
```

Пусть `e` — выражение, классифицируемое как доступ к свойству `P`. Если доступ к `P` осуществляется через нестатическое выражение e_P, имеющее тип T_P (т.е. `P` — нестатическое свойство), то

```
Norm[e op] ::=
    Norm[TP z];    Norm[eP];    GetVal[z];
    Norm[Te x];    z.get_P();   GetVal[x];
    Norm[Te y];    Te.op(x);    GetVal[y];
    z.set_P(y);    PutVal[x]
```

Иначе (`P` — статическое свойство, объявленное в типе T_P):

```
Norm[e op] ::=
    Norm[Te x];    TP.get_P();  GetVal[x];
```

```
Norm[Te y];    Te.op(x);     GetVal[y];
TP.set_P(y);   PutVal[x]
```

Пусть e — выражение, классифицируемое как доступ к индексатору через нестатическое выражение e_I, имеющее тип T_I, A_1, …, A_n — аргументы доступа к индексатору, а T_1, …, T_n — типы его формальных параметров. Тогда

```
Norm[e op] ::=
    Norm[TI z];    Norm[eI];          GetVal[z];
    Norm[T1 a1];   Norm[(T1) A1];     GetVal[a1];
    …
    Norm[Tn an];   Norm[(Tn) An];     GetVal[an];
    Norm[Te x];    z.get_Item(a1, … an); GetVal[x];
    Norm[Te y];    Te.op(x);          GetVal[y];
    z.set_Item(a1, … an, y);          PutVal[x]
```

Операция new. Определим Norm на выражении создания объекта. Пусть A_1, …, A_n — аргументы выражения создания объекта, а T_1, …, T_n — типы формальных параметров конструктора, применимого к аргументам A_1, …, A_n.

```
Norm[new Q(A1, … An)] ::=
    Norm[T1 x1];
    NormArg[A1, T1]; GetLocOrVal[x1, Var[Arg[A1, T1]]];
    …
    Norm[Tn xn];
    NormArg[An, Tn]; GetLocOrVal[xn, Var[Arg[An, Tn]]];
    Init(Q);
    new_instance(); L := upd(L, x, V0);
    T := upd(T, L(x), Loc(Q));
    new_instance(); V := upd(V, L(x), V0);
    T := upd(T, V(x), Q);
    x.IFI_Q(); x.Q(Mod[A1] x1, … , Mod[An] xn);
    PutVal[x]
```

Теперь определим Norm для выражения создания массива. Пусть T_1, …, T_n $\in$ {int, uint, long, ulong} — типы, к которым неявно могут быть преобразованы типы выражений e_1, …, e_n, и RS_{opt} — необязательные спецификаторы размерности массива. Тогда

```
Norm[new Q[e1, … en] RSopt] ::=
    Norm[T1 x1];   Norm[(T1) e1];     GetVal[x1];
    …
    Norm[Tn xn];   Norm[(Tn) en];     GetVal[xn];
    Norm[bool b];
```

```
Norm[(x1 < 0) || … || (xn < 0)]; GetVal[b];
if (b) { throw new System.OverflowException(); }
Init(Q[x1, … xn] RSopt);
new_instance(); L := upd(L, x, V0);
T := upd(T, L(x), Loc(Q[x1, … xn] RSopt));
new_instance(); V := upd(V, L(x), V0);
T := upd(T, V(x), Q[x1, … xn] RSopt);
x.IFI_Array(); x.Array();

for (int i1 = 0; i1 < x1; i1++)
    …
    for (int in = 0; in < xn; in++) {
        DefaultValue[x[i1, … in]];
    }

PutVal[x]
```

Пусть *arr-init*|$A_{j1, \ldots, jN}|1 \le j1 \le y1, \ldots, 1 \le jN \le yN$ — инициализатор N-мерного массива. Тогда

```
Norm[new Q[e1, … eN] RSopt arr-init|Aj1, …, jN|] ::=
    Norm[T1 x1];   Norm[(T1) e1];      GetVal[x1];
    …
    Norm[TN xN];   Norm[(TN) eN];      GetVal[xN];
    Norm[bool b];
    Norm[(x1 < 0) || … || (xN < 0)]; GetVal[b];
    if (b) { throw new System.OverflowException(); }
    Init(Q[x1, … xN] RSopt);
    new_instance(); L := upd(L, x, V0);
    T := upd(T, L(x), Loc(Q[x1, … xN] RSopt));
    new_instance(); V := upd(V, L(x), V0);
    T := upd(T, V(x), Q[x1, … xN] RSopt);
    x.IFI_Array(); x.Array();

    for (int i1 = 0; i1 < x1; i1++)
        …
        for (int iN = 0; iN < xN; iN++) {
            DefaultValue[x[i1, … iN]];
        }

    Norm[Q RSopt  a];
    Norm[A1, 1, …, 1]; GetVal[a];
```

```
    V := upd(V, L2(V(x), <0, 0, … 0>), V(a));
    Norm[A1, 1, …, 2]; GetVal[a];
    V := upd(V, L2(V(x), <0, 0, … 1>), V(a));
    …
    Norm[Ay1, y2, …, yN]; GetVal[a];
    V := upd(V, L2(V(x), <y1 - 1, y2 - 1, … yN - 1>), V(a));
    PutVal[x]
```

```
Norm[new Q [,, …] RSopt arr-init|Aj1, …, jN| 1 ≤ j1 ≤ y1, …, 1 ≤ jN ≤ yN] ::=
    Norm[new Q[y1, … yN] RSopt arr-init|Aj1, …, jN|]
```

Наконец, рассмотрим выражение создания делегата. Пусть аргументом такого выражения является группа методов E, состоящая из статического метода. Тогда

```
Norm[new D(E))] ::=
    Init(D);
    new_instance(); L := upd(L, x, V0);
    T := upd(T, L(x), Loc(D));
    new_instance(); V := upd(V, L(x), V0);
    T := upd(T, V(x), D);
    x.IFI_D();
    V := upd(V,
             V(x),
             add(V(V(x)),
                 <V(MGType[E]), MGMethod[E]>));
    PutVal[x]
```

Если аргументом выражения создания делегата является группа методов, состоящая из нестатического метода, а ее нестатическое выражение $e_м$ имеет ссылочный тип $T_м$, то

```
Norm[new D(e))] ::=
    Norm[Tм o];       Norm[eм];     GetVal[o];
    Norm[bool b];
    V := upd(V, L(b), eq(V(o), null));
    if (b) { throw new System.NullReferenceException(); }
    Init(D);
    new_instance(); L := upd(L, x, V0);
    T := upd(T, L(x), Loc(D));
    new_instance(); V := upd(V, L(x), V0);
    T := upd(T, V(x), D);
    x.IFI_D();
```

```
    V := upd(V,
             V(x),
             add(V(V(x))),
                 <V(o), MGMethod[e]>));
    PutVal[x]
```

Если аргументом выражения создания делегата является группа методов, состоящая из нестатического метода, а ее нестатическое выражение $e_м$ имеет value-тип $T_м$, то

```
Norm[new D(e))] ::=
    Norm[Tм z];      Norm[eм];            GetVal[z];
    Norm[object o];  Norm[(object) z];    GetVal[o];
    Init(D);
    new_instance(); L := upd(L, x, V0);
    T := upd(T, L(x), Loc(D));
    new_instance(); V := upd(V, L(x), V0);
    T := upd(T, V(x), D);
    x.IFI_D();
    V := upd(V,
             V(x),
             add(V(V(x))),
                 <V(o), MGMethod[e]>));
    PutVal[x]
```

Если аргументом выражения создания делегата является значение, имеющее тип делегата, то

```
Norm[new D(e)] ::=
    Norm[D z];    Norm[e];        GetVal[z];
    Norm[bool b];
    V := upd(V, L(b), eq(V(z), null));
    if (b) { throw new System.NullReferenceException(); }
    Init(D);
    new_instance(); L := upd(L, x, V0);
    T := upd(T, L(x), Loc(D));
    new_instance(); V := upd(V, L(x), V0);
    T := upd(T, V(x), D);
    x.IFI(D); x.D(z); PutVal[x]
```

Операция typeof. Пусть Q — тип. Тогда

```
Norm[typeof(Q)] ::= typeof(Q)
Norm[typeof(void)] ::= typeof(void)
```

5.4.3. Унарные операции

Операции +, -, ! и ~. Пусть *op* ∈ {+, -, !, ~}, e — выражение, отличное от локальной переменной. Тогда

```
Norm[op e] ::= Norm[Te x]; Norm[e]; GetVal[x]; Norm[op x]
```

Пусть x — локальная переменная типа T_x, *op* ∈ {+, -, !, ~} перегружена в типе T_x и имеет аргумент типа T_{arg}. Тогда

```
Norm[op x] ::= Norm[Targ y]; Norm[(Targ) x]; GetVal[y];
               Tx.op(y)
```

Операции префиксного инкремента и декремента. Пусть *op* ∈ {++, --}, e — выражение, классифицируемое как переменная. Тогда

```
Norm[op e] ::=
    Norm[Te x];    Norm[e];     GetLoc[x];
    Norm[Te y];    Te.op(x);    GetVal[y];
    V := upd(V, L(x), V(y));    PutVal[y]
```

Пусть e — выражение, классифицируемое как доступ к свойству P. Если доступ к P осуществляется через нестатическое выражение e_P, имеющее тип T_P (т.е. P — нестатическое свойство), то

```
Norm[op e] ::=
    Norm[TP z];    Norm[eP];    GetLocOrVal[z, Var[eP]];
    Norm[Te x];    z.get_P();   GetVal[x];
    Norm[Te y];    Te.op(x);    GetVal[y];
    z.set_P(y);    PutVal[y]
```

Иначе (P — статическое свойство, объявленное в типе T_P):

```
Norm[op e] ::=
    Norm[Te x];    TP.get_P();  GetVal[x];
    Norm[Te y];    Te.op(x);    GetVal[y];
    TP.set_P(y);   PutVal[y]
```

Пусть e — выражение, классифицируемое как доступ к индексатору через нестатическое выражение e_I, имеющее тип T_I, A_1, …, A_n — аргументы доступа к индексатору, а T_1, …, T_n — типы его формальных параметров. Тогда

```
Norm[op e] ::=
    Norm[TI z];    Norm[eI];       GetLocOrVal[z, Var[eI]];
    Norm[T1 a1];   Norm[(T1) A1];  GetVal[a1];
    …
```

```
Norm[T_n a_n];    Norm[(T_n) A_n];   GetVal[a_n];
Norm[T_e x];      T_I.get_Item(a_1, a_2, … a_n);  GetVal[x];
Norm[T_e y];      T_e.op(x);         GetVal[y];
T_I.set_Item(a_1, … a_n, y);         PutVal[y]
```

5.4.4. Операция приведения

Пусть `Q` — тип, `e` — выражение, отличное от локальной переменной. Тогда

```
Norm[(Q) e] ::= Norm[T_e x]; Norm[e]; GetVal[x];
                Norm[(Q) x]
```

Пусть `x` — локальная переменная типа `S`, а `Q` — тип, к которому следует привести `x`. Рассмотрим все случаи явных и неявных операций приведения (преобразований).

Единичные преобразования. Пусть `S = Q`. Тогда `Norm[(Q) x] ::= PutVal[x]`.

Неявные числовые преобразования. Согласно спецификации C#, данные преобразования применяются в следующих случаях:

- `S = sbyte`, `Q` ∈ {`short`, `int`, `long`, `float`, `double`, `decimal`};
- `S = byte`, `Q` ∈ {`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`};
- `S = short`, `Q` ∈ {`int`, `long`, `float`, `double`, `decimal`};
- `S = ushort`, `Q` ∈ {`int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`};
- `S = int`, `Q` ∈ {`long`, `float`, `double`, `decimal`};
- `S = uint`, `Q` ∈ {`long`, `ulong`, `float`, `double`, `decimal`};
- `S = long`, `Q` ∈ {`float`, `double`, `decimal`};
- `S = ulong`, `Q` ∈ {`float`, `double`, `decimal`};
- `S = char`, `Q` ∈ {`ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`};
- `S = float`, `Q` ∈ {`double`, `decimal`}.

В этих случаях `Norm[(Q) x] ::= PutVal[x]`.

Неявные преобразования типов перечисления. Пусть `Q` — произвольный тип перечисления. Тогда `Norm[(Q) 0] ::= V0 := 0`.

Неявные преобразования ссылочных типов. Согласно спецификации C#, данные преобразования применяются в следующих случаях:

- S — ссылочный тип, Q = object;
- S — класс, Q — класс, S — наследник Q;
- S — класс, Q — интерфейс, S реализует Q;
- S — интерфейс, Q — интерфейс, S — наследник Q;
- S — массив элементов типа S_E, Q — массив элементов типа Q_E, и при этом:
  - S и Q имеют одинаковую размерность,
  - S_E и Q_E являются ссылочными типами,
  - существует неявное преобразование ссылочных типов из S_E в Q_E;
- S — массив, Q = System.Array;
- S — делегат, Q = System.Delegate;
- S — массив или делегат, Q = System.ICloneable;
- S — тип литерала null (т.е. x есть слово null), Q — ссылочный тип.

В этих случаях Norm[(Q) x] ::= PutVal[x].

Boxing-преобразования. Пусть S — value-тип, а Q — тип object или интерфейс, реализуемый value-типом S. Тогда:

```
Norm[(Q) x] ::= new_instance(); L := upd(L, y, V0);
    T := upd(T, L(y), Loc(S));
    CopyValue[y, x]; V0 := V(y)
```

Неявные преобразования константных выражений. Согласно спецификации C#, данные преобразования применяются в следующих случаях:

- x — локальная константа типа int, Q ∈ {sbyte, byte, short, ushort, uint, ulong}, и при этом значение константы x лежит в области значений типа Q;
- x — локальная константа типа long, Q = ulong, и при этом значение константы x неотрицательно.

В этих случаях Norm[(Q) x] ::= PutVal[x].

Явные числовые преобразования. Согласно спецификации C#, явные числовые преобразования — это преобразования из числовых типов в числовые типы, для которых не существует неявного числового преобразования. Более точно, данные преобразования применяются в следующих случаях:

- S, Q ∈ {char, sbyte, byte, short, ushort, int, uint, long, ulong}, т.е. S и Q — целые типы;

- S ∈ {float, double, decimal}, Q — целый тип;
- S = double, Q = float;
- S ∈ {float, double}, Q = decimal;
- S = decimal, Q ∈ {float, double}.

В этих случаях Norm[(Q) x] ::= V0 := γ(S, Q, V(x)).

Явные преобразования типов перечисления. Если S ∈ {sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal}, а Q — тип перечисления, то

```
Norm[(Q) x] ::= V0 := γ(S, UT(Q), V(x))
```

Если S — тип перечисления, а Q ∈ {sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal}, то

```
Norm[(Q) x] ::= V0 := γ(UT(S), Q, V(x))
```

Если S и Q — типы перечисления, то

```
Norm[(Q) x] ::= V0 := γ(UT(S), UT(Q), V(x))
```

Явные преобразования ссылочных типов. Согласно спецификации C#, данные преобразования применяются в следующих случаях:

- S = object, Q — ссылочный тип;
- S — класс, Q — класс, S — базовый класс для Q;
- S — класс, Q — интерфейс, и при этом S — не sealed-класс и не реализует Q;
- S — интерфейс, Q — класс, и при этом Q — не sealed-класс или Q реализует S;
- S — интерфейс, Q — интерфейс, S не является наследником Q;
- S — массив элементов типа S_E, Q — массив элементов типа Q_E, и при этом:
  - S и Q имеют одинаковую размерность,
  - S_E и Q_E являются ссылочными типами,
  - существует явное преобразование ссылочных типов из S_E в Q_E;
- S есть System.Array или один из интерфейсов, реализуемых классом System.Array, Q — массив;
- S есть System.Delegate или один из интерфейсов, реализуемых классом System.Delegate, Q — делегат.

В этих случаях

```
Norm[(Q) x] ::=
    Norm[bool b];
    V := upd(V,
             L(b),
             and(not(eq(V(x), null)),
                 not(can_cast(T(V(x)), Q))));
    if (b) {
        Norm[System.InvalidCastException y];
        Norm[new System.InvalidCastException()];
        GetVal[y];
        E := V(y); } else { PutVal[x]; }
```

Unboxing-преобразования. Пусть `Q` — value-тип, а `S` — тип `object` или интерфейс, реализуемый value-типом `Q`. Тогда

```
Norm[(Q) x] ::=
    Norm[bool b];
    V := upd(V,
             L(b),
             or(eq(V(x), null),
                not(eq(T(V(x)), Q))));
    if (b) {
        Norm[System.InvalidCastException y];
        Norm[new System.InvalidCastException()];
        GetVal[y];
        E := V(y); } else { V0 := V(x); }
```

Пользовательские преобразования. Если для преобразования из `S` в `Q` существует пользовательское преобразование $u_{S,Q}$: $S_0 \to Q_0$, объявленное в типе $U \in \{S, Q\}$, то

```
Norm[(Q) x] ::=
    Norm[S0 x1];   Norm[(S0) x];   GetVal[x1];
    Norm[Q0 x2];   U.us,Q(x1);     GetVal[x2];
    Norm[(T) x2]
```

5.4.5. Бинарные операции

Пусть *op* $\in$ {`&`, `^`, `|`, `==`, `!=`, `<`, `>`, `<=`, `>=`, `<<`, `>>`, `*`, `/`, `%`, `+`, `-`}, E_1, E_2 — выражения, причем хотя бы одно из них отлично от локальной переменной, T_1, T_2 — типы выражений E_1, E_2. Тогда

```
Norm[E1 op E2] ::=
    Norm[T1 x];       Norm[E1];     GetVal[x];
    Norm[T2 y];       Norm[E2];     GetVal[y];
    Norm[x op y];
```

Пусть x и y — локальные переменные, имеющие типы T_x и T_y соответственно, *op* ∈ {&, ^, |, ==, !=, <, >, <=, >=, <<, >>, *, /, %, +, -} перегружена в типе U ∈ {T_x, T_y} и имеет аргументы типов T_{argx} и T_{argy}. Тогда

```
Norm[x op y] ::=
    Norm[Targx x1];    Norm[(Targx) x]; GetVal[x1];
    Norm[Targy y1];    Norm[(Targy) y]; GetVal[y1];
    U.op(x1, y1)
```

5.4.6. Операции проверки типов

Операция is. Пусть E — выражение, отличное от локальной переменной, T — тип. Тогда

```
Norm[E is T] ::= Norm[TE x]; Norm[E]; GetVal[x]; x.is(T)
```

Операция as. Пусть E — выражение, отличное от локальной переменной, T — тип. Тогда

```
Norm[E as T] ::= Norm[TE x]; Norm[E]; GetVal[x]; x.as(T)
```

5.4.7. Условные логические операции

Пусть E_1, E_2 — выражения, T_1, T_2 — их типы. Если операнды операций && или || имеют тип bool или типы, которые не определяют применимую операцию operator & или operator |, но определяют неявные преобразования в тип bool, то

```
Norm[E1 || E2] ::=
    Norm[T1 x];    Norm[E1];       GetVal[x];
    Norm[bool b]; Norm[(bool) x]; GetVal[b];
    if (b) { PutVal[true]; }
    else   { Norm[(bool) E2]; }

Norm[E1 && E2] ::=
    Norm[T1 x];    Norm[E1];       GetVal[x];
    Norm[bool b]; Norm[(bool) x]; GetVal[b];
    if (b) { Norm[(bool) E2]; }
    else   { PutVal[false]; }
```

Если операнды операций && или || имеют типы, которые определяют применимую перегруженную операцию operator & или operator | и содержат определения операций operator true и operator false, то

```
Norm[E1 || E2] ::=
    Norm[T1 x];    Norm[E1];        GetVal[x];
    Norm[T y];     Norm[(T) x];     GetVal[y];
    Norm[bool b]; T.true(y);        GetVal[b];
    if (b) { Norm[(T) x]; }
    else   { Norm[T.|(x, E2)]; }

Norm[E1 && E2] ::=
    Norm[T1 x];    Norm[E1];        GetVal[x];
    Norm[T y];     Norm[(T) x];     GetVal[y];
    Norm[bool b]; T.false(y);       GetVal[b];
    if (b) { Norm[(T) x]; }
    else   { Norm[T.&(x, E2)]; }
```

где T — тип, в котором объявляются описанные выше операции &, |, true и false.

5.4.8. Условная операция

Пусть E, E_1, E_2 — выражения, T_E, T_1, T_2 — типы выражений E, E_1, E_2 соответственно. Если первый операнд операции ?: имеет тип, который может быть неявно преобразован в тип bool, то

```
Norm[E ? E1 : E2] ::=
    Norm[TE x];    Norm[E];            GetVal[x];
    Norm[bool b]; Norm[(bool) x];      GetVal[b];
    if (b) { Norm[(T) E1]; }
    else   { Norm[(T) E2]; }
```

где $T \in \{T_1, T_2\}$ — тип условного выражения.

Если первый операнд операции ?: имеет тип, который определяет операцию operator true, то

```
Norm[E ? E1 : E2] ::=
    Norm[TE x];    Norm[E];            GetVal[x];
    Norm[bool b]; Norm[TE.true(x)];    GetVal[b];
    if (b) { Norm[(T) E1]; }
    else   { Norm[(T) E2]; }
```

где $T \in \{T_1, T_2\}$ — тип условного выражения.

5.4.9. Операции присваивания

Пусть E — выражение, *op* ∈ {&, ^, |, <<, >>, *, /, %, +, -}. Пусть U — выражение, классифицируемое как переменная. Если выражение U на верхнем уровне является элементом массива ссылочного типа, то

```
Norm[U = E] ::=
    Norm[T_U x];    Norm[U];        GetLoc[x];
    Norm[T_U y];    Norm[(T_U) E];  GetVal[y];
    Norm[bool b];
    V := upd(V,
             L(b),
             and(not(eq(V(y),
                        null)),
                 not(can_cast(T(V(y)),
                              T(V(x))))));
    if (b) {
        Norm[System.ArrayTypeMismatchException z];
        Norm[new System.ArrayTypeMismatchException()];
        GetVal[z];
        E := V(z); }
    else {
        V := upd(V, L(x), V(y));
        PutVal[y]; }

Norm[U op= E] ::=
    Norm[T_U x];    Norm[U];        GetLoc[x];
    Norm[T_E y];    Norm[E];        GetVal[y];
    Norm[T_op z_0]; Norm[x op y];   GetVal[z_0];
    Norm[T_U z];    Norm[(T_U) z_0]; GetVal[z];
    Norm[bool b];
    V := upd(V,
             L(b),
             and(not(eq(V(z),
                        null)),
                 not(can_cast(T(V(z)),
                              T(V(x))))));
    if (b) {
        Norm[System.ArrayTypeMismatchException z_1];
        Norm[new System.ArrayTypeMismatchException()];
        GetVal[z_1];
        E := V(z_1); }
    else {
```

```
        V := upd(V, L(x), V(z));
        PutVal[z]; }
```

Иначе:

```
Norm[U = E] ::=
    Norm[T_U x];    Norm[U];        GetLoc[x];
    Norm[T_U y];    Norm[(T_U) E];  GetVal[y];
    CopyValue[x, y];   PutVal[y]

Norm[U op= E] ::=
    Norm[T_U x];    Norm[U];        GetLoc[x];
    Norm[T_E y];    Norm[E];        GetVal[y];
    Norm[T_op z_0];  Norm[x op y];  GetVal[z_0];
    Norm[T_U z];    Norm[(T_U) z_0];  GetVal[z];
    CopyValue[x, y];   PutVal[z]
```

Пусть `U` — выражение, классифицируемое как доступ к свойству `P`. Если доступ к `P` осуществляется через нестатическое выражение E_P, имеющее тип T_P (т.е. `P` — нестатическое свойство), то

```
Norm[U = E] ::=
    Norm[T_P z];    Norm[E_P];      GetLocOrVal[z, Var[E_P]];
    Norm[T_U y];    Norm[(T_U) E];  GetVal[y];
    z.set_P(y);     PutVal[y]

Norm[U op= E] ::=
    Norm[T_P w];    Norm[E_P];      GetLocOrVal[w, Var[E_P]];
    Norm[T_U x];    w.get_P();      GetVal[x];
    Norm[T_E y];    Norm[E];        GetVal[y];
    Norm[T_op z_0];  Norm[x op y];  GetVal[z_0];
    Norm[T_U z];    Norm[(T_U) z_0];  GetVal[z];
    w.set_P(z);     PutVal[z]
```

Иначе (`P` — статическое свойство, объявленное в типе T_P):

```
Norm[U = E] ::=
    Norm[T_U y];    Norm[(T_U) E];  GetVal[y];
    T_P.set_P(y);   PutVal[y]

Norm[U op= E] ::=
    Norm[T_U x];    T_P.get_P();    GetVal[x];
    Norm[T_E y];    Norm[E];        GetVal[y];
    Norm[T_op z_0];  Norm[x op y];  GetVal[z_0];
```

```
    Norm[TU z];    Norm[(TU) z0];  GetVal[z];
    TP.set_P(z);   PutVal[z]
```

Пусть U — выражение, классифицируемое как доступ к индексатору через нестатическое выражение E_I, имеющее тип T_I, A_1, …, A_n — аргументы доступа к индексатору, а T_1, …, T_n — типы его формальных параметров. Тогда

```
Norm[U = E] ::=
    Norm[TI z];    Norm[EI];       GetLocOrVal[z, Var[EI]];
    Norm[T1 a1];   Norm[(T1) A1];  GetVal[a1];
    …
    Norm[Tn an];   Norm[(Tn) An];  GetVal[an];
    Norm[TU y];    Norm[(TU) E];   GetVal[y];
    z.set_Item(a1, … an, y);       PutVal[y]

Norm[U op= E] ::=
    Norm[TI w];    Norm[EI];       GetLocOrVal[w, Var[EI]];
    Norm[T1 a1];   Norm[(T1) A1];  GetVal[a1];
    …
    Norm[Tn an];   Norm[(Tn) An];  GetVal[an];
    Norm[TU x];
    w.get_Item(a1, … an);          GetVal[x];
    Norm[TE y];    Norm[E];        GetVal[y];
    Norm[Top z0];  Norm[x op y];   GetVal[z0];
    Norm[TU z];    Norm[(TU) z0];  GetVal[z];
    w.set_Item(a1, … an, z);       PutVal[z]
```

Пусть U — выражение, классифицируемое как доступ к событию Evt, в объявлении которого присутствуют функциональные члены add и remove. Если доступ к Evt осуществляется через нестатическое выражение E_{Evt}, имеющее тип T_{Evt} (т.е. Evt — нестатическое событие), то

```
Norm[U += E] ::=
    Norm[TEvt z];  Norm[EEvt];   GetLocOrVal[z, Var[EEvt]];
    Norm[TE x];    Norm[E];        GetVal[x];
    Norm[TU y];    Norm[(TU) x];   GetVal[y];
    z.add_Evt(y);

Norm[U -= E] ::=
    Norm[TEvt z];  Norm[EEvt];   GetLocOrVal[z, Var[EEvt]];
    Norm[TE x];    Norm[E];        GetVal[x];
    Norm[TU y];    Norm[(TU) x];   GetVal[y];
    z.remove_Evt(y);
```

Иначе (Evt — статическое событие, объявленное в типе T_{Evt}):

```
Norm[U += E] ::=
    Norm[T_E x];    Norm[E];        GetVal[x];
    Norm[T_U y];    Norm[(T_U) x];  GetVal[y];
    T_Evt.add_Evt(y);

Norm[U -= E] ::=
    Norm[T_E x];    Norm[E];        GetVal[x];
    Norm[T_U y];    Norm[(T_U) x];  GetVal[y];
    T_Evt.remove_Evt(y);
```

5.5. Операторы while, for, do, foreach

Нормализация операторов циклов подготавливает их к применению правил элиминации, расставляя дополнительные фигурные скобки.

Правило NIS1. Фрагмент вида

```
while (b) A
```

где A — оператор, не являющийся блоком, заменяется фрагментом

```
while (b) {A}
```

Правило NIS2. Фрагмент вида

```
for (B) A
```

где A — оператор, не являющийся блоком, заменяется фрагментом

```
for (B) {A}
```

Правило NIS3. Фрагмент вида

```
for (A_initializer; ; A_iterator) {A}
```

заменяется фрагментом

```
for (A_initializer; true; A_iterator) {A}
```

Правило NIS4. Фрагмент вида

```
do A while (b);
```

где A — оператор, не являющийся блоком, заменяется фрагментом

```
do {A} while (b);
```

Правило NIS5. Фрагмент вида

```
foreach (B) A
```

где A — оператор, не являющийся блоком, заменяется фрагментом

```
foreach (B) {A}
```

5.6. Операторы switch, if

Нормализация операторов выбора, как и в случае нормализации циклов, подготавливает их к применению правил элиминации. Условием операторов `if` и `switch` становится значение переменной, а не выражения. Сами операторы получают недостающие ветви `else` и `default`. Существующие метки `default` перемещаются в конец тела `switch`. В начало каждой switch- и default-секции помещается новая метка. Эти метки используются алгоритмом нормализации `goto case` и `goto default`.

Правило NSS1. Фрагмент вида

```
switch (e) {A}
```

заменяется фрагментом

```
{ Te x; x = e; switch (x) {A} }
```

где e — выражение, отличное от локальной переменной, T_e — тип выражения e, x — новая локальная переменная.

Правило NSS2. Фрагмент вида

```
switch (x) {A}
```

где x — переменная, A не содержит default-секцию на верхнем уровне, заменяется фрагментом

```
switch (x) { A default: break; }
```

Правило NSS3. Фрагмент вида

```
switch (x) { A case v1: … case vk: default: case vk+1: …
             case vn: B }
```

где x — переменная, заменяется фрагментом

```
switch (x) { A default: B }
```

Правило NSS4. Фрагмент вида

```
switch (x) { A B C }
```

где `x` — переменная, `B` — `switch`-секция, содержащая `default`, `C` содержит `switch`-секции, заменяется фрагментом

```
switch (x) { A C B }
```

Правило NSS5. Фрагмент вида

switch-labels statement-list

где *statement-list* не начинается с помеченного оператора вида *identifier*`:;`, порожденного данным правилом, заменяется фрагментом

switch-labels `L:;` *statement-list*

где `L` — новая уникальная метка.

Правило NSS6. Фрагмент вида

```
if (b) A
```

где `A` — оператор, не являющийся блоком, заменяется фрагментом

```
if (b) {A}
```

Правило NSS7. Фрагмент вида

```
if (b) {A} else B
```

где `B` — оператор, не являющийся блоком, заменяется фрагментом

```
if (b) {A} else {B}
```

Правило NSS8. Фрагмент вида

```
if (b) {A} S
```

где `S` ≠ `else`, заменяется фрагментом

```
if (b) {A} else {}
```

Правило NSS9. Фрагмент вида

```
if (b) {A} else {B}
```

заменяется фрагментом

```
{ Tb x; x = b; if (x) {A} else {B} }
```

где b — выражение, отличное от локальной переменной и метаинструкции, T_b — тип выражения b, x — новая локальная переменная.

Далее, рассмотрим правило нормализации тел функциональных членов.

5.7. Тела функциональных членов

Функциональными членами являются методы, свойства, события, индексаторы, операции, деструкторы и статические и нестатические конструкторы. В тело каждого функционального члена после всех операторов помещается новая метка выхода `__ExitPoint`, используемая в дальнейшем другими преобразованиями.

Правило NFMB. Пусть тело некоторого функционального члена имеет вид

`{` *statement-list*$_{opt}$ `}`

Тогда такой фрагмент заменяется фрагментом

`{` *statement-list*$_{opt}$ `__ExitPoint: ; }`

где `__ExitPoint` — новая уникальная метка.

Теперь перейдем к описанию алгоритмов нормализации операторов `goto`, `goto case` и `goto default`. Такая нормализация является частью процесса элиминации механизма обработки исключений и заключается в приведении программы к форме, в которой операторы перехода не передают управление за пределы операторов `try`.

5.8. Операторы goto, goto case и goto default

С элиминацией механизма обработки исключений связано две проблемы. Во-первых, невозможно локализовать преобразуемый фрагмент программы, в отличие, например, от элиминации циклов. Во-вторых, прежде чем удалять `try`, необходимо корректно преобразовать (нормализовать) те операторы `goto`, `goto case` и `goto default`, которые передают управление за пределы операторов `try`, имеющих finally-блоки. Для решения этих проблем разработан алгоритм, состоящий из нескольких шагов, каждый из которых сохраняет семантическую корректность и функциональную эквивалентность программы. Тем не менее, применение такого алгоритма требует введения на промежуточных шагах расширенного оператора `try`, т.е. оператора `try` с помеченным finally-блоком.

Правило NGOTO. Пусть *LS* — помеченный оператор вида `L: S;`. Блок, содержащий *LS* на верхнем уровне, обозначим через B_{LS}.

Пусть *Lab* = {$\mathtt{L}_1$, …, $\mathtt{L}_n$} — множество всех меток в программе. Множество всех меток верхнего уровня в некотором блоке *B* обозначим через *Lab*(*B*). Элементы множества *Lab*(*B*) обозначим $\mathtt{L}_1(B)$, …, $\mathtt{L}_K(B)$.

1. В глобальное пространство имен добавляется объявление следующего вида:

```
enum GT_LABELS { none }
class GT {
    public static void SFI() {
        gt = 0; gt = GT_LABELS.none;
    }
    public void IFI() {}
    public static GT_LABELS gt;
}
```

2. Для всех помеченных операторов *LS* фрагмент `L: S;` заменяется фрагментом

```
L:; GT.gt = GT_LABELS.none; S;
```

   и для всех операторов `goto L;` в блоке B_{LS} метка `L` заменяется уникальным идентификатором.

3. В перечисление `GT_LABELS` добавляются новые различные элементы $\mathtt{l}_1$, …, $\mathtt{l}_n$. Определим функцию *v*: *Lab* → `GT_LABELS` такую, что $v(\mathtt{L}_i) = \mathtt{l}_i$, $1 \le i \le n$.

4. Для всех блоков *B*, для всех finally-блоков в блоке *B* на верхнем уровне `finally {A}` заменяется фрагментом

```
finally(f_{j+1}) {
    A
    if (GT.gt == GT_LABELS.v(L_1(B))) goto L_1(B);
    else if (GT.gt == GT_LABELS.v(L_2(B))) goto L_2(B);
    …
    else if (GT.gt == GT_LABELS.v(L_K(B))) goto L_K(B);
    else if (GT.gt != GT_LABELS.none) goto f_j;
}
```

   где $\mathtt{f}_{j+1}$ — новая уникальная метка, $j \ge 0$, $\mathtt{f}_0$ — метка, соответствующая выходу из тела функционального члена (т.е. `__ExitPoint` до выполне-

ния шага 2). Перечисление по блокам B проводится рекурсивно, начиная с тел функциональных членов.

5. Для всех помеченных операторов LS, для всех операторов `goto L;` в блоке B_{LS}, если `goto L;` передает управление из try-оператора с finally-блоком, помеченным меткой $\mathtt{f}_j$, то `goto L;` заменяется фрагментом `GT.gt = GT_LABELS.`v`(L); goto` $\mathtt{f}_j$`;`.

Заметим, что в пунктах 4, 5 стратегия применения кванторов всеобщности по блокам имеет вид рекурсивного обхода блоков от объемлющего к вложенному. Однако операторы `goto` обрабатываются в текстуальном порядке сверху вниз. Аналогичное замечание действует и в случае следующего правила.

Правило NGOTOC. Данное правило применяется после применения правила NGOTO.

Пусть V — множество всех значений константных выражений, допустимых в switch-операторах.

1. Все switch-операторы в программе пронумеровываются.

2. Пусть Lab^{new} — множество всех меток, порожденных правилом **NSS5** на этапе нормализации операторов switch, а $Lab^{new}(S) = \{\mathtt{L}_1(S), \ldots, \mathtt{L}_{n(S)}(S)\}$ — множество таких меток в switch-операторе S. Определим соответствия $CaseL$: $V \times Nat \to Lab$ и $DefL$: $Nat \to Lab$ между метками switch-операторов и новыми метками следующим образом. Для всех меток вида `case E:`, входящих в *switch-labels* оператора S_i, $CaseL$(`E`, i) `:= L`, где `L` — метка, порожденная правилом **NSS5** для switch-секции, содержащей `case E:`. Для меток вида `default:`, входящих в *switch-labels* оператора S_i, $DefL(i)$ `:= L`.

3. В перечисление `GT_LABELS` добавляются новые различные элементы $\mathtt{l}_1$, …, $\mathtt{l}_N$, $N = |Lab^{new}|$. Пусть функция v: $Lab^{new} \to$ `GT_LABELS` устанавливает взаимнооднозначное соответствие между элементами множества Lab^{new} и $\mathtt{l}_1$, …, $\mathtt{l}_N$.

4. Для всех switch-операторов S, для всех блоков B в операторе S, кроме тел вложенных switch-операторов, для всех finally-блоков в блоке B на верхнем уровне, если finally-блок `finally {A}` является самым верхним в S из всех finally-блоков, то он заменяется фрагментом

```
finally(f0) {
    A
```

```
if (GT.gt == GT_LABELS.v(L1(S))) goto L1(S);
else if (GT.gt == GT_LABELS.v(L2(S))) goto L2(S);
…
else if (GT.gt == GT_LABELS.v(Ln(S))(S))) goto Ln(S)(S); }
```

иначе он заменяется фрагментом

```
finally(fj+1) {
    A
    if (GT.gt != GT_LABELS.none) goto fj; }
```

где $\mathtt{f}_j$ — новые различные метки, $j \geq 0$. Перечисление по блокам B проводится рекурсивно, начиная с тела оператора S.

5. Для всех switch-операторов S_i, для всех операторов `goto case E;` в операторе S_i, если `goto case E;` передает управление из try-оператора с finally-блоком, помеченным меткой $\mathtt{f}_j$, то `goto case E;` заменяется фрагментом `GT.gt = GT_LABELS.`v`(`*CaseL*`(E, `i`)); goto f`$_j$`;`. Иначе `goto case E;` заменяется фрагментом `goto` *CaseL*`(E, `i`);`.

6. Для всех switch-операторов S_i, для всех операторов `goto default;` в операторе S_i, если `goto default;` передает управление из try-оператора с finally-блоком, помеченным меткой $\mathtt{f}_j$, то `goto default;` заменяется фрагментом `GT.gt = GT_LABELS.`v`(`*DefL*`(`i`)); goto f`$_j$`;`. Иначе `goto default;` заменяется фрагментом `goto` *DefL*`(`i`);`.

Таким образом, алгоритмы, задаваемые правилами **NGOTO**, **NGOTOC**, преобразуют операторы `goto`, `goto case` и `goto default` в обычный оператор `goto`. В дополнение к этому, в полученной на выходе программе операторы `goto` не передают управление за пределы `try`. В программе, обладающей таким свойством, элиминация `try` является законной. Удаление операторов `try` — это уже локальное преобразование, реализуемое набором недетерминировано применяемых правил, которые обсуждаются в следующих разделах.

6. ЭЛИМИНАЦИЯ

В этом разделе детально рассмотрены алгоритмы элиминации. Они удаляют из C#-light программы конструкции, которые запрещены в C#-kernel.

6.1. Оператор-объявление

Для нормализации операторов-выражений (разд. 0) и элиминации операторов-объявлений языка C#-light используется преобразование `Norm`.

Правило EDS. Фрагмент вида

type identifier;

заменяется фрагментом

`Norm[`*type identifier*`];`

Определим `Norm` для каждого типа объявлений локальных переменных.

6.1.1. Локальные объявления

Локальные переменные. Пусть `I` — идентификатор, `Q` — тип. Тогда

```
Norm[Q I] ::= Init(Q); new_instance();
              L := upd(L, I, V0);
              T := upd(T, L(I), Loc(Q))
```

Локальные константы. Пусть `I` — идентификатор, `Q` тип, допустимый в объявлениях локальных констант, т.е. `Q` — ссылочный тип, тип перечисления или `Q` ∈ {`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`}, `e` — константное выражение. Тогда

```
Norm[const Q I = e] ::=
    Init(const Q); new_instance();
    L := upd(L, I, V0);
    T := upd(T, L(I), Loc(const Q));
    Norm[e]; GetVal[I]
```

6.2. Оператор break

Оператор `break` заменяется оператором `goto`.

Правило EBREAK1. Фрагмент вида

```
switch (e) { A break; B }
```

заменяется фрагментом

```
{ switch (e) { A goto L; B } L: ; }
```

где L — новая метка.

Правило EBREAK2. Фрагмент вида

```
while (b) { A break; B }
```

заменяется фрагментом

```
{ while (b) { A goto L; B } L: ; }
```

где L — новая метка.

Правило EBREAK3. Фрагмент вида

```
for (C) { A break; B }
```

заменяется фрагментом

```
{ for (C) { A goto L; B } L: ; }
```

где L — новая метка.

Правило EBREAK4. Фрагмент вида

```
do { A break; B } while (b);
```

заменяется фрагментом

```
{ do { A goto L; B } while (b); L: ; }
```

где L — новая метка.

Правило EBREAK5. Фрагмент вида

```
foreach (C) { A break; B }
```

заменяется фрагментом

```
{ foreach (C) { A goto L; B } L: ; }
```

где L — новая метка.

6.3. Оператор continue

Оператор `continue` заменяется оператором `goto`.

Правило ECONT1. Фрагмент вида

```
while (b) { A continue; B }
```

заменяется фрагментом

```
while (b) { A goto L; B L: ; }
```

где L — новая метка.

Правило ECONT2. Фрагмент вида

```
for (C) { A continue; B }
```

заменяется фрагментом

```
for (C) { A goto L; B L: ; }
```

где L — новая метка.

Правило ECONT3. Фрагмент вида

```
do { A continue; B } while (b);
```

заменяется фрагментом

```
do { A goto L; B L: ; } while (b);
```

где L — новая метка.

Правило ECONT4. Фрагмент вида

```
foreach (C) { A continue; B }
```

заменяется фрагментом

```
foreach (C) { A goto L; B L: ; }
```

где L — новая метка.

6.4. Оператор return

Оператор `return` заменяется оператором `goto`, передающим управление на метку `__ExitPoint`, расставляемую правилом **NFMB** (разд. 0).

Правило ERET1. Фрагмент вида

```
return e;
```

заменяется фрагментом

```
{ Norm[Te x]; Norm[e]; ReadVal[x];
  Norm[Te y]; CopyValue[y, x]; PutVal[y];
  goto __ExitPoint; }
```

Правило ERET2. Фрагмент вида

```
return;
```

заменяется фрагментом

```
{ goto __ExitPoint; }
```

6.5. Оператор throw

Оператор `throw` заменяется серией новых операторов-выражений, помещающих исключение в ячейку `Exc`. В случае оператора `throw` без аргументов, который может использоваться только в catch-секциях операторов `try`, в `Exc` помещается значение переменной `eSaved`, объявляемой правилами элиминации `try`, описанными в следующих разделах.

Правило ETHROW1. Фрагмент вида

```
throw e;
```

заменяется фрагментом

```
{   Norm[Te x]; e; GetVal[x];
    Norm[bool b];
    V := upd(V, L(b), eq(V(x), null));
    if (b) {
        Norm[System.NullReferenceException y];
        Norm[new System.NullReferenceException()];
        GetVal[y];
        E := V(y);
    }
    else  { E := V(x); }
}
```

Правило ETHROW2. Пусть оператор `throw;` находится в catch-блоке, в котором текущее исключение записано в переменную `eSaved` (см. элиминацию оператора `try`). Тогда фрагмент вида

```
throw;
```

заменяется фрагментом

```
{ E := V(eSaved); }
```

6.6. Оператор foreach

Оператор `foreach` преобразуется в оператор `for`, чтобы затем применить правила элиминации `for`. Замена происходит в точности по спецификации исполнения цикла `foreach` [6].

Правило EFOREACH1. Фрагмент вида

```
foreach (type id in e) {A}
```

в котором тип выражения `e` реализует схему совокупности (collection pattern), заменяется фрагментом

```
{   E enumerator = (e).GetEnumerator();
    try {
        while (enumerator.MoveNext()) {
            type id = (type) enumerator.Current;
            {A}
        }
    }
    finally {
        ((System.IDisposable)(enumerator)).Dispose();
    }
}
```

если `E` реализует интерфейс `System.IDisposable`, где `E` — тип возвращаемого значения метода `(e).GetEnumerator()`.

Правило EFOREACH2. Фрагмент вида

```
foreach (type id in e) {A}
```

в котором тип выражения `e` реализует схему совокупности (collection pattern), заменяется фрагментом

```
{   E enumerator = (e).GetEnumerator();
    while (enumerator.MoveNext()) {
        type id = (type) enumerator.Current;
        {A}
    }
}
```

если `E` не реализует интерфейс `System.IDisposable`, где `E` — тип возвращаемого значения метода `(e).GetEnumerator()`.

Правило EFOREACH3. Фрагмент вида

```
foreach (type id in e) {A}
```

в котором тип выражения `e` реализует интерфейс `System.IEnumerable`, заменяется фрагментом

```
{   System.IEnumerator enumerator =
            ((System.IEnumerable)(e)).GetEnumerator();
    try {
        while (enumerator.MoveNext()) {
            type id = (type) enumerator.Current;
            {A}
        }
    }
    finally {
        System.IDisposable disposable =
                (enumerator as System.IDisposable);
        if (disposable != null)
            disposable.Dispose();
    }
}
```

Таким образом, после применения данных правил в программе остается цикл `for`, который затем удаляется остальными правилами элиминации циклов.

6.7. Оператор for

Оператор `for` преобразуется в оператор `while`, чтобы затем применить правила элиминации `while`.

Правило EFOR1. Фрагмент вида

```
for (d; b; e1, e2, … en) {A}
```

где `d` — объявление локальной переменной, $n \geq 0$, `A` не содержит `continue`, `break` на верхнем уровне и не содержит `return`, заменяется фрагментом

```
{ d; while (b) { A e1; e2; … en; } }
```

Правило EFOR2. Фрагмент вида

```
for (e1, e2, … em; b; e1, e2, … en) {A}
```

где $m \geq 0$, $n \geq 0$, `A` не содержит `continue`, `break` на верхнем уровне и не содержит `return`, заменяется фрагментом

```
{ e[1]; e[2]; … e[m]; while (b) { A e1; e2; … en; } }
```

6.8. Оператор do

Оператор `do`, как и `for`, преобразуется в оператор `while`.

Правило EDO. Фрагмент вида

```
do {A} while (b);
```

где `A` не содержит `continue`, `break` на верхнем уровне и не содержит `return`, заменяется фрагментом

```
{ while (true) { A if (b) {} else { goto L; } } L: ; }
```

где `L` — новая метка.

6.9. Оператор while

Правило EWHILE. Фрагмент вида

```
while (b) {A}
```

где `A` не содержит `continue`, `break` на верхнем уровне и не содержит `return`, заменяется фрагментом

```
{ L:; if (b) { A goto L; } }
```

где `L` — новая метка.

6.10. Оператор try

Элиминация оператора `try` происходит после нормализации операторов `goto`, `goto case` и `goto default`.

Правило ETRY1. Фрагмент вида

```
try {A} finally(L) {B}
```

заменяется фрагментом

```
{A} L:;
    if (catch(x)) {
        B
        E := V(x); goto M;
    }
```

```
    {B}
    M:;
```

где M — новая метка.

Правило ETRY2. Фрагмент вида

```
try {A} catch (T1 x) {C1}
        catch (T2 x) {C2}
        …
        catch (Tn x) {Cn}
        catch        {Cg}
```

заменяется фрагментом

```
{A} if      (catch(T1, x)) { T1 eSaved; eSaved = x; C1 }
    else if (catch(T2, x)) { T2 eSaved; eSaved = x; C2 }
    …
    else if (catch(Tn, x)) { Tn eSaved; eSaved = x; Cn }
    else if (catch(eSaved)) { Cg }
```

Правило ETRY3. Фрагмент вида

```
try {A} catch (T1 x) {C1}
        catch (T2 x) {C2}
        …
        catch (Tn x) {Cn}
        catch        {Cg} finally(L) {B}
```

заменяется фрагментом

```
{A} if      (catch(T1, x)) { T1 eSaved; eSaved = x; C1 }
    else if (catch(T2, x)) { T2 eSaved; eSaved = x; C2 }
    …
    else if (catch(Tn, x)) { Tn eSaved; eSaved = x; Cn }
    else if (catch(eSaved)) { Cg }
    L:;
    if (catch(x)) {
        B
        E := V(x); goto M;
    }
    {B}
    M:;
```

где M — новая метка.

6.11. Оператор switch

Оператор `switch` преобразуется к условному оператору `if` собиранием ветвей. Такая замена позволяет не усложнять аксиоматическую семантику громоздкими правилами вывода.

Правило ESWITCH1. Фрагмент вида

```
switch (x) { A default: L:; C }
```

где `x` — переменная, `A` — не пусто, `C` — список операторов, не содержащий `break`, `goto case` и `goto default` на верхнем уровне, `L` — метка, добавленная правилом **NSS5** на этапе нормализации оператора switch, заменяется фрагментом

```
switch (x) { A default: goto L; } L:; {C}
```

Правило ESWITCH2. Фрагмент вида

switch (x) { A case v_1: … case v_n: L:; B default: C }

где `x` — переменная, $n \geq 1$, `B` — список операторов, не содержащий `break`, `goto case` и `goto default` на верхнем уровне, `L` — метка, добавленная правилом **NSS5** на этапе нормализации оператора switch, `C` не начинается с помеченного оператора, заменяется фрагментом

switch (x)
{ A default: if ((x == v_1)||…||(x == v_n))
{ goto L; } else {C}
} L:; {B}

Правило ESWITCH3. Фрагмент вида

```
switch (x) { default: A }
```

где `x` — переменная, `A` не содержит `case`, `break`, `goto case` и `goto default` на верхнем уровне, заменяется фрагментом

```
{ A }
```

Теперь рассмотрим элиминацию using-директив.

6.12. Директива using

Правило EUSING.
1. Все вхождения имен типов и пространств имен (*namespace-or-type-name*) заменяются своими полностью квалифицированными именами.
2. Все using-директивы удаляются.

6.13. Пространство имен

Правило ENSPACE. Данное правило применяется после элиминации using-директив.
1. Для всех типов, объявленных в пространствах имен (кроме типов, вложенных в другие типы), создаются копии в глобальном пространстве имен.
2. Все вхождения имен типов заменяются именами своих копий в глобальном пространстве имен.
3. Все пространства имен удаляются вместе с объявленными в них типами.

7. ЗАКЛЮЧЕНИЕ

В данной работе представлен перевод из языка C#-light в язык C#-kernel в рамках трехуровневого подхода к верификации C#-программ [8]. Ранее подобная проблема исследовалась в работах [3, 9] для языка C. Процесс перевода имеет вид последовательного применения трансформаций.

По типу воздействия на программу все алгоритмы можно разбить на три группы:

– локальные преобразования;
– нелокальные преобразования;
– псевдо-локальные преобразования.

К первой группе относятся такие преобразования, как элиминация операторов `break`, `continue`, `return`. Ко второй группе можно отнести элиминацию директив `using` и пространств имен. В третью группу попадают нормализация `goto`, `goto case`, `goto default` и элиминация оператора `try`. Преобразования третьей группы характерны тем, что, с одной стороны, их действие можно локализовать, а с другой, их применение требует

расширения семантики C#-light путем введения новых вспомогательных конструкций, что отличает их от обычных локальных преобразований.

Элиминация механизма обработки исключений, реализуемая группой псевдо-локальных преобразований, — принципиальное отличие данной работы от других по этой тематике. Сложности такой элиминации связаны с finally-блоками и операторами `goto`, `goto case`, `goto default`. Другим существенным отличием служит нормализация выражений. В отличие от [4], в этой работе потребовалось применение рекурсивного преобразования `Norm`, задаваемого серией нетривиальных правил.

Следующим закономерным шагом является формальное обоснование корректности перевода. Специфика этой проблемы заключается в том, что входная и выходная программы аннотированы пред-, постусловиями и инвариантами циклов. Обоснование корректности перевода аннотированных программ — неисследованная область, и для решения этой задачи потребуется разработка новых методов доказательства, что и будет служить предметом дальнейшей работы.

СПИСОК ЛИТЕРАТУРЫ

1. **Дубрановский И.В.** Верификация C#-программ: перевод из языка C#-light в язык C#-kernel. — Новосибирск, 2004. — 60 с. — (Препр. / РАН. Сиб отд-ние. ИСИ; № 120).
2. **Непомнящий В.А., Ануреев И.С., Дубрановский И.В., Промский А.В.** На пути к верификации C#-программ: трехуровневый подход // Программирование. — 2006. — № 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. — № 6. — С. 19-30.
4. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации C-программ. Перевод из языка C-light в язык C-light-kernel и его формальное обоснование. Часть 3. — Новосибирск, 2002. — 84 с. — (Препр. / РАН. Сиб отд-ние. ИСИ; № 97).
5. **Börger E.**, **Schulte W.** A programmer friendly modular definition of the semantics of Java // Lect. Notes Comput. Sci. — 1999. — Vol. 1523. — P. 353–404.
6. Standard ECMA-334 — C# Language Specification. — December 2001. — http://www.ecma.ch.
7. **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.

8. **Nepomniaschy V.A., Anureev I.S., Dubranovsky I.V., Promsky A.V.** A three-level approach to C# program verification // Joint Bulletin of NCC and IIS. — 2004. — Vol. 20. — P. 61-86.
9. **Nepomniaschy V.A., Anureev I.S., Promsky A.V.** Verification-oriented language C-light and its structural operational semantics // Proc. of PSI'03. — Lect. Notes Comput. Sci. — 2003. — Vol. 2890. — P. 103-111.
10. **Oheimb D.V.** Hoare logic for Java in Isabelle/HOL // Concurrency and Computation: Practice and Experience. — 2001. — Vol. 13, № 13. — P. 1173-1214.

И. В. Дубрановский

НА ПУТИ К ВЕРИФИКАЦИИ C#-ПРОГРАММ: АЛГОРИТМЫ ПЕРЕВОДА ИЗ C#-LIGHT В C#-KERNEL

Препринт
140

Рукопись поступила в редакцию 11.12.06
Рецензент А.В. Промский
Редактор З.В. Скок

Подписано в печать 28.12.06
Формат бумаги 60 × 84 1/16 Объем 3.2 уч.-изд.л., 3.5 п.л.
Тираж 60 экз.

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