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

И. В. Дубрановский

ВЕРИФИКАЦИЯ C#-ПРОГРАММ: ПЕРЕВОД ИЗ ЯЗЫКА C#-LIGHT В ЯЗЫК C#-KERNEL

Препринт
120

Новосибирск 2004

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.

© Институт систем информатики им А. П. Ершова СО РАН, 2004

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

Igor V. Dubranovsky

C#-PROGRAM VERIFICATION: TRANSLATION FROM C#-LIGHT INTO C#-KERNEL

Preprint
120

Novosibirsk 2004

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 of translation from C#-light into C#-kernel are considered.

© A. P. Ershov Institute of Informatics Systems, 2004

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 [3]. 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 конструкциями, как перегрузка, динамическое связывание методов, обработка исключений, статическая инициализация классов. Аксиоматическая семантика была предложена в [5, 8] 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 [8] семантика оказалась громоздкой и неудобной для практического использования.

В этой работе детально представлен этап перевода нового трехуровневого подхода [6], совмещающего операционную и аксиоматическую семантику. 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 – это один из критических моментов, требующих аккуратного описания и формального обоснования. Подобная проблема решается в [2] для языка C. Но там этап перевода значительно менее трудоемкий и носит характер локальных трансформаций фрагментов программы, в то

время как в данной работе часть трансформаций имеет более сложный вид. В некоторой степени это связано с объектной ориентированностью языка C# и с механизмом обработки исключений, которого нет в C.

Работа состоит из четырех разделов. Язык аннотаций описан в разд. 0. Краткое описание языков C#-light и C#-kernel приводится в разд. 0. Алгоритмы перевода из C#-light в C#-kernel рассматриваются в разд. 0. Результаты и перспективы дальнейшей работы обсуждаются в разд. 0.

Эта работа частично поддержана грантом РФФИ 04-01-00114a.

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.

Типы и алфавит. Допустимыми типами языка аннотаций являются:

- базовые типы: `CT#`, `Names`, `Locations`, `TypeSpecs`,
- функции: $\mathtt{T}_1 \to \mathtt{T}$,
- декартовы произведения: $\mathtt{T}_1 \times \mathtt{T}_2$,

где `CT#` – объединение всех допустимых типов C#-light, `Names` – множество идентификаторов объектов в программе, `Locations` – множество ячеек памяти, `TypeSpecs` – множество абстрактных имен типов. Тип `Locations` содержит специальные ячейки `Val` и `Exc`, используемые для хранения значения, соответственно последнего вычисленного (под)выражения и брошенного исключения.

Алфавит языка аннотаций состоит из следующих классов символов:

- *переменные* и *константы*;
- *функциональные* и *предикатные* символы;
- *круглые скобки*, *треугольные скобки* и *запятая*;
- символы *операций*: стандартные операции C#-light и $\cup$.

Выражения. Переменные могут иметь только допустимые типы. Константы могут иметь любой тип, за исключением `void`. Помимо общепринятых значений, базовые типы C#-light могут содержать неопределенное значение, обозначаемое через `Om`.

Выражения языка аннотаций определяются индуктивно:

– константа c типа T является выражением типа T;
– переменная v типа T является выражением типа T;
– если s_1, …, s_n – выражения типов T_1, …, T_n соответственно, то <s_1, …, s_n> – выражение типа $T_1 \times \ldots \times T_n$;
– если s_1, …, s_n – выражения типов T_1, …, T_n соответственно, а s – выражение типа $T_1 \times \ldots \times T_n \to T$, то s($s_1$, …, s_n) – выражение типа T.

Логические выражения, или просто *аннотации*, строятся из выражений типа bool с помощью логических связок и кванторов как обычно. В связи с тем что выражения имеют префиксную нотацию, стандартные логические связки обозначаются через следующие функции-константы: and(,), or(,), not(), imply(,), exist(,), any(,). Предикат равенства обозначается через eq(,). Следует отметить, что операции C#-light также 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.

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

1. upd – функция замены значения. Если s – выражение типа $T \to T'$, а выражения e_1, e_2 имеют типы T и T′ соответственно, то терм upd(s, e_1, e_2) является выражением s′ типа $T \to T'$, которое совпадает с s на всех аргументах кроме, возможно, e_1 и $s'(e_1) = e_2$.

2. member – функция доступа к элементу. Если s – выражение типа Names → Locations, e – выражение типа Names и значение e лежит в области определения s, то member(s, e) является выражением типа Locations и member(s, e) = s(e); иначе, member(s, e) = Om.

3. γ – функция округления. Если T_1, T_2 – числовые типы, а выражение e имеет тип CT#, то γ(T_1, T_2, e) тоже имеет тип CT# и возвращает округленное по правилам округления числовых типов [4] значение выражения e. Отметим, что в случаях, предусмотренных спецификацией [4], γ имеет побочный эффект в виде записи объекта исключения в ячейку Exc. Кроме того, если существует неявное числовое преобразование из T_1 в T_2, то γ(T_1, T_2, e) = e.

4. add – функция добавления метода в делегат. Если s – выражение типа N → CT# × Names, а m – выражение типа CT# × Names, то add(s, m) яв-

ляется выражением типа $\mathbf{N} \to$ CT# $\times$ Names и add(s, m) = upd(s, n, m), где n $\in \mathbf{N}$, n = max(Dom(s)) + 1; иначе, add(s, m) = Om.

5. 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 [4], то выражение can_cast(T_1, T_2) истинно; иначе это выражение ложно.

Метапеременные. Модификация переменных в программе осуществляется через следующие метапеременные:

1) Mem – переменная типа Names $\cup$ TypeSpecs $\to$ Locations, т.е. управление памятью (**Me**mory **M**anagement);

2) MD – переменная типа Locations $\to$ CT# $\cup$ Locations, т.е. дамп памяти (**M**emory **D**ump);

3) TP – переменная типа Names $\cup$ Locations $\cup$ CT# $\to$ TypeSpecs. Задает типы идентификаторов, ячеек памяти и констант типов C#-light;

4) UT – переменная типа TypeSpecs $\to$ TypeSpecs, т.е. **U**nderlying **T**ype. Для каждого типа перечисления задает его базовый тип.

Например, тот факт, что переменная x в программе равна 3, в нашем подходе записывается в виде eq(MD(Mem(x)), 3), где x – уже константа.

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, основанный на синтаксисе языка C#-light. Он определяется в два этапа. Сначала строится подмножество S языка C#-light с помощью следующих ограничений:

- S не содержит пространства имен и using-директивы;
- S не содержит следующие операторы:
  - операторы перехода `break`, `continue`, `return`, `goto case`, `goto default` и `throw`,
  - оператор `try`,
  - оператор выбора `switch`,
  - все операторы циклов,
  - операторы-объявления;
- S не содержит операторы `if`, в которых условное выражение не является переменной типа boolean;
- в программах, принадлежащих подмножеству S, вызов статических функциональных членов может производиться только в тех местах в программе, в которых выполнена статическая инициализация соответствующих классов или структур;
- в программах, принадлежащих подмножеству S, все метки, имена локальных переменных и имена локальных констант должны быть уникальны;
- в программах, принадлежащих подмножеству S, множества меток, имен типов, имен локальных переменных и имен локальных констант не пересекаются;
- все комментарии в программе, принадлежащей подмножеству S, являются аннотациями.

После этого, C#-kernel определяется как расширение подмножества S. Новыми конструкциями, не принадлежащими S, являются метаинструкции, модифицированные операторы-выражения и модифицированные декларации классов и структур.

Метаинструкции

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

1) `x := E` присваивает метапеременной `x` значение выражения `E`, которое записывается на языке аннотаций;

2) `new_instance(x)` ассоциирует идентификатор `x` с новой ячейкой памяти;

3) `Init(C)` выполняет статическую инициализацию класса `C`, если этот класс не инициализирован; иначе, ничего не делает;

4) `catch(T, x)` возвращает `true`, если в ячейке `Exc` лежит значение типа `T`; иначе, возвращает `false`. Если в `Exc` лежит значение типа `T`, объект-исключение, находящийся в `Exc`, записывается в переменную `x`, а затем удаляется из `Exc`, чтобы показать, что исключение перехвачено. Эта метаинструкция может использоваться только как условное выражение в операторе `if`;

5) `catch(x)` возвращает `true`, если значение ячейки `Exc` определено; иначе, возвращает `false`. Так же, как в случае с `catch(T, x)`, объект-исключение, находящийся в `Exc`, записывается в переменную `x`, а затем удаляется из `Exc`. Эта метаинструкция может использоваться только как условное выражение в операторе `if`, в котором отсутствует ветвь `else`, и моделирует общую `catch`-секцию оператора `try`.

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 на выражения языка C#-light:

- нормализованное выражение имеет вид `x.y(`z_1`, …, `z_n`)` или `y(`z_1`, …, `z_n`)`, где `x`, `y` – имена, z_1, …, z_n – имена или литералы;
- в нормализованных выражениях функциональные члены могут быть вызваны только в нормальной форме [4];
- в нормализованных выражениях логические операции `||` и `&&`, условная операция `?:`, операция `new` и все операции присваивания запрещены.

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 instance-полей соответственно. Инициализация констант и статических полей выполняется в методе

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

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 instance-полей выполняется в методе

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

Эти методы расширяют контекст прямого присваивания в `readonly`-поле. Все выражения, встречающиеся в телах этих методов, должны быть выражениями языка C#-kernel.

ПЕРЕВОД ИЗ C#-LIGHT В C#-KERNEL

Перевод из C#-light в C#-kernel имеет вид последовательного применения алгоритмов перевода. Алгоритмы перевода строятся на базе трансформаций. Некоторые алгоритмы определяются неявно набором трансформаций, недетерминировано применяемых к преобразуемой программе. Остальные 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.

Применение одного алгоритма перевода может привести к возможности применения другого, даже если первоначально его применение было невозможно. В этом смысле алгоритмы перевода имеют характер

можно. В этом смысле алгоритмы перевода имеют характер оптимизирующих преобразований, когда одно преобразование является тупиковым или открытым по отношению к другому.

Все алгоритмы для удобства разделены на два класса:

- алгоритмы нормализации;
- алгоритмы элиминации.

Далее, введем несколько соглашений и обозначений, а затем перейдем к описанию алгоритмов перевода из C#-light в C#-kernel.

Обозначения

Для обозначения выражений произвольного вида используются символы e, E, e_1, E_1 и т.д. Для обозначения boolean-выражений используется символ b. Пустая строка обозначается через *e*.

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.

Далее, опишем алгоритмы перевода из C#-light в C#-kernel.

Алгоритмы нормализации

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.

Декларация класса и структуры

Для каждого свойства, индексатора и события с пользовательскими аксессорами в объявление класса или структуры добавляются соответствующие методы, по одному на каждый из аксессоров. Имена и сигнатуры методов строятся по правилам для зарезервированных имен членов (reserved member names) [4]. Тела методов дублируют тела соответствующих аксессоров. 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 аксессоров преобразуется в объявления делегатов путем удаления ключевого слова `event`. Это законно, так как такие события имеют такую же семантику, как и 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
}
```

Правило NINSTINIT. В объявление класса добавляется метод

```
public void IFI()
{
    instance-field-default-initialization
```

instance-field-initialization
}

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.

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.

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 [4] 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

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

являющийся выражением создания объекта, в котором выбранный instance-конструктор применим в расширенной форме [4] к списку аргументов A_1, A_2, …, A_n, заменяется фрагментом

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

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

Правило NFVNA3. Фрагмент вида

$E[E_1, E_2, \ldots E_n]$

[1] Определение понятий *расширенная форма* и *нормальная форма* вызова функционального члена можно найти в спецификации C# [4].

являющийся выражением доступа к индексатору, в котором выбранный индексатор применим в расширенной форме [4] 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, которую мы рассмотрим далее.

Оператор-выражение

Для нормализации операторов-выражений и элиминации операторов-объявлений (см. 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. Мы 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.

Вспомогательные преобразования

При определении `Norm` используются вспомогательные преобразования, которые мы опишем в этом разделе.

`Mod`. Пусть `E` – выражение. Тогда

```
Mod[E] ::= e
Mod[ref E] ::= ref
```

```
Mod[out E] ::= out
```

Здесь и далее через *e* обозначается пустая строка.

Arg. Пусть E – выражение. Тогда

```
Arg[E] ::= E
Arg[ref E] ::= E
Arg[out E] ::= E
```

Var. Пусть E – выражение. Если E классифицируется как переменная, то

```
Var[E] ::= true
```

Иначе

```
Var[E] ::= false
```

ReadVal. Пусть x – локальная переменная. Тогда

```
ReadVal[x, true] ::= MeM := upd(MeM, x, MD(Val))
ReadVal[x, false] ::= MD := upd(MD, MeM(x), MD(Val))
```

WriteVal. Пусть x – локальная переменная или this. Тогда

```
WriteVal[x, true] ::= MD := upd(MD, Val, MeM(x))
WriteVal[x, false] ::= MD := upd(MD, Val, MD(MeM(x)))
```

WriteMemberVal. Пусть E – тип, в котором объявлено поле x. Тогда

```
WriteMemberVal[E, x] ::=
        MD := upd(MD, Val, member(MeM(E), x))
```

Пусть E – переменная value-типа, в котором объявлено поле x. Тогда

```
WriteMemberVal[E, x] ::=
        MD := upd(MD, Val, member(MeM(E), x))
```

Пусть E – переменная ссылочного типа, в котором объявлено поле x. Тогда

```
WriteMemberVal[E, x] ::=
        MD := upd(MD, Val, member(MD(MeM(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.a2];
                    …
                    DefaultValue[E.an]
```

MGType. Пусть E – выражение, классифицируемое как группа методов, которое порождает статический метод, объявленный в типе T. Тогда

```
MGType[E] ::= T
```

MGMethod. Пусть E – выражение, классифицируемое как группа методов, которое порождает статический или instance-метод M. Тогда

```
MGMethod[E] ::= M
```

NormArg. Пусть E – выражение. Тогда

```
NormArg[E, T] ::= Norm[(T) E]
```

Пусть E – выражение, классифицируемое как переменная, T_E – тип выражения E.

```
NormArg[ref E, T] ::= Norm[E]

NormArg[out E, T] ::= Norm[E]
```

Первичные выражения

Литерал. Пусть I – литерал. Тогда

```
Norm[I] ::= MD := upd(MD, Val, I)
```

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

```
Norm[I] ::= WriteVal[I, true]
```

Если I является instance-членом (кроме instance-метода), то

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

Иначе:

```
Norm[E.I] ::= Init(E);
              WriteMemberVal[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[T_E x];
              Norm[E];
```

```
                ReadVal[x, Var[E]];
                x.get_I()
```

Иначе:

```
Norm[E.I] ::= Norm[TE x];
              Norm[E];
              ReadVal[x, Var[E]];
              WriteMemberVal[x, I]
```

Выражение вызова. Пусть первичное выражение P в выражении вызова является группой методов, состоящей из статического метода, а T_1, T_2, …, T_n – типы формальных параметров метода. Тогда

```
Norm[P(A1, A2, … An)] ::=
    Norm[T1 x1]; NormArg[A1, T1]; ReadVal[x1, Var[Arg[A1]]];
    Norm[T2 x2]; NormArg[A2, T2]; ReadVal[x2, Var[Arg[A2]]];
    …
    Norm[Tn xn]; NormArg[An, Tn]; ReadVal[xn, Var[Arg[An]]];
    P(Mod[A1] x1, Mod[A2] x2, … , Mod[An] xn)

Norm[base(A1, A2, … An)] ::=
    Norm[T1 x1]; NormArg[A1, T1]; ReadVal[x1, Var[Arg[A1]]];
    Norm[T2 x2]; NormArg[A2, T2]; ReadVal[x2, Var[Arg[A2]]];
    …
    Norm[Tn xn]; NormArg[An, Tn]; ReadVal[xn, Var[Arg[An]]];
    base(Mod[A1] x1, Mod[A2] x2, … , Mod[An] xn)
```

Пусть первичное выражение P в выражении вызова является группой методов, состоящей из instance-метода и instance-выражения E, имеющего тип T_E, а T_1, T_2, …, T_n – типы формальных параметров instance-метода. Тогда

```
Norm[P(A1, A2, … An)] ::=
    Norm[TE x0]; Norm[E];          ReadVal[x0, Var[E]];
    Norm[T1 x1]; NormArg[A1, T1]; ReadVal[x1, Var[Arg[A1]]];
    Norm[T2 x2]; NormArg[A2, T2]; ReadVal[x2, Var[Arg[A2]]];
    …
    Norm[Tn xn]; NormArg[An, Tn]; ReadVal[xn, Var[Arg[An]]];
    x0.MGMethod[P](Mod[A1] x1, Mod[A2] x2, … , Mod[An] xn)
```

Пусть первичное выражение P в выражении вызова имеет тип делегата, а T_1, T_2, …, T_n – типы формальных параметров делегата. Тогда

```
Norm[P(A1, A2, … An)] ::=
    Norm[TP x];  Norm[P];          ReadVal[x, Var[P]];
```

```
    Norm[T1 x1]; NormArg[A1, T1]; ReadVal[x1, Var[Arg[A1]]];
    Norm[T2 x2]; NormArg[A2, T2]; ReadVal[x2, Var[Arg[A2]]];
    …
    Norm[Tn xn]; NormArg[An, Tn]; ReadVal[xn, Var[Arg[An]]];
    x(Mod[A1] x1, Mod[A2] x2, … , Mod[An] xn)
```

Доступ к элементу. Пусть `E` – выражение, имеющее тип массива, T_1, T_2, …, $T_n \in$ {`int`, `uint`, `long`, `ulong`} – типы, к которым неявно могут быть преобразованы типы выражений E_1, E_2, …, E_n. Тогда

```
Norm[E[E1, E2, … En]] ::=
    Norm[TE x];    Norm[E];        ReadVal[x, Var[E]];
    Norm[T1 x1];   Norm[(T1) E1];  ReadVal[x1, false];
    Norm[T2 x2];   Norm[(T2) E2];  ReadVal[x2, false];
    …
    Norm[Tn xn];   Norm[(Tn) En];  ReadVal[xn, false];
    WriteMemberVal[x, <x1, x2, … xn>]
```

Пусть `E` – выражение, имеющее тип класса, структуры или интерфейса, который реализует применимый индексатор, а T_1, T_2, …, T_n – типы формальных параметров индексатора. Тогда

```
Norm[E[E1, E2, … En]] ::=
    Norm[TE x];    Norm[E];   ReadVal[x, Var[E]];
    Norm[T1 x1];   Norm[(T1) E1];  ReadVal[x1, false];
    Norm[T2 x2];   Norm[(T2) E2];  ReadVal[x2, false];
    …
    Norm[Tn xn];   Norm[(Tn) En];  ReadVal[xn, false];
    x.get_Item(x1, x2, … xn)
```

This-доступ. Если `this` классифицируется как переменная, то

```
Norm[this] ::= WriteVal[this, true]
```

Если `this` классифицируется как значение, то

```
Norm[this] ::= WriteVal[this, false]
```

Base-доступ. Пусть `I` – идентификатор, E_1, E_2, …, E_n – выражения. Тогда

```
Norm[base[E1, E2, … En]] ::=
    Norm[T1 x1];   Norm[(T1) E1];  ReadVal[x1, false];
    Norm[T2 x2];   Norm[(T2) E2];  ReadVal[x2, false];
    …
```

```
    Norm[T_n x_n];   Norm[(T_n) E_n];   ReadVal[x_n, false];
    base.get_Item(x_1, x_2, … x_n)
```

Если `I` – свойство, то

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

Иначе:

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

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

```
Norm[E op] ::=
    Norm[T_E x];    Norm[E];      ReadVal[x, true];
    Norm[T_E y];    T_E.op(x);    ReadVal[y, false];
    MD := upd(MD, MeM(x), MD(MeM(y)));
    WriteVal[x, false]
```

Пусть `E` – выражение, классифицируемое как доступ к свойству `P`. Если доступ к `P` осуществляется через instance-выражение E_P, имеющее тип T_P (т.е. `P` – instance-свойство), то

```
Norm[E op] ::=
    Norm[T_P z];    Norm[E_P];    ReadVal[z, Var[E_P]];
    Norm[T_E x];    z.get_P();    ReadVal[x, false];
    Norm[T_E y];    T_E.op(x);    ReadVal[y, false];
    z.set_P(y);
    WriteVal[x, false]
```

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

```
Norm[E op] ::=
    Norm[T_E x];    T_P.get_P();  ReadVal[x, false];
    Norm[T_E y];    T_E.op(x);    ReadVal[y, false];
    T_P.set_P(y);
    WriteVal[x, false]
```

Пусть `E` – выражение, классифицируемое как доступ к индексатору через instance-выражение E_I, имеющее тип T_I, A_1, A_2, …, A_n – аргументы доступа к индексатору, а T_1, T_2, …, T_n – типы его формальных параметров. Тогда

```
Norm[E op] ::=
    Norm[T_I z];    Norm[E_I];        ReadVal[z, Var[E_I]];
    Norm[T_1 a_1];  Norm[(T_1) A_1];  ReadVal[a_1, false];
```

```
Norm[T2 a2];   Norm[(T2) A2];   ReadVal[a2, false];
…
Norm[Tn an];   Norm[(Tn) An];   ReadVal[an, false];
Norm[TE x];
z.get_Item(a1, a2, … an);
ReadVal[x, false];
Norm[TE y];
TE.op(x);
ReadVal[y, false];
z.set_Item(a1, a2, … an, y);
WriteVal[x, false]
```

Операция new. Определим Norm на выражении создания объекта. Пусть A_1, A_2, …, A_n – аргументы выражения создания объекта, а T_1, T_2, …, T_n – типы формальных параметров конструктора, применимого к аргументам A_1, A_2, …, A_n. Если T – value-тип, то

```
Norm[new T(A1, A2, … An)] ::=
    Norm[T1 x1]; NormArg[A1, T1]; ReadVal[x1, Var[Arg[A1]]];
    Norm[T2 x2]; NormArg[A2, T2]; ReadVal[x2, Var[Arg[A2]]];
    …
    Norm[Tn xn]; NormArg[An, Tn]; ReadVal[xn, Var[Arg[An]]];
    Init(T);
    new_instance(x);
    TP := upd(TP, x, T);
    TP := upd(TP, MeM(x), T);
    x.T(Mod[A1] x1, Mod[A2] x2, … , Mod[An] xn);
    WriteVal[x, false]
```

Иначе:

```
Norm[new T(A1, A2, … An)] ::=
    Norm[T1 x1]; NormArg[A1, T1]; ReadVal[x1, Var[Arg[A1]]];
    Norm[T2 x2]; NormArg[A2, T2]; ReadVal[x2, Var[Arg[A2]]];
    …
    Norm[Tn xn]; NormArg[An, Tn]; ReadVal[xn, Var[Arg[An]]];
    new_instance(x);
    TP := upd(TP, x, T);
    Init(T);
    new_instance(y);
    TP := upd(TP, MeM(y), T);
    MD := upd(MD, MeM(x), MeM(y));
    x.IFI();
```

```
x.T(Mod[A1] x1, Mod[A2] x2, … , Mod[An] xn);
WriteVal[x, false]
```

Теперь определим Norm для выражения создания массива. Пусть T_1, T_2, …, $T_n \in$ {int, uint, long, ulong} – типы, к которым неявно могут быть преобразованы типы выражений E_1, E_2, …, E_n, и RS_{opt} – необязательные спецификаторы размерности массива. Тогда

```
Norm[new T[E1, E2, … En] RSopt] ::=
    Norm[T1 x1];   Norm[(T1) E1];   ReadVal[x1, false];
    Norm[T2 x2];   Norm[(T2) E2];   ReadVal[x2, false];
    …
    Norm[Tn xn];   Norm[(Tn) En];   ReadVal[xn, false];

    Norm[bool b];
    Norm[(x1 < 0) || (x2 < 0) || … || (xn < 0)];
    ReadVal[b, false];
    if (b)
    {
        throw new System.OverflowException();
    }

    new_instance(x);
    TP := upd(TP, x, T[x1, x2, … xn] RSopt);

    Init(T[x1, x2, … xn] RSopt);
    new_instance(y);
    TP := upd(TP, MeM(y), T[x1, x2, … xn] RSopt);
    MD := upd(MD, MeM(x), MeM(y));

    for (int i1 = 0; i1 < x1; i1++)
        for (int i2 = 0; i2 < x2; i2++)
            …
            for (int in = 0; in < xn; in++)
            {
                DefaultValue[x[i1, i2, … in]];
            }

    WriteVal[x, false]
```

Пусть *arr-init*$|A_{j1, j2, \ldots, jN}|1 \le j1 \le y1, \ldots, 1 \le jN \le yN$ – инициализатор N-мерного массива. Тогда

```
Norm[new T[E1, E2, … EN] RSopt arr-init|Aj1, j2, …, jN|] ::=
    Norm[T1 x1];   Norm[(T1) E1];   ReadVal[x1, false];
    Norm[T2 x2];   Norm[(T2) E2];   ReadVal[x2, false];
    …
    Norm[TN xN];   Norm[(TN) EN];   ReadVal[xN, false];

    Norm[bool b];
    Norm[(x1 < 0) || (x2 < 0) || … || (xN < 0)];
    ReadVal[b, false];
    if (b)
    {
        throw new System.OverflowException();
    }

    new_instance(x);
    TP := upd(TP, x, T[x1, x2, … xN] RSopt);

    Init(T[x1, x2, … xN] RSopt);
    new_instance(y);
    TP := upd(TP, MeM(y), T[x1, x2, … xN] RSopt);
    MD := upd(MD, MeM(x), MeM(y));

    for (int i1 = 0; i1 < x1; i1++)
        for (int i2 = 0; i2 < x2; i2++)
            …
            for (int iN = 0; iN < xN; iN++)
            {
                DefaultValue[x[i1, i2, … iN]];
            }

    Norm[T RSopt  a];

    Norm[A1, 1, …, 1];
    ReadVal[a, Var[A1, 1, …, 1]];
    MD := upd(MD,
              MeM(x),
              upd(MD(MeM(x)),
                  <0, 0, … 0>,
                  MD(MeM(a))));
    Norm[A1, 1, …, 2];
    ReadVal[a, Var[A1, 1, …, 2]];
    MD := upd(MD,
```

```
                MeM(x),
                upd(MD(MeM(x)),
                    <0, 0, … 1>,
                    MD(MeM(a))));
    …
    Norm[A_{y1, y2, …, yN}];
    ReadVal[a, Var[A_{y1, y2, …, yN}]];
    MD := upd(MD,
                MeM(x),
                upd(MD(MeM(x)),
                    <y1 - 1, y2 - 1, … yN - 1>,
                    MD(MeM(a))));
    WriteVal[x, false]

Norm[new T [,, …] RSopt arr-init|A_{j1, j2, …, jN}|1 ≤ j1 ≤ y1, …, 1 ≤ jN ≤ yN] ::=
    Norm[new T[y1, y2, … yN] RSopt arr-init|A_{j1, j2, …, jN}|]
```

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

```
Norm[new D(E))] ::=
    new_instance(x);
    TP := upd(TP, x, D);
    Init(D);
    new_instance(y);
    TP := upd(TP, MeM(y), D);
    MD := upd(MD, MeM(x), MeM(y));
    MD := upd(MD,
                MeM(y),
                add(MD(MeM(y)),
                    <MD(MeM(MGType[E])), MGMethod[E]>));
    WriteVal[x, false]
```

Если аргументом выражения создания делегата является группа методов, состоящая из instance-метода, а ее instance-выражение $E_м$ имеет ссылочный тип $T_м$, то

```
Norm[new D(E))] ::=
    Norm[Tм o];    Norm[Eм];    ReadVal[o, Var[Eм]];
    Norm[bool b];
    MD := upd(MD, b, eq(MD(MeM(o)), null));
    if (b)
```

```
    {
        throw new System.NullReferenceException();
    }
    new_instance(x);
    TP := upd(TP, x, D);
    Init(D);
    new_instance(y);
    TP := upd(TP, MeM(y), D);
    MD := upd(MD, MeM(x), MeM(y));
    MD := upd(MD,
              MeM(y),
              add(MD(MeM(y)),
                  <MD(MD(MeM(o))), MGMethod[E]>));
    WriteVal[x, false]
```

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

```
Norm[new D(E))] ::=
    Norm[Tм z];    Norm[Eм];     ReadVal[z, Var[Eм]];
    Norm[object o];
    Norm[(object) z];
    ReadVal[o, false];
    new_instance(x);
    TP := upd(TP, x, D);
    Init(D);
    new_instance(y);
    TP := upd(TP, MeM(y), D);
    MD := upd(MD, MeM(x), MeM(y));
    MD := upd(MD,
              MeM(y),
              add(MD(MeM(y)),
                  <MD(MD(MeM(o))), MGMethod[E]>));
    WriteVal[x, false]
```

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

```
Norm[new D(E)] ::=
    Norm[D z];    Norm[E];        ReadVal[z, Var[E]];
    Norm[bool b];
    MD := upd(MD, b, eq(MD(MeM(z)), null));
```

```
if (b)
{
    throw new System.NullReferenceException();
}
new_instance(x);
TP := upd(TP, x, D);
Init(D);
new_instance(y);
TP := upd(TP, MeM(y), D);
MD := upd(MD, MeM(y), MD(MD(MeM(z))));
WriteVal[x, false]
```

Операция typeof. Пусть T – тип. Тогда

```
Norm[typeof(T)] ::= typeof(T)
```

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

Унарные операции

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

```
Norm[op E] ::= Norm[TE x];
               Norm[E];
               ReadVal[x, Var[E]];
               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];
               ReadVal[y, false];
               Tx.op(y)
```

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

```
Norm[op E] ::=
    Norm[TE x];    Norm[E];     ReadVal[x, true];
    Norm[TE y];    TE.op(x);    ReadVal[y, false];
    MD := upd(MD, MeM(x), MD(MeM(y)));
    WriteVal[y, false]
```

Пусть E – выражение, классифицируемое как доступ к свойству P. Если доступ к P осуществляется через instance-выражение E_P, имеющее тип T_P (т.е. P – instance-свойство), то

```
Norm[op E] ::=
    Norm[TP z];    Norm[EP];     ReadVal[z, Var[EP]];
    Norm[TE x];    z.get_P();    ReadVal[x, false];
    Norm[TE y];    TE.op(x);     ReadVal[y, false];
    z.set_P(y);
    WriteVal[y, false]
```

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

```
Norm[op E] ::=
    Norm[TE x];    TP.get_P();   ReadVal[x, false];
    Norm[TE y];    TE.op(x);     ReadVal[y, false];
    TP.set_P(y);
    WriteVal[y, false]
```

Пусть E – выражение, классифицируемое как доступ к индексатору через instance-выражение E_I, имеющее тип T_I, A_1, A_2, …, A_n – аргументы доступа к индексатору, а T_1, T_2, …, T_n – типы его формальных параметров. Тогда

```
Norm[op E] ::=
    Norm[TI z];    Norm[EI];         ReadVal[z, Var[EI]];
    Norm[T1 a1];   Norm[(T1) A1];    ReadVal[a1, false];
    Norm[T2 a2];   Norm[(T2) A2];    ReadVal[a2, false];
    …
    Norm[Tn an];   Norm[(Tn) An];    ReadVal[an, false];
    Norm[TE x];
    TI.get_Item(a1, a2, … an);
    ReadVal[x, false];
    Norm[TE y];
    TE.op(x);
    ReadVal[y, false];
    TI.set_Item(a1, a2, … an, y);
    WriteVal[y, false]
```

Операция приведения

Пусть T – тип, E – выражение, отличное от локальной переменной. Тогда

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

```
ReadVal[x, Var[E]];
Norm[(T) x]
```

Пусть x – локальная переменная типа S, а T – тип, к которому следует привести x. Рассмотрим теперь все случаи явных и неявных операций приведения (преобразований) по порядку.

Единичные преобразования. Пусть S = T. Тогда

```
Norm[(T) x] ::= Norm[x]
```

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

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

В этих случаях

```
Norm[(T) x] ::= WriteVal[x, false]
```

Неявные преобразования типов перечисления. Пусть T – произвольный тип перечисления. Тогда

```
Norm[(T) 0] ::= MD := upd(MD, Val, 0)
```

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

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

В этих случаях

```
Norm[(T) x] ::= WriteVal[x, false]
```

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

```
Norm[(T) x] ::=
    new_instance(y);
    TP := upd(TP, MeM(y), S);
    MD := upd(MD, MeM(y), MD(MeM(x)));
    MD := upd(MD, Val, MeM(y))
```

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

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

В этих случаях

```
Norm[(T) x] ::= WriteVal[x, false]
```

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, T ∈ {char, sbyte, byte, short, ushort, int, uint, long, ulong}, т.е. S и T – целые типы;
- S ∈ {float, double, decimal}, T – целый тип;
- S = double, T = float;
- S ∈ {float, double}, T = decimal;
- S = decimal, T ∈ {float, double}.

В этих случаях

```
Norm[(T) x] ::=
    MD := upd(MD, Val, γ(S, T, MD(MeM(x))))
```

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

```
Norm[(T) x] ::=
    MD := upd(MD, Val, γ(S, UT(T), MD(MeM(x))))
```

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

```
Norm[(T) x] ::=
    MD := upd(MD, Val, γ(UT(S), T, MD(MeM(x))))
```

Если S и T – типы перечисления, то

```
Norm[(T) x] ::=
    MD := upd(MD, Val, γ(UT(S), UT(T), MD(MeM(x))))
```

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

- S = object, T – ссылочный тип;

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

В этих случаях

```
Norm[(T) x] ::=
    Norm[bool b];
    MD := upd(MD,
              MeM(b),
              and(not(eq(MD(MeM(x)), null)),
                  not(can_cast(TP(MD(MeM(x))), T))));
    if (b)
    {
        Norm[System.InvalidCastException y];
        Norm[new System.InvalidCastException()];
        ReadVal[y, false];
        MD := upd(MD, Exc, MD(MeM(y)));
    }
    else
    {
        WriteVal[x, false];
    }
```

Unboxing-преобразования. Пусть T – value-тип, а S – тип object или интерфейс, реализуемый value-типом T. Тогда

```
Norm[(T) x] ::=
    Norm[bool b];
    MD := upd(MD,
              MeM(b),
              or(eq(MD(MeM(x)), null),
                 not(eq(TP(MD(MeM(x))), T))));
    if (b)
    {
        Norm[System.InvalidCastException y];
        Norm[new System.InvalidCastException()];
        ReadVal[y, false];
        MD := upd(MD, Exc, MD(MeM(y)));
    }
    else
    {
        MD := upd(MD, Val, MD(MD(MeM(x))));
    }
```

Пользовательские преобразования. Если для преобразования из S в T существует пользовательское преобразование $u_{S,T}$: $S_0 \to T_0$, объявленное в типе $U \in \{S, T\}$, то

```
Norm[(T) x] ::=
    Norm[S0 x1];   Norm[(S0) x];   ReadVal[x1, false];
    Norm[T0 x2];   U.us,T(x1);     ReadVal[x2, false];
    Norm[(T) x2]
```

Бинарные операции

Пусть *op* ∈ {&, ^, |, ==, !=, <, >, <=, >=, <<, >>, *, /, %, +, -}, 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];     ReadVal[x, Var[E1]];
    Norm[T2 y];      Norm[E2];     ReadVal[y, Var[E2]];
    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]; ReadVal[x1, false];
    Norm[Targy y1];    Norm[(Targy) y]; ReadVal[y1, false];
    U.op(x1, y1)
```

Операции проверки типов

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

```
Norm[E is T] ::= Norm[TE x];
                 Norm[E];
                 ReadVal[x, Var[E]];
                 x.is(T)
```

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

```
Norm[E as T] ::= Norm[TE x];
                 Norm[E];
                 ReadVal[x, Var[E]];
                 x.as(T)
```

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];        ReadVal[x, Var[E1]];
    Norm[bool b]; Norm[(bool) x]; ReadVal[b, false];
    if (b)
    { WriteVal[true, false]; }
    else
    { Norm[(bool) E2]; }

Norm[E1 && E2] ::=
    Norm[T1 x];    Norm[E1];        ReadVal[x, Var[E1]];
```

```
    Norm[bool b]; Norm[(bool) x]; ReadVal[b, false];
    if (b)
    { Norm[(bool) E2]; }
    else
    { WriteVal[false, 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];        ReadVal[x, Var[E1]];
    Norm[T y];     Norm[(T) x];     ReadVal[y, false];
    Norm[bool b]; T.true(y);        ReadVal[b, false];
    if (b)
    { Norm[(T) x]; }
    else
    { Norm[T.|(x, E2)]; }

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

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

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];           ReadVal[x, Var[E]];
    Norm[bool b]; Norm[(bool) x];     ReadVal[b, false];
    if (b)
    { Norm[(T) E1]; }
    else
    { Norm[(T) E2]; }
```

где T ∈ {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];           ReadVal[x, Var[E]];
    Norm[bool b]; Norm[TE.true(x)];  ReadVal[b, false];
    if (b)
    { Norm[(T) E1]; }
    else
    { Norm[(T) E2]; }
```

где T ∈ {T_1, T_2} – тип условного выражения.

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[TU x];    Norm[U];        ReadVal[x, true];
    Norm[TU y];    Norm[(TU) E];   ReadVal[y, false];
    Norm[bool b];
    MD := upd(MD,
              MeM(b),
              and(not(eq(MD(MeM(y)),
                         null)),
                  not(can_cast(TP(MD(MeM(y))),
                               TP(MD(MeM(x)))))));
    if (b)
    {
        Norm[System.ArrayTypeMismatchException z];
        Norm[new System.ArrayTypeMismatchException()];
        ReadVal[z, false];
        MD := upd(MD, Exc, MD(MeM(z)));
    }
    else
    {
        MD := upd(MD, MeM(x), MD(MeM(y)));
        WriteVal[y, false];
    }
```

```
Norm[U op= E] ::=
    Norm[T_U x];    Norm[U];        ReadVal[x, true];
    Norm[T_E y];    Norm[E];        ReadVal[y, Var[E]];
    Norm[T_op z_0];  Norm[x op y];   ReadVal[z_0, false];
    Norm[T_U z];    Norm[(T_U) z_0];  ReadVal[z, false];
    Norm[bool b];
    MD := upd(MD,
              MeM(b),
              and(not(eq(MD(MeM(z)),
                         null)),
                  not(can_cast(TP(MD(MeM(z))),
                               TP(MD(MeM(x)))))));
    if (b)
    {
        Norm[System.ArrayTypeMismatchException z_1];
        Norm[new System.ArrayTypeMismatchException()];
        ReadVal[z_1, false];
        MD := upd(MD, Exc, MD(MeM(z_1)));
    }
    else
    {
        MD := upd(MD, MeM(x), MD(MeM(z)));
        WriteVal[z, false];
    }
```

Иначе:

```
Norm[U = E] ::=
    Norm[T_U x];    Norm[U];        ReadVal[x, true];
    Norm[T_U y];    Norm[(T_U) E];  ReadVal[y, false];
    MD := upd(MD, MeM(x), MD(MeM(y)));
    WriteVal[y, false]

Norm[U op= E] ::=
    Norm[T_U x];    Norm[U];        ReadVal[x, true];
    Norm[T_E y];    Norm[E];        ReadVal[y, Var[E]];
    Norm[T_op z_0];  Norm[x op y];   ReadVal[z_0, false];
    Norm[T_U z];    Norm[(T_U) z_0];  ReadVal[z, false];
    MD := upd(MD, MeM(x), MD(MeM(z)));
    WriteVal[z, false]
```

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

```
Norm[U = E] ::=
    Norm[T_P z];    Norm[E_P];        ReadVal[z, Var[E_P]];
    Norm[T_U y];    Norm[(T_U) E];    ReadVal[y, false];
    z.set_P(y);
    WriteVal[y, false]

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

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

```
Norm[U = E] ::=
    Norm[T_U y];    Norm[(T_U) E];    ReadVal[y, false];
    T_P.set_P(y);
    WriteVal[y, false]

Norm[U op= E] ::=
    Norm[T_U x];    T_P.get_P();      ReadVal[x, false];
    Norm[T_E y];    Norm[E];          ReadVal[y, Var[E]];
    Norm[T_op z_0]; Norm[x op y];     ReadVal[z_0, false];
    Norm[T_U z];    Norm[(T_U) z_0];  ReadVal[z, false];
    T_P.set_P(z);
    WriteVal[z, false]
```

Пусть U – выражение, классифицируемое как доступ к индексатору через instance-выражение E_I, имеющее тип T_I, A_1, A_2, …, A_n – аргументы доступа к индексатору, а T_1, T_2, …, T_n – типы его формальных параметров. Тогда

```
Norm[U = E] ::=
    Norm[T_I z];    Norm[E_I];        ReadVal[z, Var[E_I]];
    Norm[T_1 a_1];  Norm[(T_1) A_1];  ReadVal[a_1, false];
    Norm[T_2 a_2];  Norm[(T_2) A_2];  ReadVal[a_2, false];
    …
    Norm[T_n a_n];  Norm[(T_n) A_n];  ReadVal[a_n, false];
```

```
    Norm[TU y];    Norm[(TU) E];   ReadVal[y, false];
    z.set_Item(a1, a2, … an, y);
    WriteVal[y, false]

Norm[U op= E] ::=
    Norm[TI w];    Norm[EI];       ReadVal[w, Var[EI]];
    Norm[T1 a1];   Norm[(T1) A1];  ReadVal[a1, false];
    Norm[T2 a2];   Norm[(T2) A2];  ReadVal[a2, false];
    …
    Norm[Tn an];   Norm[(Tn) An];  ReadVal[an, false];
    Norm[TU x];
    w.get_Item(a1, a2, … an);
    ReadVal[x, false];
    Norm[TE y];    Norm[E];        ReadVal[y, Var[E]];
    Norm[Top z0];  Norm[x op y];   ReadVal[z0, false];
    Norm[TU z];    Norm[(TU) z0];  ReadVal[z, false];
    w.set_Item(a1, a2, … an, z);
    WriteVal[z, false]
```

Пусть `U` – выражение, классифицируемое как доступ к событию `Evt`, в объявлении которого присутствуют аксессоры `add` и `remove`. Если доступ к `Evt` осуществляется через instance-выражение E_{Evt}, имеющее тип T_{Evt} (т.е. `Evt` – instance-событие), то

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

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

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

```
Norm[U += E] ::=
    Norm[TE x];    Norm[E];        ReadVal[x, Var[E]];
    Norm[TU y];    Norm[(TU) x];   ReadVal[y, false];
    TEvt.add_Evt(y);
```

```
Norm[U -= E] ::=
    Norm[TE x];    Norm[E];        ReadVal[x, Var[E]];
    Norm[TU y];    Norm[(TU) x];   ReadVal[y, false];
    TEvt.remove_Evt(y);
```

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 (Ainitializer; ; Aiterator) {A}
```

заменяется фрагментом

```
for (Ainitializer; true; Aiterator) {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}
```

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, `Tb` – тип выражения `b`, `x` – новая локальная переменная.

Далее, рассмотрим правило нормализации тел функциональных членов.

Тела функциональных членов

Функциональными членами являются методы, свойства, события, индексаторы, операции, статические конструкторы, instance-конструкторы и

деструкторы. 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`.

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 = GT_LABELS.none;
}
```

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

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

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

3. В перечисление `GT_LABELS` добавляются новые различные элементы $\mathtt{l}_1$, …, $\mathtt{l}_n$. Определим функцию v: $Lab \to$ `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 выполнения 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 $CaseL: V \times \mathtt{N} \to Lab$ и $DefL: \mathtt{N} \to Lab$ между метками switch-операторов и новыми метками следующим образом. Для всех меток вида `case E:`, входящих в *switch-labels* оператора S_i, $CaseL(\mathtt{E}, i)$ `:= 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 $\mathtt{l}_1, \ldots, \mathtt{l}_N$, $N = |Lab^{new}|$. Пусть функция $v: Lab^{new} \to$ `GT_LABELS` устанавливает взаимнооднозначное соответствие между элементами множества Lab^{new} и $\mathtt{l}_1, \ldots, \mathtt{l}_N$.
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(f_{j+1}) {
    A
    if (GT.gt != GT_LABELS.none) goto f_j;
}
```

где 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 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 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.

Алгоритмы элиминации

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

Оператор-объявление

Для нормализации операторов-выражений (см. 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.

Локальные объявления

Локальные переменные. Пусть `I` – идентификатор, `T` – тип. Если `T` является value-типом, то

```
Norm[T I] ::= Init(T);
              new_instance(I);
              TP := upd(TP, I, T);
              TP := upd(TP, MeM(I), T)
```

Иначе:

```
Norm[T I] ::= new_instance(I);
              TP := upd(TP, I, T)
```

Локальные константы. Пусть `I` – идентификатор, `T` – тип, допустимый в объявлениях локальных констант, т.е. `T` – ссылочный тип, тип перечисления или `T` ∈ {`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `char`, `float`, `double`, `decimal`, `bool`, `string`}, `E` – константное выражение. Если `T` является value-типом, то

```
Norm[const T I = E] ::=
    Init(const T);
    new_instance(I);
    TP := upd(TP, I, const T);
    TP := upd(TP, MeM(I), const T);
    Norm[E];
    ReadVal[I, Var[E]]
```

Иначе:

```
Norm[const T I = E] ::=
    new_instance(I);
    TP := upd(TP, I, T)
    Norm[E];
    ReadVal[I, Var[E]]
```

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.

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.

Оператор return

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

Правило ERET1. Фрагмент вида

```
return e;
```

заменяется фрагментом

```
{ e; goto __ExitPoint; }
```

Правило ERET2. Фрагмент вида

```
return;
```

заменяется фрагментом

```
{ goto __ExitPoint; }
```

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; ReadVal[x, Var[e]];
    Norm[bool b];
    MD := upd(MD, b, eq(MD(MeM(x)), null));
    if (b)
    {
        Norm[System.NullReferenceException y];
        Norm[new System.NullReferenceException()];
        ReadVal[y, false];
        MD := upd(MD, Exc, MD(MeM(y)));
    }
    else
    {
        MD := upd(MD, Exc, MD(MeM(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

```
{ MD := upd(MD, Exc, MD(MeM(eSaved))); }
```

Оператор foreach

Оператор `foreach` преобразуется в оператор `for`, чтобы затем применить правила элиминации `for`. Замена происходит в точности по спецификации исполнения цикла `foreach` [4].

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.

Onepamop 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

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

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.

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.

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
        MD := upd(MD, Exc, MD(MeM(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
        MD := upd(MD, Exc, MD(MeM(x)));
        goto M;
    }
    {B}
    M:;
```

где `M` – новая метка.

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 v1: … case vn: 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 == v1)||…||(x == vn))
             { 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.

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.

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.

ЗАКЛЮЧЕНИЕ

В данной работе представлен перевод из языка C#-light в язык C#-kernel в рамках трехуровневого подхода к верификации C#-программ [6]. Ранее подобная проблема исследовалась в работах [1, 7] для языка C. Процесс

перевода имеет вид последовательного применения алгоритмов перевода, которые строятся на базе 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 алгоритмы, как, например, элиминация операторов `break`, `continue`, `return`. Ко второй группе можно отнести элиминацию директив `using` и пространств имен. В третью группу попадает 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, что отличает их от обычных локальных алгоритмов.

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 [2], 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. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации С-программ. Язык C-light и его формальная семантика // Программирование. — 2002. — № 6. — С. 19-30.
2. **Непомнящий В.А., Ануреев И.С., Михайлов И.Н., Промский А.В.** На пути к верификации С-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).
3. **Börger E.**, **Schulte W.** A programmer friendly modular definition of the semantics of Java // Lect. Notes Comput. Sci. — 1999. — Vol. 1523. — P. 353–404.
4. Standard ECMA-334 — C# Language Specification. — December 2001. — http://www.ecma.ch.
5. **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.
6. **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.
7. **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.
8. **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

ВЕРИФИКАЦИЯ C#-ПРОГРАММ: ПЕРЕВОД ИЗ ЯЗЫКА C#-LIGHT В ЯЗЫК C#-KERNEL

Препринт
120

Рукопись поступила в редакцию 04.11.04
Рецензент И.С. Ануреев
Редактор З.В. Скок

Подписано в печать 14.12.04
Формат бумаги 60 × 84 1/16 Объем 3.4 уч.-изд.л., 3.8 п.л.
Тираж 60 экз.

ЗАО РИЦ «Прайс-курьер»
630090, г. Новосибирск, пр. Акад. Лаврентьева, 6, тел. (383-2) 34-22-02