Русскоязычный перевод документации Eiffel по технологии безопасности void safety

Аннотация. При проектировании структур данных в ПО часто приходится иметь дело с несуществующими ссылками (null в терминологии C/C++, Void в терминологии Eiffel). Распространенной проблемой при этом является разыменование пустого указателя, что приводит к ошибкам времени выполнения. В статье рассмотрены вопросы безопасности ПО, а также решение этой проблемы в языке Eiffel на основе понятий прикрепленных и открепляемых типов, а также сертифицированных шаблонов прикрепления. Предложенное решение совместимо с уже существующим программным кодом, не являющимся безопасным, а также полностью согласовано с остальными элементами и концепциями языка Eiffel.

Безопасность – предпосылки, определение, инструменты

Void-safety: Background, definition, and tools

Перевод: Семченков Сергей
Редактор перевода: Когтенков Александр

Содержание

1. Предпосылки

В Eiffel основное внимание уделено качеству ПО. Технология безопасности, как и статическая типизация, – это еще одна возможность для улучшения качества ПО. Безопасное ПО защищено от ошибок времени выполнения по несуществующим ссылкам, а следовательно более надежно, чем ПО, в котором такие вызовы допускаются. Можно провести аналогию со статической типизацией. Возможности безопасности могут рассматриваться как расширение системы типов и как следующий шаг статической типизации, так как механизм для обеспечения безопасности интегрирован в систему типов.

1.1 Статическая типизация

Известно, что статическая типизация устраняет целый класс ошибок ПО. Это достигается путем проверки во время компиляции вызовов вида

```
x.f (a)
```

Такой вызов компонента является допустимым во время компиляции, если тип x содержит компонент f, и любые аргументы, обозначаемые через a, совпадают по количеству с формальными аргументами f, а также совместимы с ними по типам.

В статически типизированных языках, таких как Eiffel, компилятор гарантирует невозможность возникновения ситуации, при которой компонент f не применим к объекту, прикрепленному к x. Если Вы когда-либо программировали на языке Smalltalk, Вы должны быть знакомы с ошибкой подобного рода, которая проявляется как "Сообщение не распознано". Так происходит потому, что Smalltalk не является статически типизированным языком.

1.2 ПО, не являющееся безопасным

Статическая типизация позволяет удостовериться, что в рассмотренном примере существует определенный компонент f, который может быть применен во время выполнения к x. Но она не позволяет проверить, что в любое время выполнения x.f (a) будет существовать объект, к которому прикреплена x.

Эта проблема не уникальна для Eiffel. Другие среды, которые поддерживают семантику ссылок на объекты, также допускают небезопасные вызовы времени выполнения. Если Вы работали с Java и .NET, Вы могли встретить исключение NullReferenceException. Иногда эта ошибка выражена в более благозвучном сообщении: "Ссылка на объект не установлена на экземпляр объекта". В Eiffel Вы увидите сообщение "Вызов компонента для несуществующей цели". Это признак ошибок времени выполнения ПО, не являющегося безопасным.

> **Примечание.** Если Вам нужен обзор различий между ссылочными и развернутыми типами, смотрите главу из руководства Eiffel, посвященного модели выполнения Eiffel.

Конечно, это не является проблемой для экземпляров развернутых (expanded) типов, так как эти экземпляры "разворачиваются" внутри родительских объектов. Однако невозможно построить все ПО только на развернутых типах. Ссылки являются важными с точки зрения производительности, а также для целей моделирования. Например, рассмотрим автомобиль, который имеет двигатель и производителя. Когда мы моделируем автомобили, для двигателей может быть подходящим использование развернутых типов, так как каждый автомобиль имеет один двигатель. С другой стороны несколько автомобилей будут разделять одного производителя посредством ссылки.

Поэтому ссылки необходимы, но не должны создавать проблем.

1.3 Безопасное ПО

Безопасное ПО – это такое ПО, в котором компилятор, используя статический анализ кода, может дать гарантию того, что во время выполнения при применении компонента к ссылке, эта ссылка будет иметь прикрепленный к ней объект. Это означает, что вызов компонента

```
x.f (a)
```

является допустимым только в случае, если мы гарантировали, что x будет прикреплена к объекту, когда происходит выполнение вызова.

Примечание. Это правило называется **правилом цели**, оно является главным правилом безопасности. В последующем обсуждении встретятся и другие правила. Формальное определение всех правил корректности можно посмотреть в стандарте ISO/ECMA, доступном онлайн.

Если мы принимаем это правило, то должны иметь стратегию для компиляции с использованием этого правила.

2. Элементы стратегии безопасности

Рассмотрим инструменты для создания безопасного ПО. Каждый их них будет подробно описан далее. Изучение этих элементов поможет рассмотреть вопрос с точки зрения компилятора, от которого мы ожидаем гарантий того, что код является безопасным.

Для начала рассмотрим несколько неработающих подходов.

Можно принудительно удовлетворить правило цели, если исключить концепцию несуществующих ссылок. Однако это нереально на практике. Void – это значимая абстракция, полезная во многих ситуациях, таких как void ссылки в структурах. Поэтому мы должны сохранить void, не нарушая безопасности.

Еще одна возможная идея – заставить компилятор сделать всю работу за нас. В этом случае компилятору понадобятся недопустимо большие временные затраты для того, чтобы проанализировать все потенциальные пути выполнения в программной системе и убедиться, что каждый потенциально возможный вызов компонента был сделан по прикрепленной ссылке.

Всё сводится к тому, что необходимо предпринять определенные действия для помощи компилятору. Об этом пойдет речь в следующих разделах.

2.1 Сертифицированные Шаблоны Прикрепления (СШП)

Известно, что в контексте определенных шаблонов кода ссылка не может принимать значение **Void**. Эти шаблоны уже выявлены, мы называем их СШП – Сертифицированные Шаблоны Прикрепления (Certified Attachment Patterns). Рассмотрим простой пример, синтаксис которого знаком всем разработчикам на языке Eiffel.

```
if x /= Void then
    -- ... Любые операторы, не содержащие присваивания ссылке x
    x.f (a)
end
```

Проверка обеспечивает, что x не равно **Void**. До тех пор, пока нет присваиваний ссылке x, компонент f может быть применен к x с гарантией того, что x всё время прикреплена. Причём это можно определить во время компиляции. Будем говорить, что этот шаблон кода является СШП для x.

Следует учитывать, что в этом примере (а также с другими СШП) x может быть только локальной переменной или формальным аргументом. x не может быть атрибутом или выражением (за исключением одного случая, рассмотренного далее). Непосредственный доступ к атрибутам класса не может быть разрешен с помощью СШП потому, что им может быть присвоено **Void** путем выполнения подпрограммы, вызванной промежуточными инструкциями или даже другим потоком. В разделе 2.3 будет показано, что это не является ограничением, как может показаться вначале.

Примечание. Дополнительную информацию о СШП можно найти в разделе "Дополнительная информация о СШП". Раздел "Из чего состоит СШП" содержит информацию о том, как можно определить СШП в коде.

2.2 Синтаксис прикрепления (тест объекта)

С точки зрения безопасности, **синтаксис прикрепления** выполняет две функции. Он позволяет убедиться в том, что ссылка является прикрепленной, а также обеспечивает безопасный доступ к объектам, прикрепленным к атрибутам класса. Ранее мы отметили, что код

```
if x /= Void then
    -- ... Любые операторы, не содержащие присваивания ссылке x
    x.f (a)
end
```

является СШП для вызова компонента на x только в том случае, если x является локальной переменной или формальным аргументом.

Используя **синтаксис прикрепления**, мы можем выполнить **тест объекта** для любого выражения. Синтаксис прикрепления – это выражение логического типа (BOOLEAN), с помощью которого можно ответить на вопрос: "выражение прикреплено к объекту?" (В нашем случае – “переменная x прикреплена к объекту?”) Если это так, то мы получаем новую локальную переменную, прикрепленную к тому же самому объекту, что и x, к этой переменной можно применять вызовы компонентов.

```
if attached x as l_x then
    l_x.f (a)
end
```

В этом примере x проверяется на прикрепление к объекту. Если x прикреплена, то новая локальная переменная l_x прикрепляется к тому же объекту, что и x. Объект может быть безопасно использован, даже если x является атрибутом класса. Синтаксис прикрепления является еще одним СШП, так как предоставляет ясную проверку того, что цели вызова не будут равны **Void**.

Примечание. Синтаксис прикрепления имеет другие формы и применения. Подробнее об этом рассказано здесь.

Единственным способом проверить выполнение правила цели является постоянное использование СШП или синтаксиса прикрепления всякий раз, когда необходимо применить компонент к ссылке на объект. Однако такой подход непрактичен, так как не совместим с большим количеством существующего кода на Eiffel.

2.3 Прикрепленные и открепляемые типы

Вместо того, чтобы защищать каждый вызов компонента, Eiffel позволяет объявить любую переменную как имеющую **прикрепленный тип**. Это важное расширение системы типов Eiffel.

До введения возможностей безопасности, в Eiffel любая ссылочная переменная могла получить значение **Void**. Таким образом, все переменные рассматривались как открепляемые.

Текущий вариант Eiffel поддерживает понятия **прикрепленных** и **открепляемых** типов. Когда объявлена переменная прикрепленного типа, как в следующем примере, компилятор предотвращает присваивание ей значения **Void**, или чего угодно, что может принимать значение **Void**.

```
my_attached_string: attached STRING
```

Легко понять, что чем больше объявлено прикрепленных типов, тем легче гарантировать, что вызов к несуществующей цели во время выполнения не будет иметь места. Если бы каждое объявление было гарантированно прикрепленным, то этого было бы достаточно для выполнения правила цели.

Однако такое решение не работает, так как иногда необходимо разрешить ссылки со значением **Void**.

В этом случае объявление может использовать признак откреплямого типа, как в следующем примере.

```
my_detachable_string: detachable STRING
```

Это не означает, что каждое объявление должно содержать признаки прикрепленного или откреплямого типа. Допустимы объявления, не содержащие признака. Следует ли такие объявления рассматривать как прикрепленные или откреплямые, зависит от значения настройки проекта "Являются ли типы прикрепленными по умолчанию?" (Are types attached by default?). Эта настройка может быть выставлена по-разному в разных частях проекта, что удобно при конвертации существующего ПО или смешивании библиотек с разными уровнями безопасности.

Таким образом, в Eiffel все объявления будут иметь или **прикрепленные** или **откреплямые** типы. Как результат, использовать СШП и синтаксис прикрепления необходимо только с откреплямыми типами. Следует помнить, что прямой доступ к атрибутам класса откреплямого типа никогда не является безопасным.

При проектировании безопасного ПО в Eiffel практически в каждом случае лучше всего установить значение настройки "Являются ли типы прикрепленными по умолчанию?" в "Истина" (True). Это означает, что если объявление типа не содержит ни **attached** (прикрепленный), ни **detachable** (откреплямый), то предполагается **attached** (прикрепленный).

2.3.1 Прикрепление и согласованность

Различие между прикрепленными и открепляемыми типами выражается в небольшом, но важном дополнении к правилам согласованности. Поскольку переменные прикрепленных типов никогда не могут принимать значение **Void**, нельзя разрешать присваивание открепляемого источника к прикрепленной цели. Присваивание прикрепленного источника к открепляемой цели допустимо. Следующий код показывает оба случая в предположении, что типы по умолчанию являются прикрепленными.

```
my_attached_string: STRING
my_detachable_string: detachable STRING
    ...
my_attached_string := my_detachable_string    -- Недопустимо
my_detachable_string := my_attached_string    -- Допустимо
```

2.4 Правило инициализации

Если у нас есть прикрепленные типы, то мы можем полагать, что переменные этих типов, будучи единожды прикрепленными, всегда будут прикрепленными. Однако как они прикрепляются в первый раз? Это является главной темой правила инициализации.

Правило говорит о том, что в каждом месте, где доступна переменная, она должна быть **правильно определена**. Правильность определения переменной имеет чёткое, однако не простое определение в стандарте Eiffel.

> **Примечание.** Формальное определение правила инициализации переменных и связанных с ним понятий, таких как правильно определенные переменные, приведено в стандарте ISO/ECMA.

Несложно понять основы инициализации переменных прикрепленных типов:

- Для инициализации атрибутов класса можно применить правило, аналогичное правилу первоначального выполнения инвариантов класса, заключающегося в том, что все должно быть в порядке по завершении процедуры создания. Если атрибут класса – прикрепленного типа, то каждая процедура создания отвечает за то, чтобы атрибут был прикреплен к объекту при её завершении.
- Локальная переменная рассматривается как правильно определенная, если она инициализирована в некоторой точке, **предшествующей** ее использованию для каждого пути выполнения, в котором она используется. Непосредственно после выполнения оператора **create** локальная переменная считается правильно определенной. Однако если **create** встречается в части **then** условного оператора **if**, то локальная переменная не будет правильно определенной в **else** части этого же оператора **if**.

```
my_routine
        -- Пример правильно определенной локальной переменной
    local
        l_my_string: STRING
    do
        if my_condition then
            create l_my_string.make_empty
```

```
            -- ... l_my_string является здесь правильно определенной
        else
            -- ... l_my_string не является здесь правильно определенной
        end
    end
```

- Переменная рассматривается как правильно определенная, если она является **самоинициализируемой**. Об этом речь идет в следующем разделе.

2.5 Самоинициализируемые атрибуты

Самоинициализируемый атрибут гарантированно имеет значение при обращении к нему во время выполнения. Для объявлений самоинициализируемых атрибутов характерно наличие ключевого слова **attribute**. Код, следующий за ключевым словом **attribute**, выполняется для инициализации атрибута в случае, если к атрибуту обращаются прежде, чем он был инициализирован каким-либо другим способом.

Таким образом, самоинициализируемые атрибуты – это обычные атрибуты, которые одновременно принадлежат прикрепленному и ссылочному типу (не являются константами или переменными развернутых типов). Самоинициализируемые атрибуты могут инициализироваться и, как правило, инициализируются традиционными способами. Различие состоит в том, что код после ключевого слова **attribute** гарантирует, что самоинициализируемый атрибут не примет значение **Void**, даже если к нему будет обращение до инициализации одним из традиционных способов.

```
value: STRING
    attribute
        create Result.make_empty
    end
```

В рассмотренном примере атрибут value будет прикреплен к объекту типа STRING (пустая строка), если перед первым использованием он не будет инициализирован как-то ещё.

2.6 Правило согласованности

Система типов Eiffel допускает оператор присваивания

```
x := y
```

только тогда, когда тип y **совместим** с типом x. Совместимость означает либо **конвертируемость** либо **согласованность**.

Тот факт, что все типы являются либо **прикрепленными** либо **открепляемыми**, добавляет еще один пункт к правилу согласованности.

- Если x принадлежит прикрепленному типу, то y также должен быть прикрепленного типа.

Это правило препятствует изменению статуса прикрепленности во время выполнения. Если x принадлежит открепляемому типу, то y может быть открепляемого или прикреплённого типа.

То же самое относится и к вызову подпрограмм. Пусть в вызове

```
z.r (y)
```

формальный аргумент принадлежит прикрепленному типу, тогда и фактический аргумент у должен быть прикрепленного типа.

2.7 Стабильные атрибуты

Стабильные атрибуты являются откреплямыми, так как добавление концепции стабильности имеет смысл только для откреплямых атрибутов. Объявление откреплямого атрибута стабильным означает, что его поведение совпадает с таковым у откреплямых атрибутов, за исключением правил присваивания, которые копируют таковые у прикрепленных атрибутов. Другими словами, стабильные атрибуты не обязаны быть прикреплёнными (как изначально прикреплённые) в процессе создания объекта. Однако как и атрибуты прикрепленных типов, стабильные атрибуты никогда не могут быть целью присваивания, в котором источником является **Void** или откреплямый тип.

```
my_test: detachable TEST
    note
        option: stable
    attribute
    end
```

Это означает, что стабильные атрибуты не обязаны быть инициализированы подобно атрибутам прикрепленных типов, но как только они прикреплены, они не могут принимать значение **Void** снова.

Стабильные атрибуты интересны еще тем, что они являются единственным исключением из правила раздела "Сертифицированные Шаблоны Прикрепления (СШП)", в котором говорится о том, что прямой доступ к атрибутам не может быть защищен СШП. Стабильный атрибут может быть защищен СШП. Это объясняется тем, что как только стабильный атрибут прикреплен к объекту, он не может принимать значение **Void**. Соответственно можно не беспокоиться о неожиданном изменении состояния атрибута из прикрепленного в неприкрепленное вследствие действий других подпрограмм или потоков.

2.8 Правило для параметров классов

Универсальные классы добавляют еще один вопрос. Универсальный класс

```
class
    C [G]
        ...
```

позволяет нам создать тип, указав фактический параметр класса для формального параметра G.

Рассмотрим два допустимых объявления:

```
my_integer_derivation: C [INTEGER]
```

и

```
my_employee_derivation: C [EMPLOYEE]
```

Пусть класс C содержит объявление

```
x: G
```

Что можно сказать о безопасности атрибута x?

В случае типа INTEGER атрибут x является безопасным, так как INTEGER –

развернутый тип. Типы, такие как EMPLOYEE, часто являются ссылочными и могут принимать значение **Void** во время выполнения.

Для классов, подобных C [G], G рассматривается как открепляемый тип. В результате, вследствие правила согласованности, любой класс может быть использован в качестве фактического параметра класса. Это означает, что оба следующих объявления являются допустимыми:

```
my_detachable_string_derivation: C [detachable STRING]
my_attached_string_derivation: C [attached STRING]
```

Если C содержит объявление x: G, то применение компонентов к x должно включать проверку прикрепления (СШП, синтаксис прикрепления и т.д.).

Ограниченная универсальность может быть использована для создания универсальных классов, в которых параметр класса представляет прикрепленный тип. Пусть класс C определен следующим образом:

```
class C [G -> attached ANY]
...
```

Тогда x в классе G представляет собой прикрепленный тип. Следовательно, фактический параметр класса в каждом частном случае должен быть прикреплен, и все вызовы компонента на x безопасны.

2.9 Правило для типа ARRAY

Правило для параметров классов применяется для всех универсальных типов, за исключением типа ARRAY. При обычном создании массива (ARRAY) можно указать минимальный и максимальный индекс.

```
my_array: ARRAY [STRING]
...
create my_array.make (1, 100)
```

В процессе создания также создается область для хранения соответствующего количества элементов. В зависимости от фактического параметра класса эти элементы являются либо объектами для развернутых типов либо ссылками для ссылочных типов.

В случае фактического параметра класса прикрепленного ссылочного типа, все элементы должны быть прикреплены к экземплярам типа в процессе создания массива. Процедура make не обеспечивает этого. Создание массива, в котором фактический параметр класса прикреплен, должно производиться с помощью процедуры make_filled.

```
create my_array.make_filled ("", 1, 100)
```

Первый аргумент – это объект с типом фактического параметра, в данном случае – пустая строка. Каждый элемент в создаваемом массиве будет инициализирован ссылкой на этот объект.

Более подробную информацию о безопасном использовании массивов и других универсальных классов можно получить в разделе "Использование универсальных классов".