Санкт–Петербургский государственный университет
Кафедра системного программирования
Группа 21.Б15-мм

Бучин Вячеслав Андреевич

Обобщенные кандидаты при решении ограничений системы типов .NET

Отчёт по учебной практике
в форме «Решение»

Научный руководитель:
кандидат физико-математических наук
Д.А. Мордвинов
Консультант:
инженер-исследователь кафедры системного программирования
М.П. Костицын

Санкт-Петербург
2023 г.

Содержание

Введение

В современном мире с каждым годом растет объем используемого программного обеспечения (ПО). Неотъемлемой частью разработки которого является обеспечение контроля качества, которое достигается, в частности, с помощью модульного тестирования. Существуют различные способы и подходы, позволяющие автоматизировать данный процесс. Одним из которых является символьное исполнение.

Символьное исполнение [7, 1, 6] — техника, которая представляет из себя точный метод анализа программ в условиях неопределенности входных данных. Они заменяются на *символьные переменные*, которые далее участвуют в процессе исследования. Во время этого процесса возникают различные ограничения на символьные переменные. Среди них можно выделить ограничения на типы. Для их решения чаще всего используются решатели: решатель для типов для ограничений на типы и SMT-решатели (Satisfiability Modulo Theories) [2, 8, 5] для остальных. Если решателю удается подобрать значения символьных переменных, отвечающие ограничением, то исследование продолжается, иначе оно останавливается. Поэтому для увеличения покрытия важно, чтобы решатель чаще находил эти значения в случаях, когда они существуют.

При решении ограничений на типы может возникнуть ситуация, в которой под них подходят исключительно обобщенные типы с некоторыми наборами параметров. Подбор параметров является тяжелой вычислительной задачей, потому что количество их возможных значений может быть потенциально бесконечным, даже если запретить параметрам быть обобщенными типами, оно растет экспоненциально с ростом количества параметров. Поэтому важно как можно сильнее сузить круг перебора параметров при решении ограничений. Кроме того, не любой тип подойдет в качестве значения параметра обобщенного типа, так как параметры могут иметь свои собственные ограничения, обладать различной вариантностью, они могут также зависеть друг от друга нетривиальным образом [3]. Вообще говоря данная проблема является неразрешимой, однако если решение существует, то его всегда возможно можно отыскать [4].

Один из способов решения вышеописанной проблемы — создание системы, позволяющей находить значения параметров обобщенного типа, а также реализация и ее прототипа.

Постановка задачи

Целью учебной практики является увеличение покрытия символьного исполнения. Для достижения цели были поставлены следующие задачи:

- выполнить обзор системы типов .NET,
- разработать систему, позволяющую решателю типов использовать обобщенные типы,
- реализовать эту систему в рамках проекта V#,
- провести тестирование реализованной системы.

1. Обзор

В данной главе проведен обзор системы типов .NET (в разд. 1.1), техники символьного исполнения (в разд. 1.2), а также в разд. 1.3 рассмотрен проект V# в рамках которого выполнена данная работа.

1.1 Система типов .NET

Согласно спецификации CIL (Common Intermediate Language) платформы .NET [3] все типы системы типов .NET можно разделить на следующие.

Базовые типы (англ. value types) являются наследниками класса ValueType. К ним относятся следующие типы:

- знаковые и беззнаковые целые числа (uint, long, sbyte и т.д.);
- числа с плавающей запятой (float, double, decimal);
- bool;
- char;
- перечисления;
- структуры;
- кортежи.

Ссылочные типы. К ним относятся:

- классы;
- интерфейсы;
- делегаты (англ. delegate);
- записи (англ. record).

Обобщенные типы. К ним относятся ссылочные типы или структуры, которые параметризованы другими типами. При объявлении такого типа можно наложить дополнительные ограничения на его параметры. Ограничения могут быть следующего вида:

- является структурой;
- должен быть ссылочным типом;
- не может быть нулевой ссылкой;
- имеет конструктор по умолчанию;
- наследуется от указанного класса;
- реализует указанный интерфейс;
- наследует другой параметр.

1.2 Символьное исполнение

Символьное исполнение — техника исследования программ, которая позволяет исполнять программный код в условиях неопределённости входных данных.

Изначально входные данные заменяются *символьными значениями* — абстракцией, позволяющей исследовать программу на совокупности значений входных данных сразу, а не на каких-то одних.

Во время ветвления обычный вызов функции прошел бы по одной из веток, а символьное исполнение сможет исследовать обе. Условие попадания в каждую из веток накапливается в процессе исследования — его называют *условием пути*. Для функции из лист. 1 при первом ветвлении условия пути будут соответственно $x > 0$ и $x \leq 0$ для каждой из веток.

Часть веток может оказаться недостижимыми. Для проверки данного факта зачастую используются различные решатели. Если ветка оказалась недостижимой, то ее исследование прекращается.

Решатели также позволяют находить значения символьных переменных, отвечающие условию пути. Запустив программу на этих значениях входных

```
public static int Abs(int x)
{
    int y;
    if (x > 0)
        y = x;
    else
        y = -x;

    if (y < 0)
        throw new Exception();

    return y;
}
```

Листинг 1: Пример функции для символьного исполнения

данных, ее исполнение гарантированно попадет в ветку, чье условие пути проверялось решателем. Так, например, во время второго ветвления в функции с лист. 1 может выброситься исключение при вызове `Abs(Integer.MinValue)`.

1.3 Проект V#

V#[1] — инструмент для анализа программ, он исследует CIL (Common Intermediate Language) с использованием техники символьного исполнения. Далее будет рассмотрена архитектура проекта V# (рис. 1). Цветом выделена компонента, модифицированная в рамках данной работы.

[1] https://github.com/VSharp-team/VSharp/

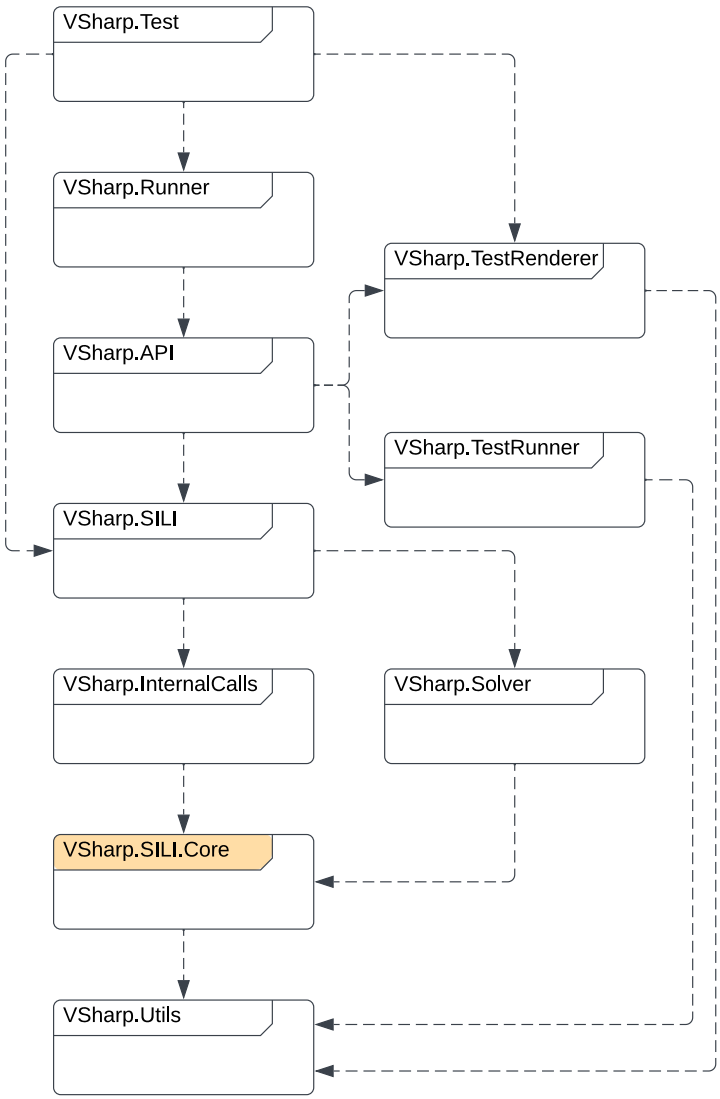

Рис. 1: Архитектура проекта V#

Рассмотрим каждую компоненту подробнее.

VSharp.Test отвечает за проведение интеграционного тестирования.

VSharp.Runner предоставляет CLI (Command Line Interface) для использования V# через терминал.

VSharp.API предоставляет интерфейс для использования V# из других проектов.

VSharp.TestRenderer занимается генерацией кода тестов на языке C#.

VSharp.TestRunner используется для запуска сгенерированных тестов.

VSharp.SILI (Symbolic Intermediate Language Interpreter) является символьным интерпретатором CIL.

VSharp.InternalCalls содержит символьные модели некоторых функций из CLR (Common Language Runtime), что позволяет вместо полноценного их исследования подставить готовые модели.

VSharp.Solver реализует систему взаимодействия с SMT-решателем.

VSharp.SILI.Core представляет из себя «ядро» проекта. В нем происходят все нетривиальные операции внутри символьной машины, например, работа с символьной памятью, решение ограничений на типы.

VSharp.Utils содержит вспомогательные функции для остальных компонент.

2. Обобщенные кандидаты при решении ограничений в системе типов .NET

Данная глава посвящена построению наиболее полной[2] последовательности *кандидатов* — типов, подходящих под ограничения, возникающие в процессе работы символьного исполнения.

Сначала, в разделе 2.1 рассматриваются основные понятия, далее используемые для построения алгоритма нахождения кандидатов. Далее, в разделе 2.2 дается определение кандидата и ограничений. А также вводится алгоритм конструирования последовательности кандидатов, за исключением одной его составляющей — алгоритма построения обобщенных кандидатов, который рассматривается далее.

Искомый алгоритм описывается постепенно, с каждым разом усложняя поставленную задачу. Сначала отсутствуют какие-либо ограничения для обобщенного кандидата. Так, в разделе 2.3 предполагается отсутствие зависимостей между параметрами обобщенного типа. Далее, в разделе 2.4 допускаются ациклические зависимости. Наконец, раздел 2.5 посвящен решению задачи с ограничениями для обобщенного кандидата.

2.1 Система типов .NET

Система типов .NET позволяет создавать *обобщенные типы* — те, что параметризованы другими, возможно тоже обобщенными, типами.

Обозначим за P — множество возможных параметров для обобщенного типа, тогда кортеж $(td, p_1, p_2, \ldots, p_n)$, где $p_i \in P$, назовем *определением обобщенного типа* (англ. *generic type definition*), которое будем обозначать $td\langle p_1, p_2, \ldots, p_n\rangle$.

Введем еще несколько обозначений множеств типов.

- N — содержит типы, не являющиеся обобщенными.
- D — состоит из определений обобщенных типов.
- G — множество обобщенных типов.

[2] содержащей как можно больше подходящих типов

- $GT \stackrel{\text{def}}{=} N \cup G$ — *базовые типы* (англ. *ground types*).
- H — множество определений обобщенных типов, в которых некоторые параметры заменены на базовые типы (например, $T \in P$, $int \in GT$, тогда Dictionary<int, T> $\in H$).
- $T \stackrel{\text{def}}{=} N \cup D \cup G \cup H$ — множество всех типов .NET.

Вместо параметров в определении обобщенного типа можно подставить базовые типы, тем самым получив базовый тип. Любую частичную функцию $subst : P \to GT$ назовем *подстановкой*, тогда можно определить функцию *substitute*, позволяющую по определению обобщенного типа и подстановке s получить базовый тип.

$$substitute(s,t) \stackrel{\text{def}}{=} \begin{cases} s(t), & t \in P, \\ td\text{<}substitute(s,p_1), \ldots, substitute(s,p_n)\text{>}, & t \in H, \\ t, & \text{иначе.} \end{cases}$$

Также в дальнейшем будет необходимо расширять подстановку s, сопоставляя параметру $p \in P$ базовый тип $t \in GT$, для этого построим функцию *add*.

$$add(p,t,s,x) \stackrel{\text{def}}{=} \begin{cases} t, & p = x, \\ s(x), & \text{иначе.} \end{cases}$$

В связи с тем, что система типов .NET позволяет задавать ограничения на параметры, необходимо учитывать их при построении подстановок. Для этого подробнее рассмотрим устройство элементов множества P. Параметр $p = (pn, supertypes) \in P$, где pn — некоторый уникальный идентификатор параметра, а $supertypes \in 2^T$ — *надтипы*, которые должен подтипировать подставляемый вместо параметра p тип. Множество *supertypes* будем называть *ограничениями на параметр*.

Ограничения на параметр также могут включать и другие параметры из определения обобщенного типа. Рассмотрим этот случай подробнее. Для

этого определим функцию *collectParams*, которая сопоставляет типу $t \in T$ множество параметров, входящих в него.

$$collectParams(t) \stackrel{\text{def}}{=} \begin{cases} \{t\}, & t \in P, \\ \bigcup\limits_{i=1}^{n} collectParams(p_i), & t = td\text{<}p_1, \ldots, p_n\text{>}, \\ \varnothing, & \text{иначе.} \end{cases}$$

Будем говорить, что *параметр* $p_i = (pn_i, C_i)$ *зависит от параметра* p_j и обозначать $p_i \leftarrow p_j$, если $p_j \in \bigcup\limits_{c \in C_i} collectParams(c)$.

Графом зависимостей для некоторого множества параметров $\mathcal{P} \subset P$ назовем граф $G = (V, E)$, где вершины являются параметрами: $V = \mathcal{P}$, а ребра — зависимостями между параметрами: $E = \{(p_j, p_i) \,|\, p_i, p_j \in \mathcal{P}, p_i \leftarrow p_j\}$.

```
public class GenericClass<T, U, V, K, G>
    where T : U
    where U : IEnumerable<V>, IDictionary<K, G>
    where K : G
{
}
```

Листинг 2: Пример обобщенного типа с зависимостями между параметрами

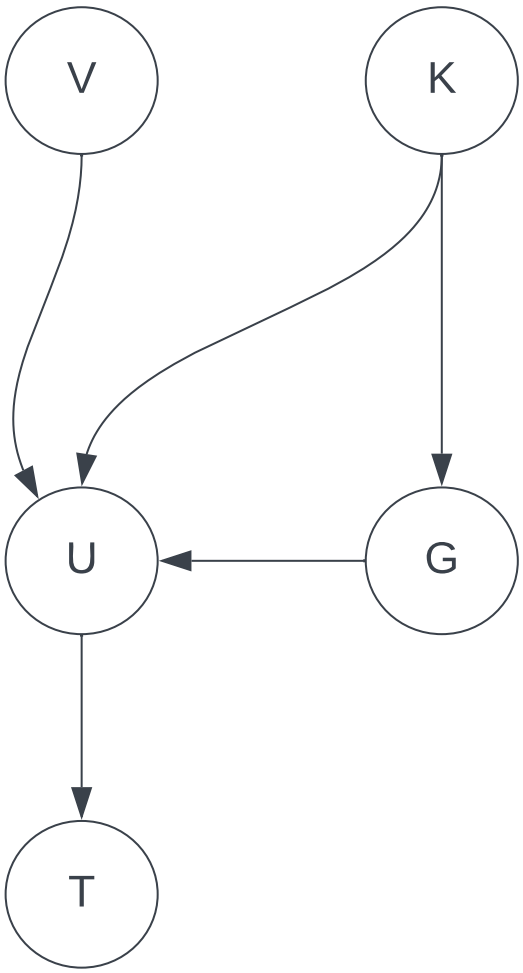

Рис. 2: Пример графа зависимостей

На лист. 2 представлен обобщенный тип с несколькими параметрами. А на рис. 2 изображен граф зависимостей его параметров. Он имеет следующие зависимости: $T \leftarrow U$, $U \leftarrow V$, $U \leftarrow K$, $U \leftarrow G$, $K \leftarrow G$.

2.2 Последовательность кандидатов

Данный и последующие разделы главы 2 описывают построение последовательности *кандидатов* — типов, подходящих под ограничения, возникающие в процессе работы символьного исполнения.

В первую очередь необходимо определить следующие бинарные отношения на множестве T:

- подтипирование — $t <: \tau$, если t является наследником, а также, когда τ — интерфейс, t реализует τ;
- надтипирование — $t :> \tau$, если $\tau <: t$;

- неподтипирование — $t \not<: \tau$, если не выполняется $t <: \tau$;
- ненадтипирование — $t :\not> \tau$, если $\tau \not<: t$.

Ограничениями будем называть упорядоченную четверку $C = \langle C_{<:}, C_{:>}, C_{\not<:}, C_{:\not>} \rangle$ подмножеств T — типы, которые искомые кандидаты должны подтипировать, надтипировать, не подтипировать, не надтипировать. Определим также предикат *satisfies*, позволяющий проверить, подходит ли тип t под ограничения C.

$$\begin{aligned} &\mathit{satisfies}(t, \langle C_{<:}, C_{:>}, C_{\not<:}, C_{:\not>} \rangle) \stackrel{\text{def}}{=} \\ &\qquad \bigwedge_{\mathit{supertype} \in C_{<:}} t <: \mathit{supertype} \quad \wedge \quad \bigwedge_{\mathit{subtype} \in C_{:>}} t :> \mathit{subtype} \\ &\wedge \bigwedge_{\mathit{nSupertype} \in C_{\not<:}} t \not<: \mathit{nSupertype} \quad \wedge \quad \bigwedge_{\mathit{nSubtype} \in C_{:\not>}} t :\not> \mathit{nSubtype} \end{aligned}$$

Следующим шагом будет определение кандидата. Он задается грамматикой:

$$\begin{aligned} \mathit{candidate} ::= &\ \texttt{GROUND}(\mathit{type}) \\ &\mid \texttt{GENERIC}(\mathit{typedef}, \mathit{substitutions}) \end{aligned}$$

Разберем устройство кандидата более подробно. Им может быть базовый тип ($\texttt{GROUND}(\mathit{type})$), то есть не обобщенный тип, либо же обобщенный, в котором все параметры заменены другими базовыми типами. Также кандидат может быть обобщенным ($\texttt{GENERIC}(\mathit{typedef}, \mathit{substitutions})$), в котором *typedef* — определение обобщенного типа, а *substitutions* — множество подстановок для его параметров.

Дальнейшей задачей станет построение этого множества подстановок. В следующих разделах объявляется и дополняется функция *makeGenericCandidate*, которая позволяет получить обобщенного кандидата из определения обобщенного типа и ограничений на него. С ее помощью можно определить функцию $\mathit{makeCandidate} : GT \cup D \to \mathit{candidate}$:

$$makeCandidate(t, C) = \begin{cases} \{\texttt{GROUND}(t)\}, & t \in GT \wedge satisfies(t, C), \\ makeGenericCandidate(t, C), & t \in D, \\ \varnothing, & \text{иначе.} \end{cases}$$

Данная функция в свою очередь необходима для определения *solve* — функции, которая сопоставляет ограничениям последовательность кандидатов.

$$solve(C) = \bigcup_{c \in \mathcal{C}(C)} c,$$

где

$$\mathcal{C}(C) = \{makeCandidate(t, C) \mid t \in GT \cup D\}.$$

2.3 Множество подстановок при отсутствии зависимостей между параметрами

Сначала определим функцию *makeSubstitution*, которая по параметрам обобщенного типа и кандидатам для каждого из них создаст подстановку.

---

function makeSubstitution(params, candidates)
  $(p_1, p_2, \ldots, p_n) \leftarrow params$
  $(c_1, c_2, \ldots, c_n) \leftarrow candidates$
  return $\{(p_1, c_1), (p_2, c_2), \ldots, (p_n, c_n)\}$
end function

Теперь рассмотрим самый простой случай при построении подстановок — случай, когда между параметрами нет никаких зависимостей, а ограничения отсутствуют.

Алгоритм 1 конструирует обобщенного кандидата в случае, когда отсутствуют зависимости между параметрами. В начале для каждого параметра по отдельности строится последовательность кандидатов. Затем, так как параметры независимы друг от друга, берется декартово произведение получившихся последовательностей кандидатов. Далее из этого составляются подстановки.

Algorithm 1 makeGenericCandidate в случае отсутствия зависимостей между параметрами

function makeGenericCandidate(typedef, C)
    $td\langle params\rangle \leftarrow typedef$
    $(pn_1, C_1), \ldots, (pn_n, C_n) \leftarrow params$
    $product \leftarrow solve(C_1) \times solve(C_2) \times \cdots \times solve(C_n)$
    $substitutions \leftarrow \{makeSubstitution(params, candidates) \mid candidates \in product\}$
    $candidate \leftarrow \texttt{GENERIC}(typedef, substitutions)$
    return $\{candidate\}$
end function

2.4 Множество подстановок при отсутствии циклических зависимостей

В случае когда зависимости между параметрами присутствуют, но при этом их граф не содержит цикла можно свести задачу построения подстановок к задаче, рассмотренном в разделе 2.3.

Для этого проведем *расслоение* графа зависимостей — поделим вершины графа (параметры) на группы таким образом, чтобы не существовало ребра (зависимости) между вершинами одной группы.

Будем считать, что имеется граф зависимостей G в виде списка смежности, а также словарь *depsCount*, где каждой вершине сопоставлено количество ребер, входящих в нее.

Algorithm 2 Расслоение графа зависимостей

$layers \leftarrow \texttt{List}\langle\texttt{List}\langle\texttt{P}\rangle\rangle\texttt{()}$
while $depsCount.size > 0$ **do**
    $zeroes \leftarrow \{p \mid depsCount[p] = 0\}$
    for $zero \in zeroes$ **do**
        $depsCount.remove(zero)$
        for $dep \in G[zero]$ **do**
            $depsCount[dep] \leftarrow depsCount[dep] - 1$
        end for
    end for
    $layers.append(zeroes)$
end while
return *layers*

Алгоритм 2 описывает процесс расслоения графа зависимостей. На каждом шаге алгоритма берутся те вершины, в которые не входит ни одного ребра — это и есть очередной слой. Далее эти вершины удаляются из графа, а также вместе с ними удаляются и все ребра, исходящие из них. Этот процесс повторяется до тех пор, пока в графе не останется вершин.

Теперь внутри одного слоя задача свелась к предыдущей. Кроме того, любое ребро теперь ведет из слоя с меньшим номером к слою с большим: $\forall (p_1, p_2) \in E$ верно, что $\exists i, j : p_1 \in layers[i], p_2 \in layers[j], i < j$.

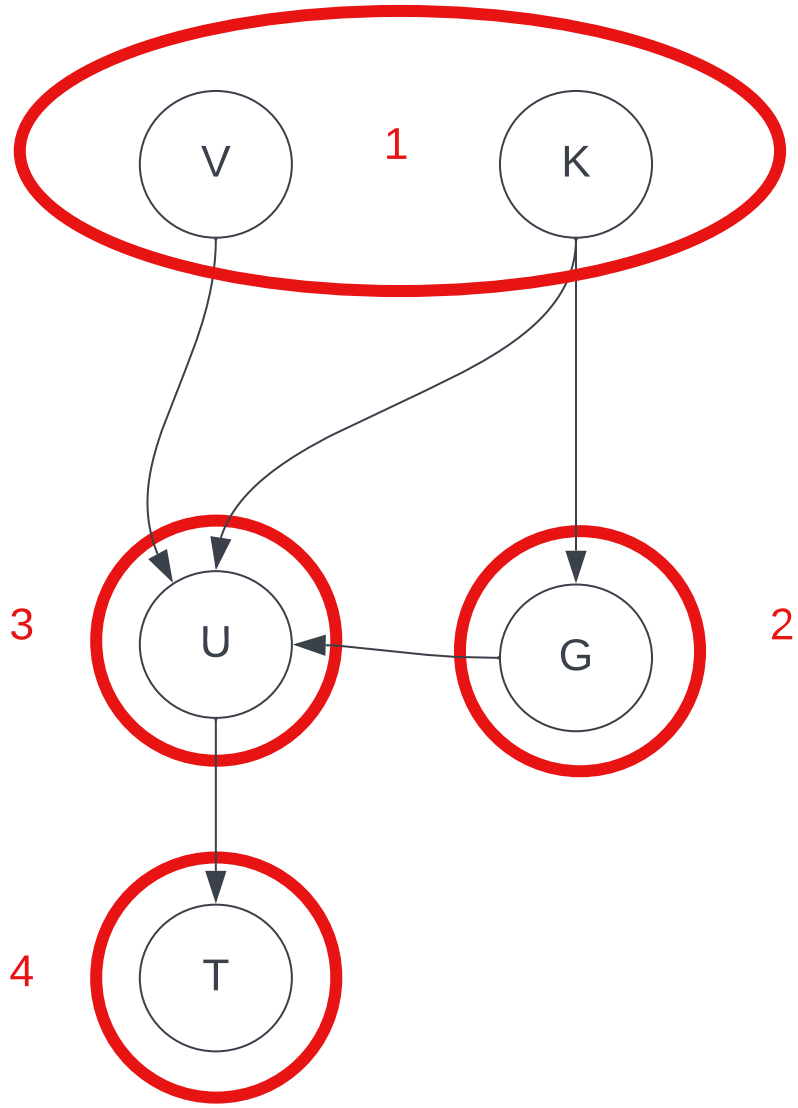

Рис. 3: Пример деления на слои графа зависимостей

На рис. 3 изображен граф зависимостей типа с лист. 2. На нем красными эллипсами выделены слои, а числами рядом с ними их номера в том порядке, в котором они строятся. Теперь рассмотрим процесс создания обобщенного

кандидата.

Algorithm 3 makeGenericCandidate в случае, когда зависимости не являются циклическими

function merge(s1, s2)
  for $(parameter, candidate) \in s2$ **do**
    $s2(p) \leftarrow add(parameter, candidate, s2, p)$
  end for
  return $s2$
end function
function makeGenericCandidate(typedef, C)
  td<params> $\leftarrow$ *typedef*
  $layers \leftarrow makeLayers(params)$
  $substs \leftarrow \{emptySubstitution\}$
  for $layer \in layers$ **do**
    $\langle (pn_{i_1}, C_{i_1}), \ldots, (pn_{i_k}, C_{i_k}) \rangle \leftarrow layer$
    $newSubsts \leftarrow \{\}$
    for $subst \in substitutions$ **do**
      $product \leftarrow solve(C_{i_1}) \times \cdots \times solve(C_{i_k})$
      $layerSubsts \leftarrow \{makeSubstitution(layer, cs) \mid cs \in product\}$
      for $layerSubst \in layerSubsts$ **do**
        $newSubsts \leftarrow newSubsts \cup merge(subst, layerSubst)$
      end for
    end for
    $substs \leftarrow newSubsts$
  end for
  $candidate \leftarrow$ `GENERIC`$(typedef, substs)$
  return $\{candidate\}$
end function

Алгоритм 3 описывает построение обобщенного кандидата в случае, когда зависимости не являются циклическими (в частности, когда их совсем нет). В первую очередь параметры делятся на слои с помощью алгоритма 2. Затем происходит обработка слоев. На каждом этапе известно, какие подста-

новки получены обработкой предыдущих слоев (изначально это единственная пустая подстановка). Каждая из этих подстановок дополняется подстановками, сконструированными для текущего слоя независимым решением ограничений и декартовым произведением.

2.5 Множество подстановок при наличии внешних ограничений на кандидатов

В данном разделе рассматривается следующее усложнение задачи — теперь на кандидатов накладываются дополнительные ограничения C. Далее будет представлен алгоритм *проталкивания* ограничений — алгоритм получения ограничений на каждый отдельный параметр из общих ограничений $C = \langle C_{<:}, C_{:>}, C_{\not<:}, C_{:\not>} \rangle$.

Сначала рассмотрим функцию *trackIndices*, которая позволяет отследить индексы параметров при наследовании.

Algorithm 4 Отслеживание индексов параметров при наследовании

function trackIndices(t, s, indices)
  $td\langle p_1, \ldots, p_n \rangle \leftarrow t$
  $sd\langle sparams \rangle \leftarrow s$
  if $td = sd$ **then**
    return *indices*
  else
    $baseType \leftarrow t.baseType$
    $btd\langle btparams \rangle \leftarrow baseType$
    $mapping \leftarrow$ `Dict()`
    for $p_i \in btparams$ **do**
      $mapping[p_i].append(i)$
    end for
    $track \leftarrow trackIndices(baseType, s, \{mapping[p_i] \mid p_i \in btparams\})$
    $traceback(x) \leftarrow \{track[ind] \mid i \in x, inds \in mapping[p_i], ind \in inds\}$
    return $\{ \bigcup_{is \in traceback(inds)} is \mid inds \in indices \}$
  end if
end function

Алгоритм 4 описывает отслеживание индексов при наследовании типов от t к s Также она принимает набор переходов *indices* — это множество отличается от математического множества тем, что оно упорядоченное, таким образом $indices[i]$ — это набор индексов в типе t в которые перешел при наследовании параметр с номером i из некоторого базового типа. По умолчанию $indices[i] = \{i\}$ Данная функция является рекурсивной. Если t оказывается равен s, то делать ничего не нужно, так как мы уже достигли по дереву наследования искомого типа s, поэтому возвращаем *indices*. Иначе же у типа t берется его базовый тип, строится новый набор переходов уже от t к $t.baseType$. После чего функция вызывается снова на t и $t.baseType$ для получения набора переходов от $t.baseType$ к s. Затем три получившихся набора переходов «склеиваются» путем «композиции» в один и возвращаются как результат работы функции.

Проталкивание ограничений на подтипирование. Далее рассмотрен алгоритм проталкивания ограничений для надтипа.

Algorithm 5 Проталкивание ограничения для надтипа

function propagateSupertype(t, supertype)
  $td\texttt{<}params\texttt{>} \leftarrow t$
  $sd\texttt{<}\langle a_1, \ldots, a_n \rangle\texttt{>} \leftarrow supertype$
  $trackedIndices \leftarrow trackIndices(t, supertype)$
  $propagated \leftarrow \{\}$
  for $p_i \in params$ **do**
    $indices \leftarrow trackedIndices[i]$
    $propagated \leftarrow propagated \cup \{\{a_{ind} \mid ind \in indices\}\}$
  end for
  return $\{(types, types, \varnothing, \varnothing) \mid types \in propagated\}$
end function

Алгоритм 5, используя ранее определенную функцию *trackIndices*, строит переход индексов при наследовании от обобщенного типа t к типу s, а затем сопоставляет соответствующие значения параметров в *supertype* параметрам

t. Так как параметры классов являются инвариантными, то итоговые ограничения содержат совпадающие множества $C_{<:}$ и $C_{:>}$.

Проталкивание ограничений на надтипирование. В данном случае ситуация обратная. Рассмотрим алгоритм подробнее.

Algorithm 6 Проталкивание для подтипа

```
function propagateSubtype(t, subtype)
    sd<p_1, ..., p_n> ← subtype
    td<tparams> ← t
    i ← 0
    propagated ← {∅ | p ∈ tparams}          ▷ здесь также подразумевается «множество» как в trackIndices
    for indices ∈ trackIndices(subtype, t) do
        for j ∈ indices do
            propagated[j] ← propagated[j] ∪ p_i
        end for
        i ← i + 1
    end for
    return {(types, types, ∅, ∅) | types ∈ propagated}
end function
```

Алгоритм 6 действует похожим на алгоритм 5 образом. Он строит переход индексов от *subtype* к t, а затем «разворачивает» этот переход и сопоставляет соответствующие индексы и значения параметров. Также применимо рассуждение об инвариантности параметров класса.

Проталкивание ограничений для интерфейсов. В данном случае необходимо поступить тем же образом, что и раньше, за исключением того, что построение перехода индексов может оказаться ненужным, в случае, если t является классом. Также необходимо учесть вариантность параметра интерфейса.

Для построения функции *propagate* достаточно применить поочередно описанные выше методы, а затем соединить получившиеся ограничения

воедино для каждого параметра.

Теперь рассмотрим финальную версию *makeGenericCandidate*.

Algorithm 7 Финальная версия *makeGenericCandidate*

function makeGenericCandidate(typedef, C)
  td<params> $\leftarrow$ *typedef*
  layers $\leftarrow$ *makeLayers*(*params*)
  substs $\leftarrow$ {*emptySubstitution*}
  $(c_1, \ldots, c_n) \leftarrow$ *propagate*(*typedef*, *C*)
  for *layer* $\in$ *layers* **do**
    $\langle (pn_{i_1}, C_{i_1}), \ldots, (pn_{i_k}, C_{i_k}) \rangle \leftarrow$ *layer*
    newSubsts $\leftarrow$ {}
    for *subst* $\in$ *substitutions* **do**
      product $\leftarrow$ *solve*$(C_{i_1} \cup c_{i_1}) \times \cdots \times$ *solve*$(C_{i_k} \cup c_{i_k})$
      layerSubsts $\leftarrow$ {*makeSubstitution*(*layer*, *cs*) | *cs* $\in$ *product*}
      for *layerSubst* $\in$ *layerSubsts* **do**
        newSubsts $\leftarrow$ *newSubsts* $\cup$ *merge*(*subst*, *layerSubst*)
      end for
    end for
    substs $\leftarrow$ *newSubsts*
  end for
  candidate $\leftarrow$ `GENERIC`(*typedef*, *substs*)
  return {*candidate*}
end function

Алгоритм 7 отличается от предыдущей версии тем, что использует проталкивание ограничений. Которые используются затем при построении последовательности кандидатов. Под объединением ограничений здесь подразумевается объединение соответствующих элементов четверки.

3. Архитектура и детали реализации

В данной главе рассматриваются архитектура (разд. 3.1) и детали реализации (разд. 3.2) разработанной системы.

3.1 Архитектура

На рис. 4 представлена архитектура модуля решателя для типов. Зеленым цветом выделены компоненты, полностью разработанные в рамках данной учебной практики, желтым — частично модифицированные.

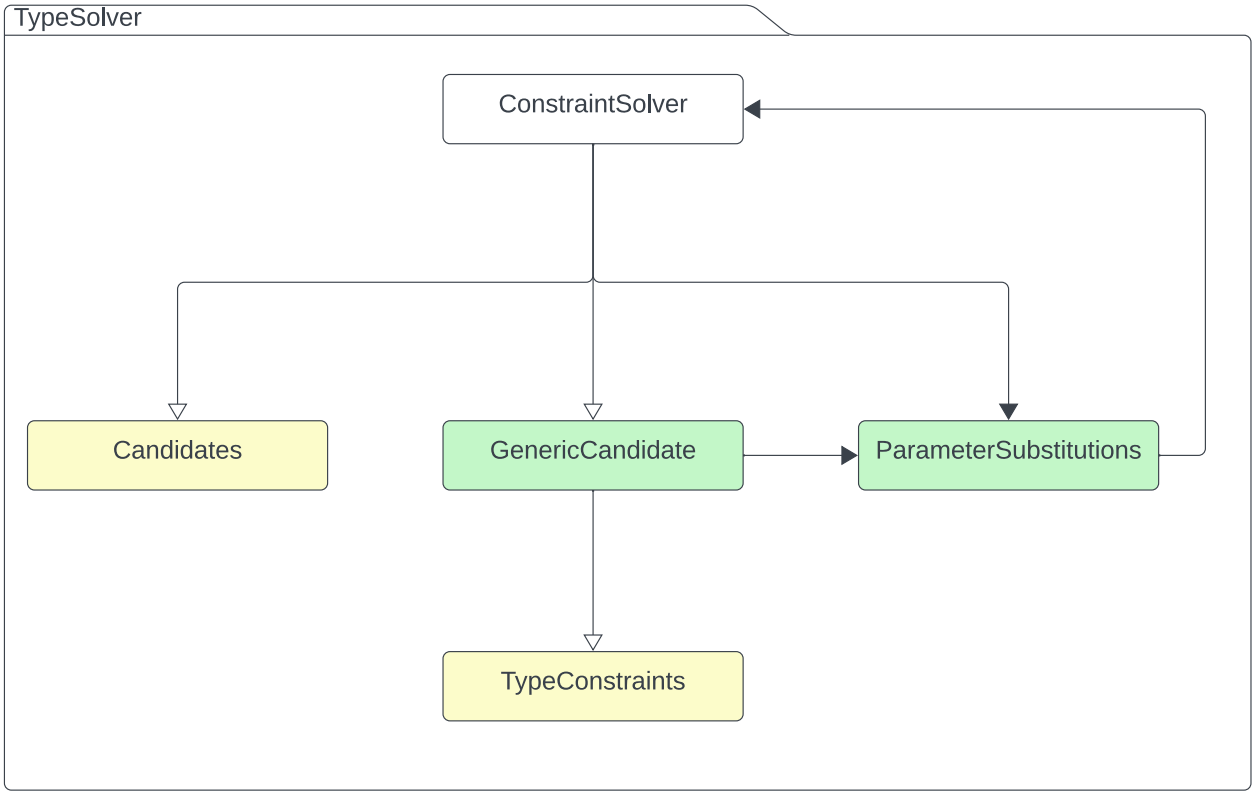

Рис. 4: Диаграмма сущностей решателя для типов

Рассмотрим каждую компоненту подробнее.

ConstraintSolver — главная компонента решателя для типов. Она была разработана в рамках осенней учебной практики. Отвечает за решение ограничений, создание кандидатов, а также составление последовательности из них. Также использует компоненту *ParameterSubstitutions* для подбора параметров исследуемого символьным исполнением метода/класса.

Candidates. Данная компонента содержит в себе логику работы с последовательностью кандидатов. Была модифицирована в рамках этой учебной практики для работы с обобщенными кандидатами.

GenericCandidate — компонента, представляющая обобщенного кандидата. Именно она отвечает за проталкивание ограничений. Использует компоненту *ParameterSubstitutions* для подбора параметров обобщенного типа.

ParameterSubstitutions — компонента, отвечающая за составления подходящих под ограничения подстановок. Она выполняет деление параметров на слои, а также использует *ConstraintSolver* для подбора кандидатов для параметров.

TypeConstraints. Данная компонента содержит внутреннее устройство ограничений на типы, а также интерфейс взаимодействия с ними. В рамках этой работы была модифицирована для использования в алгоритме проталкивания.

3.2 Детали реализации

В данном разделе описаны детали реализации, не вошедшие в главу 2.

3.2.1 Инкрементальность

Инкрементальность — свойство решателя переиспользовать результаты, полученные ранее для ускорения своей работы.

Во время работы символьного исполнения возникают ограничения, которые со временем могут стать только строже. Поэтому перед решателем для типов, реализованным внутри символьной машины проекта V# , стоит задача нахождения наиболее полной последовательности кандидатов для каждой символьной переменной. Имея набор всех подходящих кандидатов, становится возможным далее при возникновении новых ограничений лишь сужать его.

3.2.2 Ленивость вычислений

Как было сказано ранее, возможных кандидатов может оказаться потенциально бесконечно много за счет вложенности обобщенных типов. Одним из решений данной проблемы может стать ленивость вычисления последовательности кандидатов.

Также ленивости способствует деление кандидатов на базовых и обобщенных. Это позволяет отложить вычисление подстановок. А рассмотренный в разд. 2.5 алгоритм проталкивания ограничений значительно сокращает количество вычислений, необходимых для построения подходящей подстановки.

4. Тестирование

4.1 Способ тестирования

Тестирование в проекте V# производится преимущественно интеграционными тестами, представляющие из себя методы, написанные на языке C#, для которых необходимо сгенерировать тесты. Далее полученные тесты запускаются на виртуальной машине .NET, а также рассчитывается полученное покрытие. Если результаты сгенерированных тестов совпадают с ожидаемыми, а процент покрытия равен заданному числу, то тест считается успешно пройденным.

Для тестирования разработанной системы были созданы наборы тестов, в которых при решении ограничений символьной машины возникают обобщенные типы с зависимостями различного уровня сложности.

4.2 Сравнение результатов

В табл. 1 представлены результаты проведенного тестирования. Первый столбец таблицы содержит название тестого метода. Далее в каждом столбце указано либо «Old», либо «New» — результаты до и после внедрения разработанной системы соответственно. Второй и третий столбцы содержат информацию о количестве созданных типов-заглушек за время прохождения тестов. В четвертом и пятом столбцах указано итоговое покрытие по всем тестам из группы. Шестой и седьмой столбцы показывают количество пройденных тестов из группы. В восьмом столбце содержится общее число тестов в группе.

Название группы тестов	Заглушки «Old», ед.	Заглушки «New», ед.	Покрытие «Old», %	Покрытие «New», %	Тестов пройдено «Old», %	Тестов пройдено «New», %	Тестов всего, ед.
MethodParameters	3	0	100	100	4	4	4
MockRelocation	1	0	50	100	1	2	2
DeepPropagating	4	0	66	100	2	3	3
Nested	1	0	35	100	1	2	2
NoSolution	4	3	100	100	2	2	2
GenericCandidates	8	0	66	100	2	3	3
NestedGenerics	0	0	100	100	2	2	2

Таблица 1: Результаты тестирования разработанной системы

Полученные результаты свидетельствуют о том, что в случаях, когда под ограничения подходит только обобщенный тип, разработанная система находит его, вместо создания типа-заглушки. Более того, тип-заглушку возможно создать не всегда, что приводило к преждевременной остановке исследования программы до внедрения разработанной системы, однако после происходить перестало, так как кандидата возможно найти благодаря ей.

Заключение

В ходе работы были получены следующие результаты:

- выполнен обзор системы типов .NET,
- разработана система, позволяющая решателю типов использовать обобщенные типы,
- эта система реализована в рамках проекта V#,
- проведено тестирование реализованной системы.

Список литературы

[1] Cadar Cristian and Sen Koushik. Symbolic execution for software testing: three decades later // Communications of the ACM. — 2013. — Vol. 56, no. 2. — P. 82–90.

[2] De Moura Leonardo and Bjørner Nikolaj. *Z3: An efficient SMT solver* // International conference on Tools and Algorithms for the Construction and Analysis of Systems / Springer. — 2008. — P. 337–340.

[3] Ecma TC39. TG3. Common Language Infrastructure (CLI). Standard ECMA-335, June 2005.

[4] Misonizhnik Aleksandr and Mordvinov Dmitry. On Satisfiability of Nominal Subtyping with Variance // 33rd European Conference on Object-Oriented Programming (ECOOP 2019) / Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik. — 2019.

[5] Niemetz Aina and Preiner Mathias. Bitwuzla at the SMT-COMP 2020 // arXiv preprint arXiv:2006.01621. — 2020.

[6] Păsăreanu Corina S and Visser Willem. A survey of new trends in symbolic execution for software testing and analysis // International journal on software tools for technology transfer. — 2009. — Vol. 11, no. 4. — P. 339–353.

[7] Baldoni Roberto, Coppa Emilio, D'elia Daniele Cono, Demetrescu Camil, and Finocchi Irene. A survey of symbolic execution techniques // ACM Computing Surveys (CSUR). — 2018. — Vol. 51, no. 3. — P. 1–39.

[8] Barrett Clark, Conway Christopher L, Deters Morgan, Hadarean Liana, Jovanović Dejan, King Tim, Reynolds Andrew, and Tinelli Cesare. *Cvc4* // International Conference on Computer Aided Verification / Springer. — 2011. — P. 171–177.