Санкт-Петербургский государственный университет

Программная инженерия
Кафедра системного программирования

Милова Наталья Андреевна

Эффективная разрешающая процедура для задачи выполнимости в теории номинальных систем типов с вариантностью

Магистерская диссертация

Научный руководитель:
профессор кафедры СП, д.т.н, доцент Д. В. Кознов

Консультант:
старший преподаватель кафедры СП Д. А. Мордвинов

Рецензент:
программист ООО «Интеллиджей Лабс» Д. С. Косарев

Санкт-Петербург
2020

SAINT-PETERSBURG STATE UNIVERSITY

Department of Software Engineering

Milova Natalia

Effective decision procedure for satisfiability problem in the theory of nominal type systems with variance

Master's Thesis

Scientific supervisor:
Doctor of Engineering, Associate Professor Dmitry Koznov

Scientific advisor:
Senior lecturer Dmitry Mordvinov

Reviewer:
Software Developer at IntelliJ Labs Co. Ltd. Dmitry Kosarev

Saint-Petersburg
2020

Оглавление

Введение

Системы типов современных объектно-ориентированных языков программирования включают подтипы (subtyping), отношение подтипирования (правила вывода фактов подтипирования) и приведение типов (type casting) [3] и оказывают большое влияние на поведение программ во время исполнения. Поэтому, часто в программах используется дополнительный функционал, связанный с обработкой типов. Например, в платформе .NET в методах, осуществляющих сравнение объектов, генерируется исключение, если типы объектов не сравнимы. Часто бывает необходимо выполнять проверку типа аргумента, переданного в качестве типового параметра обобщенного метода, для определения стратегии дальнейшего поведения программы (для этого используются операторы `as`, `is`). Наряду с этим, если вызывается виртуальный метод (virtual method) объекта, то отношение подтипирования оказывает влияние на работу механизма динамической диспетчеризации (dynamic dispatch) виртуальных методов.

Исследование поведения таких программ может быть затруднено большим числом деталей, характерных для системы типов языка программирования, которые требуется учитывать. Например, система типов C# является номинальной с вариантностью (nominal with variance), где номинальность предполагает, что базовые включения между типами указаны явно при объявлении типов с использованием механизма наследования. В свою очередь, вариантность представляет структурное подтипирование и позволяет отразить зависимости между подтипами составных типов и задать корреляцию их аргументов. При этом массивы ковариантны по типу своего аргумента, а обобщенные типы обеспечивают ковариантность и контравариантность типовых параметров, наряду с возможностью применения к ним дополнительных ограничений.

В процессе верификации .NET-программ возникает необходимость статически оценивать влияние различных особенностей системы типов с помощью формирования запросов на подтипирование, которые долж-

ны ответить на вопрос «существуют ли такие типы, которые соответствуют заданным ограничениям?». В некоторых ситуациях это не может быть сделано компилятором .NET, потому что в данном языке не предусмотрена возможность указывать альтернативные надтипы для параметров типов. Также это не может быть выполнено с помощью современных дедуктивных верификаторов, например Spec# [27], которые не содержат в языке спецификаций средств выражения ограничений на типы. В связи с этим имеется необходимость в использовании другого способа описания запросов на подтипирование. Для .NET перспективным видится использование теории логики первого порядка номинальных систем типов с вариантностью, в рамках которой сформулирована задача выполнимости.

Доказано, что отношение подтипирования для закрытых типов (для типов не содержащих типовых аргументов) разрешимо [12]. Однако, чаще всего на практике требуются рассуждения о подтипировании между открытыми типами, которые содержат типовые переменные. К сожалению отношение подтипирования для открытых типов неразрешимо [23]. Разработка эффективной разрешающий процедуры для таких запросов является открытой проблемой.

В связи с тем, что задача выполнимости в теории номинальных систем типов с вариантностью формулируется в терминах логики первого порядка, алгоритм для ее решения может использовать системы автоматического доказательства теорем (automated theorem provers), основанные на средствах логического или реляционного программирования [15, 6], SMT-решателях [4], систему Prover9-Mace4 [20, 22] или Vampire [14]. Эффективная разрешающая процедура позволит улучшить инструменты статического анализа объектно-ориентированного кода, а также инструменты верификации, в которых требуется решение условий пути исполнения программ, включающие запросы на подтипирование. Например, алгоритм требуется для проекта кафедры системного программирования СПбГУ V#[1], который направлен на создание символьной виртуальной машины для .NET.

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

1. Постановка задачи

Основной целью данной работы является разработка эффективной разрешающей процедуры для задачи выполнимости в теории номинальных систем типов с вариантностью для .NET в проекте V#. Для достижения этой цели были сформулированы следующие задачи.

- Проанализировать основные элементы системы типов .NET.
- Исследовать задачу выполнимости в теории номинальных систем типов с вариантностью для .NET.
- Разработать и реализовать алгоритм проверки запросов на подтипирование в проекте V#.
- Провести экспериментальное исследование разработанного алгоритма.

2. Обзор

2.1. Система типов .NET

Данный раздел посвящен описанию ключевых особенностей CTS (Common Type System) — системы типов промежуточного языка CIL (Common Intermediate Language) платформы .NET, на примере языка C#[2].

Система типов C# является номинальной. Номинальность предполагает, что типы однозначно идентифицируются по имени, а отношения подтипирования задаются явно при объявлении типов с использованием механизма наследования. Все типы, кроме типов unsafe-указателей неявно наследуются от типа `System.Object` и делятся на две большие категории: ссылочные типы (reference types) и типы значений (value types).

Ссылочные типы представлены классами (classes), интерфейсами (interfaces), делегатами (delegates) и массивами (arrays). Для классов поддерживается только одиночное наследование (каждый класс может наследовать один незапечатанный класс, то есть класс без модификатора sealed) и транзитивное наследование, определяющее некоторую иерархию для набора классов. Наряду с этим для интерфесов разрешено множественное наследование. Интерфейсы не могут быть унаследованы от классов. Классы, в свою очередь, могут реализовывать несколько интерфейсов. Массивы и делегаты являются запечатанными, то есть не могут быть номинальными надтипами любого другого типа. Массивы неявно наследуются от `System.Array`, а делегаты от `System.MulticastDelegate` и в контексте подтипирования рассматриваются как интерфейсы.

К типам значений относятся структуры (например, встроенные простые типы (`int`, `byte` и др.)) и типы перечислений (enumeration types). Они неявно наследуются от специального класса `System.ValueType`, который не может быть унаследован пользовательскими классами.

[2]Описание сделано на основе [10]

К структурным элементам подтипирования в C# относятся обобщенные типы (generics), которые позволяют объявлять классы, интерфейсы, делегаты и структуры с типовыми параметрами, спецификация которых отложена до момента создания экземпляров этих типов. Примером обобщенного типа может служить стандартный контейнер `Dictionary< TKey,TValue>`, где `TKey` определяет тип ключа, а `TValue` тип значения. В объявлении обобщенного типа явно определяется вариантность типовых параметров.

Типовые параметры бывают следующих видов.

- Ковариантные, то есть позволяющие использовать подтип изначально заданного типа. Например, для класса `Base`, его номинального подтипа класса `Derived` и интерфейса `IEnumerable<out T>` с ковариантным параметром `T`, экземпляр `IEnumerable<Derived>` может быть присвоен переменной типа `IEnumerable<Base>`.
- Контравариантные, то есть позволяющие использовать надтип изначально заданного типа. Например, для класса `Base`, его номинального подтипа класса `Derived` и делегата `Action<in T>` с контравариантным параметром `T`, экземпляр `Action<Base>` может быть присвоен переменной типа `Action<Derived>`.
- Инвариантные, позволяющие использовать только изначально заданный тип. Например, для класса `Base`, который является номинальным надтипом класса `Derived` и класса `List<T>` с инвариантным параметром `T`, экземпляр `List<Base>` не может быть приведен к `List<Derived>` и наоборот.

Обобщенные классы и структуры предполагают использование только инвариантных типовых параметров. Обобщенные интерфейсы и делегаты поддерживают все виды вариантности. Несмотря на то, что массивы не являются обобщенными типами, они также поддерживают структурное подтипирование, являются ковариантными по типу своего аргумента с учетом размерности и условного деления на «`vector`» — одномерные массивы с нулевой нижней границей и «`array`» — многомерные массивы или одномерные массивы с ненулевой нижней границей. Стоит отметить, что если обобщенный тип или массив используется с

типом значения в качестве типового параметра, то в этом случае он ведет себя инвариантно.

Также на параметры обобщенного типа могут быть наложены ограничения, специфицирующие определенные свойства типов, которые могут быть использованы в качестве этих параметров. Такими ограничениями могут выступать.

- Ограничение базового класса `where T : <имя базового класса>`, предполагающее, что параметр должен иметь конкретный базовый класс или производный от него.
- Ограничение интерфейса `where T : <имя интерфейса>`, предполагающее, что параметр является данным интерфейсом или реализует его. Возможно указывать несколько ограничений такого вида.
- Ограничение `where T : U` предполагает, что параметр `T` должен быть параметром `U` или производным от него.
- Специальные ограничения.
  - `where T : struct` — параметр должен быть типом значения, не допускающим значение `Null`;
  - `where T : class?` — параметер должен быть ссылочным типом, допускающим значения `Null` или не допускающим значения `Null`;
  - `where T : new()` — параметр должен иметь конструктор по умолчанию;
  - `where T : unmanaged` — параметр должен быть неуправляемым типом, не допускающим значения `Null`, то есть типом значения, который на любом уровне вложенности не имеет ссылочных полей.

В рамках системы типов C# предусмотрены операторы проверки типа и выражение приведения типов.

- Оператор `is` проверяет совместим ли тип среды выполнения для определенного выражения с указанным типом.
- Оператор `as` явным образом преобразует выражение в указанный тип, если тип среды выполнения совместим с данным типом, в

противном случае возвращает значение `Null`.

- Выражение приведения типов преобразует значение выражения в указанный тип. Возможны исключения времени выполнения.

2.2. Проект V#

Проект V# — это система формальной верификации CIL-кода, основанная на композициональном статическом символьном исполнении [7, 1] и автоматическом доказательстве теорем над дизъюнктами Хорна с ограничениями [8].

Символьное исполнение (symbolic execution) — это популярная техника анализа, позволяющая одновременно исследовать несколько путей исполнения, которые программа может пройти с разными входными данными. Основная идея заключается в том, чтобы программа использовала символьные (symbolic), а не конкретные (concrete) входные значения (values). Исполнение происходит за счет механизма символьного исполнения, который для каждого пути (brunch) потока управления (control flow path) поддерживает: (1) формулу логики первого порядка, которая описывает условия, выполненными по ветвям взятыми вдоль этого пути (path condition) и (2) символьную память (symbolic memory store), которая отображает переменные в символьные выражения (expressions) или значения. Ветка исполнения обновляет формулу, а присваивания обновляют символьную память. Средство проверки модели, обычно основанное на SMT-решателе, используется для проверки того, есть ли какие-либо нарушения свойства вдоль каждого исследуемого пути и достижим (reachable) ли сам путь, то есть может ли формула быть выполнимой с некоторыми присваиваниями конкретных значений символьным аргументам программы [1].

Композициональность предполагает, что если одна и та же функция вызывается в программе несколько раз, то есть возможность символьного исполнения этой функции, как отдельной программы, и переиспользования результата данного исполнения для повторных вызовов.

Архитектура проекта V# представлена на рис. 1.

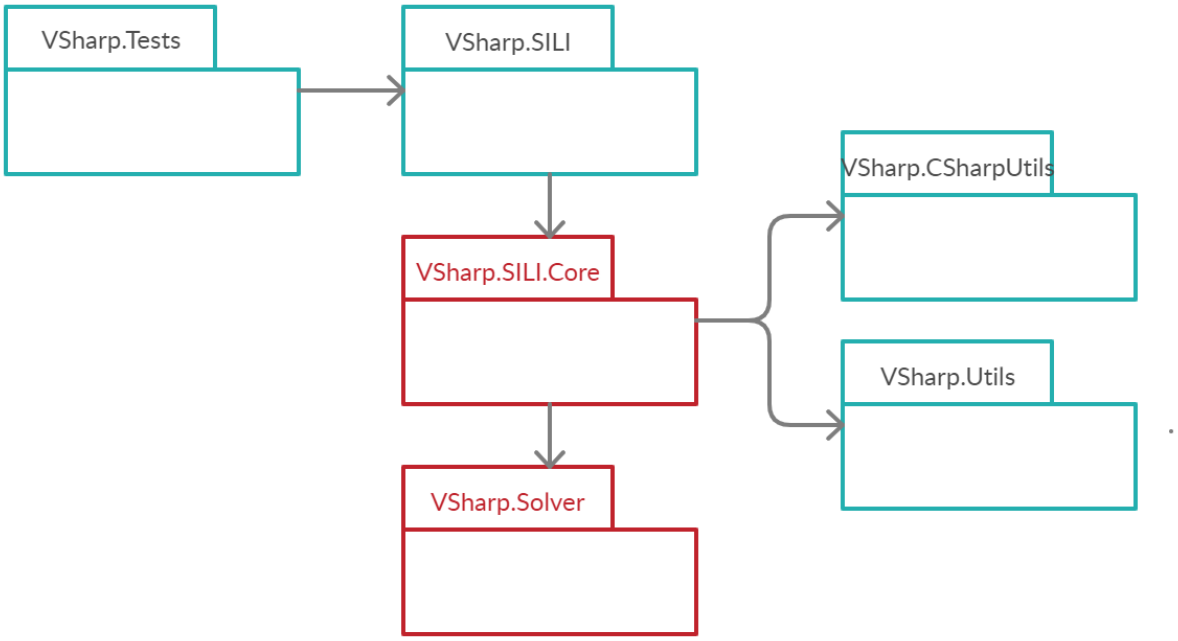

Рис. 1: Архитектура проекта V#

Проект включает следующие компоненты.

- `VSharp.SILI`[3] — подсистема, которая реализует исполнение инструкций CIL и высокоуровневый алгоритм композиционального символьного исполнения.
- `VSharp.SILI.Core` — ядро проекта, реализующее работу примитивов символьного исполнения, например, работу с символьными значениями, операции с символьной памятью программы и системой типов.
- `VSharp.Solver` — подсистема, кодирующая и выполняющий запросы к SMT-решателю.
- `VSharp.Tests` — подсистема юнит-тестирования и функционального тестирования проекта.
- `VSharp.Utils` и `VSharp.CSharpUtils` — библиотеки вспомогательных функций.

В рамках данной работы запросы на подтипирование порождаются подсистемой `VSharp.SILI.Core`, если в .NET-коде есть операторы проверки типа `as`, `is` или явные приведения типов, как в методе, представ-

[3]Аббревиатура Symbolic Intermediate Language Interpreter

ленном в примере 1. Алгоритм проверки выполнимости таких запросов затрагивает подсистемы `VSharp.SILI.Core` и `VSharp.Solver` (на рис. 1 они выделены красным цветом).

Пример 1. Метод для символьного исполнения

```
1  interface IComparable<in T> {}
2  class Base {}
3  sealed class Derived : Base, IComparable<Derived> {}
4
5  void F<T>(Base arg1, Derived arg2)
6  {
7      if (arg2 is IComparable<T> && arg1 is T)
8      {
9          ...
10     }
11 }
```

Во время символьного исполнения метода `F<T>`, ветка, которая соответствует выполнению условия строки 7, защищена следующим запросом на подтипирование.

$$TypeVariable\{arg2\} <: \texttt{IComparable<T>} \wedge \texttt{Derived} <: \texttt{IComparable<T>} \wedge TypeVariable\{arg1\} <: \texttt{T} \wedge \texttt{Base} <: \texttt{T}$$

Здесь $TypeVariable\{arg2\}$ и $TypeVariable\{arg1\}$ обозначают свежие типовые переменные.

2.3. Системы автоматического доказательства теорем

2.3.1. Логическое программирование (Prolog)

Логическое программирование — это парадигма программирования, основанная на логическом выводе информации из множества аксиом и правил, определяющих отношения между объектами предметной области. Prolog (Programming in logic) — это язык логического программирования, использующий математическую логику исчисления предика-

тов первого порядка, реализующий стратегию поиска в глубину. Базовыми элементами языка, заимствованными из логики являются термы (terms), которые являются единственной структурой данных, и утверждения (statements) — дизъюнкты Хорна (дизъюнкции с не более чем одним положительным литералом), такие как факты, правила и запросы [11, 15].

Термом может быть константа, переменная или составной терм. Константы обозначают конкретные объекты предметной области (individuals), такие как целые числа или атомы (atoms).Переменные обозначают также один объект, но не специфицированный. Составной терм состоит из функтора, называемого основным функтором терма, и последовательности одного или нескольких термов, называемых аргументами. Функтор характеризуется именем и числом аргументов или арностью. Константы рассматриваются, как функторы арности 0. Синтаксически функтор представлен следующим образом: $f(t_1, \ldots, t_n)$, где f — имя функтора, каждый t_i — терм аргумент, а n — арность функтора. Функторы с одинаковым именем, но разной арностью считаются различными. Термы называются закрытыми (ground), если они не содержат переменных, остальные являются открытыми. Цели (goals) — это атомы или составные термы, которые в большинстве случаев открытые.

Подстановка (substitution) — это возможно пустое множество пар вида $X = t$, где X — переменная, а t — это терм. Для любой подстановки $\theta = \{X_1 = t_1, \ldots, X_n = t_n\}$ и терма s запись $s\theta$ обозначает результат замены каждой переменной в терме s в соответствие с θ и называется экземпляром s.

Логическая программа — это множество дизъюнктов (clauses) или правил универсально квантифицированных логических предложений вида: $A \leftarrow B_1, \ldots, B_k$, где $k \geq 0$, A и B_i — цели. Такие предложения следует читать декларативно: «A влечется из конъюнкции B_i» и интерпретировать процедурно «Ответом запроса A является ответ конъюнктивного запроса $B_1, \ldots, B_n$». A — голова (head) дизъюнкта, а конъюнкция B_i — тело (body) дизъюнкта. Если $k = 0$, тогда дизъюнкт является фактом (fact) или единичным дизъюнктом (unit clause), записывается

«A.» и является безусловно верным утверждением. Запрос — это конъюнкция вида $A_1, \ldots, A_n?$, где $n > 0$ и A_i — цель. Переменные в запросе являются экзистенциально квантифицированными.

Вычисление логической программы P находит логически выводимый из этой программы экземпляр запроса, с использованием сопоставления с образцом и автоматического перебора с возвратами и гарантированно завершается, если оперирует термами ограниченной глубины вложенности. Цель G выводима из программы P, если существует экземпляр A цели G, где $A \leftarrow B_1, \ldots, B_n$, $n \geq 0$, закрытый экземпляр дизъюнкта в P и B_i выводимы из P. Вывод цели из фактов равенств является специальным случаем.

2.3.2. Реляционное программирование (OCanren)

Реляционное программирование является подходом, основанным на логическом программировании в ограничениях и предполагающим написание программ в виде отношений (relations), которые не делают различий между аргументами и результатом (последовательностью кортежей, где каждый кортеж принадлежит отношению), что позволяет запускать программы в «разных направлениях», например имитировать обратное исполнение (reverse execution).

Языком, реализующим данный подход, является miniKanren, который включает небольшое число примитивов и может быть реализован как встраиваемый предметно-ориентированный язык [6]. Одной из таких реализаций является OCanren, встроенный в функциональный язык OCaml [5]. В данной работе используется основная отличительная особенность OCanren от Prolog в плане поиска ответа. OCanren предполагает комбинацию поиска в ширину и в глубину так называемый (interleaving search), который чередует исполнение альтернатив при использовании сопоставления с образцом. Как и поиск в ширину interleaving search является полным поиском в том смысле, что если ответ есть, то он будет найден, возможно за длительный промежуток времени. Он похож на поиск в глубину в смысле направленности этого поиска.

2.3.3. Prover9-Mace4

`Prover9` — это резолютивная автоматическая система доказательства теорем для логики первого порядка с равенством [22]. Данная система в качестве входных данных использует дизъюнкты из раздела 2.3.1 и пытается с помощью правила резолюций вывести, что предположение логически следует из гипотез. В случае, если доказательство не может быть найдено, предусмотрена возможность использования `Mace4`.

`Mace4` [21] — это программа для поиска конечных моделей для формул логики первого порядка. Входными данными для `Mace4`, также как для `Prover9`, является множество гипотез и предположение. Основная идея поиска конечных моделей состоит в том, чтобы просто исчерпывающе проверить выполнимость универсально квантифицированных формул на моделях-кандидатах, построенных с использованием специально введенных доменных констант, в соответствие с заданным размером модели, постепенно увеличивая этот размер. Если предположение является опровержением какой-либо гипотезы, то любые найденные `Mace4` модели являются контрпримерами к этому предположению. Поиск конечных моделей и в частности `Mace4` полезен для работы с конечными алгебрами, например в верификации [18, 17, 9, 2].

2.3.4. SMT-решатели. CVC4

SMT-решатели представляют собой инструменты автоматической проверки выполнимости формулы в сигнатуре, включающей функциональные и предикатные символы заданной теории первого порядка или комбинации теорий. Решатели поддерживают такие теории, как линейная целочисленная арифметика, линейная вещественная арифметика, битовые вектора, массивы и неинтерпретированные функции [24, 29].

Если SMT-решатель сообщает ответ `SAT`[4], это означает, что формула выполнима и можно потребовать от него предъявить модель — оценку

[4] англ. satisfiable

свободных переменных. Также решатель может сообщить ответ `UNSAT`[5], говорящий о невыполнимости формулы. Наряду с этим, возможно зависание или ответ `UNKNOWN` в некоторых ситуациях, когда формула не относится к разрешимым фрагментам логики первого порядка.

В данной работе используется возможность CVC4 [24] искать конечные модели аналогично Prover9-Mace4.

2.3.5. Vampire

Vampire— система автоматического доказательства теорем для логики первого порядка. Наряду с этим, она дает дополнительную возможность эффективно рассуждать в теориях первого порядка, таких как арифметика, массивы, типы данных, и в их комбинациях, поэтому «претендует быть SMT-решателем» [14].

Входными данными для Vampire является формула-предположение (conjencture) и множество формул логики первого порядка, которое включает аксиомы (axioms) и гипотезы (assumptions), в формате `TPTP` (Thousands of Problems for Theorem Provers) или стандартном для SMT-решателей формате `SMT-LIB`. Система пытается доказать предположение, добавляя его отрицание к множеству аксиом и гипотез и проверяя невыполнимость результирующего множества. Оно невыполнимо, если предположение является логическим следствием множества аксиом и гипотез.

Доказательство невыполнимости формулы называется опровержением (refutation) этой формулы, а такие доказательства называются доказательствами опровержением (proofs by refutation). Каждая формула или вывод (inference) получается с помощью применения одного или нескольких правил вывода (inference rule). Есть несколько видов правил вывода. Некоторые используются на входных данных во время предобработки (preprocessing), например, трансформация в конъюнктивную негативную нормальную форму (cnnf transformation) и трансформация в конъюнктивную нормальную форму (cnf transformation). В

[5]англ. unsatisfiable

конечном итоге входные формулы преобразуются дизъюнкты, после чего Vampire пытается доказать невыполнимость результирующего множества формул, используя систему вывода резолюций и суперпозиций (resolution and superposition inference system), и при получении доказательства сообщает `UNSAT`.

Правила исчисления суперпозиции делятся на два вида: генерирующие (generating) и упрощающие (simplifying). Такое разделение требуется, потому что Vampire использует концепцию насыщения (saturation). Множество формул называется насыщенным относительно некоторой системы вывода, если каждый вывод с предпосылками принадлежит этому множеству и заключение этого вывода также принадлежит этому множеству. Процесс построения такого множества — алгоритм насыщения (saturation algorithm), который добавляет новые дизъюнкты или исключает избыточные. Обычно заключение вывода является логическим следствием его предпосылок, но не в общем случае. В некоторых ситуациях, если не найдено доказательство и процесс насыщения закончен, Vampire сообщает `SAT`, имея ввиду выполнимость отрицания предположения, а не выполнимость самого предположения. На ряду с этим, возможно зависание системы и ответ `UNKNOWN` после истечения предела времени исполнения. Это означает, что Vampire не может доказать невыполнимость или закончить процесс насыщения при имеющихся ограничениях ресурсов, например временных.

Правила логического вывода, используемые в Vampire, гарантируют корректность (soundness), которая означает, что логический вывод не меняет выполнимое множество формул на невыполнимое.

2.4. Существующие работы

Все исследования номинальных систем типов с вариантностью различных языков программирования можно условно разделить на две категории. К первой относятся работы в области закрытого подтипирования. Одной из основополагающей статей, на которую ссылаются многие другие, является [12]. Она формализует номинальное подтипи-

рование с вариантностью, доказывает неразрешимость закрытого подтипирования в .NET и приводит несколько разрешимых фрагментов. Одной из недавних работ является [13], в которой для обобщений в Java доказывается теорема о неразрешимости подтипирования редукцией к ней проблемы останова машины Тьюринга и их Тьюринг-полнота. В работе [28] также доказывается Тьюринг-полнота, но уже шаблонов в языке C++. Согласно работе [19] и в Scala система типов является неразрешимой. Работы по OCaml [16, 25] и Haskell с расширениями [26] показывают, что их системы типов данных языков неразрешимы.

Ко второй категории относятся работы, которые учитывают открытые типы, то есть исследующие задачу выполнимости ограничений на типы. В статье [26] задача выполнимости частичных порядков на основе типов сведена к задаче выполнимости логики первого порядка и предлагается использование SMT-решателей для номинального фрагмента Java, но без комбинации с обобщениями. В работе [23] доказывается, что даже для нерасширяющихся таблиц классов, содержащих только нульарные и унарные ковариантные и инвариантные типовые конструкторы, задача выполнимости бесквантворной формулы, состоящей из конъюнкции атомов подтипирования без отрицаний, неразрешима. Также в ней представлен разрешимый фрагмент задачи, ограничивающий вид формулы и схема получения других разрешимых фрагментов.

3. Задача выполнимости в теории номинальных систем типов с вариантностью для .NET

В данной главе адаптированы материалы статьи [23] для системы типов .NET.

Типы могут быть или типовыми переменными (обозначаются x, y, z) или сконструированными типами $C\texttt{<}\overline{T}\texttt{>}$, где C — n-арный конструктор типа, а $\overline{T}$ — вектор размерности n аргументов данного конструктора. Закрытые типы — это типы, которые не содержат типовых переменных. Открытые типы — это типы, которые не являются закрытыми.

Определение 1. Таблица классов — это конечное множество записей следующего вида:

$$C^*\texttt{<}\overline{vx^*}\texttt{>} <:: T_1, \ldots, T_n.$$

Также таблица классов включает в себя набор ограничений типовых параметров:

$$C\#i <: U_1, \ldots, U_n.$$

Символ $<::$ обозначает бинарное отношение номинального подтипирования. В ограничениях символ $<:$ обозначает отношение подтипирования из определения 3.

Каждая запись содержит уникальное объявление типового конструктора и конечный список сконструированных типов, которые являются номинальными надтипами для всех типов, сконструированных этим конструктором. Без потери общности можно считать, что в таблице классов нет одинаковых переменных (иначе переименуем их).

Левая часть записи содержит имя конструктора C и его формальные типовые параметры x_i с вариантностями v_i ($\circ$ инвариантность, $+$ ковариантность, $-$ контравариантность). Правая часть записи содержит конечный список типов T_i, полученных из конструкторов, объяв-

ленных в других записях, конструктора C и параметров x_i.

Также левая часть каждой записи содержит аннотацию для конструктора и аннотацию для формальных типовых параметров, если на них наложены специальные ограничения. Аннотация помещается вместо * и указывает является ли тип ссылочным типом (тогда записывается буква R), типом значения (V), неуправляемым типом (U) и есть ли у типа конструктор без параметров (обозначается парой пустых скобок).

Каждый i-ый формальный типовой параметр конструктора C и его вариантность обозначаются $C\#i$ и $var(C\#i)$ соответственно.

Пример 2. Таблица классов для `NullableComparer<T>`

```
struct Nullable<x>
      where x : struct {}
interface IComparer {}
interface IComparer<in x> {}
interface IComparable<in x> {}
abstract class Comparer<x>:
    IComparer, IComparer<x> {}
class NullableComparer<x>:
    Comparer<Nullable<x>>
    where x : struct , IComparable<x> {}
```

System.Object^R	<::	
System.ValueType^R	<::	System.Object
Nullable$^{V()}$<x^V>	<::	System.ValueType
IComparer^R	<::	System.Object
IComparer^R<$-y_1$>	<::	System.Object
IComparable^R<$-y_2$>	<::	System.Object
Comparer^R<y_3>	<::	IComparer
		IComparer<y_3>
NullableComparer$^{R()}$<z^V>	<::	Comparer<Nullable<z>>
z	<:	IComparable<z>

Определение 2. Подстановка — это отображение из типовых переменных в типы, которое действует тождественно везде, кроме конечного подмножества переменных. Это подмножество она отображает в сконструированные типы, у которых параметрами являются новые типовые переменные. Доменом подстановки *subst* является множество типовых переменных, отображаемое в типы, а кодоменом образ домена.

Подстановка обозначается следующим способом:

$$[\, x_1 \mapsto T_1;\ \ldots,\ x_n \mapsto T_n \,] \text{ и } [\, \overline{x} \mapsto \overline{T} \,].$$

При этом $x_1, \ldots, x_n$ являются типовыми переменными из домена подстановки, а $T_1, \ldots, T_n$ являются их образами. Применение подстановки $[\, \overline{x} \mapsto \overline{U} \,]$ к типу T обозначается как $[\, \overline{x} \mapsto \overline{U} \,]T$.

Если таблица классов содержит запись $C\texttt{<}x\texttt{>} <:: T_i$, тогда $C\texttt{<}U\texttt{>} <:: [\overline{x} \mapsto \overline{U}]T_i$. Транзитивное замыкание этого отношения обозначается $<::^+$.

Таблицы классов ограничены так, что отношение $<::^+$ ациклично и корректно по отношению к вариантности формальных типовых параметров (ковариантный параметр не может использоваться в контравариантной позиции и наоборот). Также надтипы должны быть непресекающимися: если $C\texttt{<}x\texttt{>} <:: T$ и $C\texttt{<}x\texttt{>} <:: U$, то для всех $\overline{V}$, если $[\overline{x} \mapsto \overline{V}]T = [\overline{x} \mapsto \overline{V}]U$, то $T = U$. Аннотации не должны противоречить естественным ограничениям .NET.

Определение 3. Отношение подтипирования для закрытых типов $<:$ определяется следующим набором правил:

$$\frac{T <: U}{T <:_+ U} \qquad \frac{}{T <:_\circ T} \qquad \frac{U <: T}{T <:_- U}$$

$$(\text{Var})\frac{\text{for each } i \quad T_i <:_{var(C\#i)} U_i}{C\texttt{<}T\texttt{>} <: C\texttt{<}U\texttt{>}}$$

$$(\text{Super})\frac{C\texttt{<}x\texttt{>} <:: V \quad [\overline{x} \mapsto \overline{T}]V <: D\texttt{<}U\texttt{>}}{C\texttt{<}T\texttt{>} <: D\texttt{<}U\texttt{>}}\ C \neq D$$

Данное определение может быть расширено до отношения подтипирования между открытыми типами [12] путем добавления аксиомы рефлексивности для типовых переменных $x <: x$.

Для множественного наследования правило Super может применяться разными способами.

Согласно работе [12], отношение подтипирования $<:$ разрешимо для закрытых типов, при условии, что таблица классов будет нерасширяющийся. За определением нерасширяемости автор отсылается к работе [12]. Расширяющиеся таблицы классов отвергаются средой исполнения .NET, поэтому на протяжении данной работы предполагается разрешимость отношения подтипирования из определения 3.

Предполагается, что таблица классов CT (см. определение 1) зафиксирована. $\mathcal{C}$ обозначает множество конструкторов из CT. $\Sigma = (\mathcal{C}, \{<: \})$ — сигнатура первого порядка с равенством. Функциональные симво-

лы отождествляются с конструкторами из CT. Для удобства применение функционального символа C к аргументам $\overline{U}$ записывается в виде $C\texttt{<}\overline{U}\texttt{>}$, или просто CU в случае унарного конструктора. $<:$ — бинарный предикатный символ, который будет записываться в инфиксной форме. Для удобства $\neg(T <: U)$ и $\neg(T = U)$ записываются как $T \not<: U$ и $T \neq U$.

Пусть $I_{<:}$ будет Σ-структурой с доменом $|I_{<:}|$ для всех закрытых типов, определённых CT, интерпретацией $<:$ будет отношение подтипирования из определения 3. Пусть $\mathcal{T}^{CT}_{<:}$ будет полной Σ-теорией первого порядка со структурой $I_{<:}$, то есть множеством всех Σ-предложений первого порядка, которые выполняются в $I_{<:}$ (имеется ввиду классическое определение выполнимости формулы ϕ в структуре I, обозначаемое $I \vDash \phi$). Данная формула ϕ выполнима в I по модулю теории $\mathcal{T}^{CT}_{<:}$ (обозначается $I \vDash^{CT}_{<:} \phi$) тогда и только тогда, когда $I \vDash \{\phi\} \cup \mathcal{T}^{CT}_{<:}$. В этом случае I является моделью подтипирования формулы ϕ.

$\mathcal{V}$ обозначает счетное множество переменных. Интерпретацией свободных переменных является любое отображение $v : \mathcal{V} \to |I_{<:}|$ переменных в закрытые типы. Стоит отметить, что интерпретация свободных переменных является подстановкой с замкнутым кодоменом. Формула ϕ со свободными переменными называется выполнимой (истинной, невыполнимой) если $I, v \vDash \phi$ для некоторой (любой, никакой) интерпретации свободных переменных v. $v \vDash^{CT}_{<:} \phi$ обозначает выполнимость в (I, v), а $\vDash^{CT}_{<:} \phi$ истинность соответственно.

Определение 4. *Задача выполнимости в теории номинальных систем типов с вариантностью для* .NET. Дана таблица классов CT и формула ϕ над Σ, надо найти I такую, что $I \vDash^{CT}_{<:} \phi$, или доказать, что её не существует.

Утверждение 1. Задача выполнимости в теории номинальных систем типов с вариантностью неразрешима [23].

Утверждение 2. Задача выполнимости в теории номинальных систем типов с вариантностью полуразрешима, то есть существует алгоритм, который перечисляет все возможные закрытые подстановки и в случае, если формула выполнима, завершается [23].

Задача выполнимости в теории номинальных систем типов с вариантностью для .NET может быть проиллюстрирована с помощью примера 3.

Пример 3. Задача выполнимости в теории номинальных систем типов с вариантностью для .NET.

Дана таблица классов CT.

$$\begin{array}{lll} \texttt{Object}^R & <:: & \\ \texttt{J}^R\texttt{<}+y\texttt{>} & <:: & \texttt{Object} \\ \texttt{I}^R\texttt{<}-x^R\texttt{>} & <:: & \texttt{Object} \\ \texttt{C}^R & <:: & \texttt{I<I<C>>} \end{array}$$

Требуется определить выполнимость формулы ϕ.

$$\phi \stackrel{\text{def}}{=} \texttt{C} <: \texttt{I<C>} \vee (x <: \texttt{I<}y\texttt{>} \wedge \texttt{J<}x\texttt{>} \not<: \texttt{J<}y\texttt{>})$$

Данная формула выполнима, если $x = \texttt{I<I<Object>>}$ и $y = \texttt{I<Object>}$.

4. Алгоритм проверки выполнимости запросов на подтипирование

Данная глава описывает алгоритм проверки выполнимости бескванторной формулы ϕ языка $\mathcal{L}_s$ задачи выполнимости в теории номинальных систем типов с вариантностью для .NET (см. раздел 4), учитывающий особенности системы типов .NET. Схема данного алгоритма представлена на лист. 1.

Листинг 1: Алгоритм определения выполнимости запроса на подтипирование `SolveFormula`

Вход: ϕ — бескванторная формула языка $\mathcal{L}_s$
Выход: (`SAT` + модель)/`UNSAT`/`UNKNOWN`

```
1 φ_s ← SimplifyFormula(φ)
2 if φ_s = ⊥ then return UNSAT
3 if φ_s = ⊤ then return SAT(M)                    // M - модель
4 Clauses ← Encode(φ_s)
5 Result ← CallTheoremProver(Clauses)
```

Процесс определения выполнимости формулы `SolveFormula` состоит из следующих шагов: (1) упрощение формулы ϕ в формулу ϕ_s с помощью применения правил подтипирования из определения 3 и стандартных упрощений равенств (функция `SimplifyFormula` строка 2), при необходимости (2) кодирование ϕ_s, релевантной ей таблицы классов CT, специальных ограничений типовых параметров и аксиом вариантности в соответствие с разделом 2.1 во множество *Clauses* дизъюнктов Хорна первого порядка с помощью применения функции `Encode` (строка 5), (3) использование *Clauses* в качестве входных данных для системы автоматического доказательства теорем (функция `CallTheoremProver`) (строка 6) и (4) интерпретацию результата `Result` с помощью функции `Interpret` (строка 7).

Далее подробнее рассматриваются шаги (1) и (2).

4.1. Алгоритм упрощения формулы с помощью применения правил подтипирования

Идея применения правил подтипирования и упрощения равенств в качестве этапа предобработки заключается в том, что если формула содержит атомы, которые можно упросить, то в ситуациях, когда они могут быть преобразованы в $\top$ или $\bot$, рассуждения о выполнимости формулы в целом могут не потребовать использования дополнительных средств.

Алгоритм упрощения основан на трёх взаимно рекурсивных функциях `SimplifyFormula`, `SimplifyAtom`, `ApplyRules`, представленных соответственно на лист. 2, лист. 3 и лист. 4.

Листинг 2: Функция упрощения формулы `SimplifyFormula`

```
   Вход: φ — формула
   Выход: φ_s — упрощенная формула
1  S ← GetSubformulas(φ)
2  foreach s ∈ S do
3  |   if IsAtom(s) then  s_n ← SimplifyAtom(s, s)
4  |   else  s_n ← SimplifyFormula(s)
5  |   φ_n ← Replace(φ, s, s_n)
6  end
7  φ_s ← EliminateBoolConst(φ_n)
```

Функция `SimplifyFormula` является точкой входа в алгоритм упрощения и действует следующим образом.

- С помощью `GetSubformulas` строится множество S непосредственных подформул (immediate subformulas) формулы ϕ (например, если $\phi_1 = A$ и $\phi_2 = A \wedge B$, то $S_1 = \{A\}$ и $S_2 = \{A, B\}$ соответственно).
- Для каждой подформулы $\{s\} \in S$ с помощью `IsAtom` проверяется, является ли она атомарной формулой языка $\mathcal{L}_s$. Если это так, запускается алгоритм упрощения `SimplifyAtom` лист. 3, подробности которого будут описаны ниже. В противном случае рекурсивно вызывается `SimplifyFormula` с подформулой s в качестве

входных данных.

- После процедуры упрощения подформула s в формуле ϕ заменяется на логически эквивалентную формулу s_n функцией `Replace`, формируя ϕ_n.
- Результатом применения `SimplifyFormula` является формула ϕ_s, полученная путем удаления логических констант из ϕ_n с помощью применения `EliminateBoolConst`, если какие-либо атомы исходной формулы были упрощены в $\top$ или $\bot$.

Листинг 3: Функция упрощения атома `SimplifyAtom`

Вход: $atom$ — атом для упрощения, $prevAtom$ — атом, предшествующий $atom$ в последовательности упрощений
Выход: ψ_a — упрощенная формула, логически эквивалентная $atom$

```
1 if IsSimpleAtom(atom) then ψ_a ← atom
2 else
3 |   if IsTrivial(atom) then ψ_a ← ⊤
4 |   else ψ_a ← ApplyRules(atom, prevAtom)
5 end
```

Функция `SimplifyAtom` проверяет является ли $atom$ простым. Простыми являются атомы, которые содержат минимум одну типовую переменную, кроме тривиальных атомов $x = x$ и $x <: x$, и логические константы. Например, $J2\langle x\rangle <: J1\langle Object\rangle$ или $J1\langle x\rangle = J1\langle y\rangle$ могут быть упрощены, а $x <: J1\langle Object\rangle$ и $x = y$ нет. В том случае, если атом можно упростить и проверка тривиальности `IsTrivial` в строке 4 возвращает *true*, то $atom$ тавтологически истинный. В противном случае вызывается функция применения правил упрощения `ApplyRules`, которой на вход помимо атома для упрощения передается $prevAtom$, предшествующий $atom$, в последовательности упрощений (в начале $atom = prevAtom$). Это требуется, потому что, последовательное применение правил подтипирования может зациклиться на тавтологически ложном атоме.

Функция `SimplifyAtom` возвращает формулу ψ_a, логически эквивалентную $atom$.

Листинг 4: Функция применения правил упрощения `ApplyRules`

Вход: $atom$ — атом для упрощения, $prevAtom$ — атом, предшествующий $atom$ в последовательности упрощений
Выход: ψ_a — упрощенная формула, логически эквивалентная $atom$

```
1  if IsEqualityAtom(atom) then
2  |   ψ_s ← ApplyEqRule(atom)
3  end
4  else
5  |   if HasSameHeadConstructors(atom.left, atom.right) then
6  |   |   ψ_s ← ApplyVarRule(atom)
7  |   end
8  |   else  ψ_s ← ApplySuperRule(atom)
9  end
10 if IsAtom(ψ_s) then
11 |   if ψ_s = prevAtom then  ψ_s ← ⊥
12 |   else  ψ_a ← SimplifyAtom(ψ_s, atom)
13 end
14 else  ψ_a ← SimplifyFormula(ψ_s)
```

Применение правил упрощения `ApplyRules` предполагает выяснение на основе какого предикатного символа построен $atom$ с помощью вызова функции `IsEqualityAtom`. В случае равенства используется функция `ApplyEqRule`, возвращающая $\top$ или $\bot$, или равенство аргументов открытых типов, имеющих одинаковый головной конструктор. Если $atom$ является атомом подтипирования, тогда в зависимости от того, какие в него входят элементы, возможно применение или правила Var или правила Super. Выбор происходит на основе применения функции `HasSameHeadConstructors`, которая определяет, имеют ли два типа один и тот же головной конструктор. Если это условие выполняется, тогда можно преобразовать (упростить) атом вида $C\langle\overline{T}\rangle <: C\langle\overline{U}\rangle$ в конъюнкцию $\bigwedge_i T_i <:_{var(C\#i)} U_i$, а атом вида $C <: C$ — в $\top$. В противном случае применяется правило Super. Если это сделать не удалось, то атом считается тавтологически ложным. Если же применение данного правила оказалось возможным, тогда атом вида $C\langle\overline{T}\rangle <: D\langle\overline{U}\rangle$, где $C \neq D$, является логически эквивалентным дизъюнкции $\bigvee_j D\langle W_j\rangle <: D\langle U\rangle$, где $D\langle W_j\rangle = [\,\overline{x} \mapsto \overline{T}\,]V_j$ и $C\langle x\rangle <:: V_j$. В строке 11 алгоритм проверяет, является ли результат применения правил ψ_s атомом. Если это

условие выполняется, то необходимо определить, сравнимы ли ψ_s и *prevAtom*. Это требуется для того, чтобы исключить зацикливание последовательного применения правила SUPER и правила VAR. Если они несравнимы, то запускается процесс SimplifyAtom с ψ_s в качестве атома для упрощения и *atom* в качестве *prevAtom*. В ситуации, когда ψ_s представляет собой формулу, вызывается функция `SimplifyFormula`. Алгоритм `ApplyRules` применяет правила, пока это возможно, и возвращает формулу ψ_a, логически эквивалентную *atom* и включающую в себя только простые атомы.

Поскольку недетерменированное применение правил подтипирования с проверкой повторного появления типов является разрешающей процедурой для нерасширяющегося наследования [12], то или цепочка упрощений зацикливается и атом тавтологически ложен, или процесс упрощения завершается.

Пример 4. Упрощение формулы.

Для того, чтобы продемонстрировать вышеописанный алгоритм, предлагается использовать таблицу классов CT и формулу ϕ из примера 3.

Данная формула включает непосредственные подформулы $\phi_1 = \texttt{C} <: \texttt{I<C>}$ и $\phi_2 = x <: \texttt{I<}y\texttt{>} \wedge \texttt{J<}x\texttt{>} \not<: \texttt{J<}y\texttt{>}$, связанные дизъюнкцией. Рассмотрим упрощение каждой из них.

1. $\psi_1 = \texttt{C} <: \texttt{I<C>} \xrightarrow{super} \texttt{I<I<C>>} <: \texttt{I<C>} \xrightarrow{var} \texttt{C} <: \texttt{I<C>} \xrightarrow{cycle} \bot$
2. Первый элемент подформулы ϕ_2 уже простой атом, поэтому остается упростить только второй элемент $\texttt{J<}x\texttt{>} \not<: \texttt{J<}y\texttt{>} \xrightarrow{var} x \not<: y$.
   Упрощенная подформула $\psi_2 = x <: \texttt{I<}y\texttt{>} \wedge x \not<: y$.

Согласно алгоритму, каждую подформулу исходной формулы требуется заменить на упрощенную. В данном случае $\phi_n = \psi_1 \vee \psi_2$ или $\phi_n = \bot \vee (x <: \texttt{I<}y\texttt{>} \wedge x \not<: y)$. Удаление логической константы из ϕ_n дает результирующую формулу $\phi_s = x <: \texttt{I<}y\texttt{>} \wedge x \not<: y$.

4.2. Алгоритм кодирования формулы во множество дизъюнктов Хорна

Для того, чтобы учесть особенности системы типов .NET (см. раздел 2.1) и использовать систему автоматического доказательства теорем, требуется представить формулу ϕ языка $\mathcal{L}_s$ (она находится в негативной нормальной форме), в виде множества формул *Clauses* другого языка первого порядка $\mathcal{L}_\hbar$, в котором запрещено использовать отрицание.

Язык $\mathcal{L}_\hbar$ описывается сигнатурой $\Sigma_h = (\mathcal{C}, \mathcal{P}_\hbar)$. Здесь $\mathcal{C}$ — множество функциональных символов, отождествляемое с множеством конструкторов из CT, а $\mathcal{P}_\hbar = \mathcal{P}_\hbar{}^+ \cup \mathcal{P}_\hbar{}^-$ — множество предикатных символов. Множество $\mathcal{P}_\hbar{}^+$ представляет «положительные» предикатные символы (арность указана в скобках) *covar_ subtype(2)*, *contrvar_ subtype(2)*, *subtype(2)*, *is_ reference(1)*, *is_ valuetype(1)*, *is_ unmanaged(1)*, *def_ constructor(1)*. Для каждого двухместного предикатного символа множество $\mathcal{P}_\hbar{}^-$ содержит предикатный символ, выражающий операцию отрицания *not_ covar_ subtype(2)*, *not_ contrvar_ subtype(2)*, *not_ subtype(2)* соответственно. Предикатный символ *subtype* языка $\mathcal{L}_\hbar$ соответствует предикатному символу $<:$ языка $\mathcal{L}_s$.

Для того, чтобы описать поведение обобщенных типов, в зависимости от вида передаваемых параметров, требуется определить аксиомы вариантности следующим образом.

Определение 5. Аксиомы вариантности.

1. $\forall x, y.\ is\_valuetype(x) \wedge x = y \Rightarrow covar\_subtype(x, y)$
2. $\forall x, y.\ is\_valuetype(y) \wedge x = y \Rightarrow contrvar\_subtype(x, y)$
3. $\forall x, y.\ is\_valuetype(x) \wedge x \neq y \Rightarrow not\_covar\_subtype(x, y)$
4. $\forall x, y.\ is\_valuetype(y) \wedge x \neq y \Rightarrow not\_contrvar\_subtype(x, y)$
5. $\forall x, y.\ is\_reference(x) \wedge subtype(x, y) \Rightarrow covar\_subtype(x, y)$
6. $\forall x, y.\ is\_reference(y) \wedge subtype(x, y) \Rightarrow contrvar\_subtype(x, y)$
7. $\forall x, y.\ is\_reference(x) \wedge not\_subtype(x, y) \Rightarrow not\_covar\_subtype(x, y)$
8. $\forall x, y.\ is\_reference(y) \wedge not\_subtype(x, y) \Rightarrow not\_contrvar\_subtype(x, y)$

Общая схема перевода формулы ϕ в язык $\mathcal{L}_\hbar$ представлена на лист. 5.

Листинг 5: Функция кодирование формулы `Encode`

Вход: ϕ — формула языка $\mathcal{L}_s$

Выход: $Clauses$ — множество формул языка $\mathcal{L}_h$

Исходные параметры: $VarianceAxioms$ — аксиомы вариантности

1 $Types \leftarrow$ `GetTypes`(ϕ)

2 $ClassTable \leftarrow$ `ClosureClassTable`($Types$)

3 $CTClauses \leftarrow$ `EncodeCTClauses`($ClassTable$)

4 $SpecialFacts \leftarrow$ `EncodeSpecialFacts`($ClassTable$)

5 $\phi_h \leftarrow$ `EncodeFormula` (ϕ)

6 $Clauses \leftarrow CTClauses \cup SpecialFacts \cup VarianceAxioms \cup \phi_h$

Начальным этапом кодирования является получение всех типов из формулы, а также всех ограничений типовых переменных, которые в этой формуле встречаются, с помощью функции `GetTypes`, которая возвращает множество типов *Types*.

Затем строится релевантная таблица классов для данного множества *Types*, замкнутая относительно наследования, декомпозиции и ограничений, с помощью применения функции `ClosureClassTable`.

Определение 6. Множество типов S называется замкнутым относительно наследования, декомпозиции и ограничений (далее просто замкнутым), если выполнены следующие условия.

1. $C\texttt{<}\overline{T}\texttt{>} \in S \Rightarrow \forall i, T_i \in S$
2. $T \in S \wedge T <:: V_1, \ldots, V_n \Rightarrow \forall i, V_i \in S$
3. $X \in S \Rightarrow \forall i, U_i \in S$, где U_i — элементы правой части ограничения $X <: U_1, \ldots, U_n$

Будем называть замыканием множества S минимальное замкнутое множество, содержащее S (обозначается $cl(S)$). Известно, что для нерасширяющихся таблиц классов замыкание конечного множества типов конечно.

Псевдокод алгоритма построения замыкания таблицы классов представлен на лист. 6. На вход функции `ClosureClassTable` поступает множество типов *Types*, из которых требуется сформировать таблицу классов *ClassTable*, замкнутую относительно наследования, декомпозиции и

Листинг 6: Функция построения замыкания множества типов `ClosureClassTable`

```
Вход: Types— множество типов
Выход: ClassTable — замыкание таблицы классов в виде множества пар
       ключ-значение
1  ClassTable ← ∅
2  while Types ≠ ∅ do
3      type ← GetTypes(Types)
4      if IsGenericType(type) then
5          TypeArguments ← GetGenericArguments(type)
6          foreach arg ∈ TypeArguments do
7              if not IsGenericParameter(arg) then
8                  Types ← Types ∪ {arg }
9              end
10         end
11         typeDefinition ← GetTypeDefinition(type)
12         DefinitionArguments ← GetGenericArguments(typeDefinition)
13         foreach arg ∈ DefinitionArguments do
14             Types ← Types ∪ GetArgsConstraints(arg)
15         end
16         type ← typeDefinition
17     end
18     if not ContainsKey(ClassTable, type) then
19         AddKey(ClassTable, type)
20         AddValue(ClassTable, type, GetSupertypes(type))
21     end
22 end
```

ограничений, в виде множества пар ключ-значение, где ключом является тип, а значением множество его номинальных надтипов. В цикле из множества типов *Types* с помощью функции `GetTypes` берется *type* с удалением из этого множества. Для *type* проверяется содержит ли тип аргументы с помощью `IsGenericType`. Если проверка прошла успешно, то каждый аргумент, который не является типовой переменной, добавляется в множество типов для замыкания *Types*. Затем у типа берется его «схема построения» *typeDefinition* с помощью `GetTypeDefinition` и во множество *Types* добавляются все ограничения всех типовых переменных *typeDefinition*. Вышеописанные действия относятся к замыканию таблицы классов относительно декомпозиции и ограничений. По определению 1 в левой части каждой записи должно быть уникальное объявление типа, поэтому для дальнейшего построения таблицы

классов *type* заменяется на *typeDefinition*. В том случае, если таблица классов не содержит соответствующей записи для *type*, в нее добавляется ключ *type* и значение — множество номинальных надтипов *type*, полученное с помощью применения функции `GetSupertypes`.

Пример 5. Пример замыкания таблицы классов. Для упрощенной формулы $\phi_s = x <: \texttt{I<}y\texttt{>} \wedge x \not<: y$ из примера 4 релевантная таблица классов имеет следующий вид.

$$\begin{array}{lll} \texttt{Object}^R & <:: & \\ \texttt{I}^R\texttt{<} - x^R\texttt{>} & <:: & \texttt{Object} \end{array}$$

Следующим шагом алгоритма `Encode` является перевод построенной таблицы классов *ClassTable* в множество дизъюнктов Хорна с помощью применения функции `EncodeCTClauses`, псевдокод которой представлен на лист. 7. Основная идея данной функции состоит в том, чтобы составить все возможные пары из типов, которые являются ключами таблицы классов (функция `GetKeys`), и описать при каких условиях данные типы подтипируют или не подтипируют друг друга. В случае, если в составленной паре появляются одинаковые типы, то аналогично упрощению, фактически требуется применить правило VAR с помощью `EncodeWithVarRule`. Если типы в паре различны, требуется попробовать применить правило SUPER с помощью `EncodeWithSuperRule`.

Замечание. Кодирование правил описывается в терминах раздела 2.3.1.

Функция `EncodeWithVarRule` представлена на лист. 8. Для данных двух типов *typeX* и *typeY* известно, что они имеют одинаковые конструкторы, следовательно в любом случае для них будет сгенерировано правило подтипирования. Требуется сформировать голову *subtypeHead* будущего правила с помощью применения функции `EncodePredicate`, которой на вход подается предикатный символ, в данном случае *subtype*, и кортеж его аргументов.

Типы *typeX* и *typeY* могут иметь аргументы, поэтому необходимо построить множество *ArgsConstraints* (строка 4), которое содержит

Листинг 7: Функция кодирования таблицы классов во множество дизъюнктов Хорна `EncodeCTClauses`

Вход: $ClassTable$ — замыкание таблицы классов в виде множества пар ключ-значение
Выход: $CTClauses$ — множество дизъюнктов Хорна представляющее $ClassTable$

```
1  CTClauses ← ∅
2  Types ← GetKeys(ClassTable)
3  foreach typeX ∈ Types do
4      foreach typeY ∈ Types do
5          if typeX = typeY then
6              CTClauses ← CTClauses ∪ EncodeWithVarRule(typeX, typeY)
7          end
8          else
9              CTClauses ← CTClauses ∪
                EncodeWithSuperRule(typeX, typeY, ClassTable)
10         end
11     end
12 end
```

все ограничения всех аргументов данных типов. Применение функции `GetArgsConstraints` к типу сформирует, с помощью `EncodePredicate`, для каждого его аргумента предикаты *subtype* для аргумента и его ограничения, а также предикаты *is_ reference*, *is_ valuetype*, *is_ unmanaged*, *def_ constructor* для специальных ограничений, если они накладываются на данный аргумент.

Далее требуется построить тело правила подтипирования. Для типов $typeX = C$ и $typeY = C$ тело правила выглядит так: $SubtypeBody = \varnothing$. Для $typeX = C\texttt{<}\overline{T}\texttt{>}$ и $typeY = C\texttt{<}\overline{U}\texttt{>}$ тело правила подтипирования $SubtypeBody$ является множеством предикатов *covar_ subtype*, *contrvar_ subtype* и равенств, сгенерированных в соответствие с вариантностью для каждого элемента конъюнкции $\bigwedge_i T_i <:_{var(C\#i)} U_i$. Далее формируется соответствующий дизъюнкт для отношения подтипирования и добавляется в $Clauses$ (строка 10).

Правила неподтипирования необходимо сформулировать только для $typeX = C\texttt{<}\overline{T}\texttt{>}$ и $typeY = C\texttt{<}\overline{U}\texttt{>}$. Данные типы не подтипируют друг друга, если нарушается хотя бы один $SubtypeBody$. Поэтому для формирования правила требуется сгенерировать голову $notSubtypeHead$ и

Листинг 8: Функция кодирования `EncodeWithVarRule`

Вход: $typeX$, $typeY$ — типы, для которых генерируются правила подтипирования и неподтипирования
Выход: $Clauses$ — множество дизъюнктов Хорна описывающее правила подтипирования и неподтипирования для $typeX$ и $typeY$

```
1  Clauses ← ∅
2  subtypeHead ← EncodePredicate(subtype, (typeX, typeY))
3  ArgsConstraints ← GetArgsConstraints(typeX) ∪
     GetArgsConstraints(typeY)
4  SubtypeBody ← GetArgsRelation(subtype, typeX, typeY)
5  Clauses ← Clauses ∪
     {EncodeClause(subtypeHead, SubtypeBody ∪ ArgsConstraints)}
6  notSubtypeHead ← EncodePredicate(not_subtype, (typeX, typeY))
7  NotSubtypeBody ← NegateRelations(SubtypeBody)
8  foreach bodyElement ∈ NotSubtypeBody do
9  |   Clauses ← Clauses ∪
   |     EncodeClause(notSubtypeHead, {bodyElement} ∪ ArgsConstraints)
10 end
```

элементы тела *NotSubtypeBody* данного правила, путем замены каждого элемента *SubtypeBody* на противоположный. Затем для каждого *bodyElement*, принадлежащего множеству *NotSubtypeBody* формируется дизъюнкт с *notSubtypeHead* в качестве головы и телом состоящим из объединения *bodyElement* с множеством ограничений *ArgsConstraints*, который добавляется в результирующее множество *Clauses*.

В качестве результата работы `EncodeWithVarRule` возвращает множество *Clauses* дизъюнктов Хорна.

Функция `EncodeWithSuperRule` представлена на лист. 9. Данная функция по таблице классов *ClassTable* получает множество номинальных надтипов $supersX$ для $typeX$, каждый из которых имеет головной конструктор как $typeY$ (строка 4). Если $supersX = \varnothing$, то в результирующее множество *Clauses* добавляется факт неподтипирования для $typeX$ и $typeY$ (строки 5-8). Для случая, когда $supersX$ непусто необходимо, по аналогии с `EncodeWithVarRule`, сформировать множество *ArgsConstraints* ограничений всех типовых переменных $typeX$ и $typeY$. Затем для каждого надтипа $superX$ из множества $supersX$ формируется элемент тела правила *bodyElement* с помощью `EncodePredicate` (строка 13). Данный элемент и множество ограничений используются в каче-

Листинг 9: Функция кодирования `EncodeWithSuperRule`

Вход: $typeX$, $typeY$ — типы, для которых генерируются правила подтипирования и неподтипирования, $ClassTable$ — таблица классов
Выход: $Clauses$ — множество дизъюнктов Хорна описывающее правила подтипирования и неподтипирования для $typeX$ и $typeY$

```
1  Clauses ← ∅
2  NotSubtypeBody ← ∅
3  supersX ← GetSupertypes(typeX, typeY, ClassTable)
4  notSubtypeHead ← EncodePredicate(not_subtype, typeX, typeY)
5  if supersX = ∅ then
6  |  Clauses ← Clauses ∪ {EncodeClause(notSubtypeHead, ∅)}
7  end
8  else
9  |  ArgsConstraints ← GetArgsConstraints(typeX) ∪
   |   GetArgsConstraints(typeY)
10 |  subtypeHead ← EncodePredicate(subtype, typeX, typeY)
11 |  foreach superX ∈ supersX do
12 |  |  bodyElement ← EncodePredicate(subtype, superX, typeY)
13 |  |  Clauses ← Clauses ∪
   |  |   EncodeClause(subtypeHead, {bodyElement} ∪ ArgsConstraints)
14 |  |  NotSubtypeBody ← NotSubtypeBody ∪
   |  |   EncodePredicate(not_subtype, superX, typeY)
15 |  end
16 |  NotSubtypeBody ← NotSubtypeBody ∪ ArgsConstraints
17 |  Clauses ← Clauses ∪ {EncodeClause(notSubtypeHead, NotSubtypeBody)}
18 end
```

стве тела правила, которое добавляется в результирующее множество дизъюнктов *Clauses* в строке 14. Для того, чтобы в дальнейшем описать правило неподтипирования для $typeX$ и $typeY$ формируется множество *NotSubtypeBody*, которое включает противоположные элементы для всех *bodyElement* и *ArgsConstraints*. Данное правило добавляется в результирующее множество *Clauses* в строке 18.

Пример 6. Таблица классов из примера 5 переводится в следующие дизъюнкты Хорна первого порядка.

- $\forall x.\ subtype(\texttt{I}(x)\,,\ \texttt{Object})$
- $subtype(\texttt{Object}\,,\ \texttt{Object})$
- $\forall x.\ not\_subtype(\texttt{Object}\,,\ \texttt{I}(x))$
- $\forall x, y.\ subtype(\texttt{I}(x)\,,\ \texttt{I}(y)) \leftarrow is\_reference(x) \wedge is\_reference(y) \wedge covar\_subtype(x\,,\ y)$
- $\forall x, y.\ not\_subtype(\texttt{I}(x)\,,\ \texttt{I}(y)) \leftarrow is\_reference(x) \wedge is\_reference(y) \wedge not\_covar\_subtype(x\,,\ y)$

После выполнения функции EncodeCTClauses, алгоритм `Encode` на основании таблицы классов создает множество *SpecialFacts* с помощью применения функции `EncodeSpecialFacts`. Данная функция для каждого типа из множества ключей таблицы классов *ClassTable* генерирует факт, если данный тип удовлетворяет одному или нескольким специальным ограничениям типовых параметров (*is_ reference*, *is_ valuetype*, *is_ unmanaged*, *def_ constructor*).

Пример 7. Пример множества SpecialFacts. Для таблицы классов из примера 5 множество $SpecialFacts$ определяется следующим образом.

$$SpecialFacts = \{def\_constructor(\texttt{Object}), is\_reference(\texttt{Object}), \\ is\_reference(\texttt{I})\}$$

Кодирование формулы `EncodeFormula` в алгоритме `Encode` включает замену предикатного символа подтипирования $<:$ и отрицания подтипирования $T \not<: U$ на *subtype* и *not_ subtype* соответственно. А также добавление ограничений для всех типовых переменных, которые встречаются в формуле.

Пример 8. Пример кодирования формулы.

Формула $\phi_s = x <: \texttt{I<}y\texttt{>} \wedge x \not<: y$ из примера 4 кодируется, как $\forall x, y.\ subtype(x\,,\, \texttt{I}(y)) \wedge not\_subtype(x\,,\, y)$.

Объединение множеств *SpecialFacts*, *VarianceAxioms* из определения 5 и *CTClauses* — множество гипотез, а ϕ_h — это предположение. Множество *Clauses* может быть использовано в качестве входных данных для системы автоматического доказательства теорем.

4.3. Реализация алгоритма

Вышеописанный алгоритм был реализован на языке F# в проекте V# в подсистеме `VSharp.Solver` в виде модуля `TypeSolver.fs` и метода `EncodeTypePC` модуля `Encode.fs`.

Диаграмма компонентов реализации представлена на рис. 2.

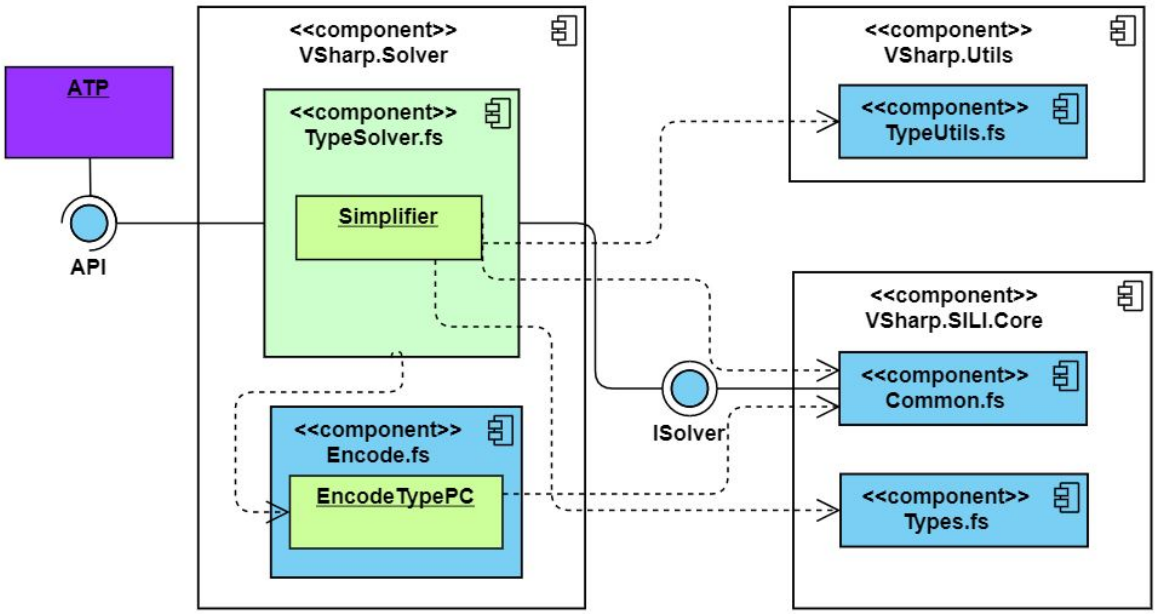

Рис. 2: Диаграмма компонентов `TypeSolver.fs`

Компонент `TypeSolver.fs` относится к подсистеме решения ограничений на условия пути `VSharp.Solver`. Он предоставляет интерфейс `ISolver`, посредством которого получает запросы на подтипирование. Компонент `TypeSolver.fs` отвечает за упрощение полученного запроса с помощью применения правил подтипирования (объект Simplifier), за принятие решения о необходимости кодирования запроса с помощью метода `EncodeTypePC`, за взаимодействие с системой автоматического доказательства теорем `ATP` через интерфейс `API` и за интерпретацию результатов полученных от нее.

Компонент `Encode.fs` подсистемы `VSharp.Solver` отвечает за кодирование ограничений на условия пути в виде формул логики первого порядка. В частности метод `EncodeTypePC` представляет запрос на подтипирование в виде множества дизъюнктов Хорна, которое в дальнейшем используется в качестве входных данных для системы автоматического доказательства теорем. Для проведения экспериментальных исследований данный метод поддерживает: синтаксис языка логического программирования Prolog, синтаксис языка реляционного программирования OCanren, синтаксис системы автоматического доказательства теорем Prover9-Mace4 и стандартный язык SMT-решателей

SMT-LIB2.

Компонент Common.fs подсистемы VSharp.SILI.Core содержит структуры данных для представления атомов подтипирования и запросов, включающих такие атомы. Данный компонент во время работы символьного исполнения формирует запросы на подтипирование, для решения которых использует интерфейс ISolver компонента TypeSolver.fs.

Компонент Types.fs подсистемы VSharp.SILI.Core содержит структуры данных, представляющие типы объектов во время символьного исполнения, и методы работы с ними.

Компонент TypeUtils.fs относится к вспомогательной подсистеме VSharp.Utils, содержащей необходимые для работы проекта прикладные структуры данных и методы их обработки. Он включает методы для работы с объектами System.Type, предоставляющими информацию о типах среды исполнения.

5. Экспериментальное исследование

В рамках экспериментального исследования разработанного алгоритма оценка эффективности работы алгоритма была проведена во время символьного исполнения методов из стандартных библиотек .NET (например, `mscorlib.dll`), а также методов библиотеки, созданной самостоятельно, с нетривиальными ограничениями типовых параметров и сложными условиями проверки типов, охватывающими все особенности системы типов .NET из раздела 2.1. На этапе упрощения было решено около 30% запросов. Для запросов, которые требуют применения системы автоматического доказательства теорем, для проекта V# рассмотрены следующие кандидаты: Prolog, OCanren, CVC4, Prover9-Mace4 и Vampire.

В ходе экспериментов было рассмотрено 56 выполнимых и 72 невыполнимых запросов на подтипирование[6]. На обработку каждого запроса был установлен лимит времени в 20 секунд.

Результаты экспериментов по количеству решенных запросов и среднему времени работы каждой системы представлены в табл. 1.

Система	Выполнимые запросы(56)		Невыполнимые запросы(72)	
	Решено	Среднее время (мс.)	Решено	Среднее время (мс.)
Prolog	50	33	38	37
OCanren	55	16	38	58
Prover9-Mace4	32	88	6	20
CVC4	48	172	6	60
Vampire	56	66	70	90

Таблица 1: Результаты экспериментов

По результатам экспериментов выявлено, что больше всего запросов было решено с использованием системы Vampire. Несмотря на то, что задача выполнимости в теории номинальных систем типов с вариантностью неразрешима, данная система решила 100% выполнимых и

[6]Тесты для каждой системы доступны по ссылке
https://github.com/MilovaNatalia/VSharp/tree/TypePC/VSharp.Test/Tests/TypeSolverTests

99% невыполнимых запросов в худшем случае за 0.2 сек. Поэтому для проекта V# в рамках алгоритма решения запросов на подтипирование в качестве системы автоматического доказательства теорем выбрана именно Vampire.

В разделах 5.1 и 5.2 проанализирована эффективность Vampire для выполнимых и невыполнимых запросов соответственно, по сравнению с остальными исследуемыми системами.

5.1. Выполнимые запросы

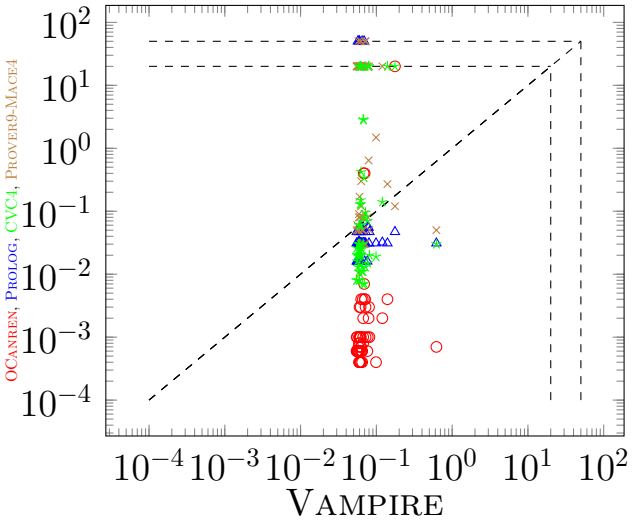

Рис. 3: Сравнение эффективности Vampire с конкурентами Prolog, OCanren, Prover9-Mace4, CVC4. Каждая фигура на графике показывает пару времен выполнения (сек × сек) Vampire (ось x) и конкурентов (ось y). Таймауты отмечены на внутренних пунктирных линиях, а завершения с ошибкой на внешних.

Первым набором тестов, на котором оценивалась эффективность работы систем автоматического доказательства, является набор из 56 выполнимых запросов на подтипирование, включающих все особенности системы типов .NETиз раздела 2.1. Проводилось сравнение системы Vampire в рамках разработанного алгоритма с конкурентами Prolog, OCanren, Prover9-Mace4, CVC4.

Система Prolog решила 50 запросов из 56 без нарушения таймаута, но с 6-ю ошибками переполнения стека, каждый запрос не более

чем за 0,05 сек. В свою очередь, OCanren справился с 55 тестами из 56 с 1-им нарушением таймаута (для 96% тестов, время решения составило не более, чем 0,01 сек.). Система Prover9-Mace4 решила 32 запроса из 56 с 19-ю нарушениями таймаута и 5-ю ошибками времени выполнения (для 94% запросов время решения не превышает 0,08 сек.). SMT-решатель CVC4 решил 48 тестов из 56 с 8-ю нарушениями таймаута (85% случаев каждый запрос обрабатывался не более чем за 0,02 сек.). Система Vampire успешно решила все 56 тестов, 95% из которых не более, чем за 0,1 сек.

Несмотря на то, что системы OCanren и Prolog показывают результаты по времени работы лучше, чем Vampire связи с тем, что задача из определения 4 полуразрешима, эффективная система доказательства должна решать, как можно больше выполнимых запросов, за приемлемое время. Основная проблема в данной работе — это подобрать систему доказательства эффективную не только с точки зрения решения выполнимых запросов, а в большей степени подходящую для решения невыполнимых запросов. Поэтому решение о использовании какой-либо системы в проекте V# принималось именно на основе набора тестов невыполнимых запросов.

5.2. Невыполнимые запросы

Вторым набором тестов, на котором оценивалась эффективность работы систем автоматического доказательства, является набор из 72 невыполнимых запросов на подтипирование.

Система Prolog решила 38 запросов из 72 с 26-ю нарушениями таймаута и с 8-ю ошибками переполнения стека, не более чем за 0,05 сек. на каждый запрос. В свою очередь, OCanren справился с таким же количеством тестов, как и Prolog но с 34-мя нарушениями таймаута (для 96% тестов, время решения составило не более, чем 0,01 сек.). Для данных систем показатель 53% успешно решенных невыполнимых запросов обусловлен тем, что для них, в связи с применяемыми стратегиями поиска, критически важным является порядок декларации

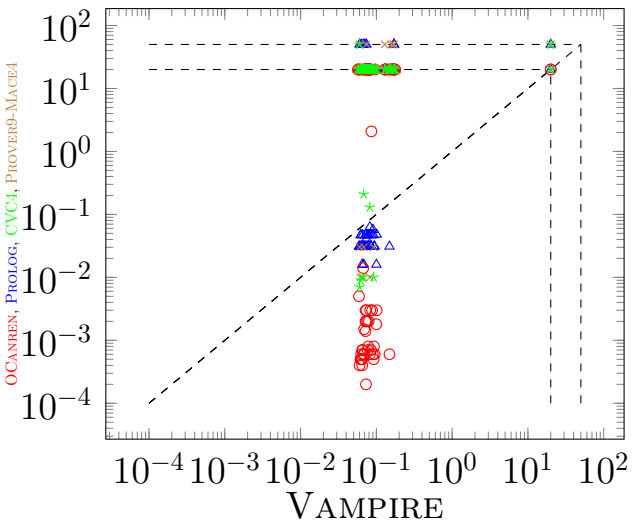

Рис. 4: Сравнение эффективности Vampire с конкурентами Prolog, OCanren, Prover9-Mace4, CVC4. Каждая фигура на графике показывает пару времен выполнения (сек × сек) Vampire (ось x) и конкурентов (ось y). Таймауты отмечены на внутренних пунктирных линиях, а завершения с ошибкой на внешних.

правил и порядок атомов в запросе, который в рамках задачи сложно регулировать.

Система Prover9-Mace4 решила 6 запросов из 72 с 58-ю нарушениями таймаута и 7-ю ошибками времени выполнения (для решенных запросов время решения не превышает 0,03 сек.). SMT-решатель CVC4 успешно решил столько же запросов, сколько Prover9-Mace4 с 66-ю нарушениями таймаута, каждый запрос не более, чем за 0,02 сек. Низкий показатель успешных решений для данных двух систем показывает, что для задачи выполнимости в теории номинальных систем типов с вариантностью поиск конечных моделей не может быть применен.

Система Vampire успешно решила 70 из 72 тестов, с 2-мя нарушениями таймаута, на специфичных запросах. Данной системой 90% решений сделаны не более, чем за 0,1 сек.

6. Заключение

В ходе данной работы были получены следующие результаты.

- В рамках анализа основных элементов системы типов на основе стандарта [10] представлены основные виды типов в .NET и описаны их характерные особенности.
- В ходе исследования задачи выполнимости в теории номинальных систем типов с вариантностью представлено формальное описание системы типов .NET и описание задачи выполнимости запросов на подтипирование для нее.
- В рамках проекта V# разработан и реализован на языке F# алгоритм проверки выполнимости запросов на подтипирование, включающий упрощение исходной формулы с помощью применения правил подтипирования и кодирование упрощенной формулы во множество дизъюнктов Хорна первого порядка с дальнейшим применением автоматических систем доказательств теорем.
- Проведены сравнительные эксперименты систем доказательств на основе Prolog, OCanren, CVC4, систем Prover9-Mace4, и Vampire по времени работы и количеству решенных запросов на подтипирование, в ходе которых использование Vampire признано более предпочтительным для проекта V#.

Список литературы

[1] A Survey of Symbolic Execution Techniques / Baldoni Roberto, Coppa Emilio, D'elia Daniele Cono et al. // ACM Comput. Surv. — 2018. — Vol. 51, no. 3.

[2] Arthan Rob, Oliva Paulo. Studying Algebraic Structures using Prover9 and Mace4. — 2019.

[3] C. Pierce Benjamin. Types and Programming Languages. — 1 edition. — MIT Press, 2002.

[4] CVC4 / Clark Barrett, Christopher L. Conway, Morgan Deters et al. // Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11) / Ed. by Ganesh Gopalakrishnan, Shaz Qadeer. — Vol. 6806 of Lecture Notes in Computer Science. — Springer, 2011. — Jul. — P. 171–177. — Snowbird, Utah.

[5] Dmitrii Kosarev, Dmitry Boulytchev. Typed Embedding of a Relational Language in OCaml // Proceedings ML Family Workshop / OCaml Users and Developers workshops, Nara, Japan, September 22-23, 2016 / Ed. by Kenichi Asai, Mark Shinwell. — Vol. 285 of Electronic Proceedings in Theoretical Computer Science. — Open Publishing Association, 2018. — P. 1–22.

[6] E. Byrd William. Relational Programming in Minikanren: Techniques, Applications, and Implementations : Ph. D. thesis / Byrd William E. — USA : Indiana University, 2009.

[7] Efficient State Merging in Symbolic Execution / Volodymyr Kuznetsov, Johannes Kinder, Stefan Bucur, George Candea // Acm Sigplan Notices. — 2012. — Vol. 47, no. 6. — P. 193–204.

[8] Horn Clause Solvers for Program Verification / Nikolaj Bjørner, Arie Gurfinkel, Ken McMillan, Andrey Rybalchenko // Fields of Logic and Computation II. — Springer, 2015. — P. 24–51.

[9] Höfner Peter, Struth Georg. Automated Reasoning in Kleene Algebra. — 2007. — 07. — P. 279–294.

[10] InternationalS ECMA. Standard ECMA-335 - Common Language Infrastructure (CLI). — 5 edition. — Geneva, Switzerland, 2010.

[11] Ivan Bratko. Prolog Programming for Artificial Intelligence. — 3 edition. — Harlow, England : Pearson Addison-Wesley, 1987.

[12] J Kennedy Andrew, C Pierce Benjamin. On decidability of nominal subtyping with variance. — 2007.

[13] Java generics are Turing complete // POPL. — ACM, 2017. — P. 73–85.

[14] Kovács Laura, Voronkov Andrei. First-Order Theorem Proving and Vampire // Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings. — 2013. — P. 1–35.

[15] L. Sterling, E. Shapiro. The Art of Prolog. — Cambridge, MA : MIT Press, 1986.

[16] Translucent Sums: A Foundation for Higher-Order Module Systems : Rep. ; Executor: Mark Lillibridge : 1997.

[17] Lisitsa Alexei. Finite Model Finding for Parameterized Verification // CoRR. — 2010. — Vol. abs/1011.0447.

[18] Lisitsa Alexei. Finite countermodels for safety verification of parameterized tree systems // CoRR. — 2011. — Vol. abs/1107.5142.

[19] Martin Odersky. Scaling DOT to Scala–soundness. — 2016.

[20] McCune William. Mace4 Reference Manual and Guide // CoRR. — 2003.

[21] McCune William. Mace4 Reference Manual and Guide // CoRR. — 2003. — Vol. cs.SC/0310055.

[22] McCune William. Prover9 Manual. — 2020. — may. — Access mode: https://github.com/theoremprover-museum/prover9/blob/master/manual/finalbook.pdf.

[23] Misonizhnik Aleksandr, 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. — P. 7:1–7:20.

[24] Reynolds Andrew, Tinelli Cesare, Barrett Clark W. Constraint Solving for Finite Model Finding in SMT Solvers // CoRR. — 2017. — Vol. abs/1706.00096.

[25] Rossberg Andreas. Undecidability of OCaml type checking. — 1999.

[26] Sherman Elena, Garvin Brady J., Dwyer Matthew B. Deciding Type-Based Partial-Order Constraints for Path-Sensitive Analysis // ACM Trans. Softw. Eng. Methodol. — 2015. — Vol. 24, no. 3. — P. 15:1–15:33.

[27] Specification and Verification: The Spec# Experience / Barnett Michael, Fähndrich Manuel, Leino K Rustan M et al. // Communications of the ACM. — 2011. — 06. — Vol. 54. — P. 81–91.

[28] Veldhuizen Todd L. C++ Templates are Turing complete. — 2003.

[29] de Moura Leonardo Mendonça, Bjørner Nikolaj. Satisfiability modulo theories: introduction and applications // Commun. ACM. — 2011. — Vol. 54. — P. 69–77.