Федеральное государственное автономное образовательное учреждение
высшего профессионального образования
«ЮЖНЫЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»

Факультет математики, механики и компьютерных наук

Направление подготовки 010300
«Фундаментальная информатика и информационные технологии»

Ю. В. Белякова

МОДЕЛЬ КОНЦЕПТОВ В ИМПЕРАТИВНОМ ЯЗЫКЕ ПРОГРАММИРОВАНИЯ

Магистерская диссертация

Научный руководитель:
доцент каф. АДМ, кандидат физ.-мат. наук
С. С. Михалкович

Рецензент:
доцент каф. ИВЭ, кандидат физ.-мат. наук
Д. В. Дубров

Ростов-на-Дону
2014

Оглавление

Введение 4

Постановка задачи 13

1 .NET концепты: дизайн и использование 14

1.1 Интерфейсы как ограничения на типы: анализ 15

1.2 Дизайн конструкций, связанных с концептами 22

1.2.1 Обзор конструкций 22

1.2.2 Концепты 24

1.2.3 Обобщённый код 31

1.2.4 Модели 35

1.3 Концепт-требования как замена ограничений-интерфейсов . 40

2 Семантический анализ и трансляция 43

2.1 Унификация ограничений 45

2.1.1 Введение 45

2.1.2 Алгоритм унификации 48

2.1.3 Завершимость алгоритма 52

2.1.4 Вычислительная сложность 57

2.2 Трансляция 59

2.2.1 Обзор 59

2.2.2 Трансляция концептов 60

2.2.3 Трансляция обобщённого кода 66

2.2.4 Трансляция моделей 71

3 Программная реализация алгоритма унификации 75

3.1 Обзор приложения 75

3.2 Особенности реализации 82

Заключение 84

Литература 86

Введение

Большинство современных языков программирования поддерживают средства **обобщённого** программирования. Обобщённое программирование — это программирование в терминах абстракций (или *концептов*), а не конкретных типов. Вместо конкретных типов используются **типовые параметры**, удовлетворяющие этим абстракциям.

Обобщённые структуры данных и обобщённые алгоритмы могут работать с любыми конкретными типами, удовлетворяющими соответствующим абстракциям, что обеспечивает возможность повторного использования кода. Например, обобщённое бинарное дерево поиска может быть использовано с любым конкретным типом элементов дерева, который обеспечивает возможность их сравнения. Таким образом, имея одну обобщённую структуру данных, можно создать множество объектов разных типов (дерево целых, дерево строк, ...).

Механизмы обобщённого программирования

Существует два основных подхода к реализации средств обобщённого программирования в языке:

«Можно только то, что явно разрешено». Обобщённое программирование на основе *явных ограничений*. Обобщённый код может

содержать только такие операции с объектами типовых параметров, которые описаны явно. Этот подход реализован в большинстве языков, например: SML (сигнатуры), Haskell (классы типов), C# (интерфейсы), Java (интерфейсы).

«Можно всё, что не запрещено». Пример такого подхода — шаблоны C++. Обобщённый код может содержать любые синтаксически-правильные конструкции.

Проблемой первого подхода является *возможная* бедность ограничений на типовые параметры, которая не позволяет программисту выражать желаемые абстракции. Шаблоны C++ (второй подход) обеспечивают гораздо большую гибкость, но обладают рядом серьёзных недостатков [2, 3]:

1. *Позднее обнаружение ошибок.* Код шаблона подвергается лишь минимальной проверке семантики. При инстанцировании шаблона каждым конкретным типом происходит полный анализ кода инстанции. Обобщённый код может содержать семантические ошибки, из-за которых его невозможно инстанцировать *ни одним* конкретным типом, но это не может быть обнаружено на этапе анализа шаблона.
2. *Сложные сообщения об ошибках.* Использование специфических операций в теле шаблона накладывает набор ограничений на типы-параметры шаблона, которыми он может быть инстанцирован. Эти требования, как правило, задаются в форме документации, которая никак не проверяется компилятором. Если шаблон не может быть инстанцирован некоторым конкретным типом из-за того, что тип не удовлетворяет этим ограничениям, соответствующее сообщение об ошибке не отражает этого в явном виде, а отсылает к исходно-

му коду шаблона. Часто такое сообщение представляется довольно сложным.

3. *Невозможность раздельной компиляции.* Код шаблона хранится в виде синтаксического дерева [3]. Его нельзя скомпилировать в объектный файл для дальнейшего использования. При компиляции инстанции шаблона необходим исходный код шаблона.

Проблему раздельной компиляции пытались решить с помощью экспорта шаблонов [4]. Экспорт предполагал, что объявление и определение шаблона могут находиться в разных единицах трансляции, однако на деле это оказалось не вполне так. Возможность экспорта была реализована лишь одним компилятором — Comeau C++, в 2002–2003-м годах. Ввиду высокой сложности реализации в компиляторе, а также несоответствия ожиданиям пользователя, экспорт был исключен из стандарта C++11 [5].

Обобщённое программирование на основе явных ограничений

В сравнении с шаблонами C++ механизм явных ограничений представляется более перспективным, если язык программирования обеспечивает достаточный для комфортного программирования набор ограничений. Исследование [17] 2007-го года посвящено анализу средств обобщённого программирования в 8-ми языках: C++, Standard ML, Objective Caml, Haskell, Eiffel, Java и C#. В результате исследования авторы выделили несколько характеристик и возможностей механизмов обобщённого программирования, существенных для удобства программирования (Рис. 0.1).

1. Ограничения на несколько типов (multi-type concepts).
2. Множественные ограничения (multiple constraints).
3. Доступ к ассоциированным типам (associated type access).
4. Ограничения на ассоциированные типы (constraints on associated types).
5. Возможность адаптации типа после его определения (retroactive modeling).
6. Синонимы (type aliases).
7. Раздельная компиляция обобщённых методов/типов и их инстанций (separate compilation).
8. Неявный вывод типов аргументов (implicit argument deduction).

Рис. 0.1: Основные возможности обобщённого программирования

Среди ограничений на типы особое место занимают требования равенства некоторых типов (same-type constraint) и подтипирования. Возможность *адаптации типа* означает, что после того, как тип уже был полностью определён, его можно адаптировать для инстанцирования обобщённого кода. Например, интерфейсы C# или Java (как ограничения на типовые параметры обобщённого кода) такую возможность не обеспечивают: если тип уже определён, его нельзя «заставить» реализовать интерфейс, не меняя типа.

Основываясь на результатах исследования, можно сделать следующий вывод: поддержка языком перечисленных свойств в совокупности с механизмом явных ограничений обеспечила бы высокую гибкость обобщённого программирования, не сопряжённую с проблемами шаблонов C++. В ряду функциональных языков программирования наиболее полно рассмотренные возможности реализует язык Haskell. Объектно-ориентированные языки поддерживают те или иные характеристики, но ни один из них не реализует их все.

Концепты

Концепты C++

В начале 2000-х годов началась работа над новой конструкцией обобщённого программирования для языка C++ — концептами C++. Изначально термин «концепт» использовался в документации STL (Standard Template Library) для неформального описания требований на типы-параметры шаблона [1]. Сейчас он означает самостоятельную конструкцию языка.

Концепты позволяют выражать требования, ограничения на типы. Они используются в обобщённых структурах данных и алгоритмах для явного описания требований к типовым параметрам обобщённого кода, то есть концепты — механизм *явных ограничений*. Обобщённый код может быть инстанцирован любым типом, который удовлетворяет заданным концептам. Для типа может быть задана *модель* концепта — описание способа, которым тип удовлетворяет концепту. Модели обеспечивают возможность адаптации типа.

Существует несколько проектов дизайна концептов C++ ([3], [6–8], [12]) и по крайней мере две экспериментальных реализации: ConceptGCC ([7], 2007) и ConceptClang ([11], 2011). Предполагалось, что концепты войдут в стандарт C++11 (C++0x), но в 2009-м году комитет исключил их [9, 10] (в основном из-за возможной сложности для «среднего программиста»). Сейчас основная работа ведётся в направлении «облегчённой» версии концептов — ConceptsLite [13].

Обобщённое программирование на основе концептов

Несмотря на принципиальные отличия C++ от языка Haskell (один из немногих языков, который поддерживает большинство характеристик, указанных на Рис. 0.1), анализ концептов и классов типов [18] показал, что возможности этих конструкций близки — концепты также обеспечивают высокий уровень поддержки важнейших средств обобщённого программирования. В сравнении с другими популярными не функциональными языками концепты обладают значительным преимуществом. Несмотря на то, что в C++ они пока не реализованы, есть попытки применить идею концептов к другим языкам программирования.

1. Докторская диссертация Джереми Сика [15] «Язык для обобщённого программирования» посвящена языку G, который представляет собой подмножество языка C++ с концептами. Компилятор G переводит код на языке G в код на C++. При этом концепты переводятся в абстрактные классы, а модели — в наследников этих классов. Обобщённые методы и типы принимают объекты соответствующих типов.
2. Работа [16] рассматривает расширение интерфейсов C# ассоциированными типами и распространением ограничений, которое несколько приближает к концептам возможности интерфейсов. Код на языке C# с расширенными интерфейсами переводится в обычный C#.
3. Идея программирования в стиле концептов на языке Scala рассматривается в [14], авторы называют этот подход **«concept pattern»**. Моделировать концепты можно и в других объектно-ориентированных языках, если писать код аналогичный тому, во что переводятся кон-

цепты языка G. Однако такой способ сложно назвать обобщённым программированием, а соответствующий код оказывается крайне объёмным и трудно поддерживаемым. Такие возможности языка Scala, как traits (нечто вроде интерфейсов с реализацией) и implicits (параметры методов и поля классов по умолчанию) облегчают программирование в стиле концептов, но это всё же не концепты, а их «низкоуровневое» моделирование: обобщённые traits используются как концепты, в качестве моделей выступают их объекты-наследники; обобщённые методы и типы принимают объекты-traits, в инстанции передаются конкретные объекты-модели (они могут передаваться неявно).

Стоит отметить, что «concept pattern» — не единственный способ обобщённого программирования в Scala, а сам язык развивается. Поэтому проводить сравнение возможностей Scala и концептов пока рано.

Проблема концептов в языке G состоит в следующем: код на C++, в который переводится код на G, теряет некоторую существенную информацию о концептах (в частности, информацию о типовых параметрах и ассоциированных типах). Это значит, что объектный файл, полученный в результате компиляции, не может быть использован в проекте на языке G, так как исходная информация о концептах не может быть восстановлена и сопоставлена с обобщённым кодом. Таким образом, язык G обеспечивает *раздельную проверку типов* (обобщённый код проверяется относительно заданных ограничений-концептов, а при инстанцировании конкретные типы проверяются на наличие моделей), в также *частично* поддерживает *раздельную компиляцию* (обобщённый код компилируется независимо от его инстанций) и *модульность* (обобщённый код компилируется в объектный файл).

Аналогичная проблема возникает в C# с расширенными интерфейсами: код на C#, который получается в результате перевода расширенного C#, не позволяет восстановить всю исходную информацию. Кроме того, расширенные интерфейсы не обеспечивают такие существенные возможности обобщённого программирования, как ограничения на несколько типов и адаптация типа. Даже методы расширения не могут решить эту проблему: классы должны реализовывать интерфейсы при определении.

Модульность и **полностью раздельная компиляция** (компиляция инстанции обобщённого алгоритма или структуры по скомпилированному модулю, а не исходному коду) находят широкое применение в современном программировании. Так, например, распространяются обобщённые .dll-библиотеки платформы .NET. Введение новых конструкций в язык не должно нарушать эти свойства. Метод трансляции в базовый язык (как это сделано для G и C# с расширениями) — один из возможных подходов к реализации новых конструкций языка программирования в компиляторе. Однако для сохранения модульности и обеспечения раздельной компиляции кода на расширенном языке *трансляция должна сохранять информацию, необходимую для восстановления исходного кода.*

В данной работе представлен проект дизайна концептов для подмножества .NET-языков на примере языка C#. Возможности концептов в целом превосходят обобщённые средства платформы .NET, но имеют одно существенное отличие: концепты, разработанные для C++, не поддерживают ограничения *подтипирования*, которые являются одним из основных видов ограничений обобщённого кода в .NET-языках. Предлагаемый дизайн концептов .NET включает эти ограничения, что усложняет семантический анализ конструкций, зависящих от концептов. Один

из основных этапов анализа — алгоритм унификации — рассматривается подробно, также представлена его реализация.

Гл. 2 посвящена методу трансляции обобщённого кода с концептами в базовый C#. Трансляция сохраняет информацию о концептах и обобщённых конструкциях, что является необходимым условием обеспечения модульности, как было упомянуто выше. При переводе активно используются специфические возможности платформы .NET: MSIL (Microsoft Intermediate Language) позволяет хранить обобщённый код и некоторую метаинформацию в форме атрибутов, которая возникает в целевом коде трансляции. Именно эти достоинства MSIL оказали решающее влияние на выбор .NET в качестве платформы для адаптации концептов — без них для хранения нужной информации пришлось бы проектировать специальные вспомогательные средства[1].

[1]Можно использовать специальный DSL-язык для описания необходимой информации. Другой вариант — хранить информацию в бинарном файле. Проблема этих решений заключается в том, что они не являются частью соответствующего языка и платформы, и поэтому требуют отдельного согласования и реализации.

Постановка задачи

Целью данной работы является *проектирование дизайна концептов* для улучшения средств обобщённого программирования на платформе .NET. Введение в язык новых конструкций требует внесения изменений в компиляторы, поэтому необходима также *разработка способа реализации концептов*: в качестве такого способа выбран метод трансляции в базовый язык.

В работе решаются следующие задачи:

1. Сравнительный анализ концептов и средств обобщённого программирования в языке C#.
2. Разработка дизайна .NET концептов.
3. Разработка концепции перевода конструкций, зависящих от концептов, в базовый язык с учётом необходимости восстановления исходной информации.
4. Построение алгоритмов семантического анализа конструкций расширенного языка.
5. Реализация одного из основных этапов семантического анализа — алгоритма унификации ограничений.

Глава 1

.NET концепты: дизайн и использование

Для исследования существующих средств обобщённого программирования в языках платформы .NET и проектирования концептов мы выбрали язык C# с некоторыми ограничениями:

1. Система типов сужается до классов (в том числе обобщённых), абстрактных классов и интерфейсов. Структуры, перечислимые и другие типы не рассматриваются: их использование не добавляет принципиальной сложности, поэтому в настоящем исследовании в целях упрощения модели такие типы исключаются как несущественные.
2. Нет перегруженных методов.

Результаты исследования применимы к *императивным* и *объектно-ориентированным* .NET-языкам со *статической* типизацией и *именным* подтипированием.

В данной главе представлен анализ использования интерфейсов в качестве ограничений на типовые параметры обобщённого кода, выделены недостатки этого подхода (Гл. 1.1). Они решаются введением .NET концептов, дизайн которых рассмотрен в Гл. 1.2.

1.1 Интерфейсы как ограничения на типы: анализ

В настоящий момент интерфейсы являются основным видом *ограничений* на типовые параметры обобщённого кода. Пример интерфейса «печатаемых» объектов и его использования в этом качестве приведены на Рис. 1.1: обобщённый метод `PrintArray<T>` зависит от типового параметра `T`, удовлетворяющего интерфейсу `IPrintable`; благодаря этому ограничению метод `Println` может быть вызван для объектов `values[i]` типа `T` в теле метода.

```
/// Интерфейс печати на консоль
interface IPrintable
{
    void Print();
    void Println();
}

class Counter : IPrintable
{
    ...
    public void Print() {...}
    public void Println() {...}
}

void PrintArray<T>(T[] values)
where T : IPrintable
{
    Counter cnt = new Counter();
    for (int i = 0; i < values.Length; ++i)
    {
        cnt.Print();
        values[i].Println();
        cnt.Inc();
    }
}
```

Рис. 1.1: Использование интерфейса `IPrintable` в качестве ограничения на типовый параметр `T`

Заметим, что «классическое» назначение интерфейса заключается в его использовании в качестве *типа*. Необобщённый метод `PrintArray`, который работает с массивом элементов типа `IPrintable` как полиморфной коллекцией печатаемых объектов, также имеет право на существование (Рис. 1.2).

```
void PrintArray(IPrintable[] values)
{
    Counter cnt = new Counter();
    for (int i = 0; i < values.Length; ++i)
    {
        cnt.Print();
        values[i].Println();
        cnt.Inc();
    }
}
```

Рис. 1.2: Использование интерфейса `IPrintable` в качестве типа

Ключевое отличие методов `PrintArray<T>` и `PrintArray` заключается в том, что в первом случае *важен тип* `T` элементов массива `values` (один и тот же интерфейс могут реализовывать абсолютно разные типы, не связанные между собой даже отношением подтипирования). Для метода `PrintArray` тип элементов не имеет значения, от объектов требуется лишь возможность быть напечатанными.

Вернёмся к `PrintArray<T>` — обобщённому методу печати на консоль гомогенной коллекции. Пусть определён класс `A` с методами `Output` и `Outputln` печати объекта на консоль:

```
class A
{
    ...
    public void Output() {...}
    public void Outputln() {...}
}
```

Семантика этих методов полностью соответствует интерфейсу `IPrintable`, но класс `A` не реализует его. А значит обобщённый метод `PrintArray<T>` не может быть инстанцирован классом `A`, так как **адаптация типов** (retroactive modeling) к интерфейсам после определения невозможна.

Адаптация типов — одна из важнейших характеристик механизмов обобщённого программирования [17] (Рис. 0.1). Обобщённые библиотеки, которые не могут быть использованы для «готовых» типов, подходящих семантически (но не соответствующих синтаксически), имеют меньшую область применения и ценность в сравнении с библиотеками, которые предусматривают такую возможность.

`IComparable<T>` и `IComparer<T>`: интерфейсы-«двойники»

Стандартное пространство имён `System` платформы .NET содержит широкий набор перегруженных версий метода сортировки массива `Array.Sort<T>` [19]. Все эти методы можно разбить на два блока в зависимости от свойств типа элементов массива (Рис. 1.3):

1. тип `T` элементов массива имеет встроенную операцию сравнения — тип удовлетворяет интерфейсу `IComparable<T>`;
2. тип `T` элементов массива не имеет встроенной операции сравнения, но для элементов этого типа определён внешний компаратор `IComparer<T>`.

Блок перегруженных методов с внешним компаратором демонстрирует механизм *низкоуровневого моделирования концептов*: объект-*адаптер* `IComparer<T>` играет роль *концепта*, а объект инстанции этого класса выступает в качестве *модели* для соответствующего конкретного типа. Заметим, что методы из блока с внешним компаратором являются

#	**Встроенное сравнение**	**Внешний компаратор**
1	`Sort<T>(T[])` `where T : IComparable<T>;`	`Sort<T>(T[], IComparer<T>);`

2	`Sort<TKey, TValue>(`
	`  TKey[], TValue[])`
	`where TKey : IComparable<TKey>;`
		`Sort<TKey, TValue>(TKey[], TValue[],`
		`  IComparer<TKey>);`

3	...	...

Рис. 1.3: Методы `System.Array.Sort<T>`

«двойниками» методов со встроенной операцией сравнения. Интерфейсы не поддерживают адаптацию типов, поэтому разработчики обобщённой библиотеки вынуждены **дублировать** все методы сортировки, заменяя использование встроенных операций типа элементов массива на методы внешнего объекта-адаптера.

`Array.Sort<T>` — не единственный набор методов-двойников. Например, большинство методов расширения для перечислимых коллекций также имеют своих адаптируемых двойников:

- ```
  GroupBy<TSource, TKey, TElement>(
      Func<TSource, TKey>, Func<TSource, TElement>);
  // и
  GroupBy<TSource, TKey, TElement>(
      Func<TSource, TKey>, Func<TSource, TElement>,
      IEqualityComparer<TKey>)
  ```

- ```
  SequenceEqual<TSource>(IEnumerable<TSource>)
  // и
  SequenceEqual<TSource>(IEnumerable<TSource>,
      IEqualityComparer<TSource>))
  ```

- etc.

Если нужен не метод, а обобщённый класс, который предусматривает возможность адаптации, объект-адаптер придётся делать его по-

```
class BST<T>
    where T : IComparable<T>
{
    public BST([args]) {...}
    // lots of code
}

class BSTflexy<T>
{
    private IComparer<T> comparator;
    public BSTflexy([args,] IComparer<T> comparator){ ... }
    // lots of code
}
```

Рис. 1.4: Классы обобщённого бинарного дерева поиска

лем. Пример обобщённых классов-двойников для бинарного дерева поиска приведён на Рис. 1.4.

Ситуация усложняется ещё больше, если обобщённый тип требует более одного ограничения. Обозначим через `ISelf`N интерфейс, задающий встроенные операции типа, а через `IExt`N — интерфейс адаптера, соответствующего `ISelf`N. Пусть нужно реализовать обобщённый метод `p<T>`, типовый параметр `T` которого удовлетворяет двум ограничениям. Чтобы обеспечить возможность адаптации типа, необходимо написать четыре версии этого метода:

```
(1) p<T>(...) where T : ISelf1<T>, ISelf2<T>;
(2) p<T>(..., IExt1<T>) where T : ISelf2<T>;
(3) p<T>(..., IExt2<T>) where T : ISelf1<T>;
(4) p<T>(..., IExt1<T>, IExt2<T>);
```

Заметим, что все эти методы необходимы. Можно облегчить разработку обобщённого кода и ограничиться реализацией только наиболее гибкой версии (4). В этом случае пользователь обобщённого метода будет вынужден определять объекты-адаптеры для каждого своего типа, даже если тип сам реализует всю необходимую функциональность. Поэтому данное решение не является удачным — использование подобной

библиотеки становится громоздким.

Количество версий обобщённых конструкций зависит от числа ограничений и растёт экспоненциально: для трёх ограничений нужно написать 8 версий, для четырёх — уже 16. В таких условиях сложность разработки обобщённых библиотек весьма высока, особенно если обобщённый код содержит нескольких типовых параметров, а компоненты связаны зависимостями. Версии обобщённых компонент почти полностью дублируют друг друга, обобщённый код плохо поддерживаем. Улучшить качество кода можно, только отказавшись от возможности адаптации типов, то есть ограничиваясь не адаптируемыми конструкциями вида (1).

Формулировка проблемы

Ограничения на типовые параметры обобщённого кода в виде интерфейсов не обеспечивают возможность адаптации типов. Это можно сделать вручную, реализовав для каждой обобщённой компоненты набор модификаций, использующих внешние объекты-адаптеры вместо встроенных операций типа, гарантируемых интерфейсами. Количество требуемых модификаций зависит от числа ограничений-интерфейсов и растёт экспоненциально, увеличивая сложность разработки и понижая качество обобщённого кода.

Другие недостатки ограничений-интерфейсов были исследованы в [16]: это, в частности, отсутствие ассоциированных типов и распространения ограничений. Взяв за основу BGL (Boost Graph Library) — обобщённую библиотеку для работы с графами на C++, авторы реализуют аналогичную обобщённую библиотеку на языке C#. В контексте этой задачи выявляется бедность средств обобщённого программирования на C# — результирующий код получается трудно поддерживае-

мым, обобщённые компоненты содержат множество типовых параметров (бóльшая часть из которых носит вспомогательный характер), ограничения на типовые параметры дублируются в различных обобщённых компонентах. Все эти проблемы решаются расширением интерфейсов C# ассоциированнми типами и распространенем ограничений (обе возможности поддерживаются и концептами), поэтому останавливаться на них подробно в данной работе не будем.

Ещё одно важное свойство механизма обобщённого программирования — ограничения на несколько типов [17] (Рис. 0.1) — также не может быть непосредственно выражено с помощью интерфейсов. Пусть, например, необходимо реализовать задачу производителя-потребителя, где производитель и потребитель являются объектами некоторых классов `P` и `C`. В базовом C# производителя и потребителя придётся представить отдельными интерфейсами, параметризованными типом «товара» `V`, с которым они работают (метод `run_base`, Рис. 1.5). В расширенном C# [16] тип товара можно сделать ассоциированным типом, но тогда необходимо в явном виде указывать, что производитель и потребитель работают с одним и тем же товаром (метод `run_ext`, Рис. 1.5). Выразить ограничение «пара классов `P` и `C` — совместимые производитель и потребитель» в *одном* выражении с помощью интерфейсов нельзя, зато это можно сделать с помощью концептов (Рис. 1.6 иллюстрирует соответствующую идею[1]). Если типовые параметры обобщённого кода связаны более сложными зависимостями, чем в нашем примере, выражение их с помощью интерфейсов потребует ещё бóльшего числа ограничений (и в этом смысле ассоциированные типы даже ухудшают ситуацию, как видно из Рис. 1.5).

[1] `CProduserConsumer` — концепт, набор ограничений, которые задают требования к типам производителя и потребителя `P` и `C`, а также их связь (в том числе тип товара)

```
// C#
void run_base<P, C, V>(...)
    where P : IProducer<V>
    where C : IConsumer<V>
{ ... }

// Extended C#
void run_ext<P, C>(...)
where   P : IProducer,
        C : IConsumer,
        P.Value == C.Value
{ ... }
```

Рис. 1.5: Задача производителя-потребителя: C# и расширенный C#

```
void run<P, C>(...)
    where CProduserConsumer[P, C]
{ ... }
```

Рис. 1.6: Задача производителя-потребителя: концепты

1.2 Дизайн конструкций, связанных с концептами

1.2.1 Обзор конструкций

Рассмотрим ключевые понятия и идеи обобщённого программирования с концептами на примере листинга Рис. 1.7.

- **Концепт** — это именованный набор ограничений на типовые параметры. `CEquatible[T]` — пример концепта от одного параметра `T`, который содержит ограничения-сигнатуры функций.
- **Модель** концепта для данных конкретных типов задаёт способ, которым эти типы удовлетворяют концепту. Говорят, что тип *удовлетворяет* концепту, если для него определена соответствующая модель. Так, например, для класса `Point` на Рис. 1.7 определена

модель `CEquatible[Point]`, поэтому тип `Point` удовлетворяет концепту `CEquatible`.

- Концепт может **уточнять** другой концепт. Например, концепт полурешётки `CSemilattice[T]` уточняет концепт `CEquatible[T]`. Уточняющий концепт включает все ограничения уточняемого, а также может добавлять новые ограничения.
- *Ограничения* (или *требования*) на типовые параметры обобщённого кода выражаются с помощью концептов (будем называть их **концепт-требованиями**). Проверка типов для обобщённого кода выполняется относительно указанных требований, поэтому механизм концептов — механизм обобщённого программирования *на основе явных ограничений*.

  Метод `contains<T>` (Рис. 1.7) требует, чтобы типовый параметр `T` удовлетворял концепту `CEquatible[T]`. Функция `Equal(T, T)` принадлежит этому концепту, поэтому её использование в теле метода допустимо.
- Обобщённый код может быть инстанцирован данным набором конкретных типов, если все типы удовлетворяют соответствующим ограничениям. Поскольку тип `Point` удовлетворяет концепту `CEquatible`, инстанция `contains<Point>` корректна.

Таким образом, механизм концептов вводит в язык три *новых конструкции*:

1. концепт;
2. модель;
3. концепт-требование.

В следующих разделах мы подробно рассмотрим каждую из этих конструкций.

```
concept CEquatible[T]
{
    bool Equal(T a, T b);
    bool NotEqual(T a, T b)
    {
        return !Equal(a, b);
    }
}

concept CSemilattice[T]
    refines CEquatible[T]
{
    T Join(T a, T b);
}

bool contains<T>(T[] values, T x)
    where CEquatible[T]
{
    foreach (T curr in values)
        if Equal(curr, x)
            return true;
    return false;
}
```

```
class Point
{
    public x;
    public y;
    public Point(x, y)
    {
        this.x = x;
        this.y = y;
    }
}

model CEquatible[Point]
{
    bool Equal(T a, T b)
    {
        return (a.x == b.x)
            && (a.y == b.y);
    }
}
```

Рис. 1.7: Программирование с концептами: вводный пример

1.2.2 Концепты

Концепты задают ограничения на один или несколько типовых параметров. Доступны следующие виды ограничений:

- объявление ассоциированного типа;
- сигнатура необобщённой функции (может включать реализацию по умолчанию);
- вложенное концепт-требование (в качестве типовых аргументов могут выступать параметры текущего концепта, его ассоциированные типы или ассоциированные типы других концептов, являющихся частью ограничений);
- ограничение равенства типов;
- ограничение подтипирования;
- ограничение надтипирования.

В качестве типовых аргументов ограничений равенства, под- и надтипирования могут выступать параметры концепта, ассоциированные типы (текущего концепта или других концептов из набора ограничений), необобщённые типы из области видимости, а также любые (конкретные и обобщённые) инстанции обобщённых типов.

Недостатки ограничений-интерфейсов были рассмотрены в Гл. 1.1, поэтому данный вид ограничений не поддерживается в концептах — его заменяют концепт-требования с ограничениями-сигнатурами функций. Наряду с ограничениями-интерфейсами, одним из основных видов ограничений для параметров обобщённого кода в языках платформы .NET являются ограничения *подтипирования* — они сохраняются.

Ограничения подтипирования в совокупности с ограничениями равенства неявным образом добавляют в концепты ограничения *надтипирования*. Например, из ограничений `{A1 <:`[2] `A2}` и `{A1 == MyQ}`, где `A1, A2` — ассоциированные типы концепта, а `MyQ` — некоторый класс, следует, что `{A2 >: MyQ}`. По этой причине ограничения надтипирования разрешены явно.

Концепт может содержать алиасы (синонимы) типов и концепт-требований. Область видимости алиасов ограничивается концептом, в котором они объявлены. Алиасы могут быть использованы для сокращения записи.

Концепт может *уточнять* другой концепт. В качестве типовых аргументов уточняемого концепта могут выступать только типовые аргументы уточняющего.

Концепт может быть объявлен как *явный* (*explicit*). Это значит, что автоматическая генерация моделей данного концепта запрещена. Явный концепт может быть полезен, если необходимо выразить некоторую се-

[2] отношение подтипирования

мантику, не зависящую от синтаксиса. К примеру, нет способа выразить семантическое требование (аксиому [8]) «операция коммутативна». Это можно сделать неформально (в форме документации) и пометить соответствующий концепт как явный. Тогда пользователь концепта будет вынужден явно определить модель (возможно даже пустую) для своего типа, тем самым взяв на себя ответственность за соблюдение семантических требований.

При анализе удобства и применимости предлагаемого дизайна концептов мы ориентировались на задачу анализа потоков данных из области оптимизирующих компиляторов. Наш опыт разработки соответствующей обобщённой библиотеки с помощью средств языка C# показал, что базовых средств недостаточно: наличие в предметной области нескольких зависимых абстракций приводит к трудно читаемому обобщённому коду, перегруженному большим количеством типовых параметров. На Рис. 1.10 приведены заголовки нескольких компонент из этой библиотеки.

```
concept CEquatible[T]
{
    // сигнатура функции
    bool Equal(T x, T y);
    // сигнатура функции
    // с реализацией по умолчанию
    bool NotEqual(T x, T y)
    { return !Equal(x, y); }
}
// уточняющий концепт
concept CComparable[T]
    refines CEquatible[T]
{
    bool Less(T x, T y);
}

concept CSemilattice[T]
    refines CEquatible[T]
{
    T Join(T x, T y);
}
```

```
concept CBoundedSemilattice[T]
    refines CSemilattice[T]
{
    T Top();
}

concept CSet[S]
{
    // ассоциированный тип
    type Element;
    // вложенное концепт требование
    require CEquatible[Element];
    // добавляет x во множество set
    void Add(S set, Element x);
    // объединяет a со множеством b
    void Union(S a, S b);
    ...
    S NewEmpty();
}
```

```
concept CDataVertex[Vertex]
{
    type Data;
    Data GetValue(Vertex v);
    type VertexSet;
    // вложенное концепт требование с алиасом
    require CSet[VertexSet] using cSet;
    // то же: {require CSet[VertexSet]; using cSet = CSet[VertexSet];}

    // ограничение равенства
    require cSet.Element == Vertex;
    void AddInVertex(Vertex vert, Vertex inVert);
    // AddOutVertex, GetInVertices, GetOutVertices
}
concept CDataGraph[Graph]
{
    type Vertex;
    require CDataVertex[Vertex] using cVert;
    // то же: {type Data; require Data == cVert.Data;}
    type Data == cVert.Data;
    // алиас типа
    using VertSet = cVert.VertexSet;
    ...
}

// явный концепт
explicit concept CBasicBlockVertex[Vertex]
    : refines CDataVertex[Vertex] {...}
```

Рис. 1.8: Примеры концептов

```
concept CFoo[T]
{
    type Data1;
}
concept CBar[S, T]
{
    require CFoo[T];
    // то же: {type Data2; require Data2 <: Data1;}
    type Data2 <: Data1;
}
```

Рис. 1.9: Ограничение подтипирования в концепте

```
class SetTransferFunction<InSetAnalysisData, AnalysisData,
                          TFInitDataType>
      : ITransferFunction<AnalysisData, Addr3Command,
                          TFInitDataType>,
        ITransferFunction<AnalysisData, BasicBlock,
                          TFInitDataType>
    where AnalysisData : SetDFADataType<InSetAnalysisData>,
                         IDFADataType<AnalysisData>, new()
{ ... }

abstract class IterativeAlgorithm<AnalysisData, VertexValueType,
                                  TFInitDataType, TF, V, G>
    where AnalysisData : class, IDFADataType<AnalysisData>
    where TF : ITransferFunction<AnalysisData, VertexValueType,
                                 TFInitDataType>, new()
    where V : IVertex<V, VertexValueType>
    where G : IGraph<V, VertexValueType>
{ ... }
```

Рис. 1.10: Компоненты библиотеки анализа потока данных (C#)

Далее в этой работе для демонстрации дизайна концептов часто будут использоваться примеры из упомянутой предметной области, поэтому коротко рассмотрим основные идеи, необходимые для их понимания.

- Основной объект анализа — *граф потока управления*. Это граф с одним источником, вершинами которого являются базовые блоки.
- Анализ выполняется на основании некоторой информации, полу-

ченной для этого графа. Сбор этой информации — *данных анализа* — выполняется с помощью итерационного алгоритма.

- Тип данных анализа является полурешёткой.
- В процессе сбора информации выполняется два основных действия:
  1. К объекту типа данных анализа применяется *передаточная функция* базового блока из вершины графа. Результат применения функции — объект того же типа.
  2. К нескольким объектам типа данных анализа применяется *оператор сбора*. Результатом также является объект данных анализа того же типа.

На Рис. 1.8 представлены некоторые примеры концептов, иллюстрирующие большинство доступных ограничений: сигнатуры функций и сигнатуры функций с реализацией по умолчанию (`CEquatible[T].NotEqual`); ассоциированные типы; ограничения равенства; вложенные концепт-требования; алиасы типов и алиасы концептов.

Концепт `CBoundedSemilattice[T]` (ограниченная полурешётка) уточняет `CSemilattice[T]` (полурешётка), который, в свою очередь, уточняет концепт `CEquatible[T]`.

Концепт `CDataGraph[G]` представляет граф. В процессе сбора данных анализа постоянно выполняется переход между соседними вершинами графа потока управления, поэтому структура графа несколько специфична: информация о дугах хранится в метавершинах (представлены концептом `CDataVertex`); каждая метавершина содержит данные (ассоциированный тип `Data`), ссылки (множество метавершин `VertexSet`) на своих предшественников и ссылки на своих потомков. Заметим, что метавершина хранит ссылки на метавершины того же типа, поэтому параметр концепта `Vertex` встречается в теле концепта в рамках ограничения

{cSet.Element == Vertex}. Описание аналогичной абстракции с помощью интерфейсов может быть не очевидно:

```
1  interface IDataVertex<V, VertexValueType>
2      where V : IDataVertex<V, VertexValueType>   // (*)
3  {
4      ...
5      HashSet<V> InVertices();
6      ...
7  }
8  interface IDataGraph<V, VertexValueType>
9      where V : IDataVertex<V, VertexValueType>
10 { ... }
```

Тип вершины V приходится вводить в интерфейс в качестве типового параметра. Требование (∗) в строке (2) выглядит несколько избыточно, но оно необходимо, например, в следующем случае:

```
class V1 : IDataVertex<V1, int>
{ ... }
class V2 : IDataVertex<V1, int>
{ ... }
```

Ограничение (∗) не позволяет системе типов допустить инстанцию графа IDataGraph<V2, int> , так как тип вершины V2 не реализует интерфейс IDataVertex<V2, int>. Без этого ограничения мы могли бы получить несогласованный граф, вершины типа V2 которого ссылаются на вершины типа V1.

Возвращаясь к коду на Рис. 1.8, нужно упомянуть явный концепт CBasicBlockVertex[Vertex]. Граф потока управления имеет особое свойство: из каждой вершины выходит не более двух дуг. Формально выразить эту особенность имеющимися средствами нельзя, но можно определить для метавершины графа потока управления явный концепт. Ещё

раз подчеркнём, что этот концепт ничего не гарантирует, однако он заставляет пользователя библиотеки в явном виде фиксировать тот факт, что его тип является метавершиной графа потока управления (удовлетворяет концепту).

Рис. 1.9 демонстрирует концепт `CBar[S, T]` с ограничением подтипирования на ассоциированный тип `Data2`.

1.2.3 Обобщённый код

Ограничения на типовые параметры обобщённого кода могут быть выражены в виде концепт-требований (вместо ограничений-интерфейсов), ограничений равенства и под-/надтипирования, которые указываются в секции `where`. Тело обобщённой конструкции также может предваряться определением алиасов для типов и концептов. Алиасы и ассоциированные типы, введённые концепт-требованиями, могут быть использованы в списке формальных параметров заголовка обобщённого метода или в заголовке определения обобщённого класса. Проверка типов обобщённого кода должна выполняться с учётом указанных ограничений. Успешная проверка типов гарантирует, что данная обобщённая конструкция может быть инстанцирована любым набором типов, удовлетворяющим всему набору ограничений.

На Рис. 1.11 приведены части определений классов обобщённого бинарного дерева поиска `BinarySearchTree<T>` и множества `BSTSet<T>`, построенного на базе БДП.

```
class BinarySearchTree<T>
    // концепт требование с алиасом
    where CComparable[T] using cCmp
{
    private BinTreeNode<T> root;
    ...
    private bool AddNested(T x, ref BinTreeNode<T> root)
    {   ...
        // ссылка на концепт по имени
        if (cCmp.Equal(x, root.data))
            return false;
        else if (Less(x, root.data))
            return AddNested(x, ref root.left);
        else
            return AddNested(x, ref root.right);
    }
    ...
}
public class BSTSet<T>
{
    // требование CComparable[T] распространено
    private BinarySearchTree<T> bst;
    ...
    public bool FindMaxLower(T x, out T maxLow)
    {
        ...
            // ссылка на слово через ключевое слово 'reqs'
            if (reqs.Comparable[T].Less(curr.data, x))
        ...
    }
}
```

Рис. 1.11: Пример обобщённых классов

Этот пример демонстрирует две важных особенности дизайна .NET концептов:

1. *Распространение ограничений* [16] позволяет не указывать для класса множества `BSTSet<T>` ограничение на тип `T` удовлетворять концепту `CComparable[T]`: это требование распространяется автоматически на класс множества, так как он содержит поле `bst` типа `BinarySearchTree<T>` (а класс БДП содержит соответствующее требование). Можно и явно добавить требование `CComparable[T]`

в определение класса `BSTSet<T>` — ошибкой это не является.

Распространение ограничений выполняется для полей класса, а также формальных параметров методов класса.

2. Обращение к концептам, доступным в текущем контексте, может быть выполнено *по точке* с помощью ключевого слова `reqs`.

   Даже один концепт может вводить довольно большое число функций, которые могут быть использованы в данном контексте. А если концепт-требований несколько, то запомнить все возможности и подавно будет непросто. При использовании конкретных типов или ограничений-интерфейсов *IntelliSense*-подобные средства среды разработки значительно упрощают процесс работы с контекстом. В то же время ни один из известных нам дизайнов концептов (C++ или G) аналогичных возможностей не предусматривает. Мы предлагаем ввести новое ключевое слово `reqs` (требования), которое обеспечивало бы доступ ко всем концептам, используемым в текущем контексте. Использование данной конструкции демонстрирует метод `FindMaxLower` класса `BSTSet<T>`.

Заметим, что альтернативой `reqs` является обращение к концепту по алиасу (`cCmp.Equal` в методе `AddNested` класса `BinarySearchTree<T>`). Этот способ удобен преимущественно при работе с «основными» концепт-требованиями. Однако, если необходимо использовать возможности «вспомогательного» концепта, наличие которого определяется основными требованиями, лучше иметь к нему непосредственный доступ. Пример такого вспомогательного концепта приведён ниже. Доступность концепта `CDataVertex[cGr.Vertex]` в контексте метода `SomeProcess` вытекает из концепт-требования `CDataGraph[G]`, так как концепт графа содержит соответствующее вложенное концепт-требование для ассоциированного

типа вершины.

```
void SomeProcess<G>(G graph)
    where CDataGraph[G] using cGr
    using cVert = CDataVertex[cGr.Vertex]
{
    // имя типа VertexSet вводится концептом CDataVertex
    var vertSet = reqs.CSet[VertexSet].NewEmpty();
    ...
        var dests = cVert.GetSuccessors(v);
    ...
}
```

Методы `NewEmpty` и `GetSuccessors` не обязательно квалифицировать концептом вершины `cVert`, как и ассоциированный тип `VertexSet` :

```
void SomeProcess<G>(G graph)
    where CDataGraph[G]
{
    VertexSet vertSet = NewEmpty();
    // var vertSet = NewEmpty();     // также корректно
    ...
        var dests = GetSuccessors(v);
    ...
}
```

А вот в следующем случае ассоциированный тип `VertexSet` концепта `CDataVertex` квалифицировать необходимо:

```
void SomeProcess<G, VertexSet>(G graph)
    where CDataGraph[G] using cGr
{
    reqs.CDataVertex[Vertex].VertexSet vertSet = NewEmpty();
    ...
}
```

1.2.4 Модели

На Рис. 1.12 представлена модель концепта `CComparable` для класса `Rational`. `CComparable` уточняет концепт `CEquatible`, поэтому модели двух этих концептов могут быть введены в одном «более широком» определении.

```
class Rational
{
    public int Num {get{...}}
    public int Denom {get{...}}
    public void Add(Rational other);
    ...
}
model CComparable[Rational]
{
    bool Equal(Rational x, Rational y)
    { return (x.Num == y.Num) && (x.Denom == y.Denom); }
    bool Less(Rational x, Rational y)
    {
        if (x.IsPositive)
            ...
    }
}
```

Рис. 1.12: Модель `CComparable[Rational]`

Функции, определённые в модели, могут быть использованы и вне обобщённого кода:

```
Rational MinPositive(Rational[] vals)
using CmpRatio = CComparable[Rational]
{
    ...
    // CComparable[Rational].Less(vals[i], rslt)
    // также допустимо
    if (vals[i].IsPositive && CmpRatio.Less(vals[i], rslt)
        rslt = vals[i];
    ...
}
```

При инстанцировании обобщённого кода проверка типов сводится к проверке наличия всех необходимых моделей. Если для каждого типа определена лишь одна модель, проблем не возникает. Но в некоторых случаях может быть действительно удобно определить несколько моделей концепта для одного и того же типа.

Множественное определение моделей. Среди задач анализа потока данных встречается несколько очень близких: в одном случае итерационный алгоритм работает с данными анализа в форме множества и оператором сбора-объединением (анализ достигающих определений), а в другом — с данными анализа в форме множества и оператором сбора-пересечением (анализ доступных выражений). В остальном алгоритмы идентичны.

Дизайн .NET концептов позволяет определить несколько *именованных* моделей одного концепта для одного и того же типа. Рис. 1.13 демонстрирует определение двух *обобщённых* моделей `MUnionSemilattice<T>` и `MIntersectSemilattice<T>` концепта полурешётки для класса множества. Модель `MUnionBoundedSemilattice<T>` ограниченной полурешётки для оператора сбора-объединения должна квалифицировать конкретную модель, которую она уточняет, поскольку в пространстве имён определено две модели обычной решётки.

В случае с множественным определением моделей запрещается задавать модели нескольких концептов в одном определении (как было сделано для модели `CComparable[Rational]`). Рис. 1.14 иллюстрирует соответствующий пример.

При наличии нескольких моделей для разрешения неоднозначности при инстанцировании обобщённого кода необходимо указать нужную модель. Так в примере на Рис. 1.15 при инстанцировании `foo<D>` необхо-

```
model<T> CEquatible[BSTSet<T>]
{
    bool Equal(BSTSet<T> x, BSTSet<T> y)
    { return x.EqualsTo(y); }
}

model MUnionSemilattice<T> of CSemilattice[BSTSet<T>]
{
    BSTSet<T> Join(BSTSet<T> x, BSTSet<T> y) {...}
}
model MIntersectSemilattice<T> of CSemilattice[BSTSet<T>]
{
    BSTSet<T> Join(BSTSet<T> x, BSTSet<T> y)
    {
        BSTSet<T> result = BSTSet<T>.EmptySet();
        result.Union(x);
        result.Intersect(y);
        return result;
    }
}

model MUnionBoundedSemilattice<T>
    of CBoundedSemilattice[BSTSet<T>]
    // уточняемая модель
    refines MUnionSemilattice<T>
{
    BSTSet<T> Top()
    { return BSTSet<T>.EmptySet(); }
}
```

Рис. 1.13: Пример множественного определения моделей

```
model<T> MIntersectSemilattice of CSemilattice[BSTSet<T>]
{...}
model<T> MUnionBoundedSemilattice
    of CBoundedSemilattice[BSTSet<T>]
{
    // ERROR!
    BSTSet<T> Join(BSTSet<T> x, BSTSet<T> y) {...}
    BSTSet<T> Top() {...}
}
```

Рис. 1.14: Пример некорректного определения модели

```
foo<D>(D x)
    where CSemilattice[D]
{...}

(1) foo[MIntersectSemilattice<int>](x)
or
(2) foo[CSemilattice[D]=MIntersectSemilattice<int>](x)
```

Рис. 1.15: Вызов обобщённой функции с квалификацией модели

димо указать, какая модель полурешётки (объединение или пересечение) должна быть использована. Это можно сделать в полной форме, как в операторе (2), или в сокращённой, как в операторе (1), поскольку определение функции содержит только одно требование концепта полурешётки.

Ещё одну проблему, связанную с использованием нескольких моделей, демонстрирует Рис. 1.16.

```
concept CTransferFunction[TF]
{
    type Data;
    require CSemilattice[Data];
    Data Apply(TF tf, Data d);
    TF Compose(TF a, TF b);
}

class ReachDefsTF {...}
model CTransferFunction[ReachDefsTF]
{
    // Неоднозначность: MUnionSemilattice или MIntersectSemilattice?
    type Data = BSTSet<ReachDef>;
    ...
}
```

Рис. 1.16: Модель передаточной функции: неоднозначность

Модель передаточной функции для класса `ReachDefsTF` задаёт ассоциированный тип `Data` равным множеству `BSTSet<ReachDef>`. Концепт `CTransferFunction` содержит вложенное концепт-требование полурешётки для типа `Data`. Но в пространстве имён определено две модели полурешётки для множества, поэтому неясно, какая из них должна быть

использована. Для разрешения неоднозначности нужную модель необходимо задать явно:

```
model CTransferFunction[ReachDefsTF]
{
    type Data = BSTSet<ReachDef>;
    using MUnionSemilattice<ReachDef>;
    ...
}
```

Замечание. Определение моделей разного уровня специализии данным дизайном не предусмотрено (например, нельзя определить модель концепта полурешётки для типа `BSTSet<ReachDef>`, поскольку уже определена обобщённая модель для типа `BSTSet<T>`, где `T` — типовый параметр), данный аспект требует дальнейшего исследования.

Неявная генерация моделей. В некоторых ситуациях компилятор может *автоматически* неявно сгенерировать модель концепта для типа. В примере ниже класс `A` содержит методы, непосредственно совместимые с сигнатурами функций из определений концептов `Equatible` и `Comparable` (метод `R Name(args)` типа `S` *совместим* с сигнатурой функции `R Name(S arg1, args)`). Поэтому определять модель `CComparable[A]` явно не нужно.

```
class A
{
    ...
    public bool Equal(A other);
    public bool NotEqual(A other);
    public bool Less(A other);
}
```

Напомним, что для *явных* концептов неявная генерация моделей запрещена.

1.3 Концепт-требования как замена ограничений-интерфейсов

Дизайн концептов, представленный в Гл. 1.2, позволяет решить проблему обеспечения адаптации типов без дублирования кода, рассмотренную в Гл. 1.1. Адаптация типов получается «бесплатно» за счёт структуры концептов: для типа нужно лишь определить необходимую модель. Например, вместо пары методов

```
(1) Sort<T>(T[])
        where T : IComparable<T>;
(2) Sort<T>(T[], IComparer<T>);
```

достаточно определить один:

```
Sort<T>(T[])
    where CComparable[T];
```

Как и вместо четырёх методов

```
(1) p<T>(...) where T : ISelf1<T>, ISelf2<T>;
(2) p<T>(..., IExt1<T>) where T : ISelf2<T>;
(3) p<T>(..., IExt2<T>) where T : ISelf1<T>;
(4) p<T>(..., IExt1<T>, IExt2<T>);
```

хватит одного:

```
p<T>(...) where Concept1[T], Concept2[T];
```

Заметим, что обобщённые методы с ограничениями-интерфейсами, описывающими встроенные операции типа, «покрываются» возможностью неявной генерации моделей: если тип определяет все сигнатуры концепта, задавать модель в явном виде не нужно.

При сравнении интерфейсов и концептов необходимо отметить следующую особенность: кроме проблем с адаптацией типов, использование интерфейса `IComparable<T>` в качестве ограничения на типовые параметры влечёт некоторую сымсловую неоднозначность. Соответствущее ограничение

```
p<T>(...)
    where T : IComparable<T>;
```

выглядит немного избыточно и используется для того, чтобы выразить семантическое требование «для элементов типа `T` определена операция сравнения». При этом сам интерфейс `IComparable<T>` (Рис. 1.17) имеет другую семантику: он определяет возможность элементов некоторого типа (реализующего данный интерфейс) быть сравнимыми с элементами типа `T`.

```
interface IComparable<T>
{
    int CompareTo(T other);
}
```

Рис. 1.17: Интерфейс `IComparable<T>`

С точки зрения семантики, следовало бы разбить этот интерфейс на два(Рис. 1.18): `IComparableTo<T>` и `IComparable<T>`.

```
interface IComparableTo<T> {...}

int find<S, T>(S[] vals, T x)
    where S : IComparableTo<T>
{...}

interface IComparable<T> where T : IComparable<T> {...}

void sort<T>(T[] vals)
    where T : IComparable<T>
{...}
```

Рис. 1.18: Разбиение интерфейса `IComparable<T>`

Тем не менее, такое разбиение не отменяет необходимости «избыточной» записи ограничения

```
where T : IComparable<T>;
```

в функции `sort`, которая исчезает, если использовать концепты (Рис. 1.19).

```
concept CAreComparable[S, T] {...}

int find<S, T>(S[] vals, T x)
    where CAreComparable[S, T]
{...}

concept CComparable[T] {...}

void sort<T>(T[] vals)
    where CComparable[T]
{...}
```

Рис. 1.19: Использование концептов `C*Comparable`

Глава 2

Семантический анализ и трансляция

В Гл. 1.2 был представлен *дизайн* конструкций обобщённого программирования на основе концептов. Внедрение механизма концептов в язык программирования (то есть *реализация*) требует модификации соответствующих компиляторов. Кроме изменений в синтаксическом анализаторе (поскольку меняется грамматика языка), на уровне семантического анализа необходимо обеспечить поддержку новых конструкций (концепты, модели, концепт-требования), а также внести изменения в процесс проверки обобщённого кода и его инстанций. В случае успешного завершения семантического анализа, необходимо выполнить **трансляцию** кода, связанного с концептами, в базовый язык.

Семантический анализ обобщённого кода и его инстанций не представляет принципиальных сложностей: проверка типов в обобщённом коде выполняется обычным образом, но контекст расширяется информацией из заданных концепт-требований; проверка типов для инстанций сводится к проверке наличия соответствующих моделей.

Более трудоёмким является процесс анализа концептов, моделей и концепт-требований, так как в этом случае необходимо выполнить спе-

```
class EncryptedData<T> {...}
class PlainData {...}

concept C1[S]
{
    type Key;
    type Data <: EncryptedData<Key>;
    Data GetData(S x);
}

concept C2[S, T]
{
    require C1[S];
    type Data <: C1[S].Data;
    ...
}
concept C2Bad[S, T] refines C2[S, T]
{
    type Data == PlainData; // Ошибка!
}
```

Рис. 2.1: Некорректный концепт `C2Bad`: противоречивое множество ограничений

циальную процедуру **проверки непротиворечивости ограничений**. Рассмотрим пример на Рис. 2.1. Ассоциированный тип `Data` концепта `C1` должен быть подтипом класса `EncryptedData<Key>`; концепт `C2` содержит вложенное концепт-требование `C1[S]`, его ассоциированный тип `Data` должен быть подтипом `C1[S].Data`, то есть должно выполняться ограничение:

$$\texttt{C2[S, T].Data} <: \texttt{C1[S].Data} <: \texttt{EncryptedData<Key>} \tag{2.1}$$

Концепт `C2Bad`, уточняющий концепт `C2`, требует от типа `C2[S, T].Data` быть равным классу `PlainData`, что противоречит ограничению (2.1), так как `PlainData` не является подтипом `EncryptedData<Key>`.

Подобные ошибки могут возникать не только в концептах, но и в моделях (определение модели может противоречить концепту), а также концепт-требованиях к обобщённому коду. Проверка непротиворечиво-

сти множества ограничений выполняется с помощью **алгоритма унификации** — он представлен в Гл. 2.1.

Гл. 2.2 посвящена **трансляции** конструкций, зависящих от концептов, в язык C#.

2.1 Унификация ограничений

2.1.1 Введение

Унификация набора ограничений является существенной частью этапа семантического анализа: она используется для проверки непротиворечивости ограничений, задаваемых концептами, моделями и концепт-требованиями.

Алгоритм унификации ограничений .NET концептов основан на алгоритме унификации Хиндли-Милнера [20]. Базовый алгоритм используется, например, для реконструкции типов термов простого типизированного лямбда-исчисления. Он применяется ко множеству ограничений *равенства* вида

$$\mathtt{S} = \mathtt{T}$$

где $\mathtt{S}$, $\mathtt{T}$ — базовые типы, функциональные типы и типовые переменные. Это ограничение выражает требование соответствия $\mathtt{S}$ и $\mathtt{T}$ одному и тому же типу. Результатом унификации множества ограничений равенства $C = \{\mathtt{S}_i = \mathtt{T}_i\}$ является подстановка типов σ

$$\sigma = \{\mathtt{V}_i \to \mathtt{A}_i \mid \mathtt{V}_i \text{ — различные типовые переменные}\}$$

унифицирующая множество C (σ **унифицирует** ограничение $\{\mathtt{S} = \mathtt{T}\}$, если $\sigma\mathtt{S} = \sigma\mathtt{T}$; σ унифицирует множество ограничений C, если унифицирует все ограничения из этого множества).

Дизайн .NET концептов включает ограничения равенства и добавляет новый вид ограничений — ограничения *подтипирования*. Ограничение подтипирования

$$\texttt{S} <: \texttt{T}$$

требует, чтобы тип `S` являлся подтипом `T` или совпадал с ним.

Сочетание ограничений равенства и подтипирования для типизированного лямбда-исчисления рассматривается в [25]: ограничения подтипирования частично сводятся к ограничениям равенства, благодаря чему для унификации удаётся использовать базовый алгоритм Хиндли-Милнера. В случае .NET концептов ограничения подтипирования имеют более сложную структуру (это, в частности, связано с тем, что ограничения задаются в концепте произвольно, а не формируются по терму, как в [25]), поэтому базовый алгоритм унификации уже не подходит — необходима модификация. Она затрагивает не только сам алгоритм, но и структуру типовых переменных, а также подстановку типов.

Мы будем использовать следующую нотацию: $typevar$ — множество типовых переменных; $stype$ — множество «обычных» классов (например, `Rational`); $genstype$ — множество закрытых сконструированных обобщённых классов (например, `BSTSet<Rational>`); $genvartype$ — множество открытых сконструированных обобщённых классов (например, `BSTSet<T>`, где `T` — типовая переменная); объединённое множество типов .NET $nettype = stype \cup genstype \cup genvartype$; а также множество всех типов $typeval = nettype \cup \{Top, Bottom\}$, где Top представляет самый «верхний» тип (являющийся надтипом любого типа), а $Bottom$ — самый «нижний» (подтип любого типа).

В отличие от базового алгоритма унификации, типовые переменные `V` снабжаются двумя атрибутами — границами в цепочке подтипиро-

вания: наибольшим подтипом V.maxLT и наименьшим надтипом V.minUT, причём V.maxLT, V.minUT : *typeval* (то есть в качестве границ не могут выступать типовые переменные, хотя они могут встречаться «внутри» типов-границ) и V.maxLT <: V.minUT. Эти атрибуты задают ограничение на множество возможных значений A типовой переменной:

$$\mathtt{V.maxLT} <: \mathtt{A} <: \mathtt{V.minUT}$$

В простейшем случае, если множество значений переменной не ограничено, V.maxLT полагается равным $Bottom$, а V.minUT $= Top$.

Подстановка σ состоит из элементов подстановки $[\mathtt{S} \to \mathtt{T}]$, каждый из которых удовлетворяет следующему правилу:

$$\begin{array}{ll} \mathtt{S.maxLT} <:= \mathtt{T.maxLT} <: \mathtt{T.minUT} <:= \mathtt{S.minUT} & \text{если } \mathtt{T} : \mathit{typevar} \\ \mathtt{S.maxLT} <:= \mathtt{T} <:= \mathtt{S.minUT} & \text{если } \mathtt{T} : \mathit{typeval} \end{array}$$

Будем говорить, что подстановка σ *унифицирует* ограничение подтипирования $\{\mathtt{S} <: \mathtt{T}\}$, если ограничение $\{\sigma\mathtt{S} <: \sigma\mathtt{T}\}$ выполнимо. При этом ограничение $\{\mathtt{S} <: \mathtt{T}\}$ называется *выполнимым*, если выполняется одно из условий:

(1) $\mathtt{S} : \mathit{typeval}, \mathtt{T} : \mathit{typeval} \Rightarrow$
$\mathtt{S} <:= \mathtt{T}$

(2) $\mathtt{S} : \mathit{typeval}, \mathtt{T} : \mathit{typevar} \Rightarrow$
$\mathtt{S} <:= \mathtt{T.maxLT}$

(3) $\mathtt{S} : \mathit{typevar}, \mathtt{T} : \mathit{typeval} \Rightarrow$
$\mathtt{S.minUT} <:= \mathtt{T}$

(4) $\mathtt{S} : \mathit{typevar}, \mathtt{T} : \mathit{typevar} \Rightarrow$
$(\forall \mathtt{S}',\ \mathtt{S.maxLT} <:= \mathtt{S}' <:= \mathtt{S.minUT}$
$\exists \mathtt{T}',\ \mathtt{T.maxLT} <:= \mathtt{T}' <:= \mathtt{T.minUT} \mid \mathtt{S}' <:= \mathtt{T}')$
и
$(\forall \mathtt{T}',\ \mathtt{T.maxLT} <:= \mathtt{T}' <:= \mathtt{T.minUT}$
$\exists \mathtt{S}',\ \mathtt{S.maxLT} <:= \mathtt{S}' <:= \mathtt{S.minUT} \mid \mathtt{S}' <:= \mathtt{T}')$

2.1.2 Алгоритм унификации

Алгоритм унификации $unify$ (частично представлен на Рис. 2.2) применяется к набору C ограничений равенства и подтипирования. Результатом работы алгоритма является подстановка типов σ, унифицирующая C, а также набор ограничений подтипирования $C' = \{\mathtt{S}_i <:= \mathtt{T}_i\}$, где $\mathtt{S}_i, \mathtt{T}_i$ — типовые переменные. Такие ограничения мы будем называть ограничениями подтипирования *второго рода*, все остальные ограничения подтипирования — *первого рода*.

Вспомогательная функция *suboreq* активно используется в $unify$, она проверяет, может ли тип `A` быть подтипом `B`. Если это возможно, но необходимы дополнительные ограничения, требуемые ограничения добавляются в набор. Рассмотрим пример: пусть в программе определены следующие классы: `class Opq, class Rst, class A<S, T>, class B<T> : A<Opq, T>, class C : B<Rst>`. Класс `C` может быть подтипом `A<X, Y>` (`X, Y` — типовые переменные), если $\mathtt{X} = \mathtt{Opq}$, $\mathtt{Y} = \mathtt{Rst}$, поэтому функция *suboreq* вернёт `True` и добавит ограничения $\{\mathtt{X} = \mathtt{Opq}, \mathtt{Y} = \mathtt{Rst}\}$ в соответствующий набор ограничений.

Замечание. Дополнительные ограничения могут быть построены только в том случае, когда хотя бы один из типов `A` и `B` является открытым сконструированным обобщённым классом (*genvartype*), то есть его типовые аргументы содержат типовые переменные, которые можно приравнять для обеспечения условия подтипирования. Именно такой тип `A<X, Y>` использован в приведённом выше примере. Отношение `C <: A<Opq, Rst>` выполняется без дополнительных ограничений, а `C <: A<Opq, Opq>` не выполнимо (*suboreq* вернёт `False`). Более точно, дополнительные огра-

ничения равенства *могут быть* добавлены в набор только в четырёх случаях:

1. $suboreq(\texttt{A} : genvartype, \texttt{B} : genvartype)$
2. $suboreq(\texttt{A} : genvartype, \texttt{B} : genstype)$
3. $suboreq(\texttt{A} : stype, \texttt{B} : genvartype)$
4. $suboreq(\texttt{A} : genstype, \texttt{B} : genvartype)$

Мы не приводим псевдокод алгоритма унификации полностью, поскольку он достаточно объёмист (в отличие от базового алгоритма Хиндли-Милнера). Следующая глава посвящена проекту реализации этого алгоритма, а здесь рассмотрим некоторые его особенности.

Набор ограничений C задаётся очередью с приоритетом. Запись $C = C' + c_h$ в начале алгоритма $unify$ означает, что мы рассматриваем ограничение c_h с наибольшим приоритетом из головы очереди, а C' представляет очередь оставшихся ограничений. Приоритет распределяется следующим образом: наибольший приоритет имеют ограничения равенства, затем «не просмотренные» ограничения подтипирования (то есть ограничения $c_h : \{S <: T\} \mid c_h.\texttt{viewCount} = 0$) и, наконец, ограничения подтипирования, которые уже были просмотрены ($c_h \mid c_h.\texttt{viewCount} = 1$). В начале алгоритма все ограничения имеют нулевое число просмотров. Запись $\sigma \circ [\texttt{S} \to \texttt{T}]$ означает композицию подстановок, $(\sigma \circ [\texttt{S} \to \texttt{T}])\texttt{X} = \sigma([\texttt{S} \to \texttt{T}]\texttt{X})$; запись $[\texttt{S} \to \texttt{T}]C'$ — применение подстановки $[\texttt{S} \to \texttt{T}]$ к очереди ограничений C'. В результате применения подстановки к очереди, атрибут `viewCount` числа просмотров каждого ограничения в этой очереди сбрасывается в ноль. Таким образом, алгоритм унификации работает по следующей схеме:

- Если набор ограничений пуст, алгоритм завершается.

```
unify(∅) = (∅, [])
unify(C) =
  C = C′ + c_h
  if (c_h.viewCount = 1)
    (C, [])
  else if (c_h : {S = T})
    ...
    // S : typevar, T : typeval
    if (T : genvartype && T.genericArgs.contains(S))
      fail
    else if suboreq(S.maxLT, T, inout C′)
      if suboreq(T, S.minUT, inout C′)
        var (C_r, σ) = unify([S → T]C′)
        (C_r, σ ∘ [S → T])
      else
        fail(T не может быть подтипом S.minUT)
    ...
  else // (c_h : {S <:= T})
    ...
    // S : typeval, T : typevar
    if eq(S, T.maxLT)
      unify(C′)
    ...
    // T.maxLT <: S
      if suboreq(S, T.minUT, inout C′)
        var T′ = new typevar(maxLT = S, minUT = T.minUT)
        var (C_r, σ) = unify([T → T′]C′)
        (C_r, σ ∘ [T → T′])
  ...

suboreq(A : stype, B : genvartype, inout C) =
  false
  var BaseA = A.baseclass
  while (BaseA ! = null)
    if (BaseA.def == B.def)
      makeeqconstrs(BaseA.genericArgs,
        B.genericArgs, inout C)
      true
      break
    BaseA = A.baseclass
```

Рис. 2.2: Unification algorithm

- После обработки ограничения из головы очереди алгоритм рекурсивно применяется к оставшемуся[1] набору ограничений.
- Прежде всего обрабатываются ограничения равенства. В результате обработки одного ограничения может возникнуть новый элемент подстановки, набор дополнительных ограничений равенства, то и другое или ничего.
- Если очередь не пуста, но не содержит ограничений равенства, обрабатывается очередное ограничение подтипирования.
  - Если есть хоть одно не просмотренное ограничение подтипирования (его атрибут `viewCount` равен 0), обрабатывается именно оно. В результате обработки возможна одна из следующих ситуаций: получен новый элемент подстановки; получен новый элемент подстановки и набор ограничений равенства; получен новый элемент подстановки и одно ограничение подтипирования второго рода; получен новый элемент подстановки, одно ограничение подтипирования второго рода и набор ограничений равенства; получен набор ограничений равенства, ограничение подтипирования второго рода сохраняется, но его атрибут числа просмотров устанавливается равным единице; ограничение подтипирования второго рода сохраняется, но его атрибут числа просмотров устанавливается равным единице.
  - Если все ограничения подтипирования уже были просмотрены, атрибут числа просмотров ограничения c_h из головы очереди $c_h.\texttt{viewCount} = 1$, и алгоритм унификации завершается. C представляет результирующий набор ограничений подтипирования второго рода.

[1] это не обязательно C', набор ограничений может быть изменён в процессе обработки текущего ограничения из головы очереди

Замечание. Интуитивно этот процесс можно описать так: в процессе работы алгоритма некоторые ограничения «уничтожаются», другие анализируются и возвращаются в набор, при этом формируются новые ограничения равенства, обеспечивающие выполнимость уже рассмотренных ограничений. Если очередь содержит только просмотренные ограничения подтипирования второго рода, это означает, что набор ограничений стабилизировался: получена подстановка, унифицирующая исходный набор ограничений, а оставшиеся ограничения представляют собой набор выполнимых ограничений подтипирования второго рода.

2.1.3 Завершимость алгоритма

Теорема. Алгоритм унификации $unify$ всегда завершается.

Доказательство. Введём несколько вспомогательных обозначений.

1. Через $|\mathtt{T}|$ обозначим *размер* типа `T`. Он определяется по следующим правилам:

$$\begin{array}{lll} (1) & |\mathtt{T}| = 1, & \mathtt{T} : stype \cup \{Top, Bottom\} \\ (2) & |\mathtt{T}| = 1 + \sum_{i=1}^{n} |\mathtt{T.genericArgs}_i|, & \mathtt{T} : genstype \cup genvartype \\ (3) & |\mathtt{T}| = 1 + |\mathtt{T.maxLT}| + |\mathtt{T.maxUT}|, & \mathtt{T} : typevar \end{array}$$

2. $|c_h|$ — размер ограничения (суммарный размер его типов), $|C|$ — суммарный размер всех ограничений в наборе C.
3. *Длину цепочки подтипирования* типа `A` из множества $nettype$ будем обозначать через $subtypelen(\mathtt{A})$, она вычисляется следующим

образом:

$$subtypelen(\mathtt{A}) = k + 1 \mid \mathtt{A} = \mathtt{A}_1,\ \mathtt{Object} = \mathtt{A}_k,\ \mathtt{A}_1 : \mathtt{A}_1 : \ldots : \mathtt{A}_k$$

где «:» — отношение непосредственного подтипирования

Замечание. Тип `B` является непосредственным подтипом `A`, если `A` — его базовый класс. Значение 1 прибавляется к реальной длине цепочки k, чтобы смоделировать отношение подтипирования с Top.

4. Пусть C — набор ограничений. *Типовым диаметром* множества ограничений $d(C)$ будем называть максимальную длину цепочки подтипирования среди всех типов, содержащихся в C.
5. Через $weight(\mathtt{T})$ будем обозначать *вес* типа $\mathtt{T} \in C$, он определяется по следующим правилам:

   (1) $weight(\mathtt{T}) = subtypelen(\mathtt{T}),$ если $\mathtt{T} : nettype$
   (2) $weight(Top) = 1$
   (3) $weight(Bottom) = d(C) + 1$
   (4) $weight(\mathtt{T}) = weight(\mathtt{T.maxLT}) - weight(\mathtt{T.minUT}) + 1,$ если $\mathtt{T} : typevar$

6. Через $TV(C)$ обозначим множество типовых переменных из набора ограничений C.
7. **Степенью** $pow(C)$ множества ограничений C будем называть тройку (sc, m, n):

   - sc — число ограничений подтипирования в наборе C;
   - $m = \sum_{\mathtt{X} \in TV(C)} weight(\mathtt{X})$ — суммарный вес множества типовых переменных в наборе C;
   - $n = |C|$ — суммарный размер типов в наборе ограничений C.

8. Через sc_0 будем обозначать количество не просмотренных ограни-

чений подтипирования второго рода $c_h \in C$, $c_h.\texttt{viewCount} = 0$. Заметим, что $sc_0 \leq sc$.

Алгоритм $unify$ рекурсивный. На каждом шаге из очереди ограничений C выбирается ограничение c_h с наибольшим приоритетом (запись $C = C' + c_h$); c_h обрабатывается, формируется очередь ограничений $C^r \supseteq C'$, и происходит рекурсивный вызов $unify(C^r)$. Перебирая все возможные варианты ограничения c_h, можно показать, что алгоритм либо завершается неудачей, либо степень и число не просмотренных ограничений подтипирования второго рода для множеств C^r и C связаны одним из следующих отношений (сравнение «<» на степенях наборов $(sc^r, m^r, n^r) < (sc, m, n)$ выражает лексикографическое сравнение; тройкой (s, m, n) обозначена степень $pow(C)$; символ «_» в записи степени означает, что конкретное число не имеет значения для результата сравнения):

1. $c_h : \{\texttt{S} = \texttt{T}\}$
   a) $\texttt{S} : typeval, \texttt{T} : typeval \Longrightarrow$
   $pow(C^r) = (sc, m, n^r) < (sc, m, n) \mid n^r < n$
   b) $\texttt{S} : typevar$ or $\texttt{T} : typevar \Longrightarrow$
   $pow(C^r) = (sc, m^r, \_) < (sc, m, n) \mid m^r < m$
2. $c_h : \{\texttt{S} <: \texttt{T}\}$
   a) $\texttt{S} : typeval, \texttt{T} : typeval \Longrightarrow$
   $pow(C^r) = (sc^r, \_, \_) < (sc, m, n) \mid sc^r < sc$
   b) $\texttt{S} : typevar$ xor $\texttt{T} : typevar \Longrightarrow$
   $pow(C^r) = (sc^r, \_, \_) < (sc, m, n) \mid sc^r < sc$
   c) $\texttt{S} : typevar, \texttt{T} : typevar \Longrightarrow$
      i. $pow(C^r) = (sc, m^r, \_) < (sc, m, n) \mid m^r < m$
      ii. $pow(C^r) = (sc, m, n) = pow(C)$
      $sc_0^r < sc_0$
      iii. $pow(C^r) = (sc, m, n^r) > (sc, m, n) \mid n^r > n$

Во всех случаях, кроме [2(c)iii], в рекурсивный вызов передаётся набор C^r, степень которого меньше степени C. Рассмотрим более подробно [2(c)iii].

Этот вариант соответствует ситуации, когда ограничение второго рода $c_h : \{\mathtt{S} <: \mathtt{T}\} \mid \mathtt{S} : typevar, \mathtt{T} : typevar$ выполнимо при условии дополнительных ограничений равенства. Это проверяется попарным сравнением границ типовых переменных (`S.maxLT`, `S.minUT`, `T.maxLT`, `T.minUT`) с помощью алгоритма *suboreq*.

- Если дополнительные ограничения не нужны, это случай [2(c)ii]: ограничение c_h возвращается в набор ограничений C (то есть $C^r = C' + c_h = C$), но его атрибут числа просмотров $c_h.\mathtt{viewCount} = 1$. Поэтому количество не просмотренных ограничений второго рода уменьшается ($sc_0^r < sc_0$).
- В случае необходимости вспомогательных ограничений работает вариант [2(c)iii]. Дополнительные ограничения равенства появляются в результате выполнения алгоритма *suboreq*. Это может случиться только в том случае, если среди типов `S.maxLT`, `S.minUT`, `T.maxLT`, `T.minUT` есть открытые сконструированные обобщённые классы (*genvartype*): соответствующие типовые переменные-аргументы (из множества *typevar*) приравниваются к значениям из $typeval \cup typevar$. Таким образом, после завершения *suboreq* множество C^r содержит ограничения равенства $\{\mathtt{S} = \mathtt{T} \mid \mathtt{S} : typevar \text{ or } \mathtt{T} : typevar\}$. Поскольку ограничения равенства имеют наивысший приоритет, на следующей итерации алгоритма будет выбрано именно такое ограничение равенства c'_h из C^r, а ему соответствует вариант обработки [1b]. Из этого следует, что в результате выполнения пары ите-

раций [2(c)iii] и [1b] степень множества ограничений уменьшается:

$$pow(C^{r\prime}) = (sc, m^{r\prime}, n^r) < (sc, m, n) \mid m^{r\prime} < m, n^r > n$$

Итак, для всех видов ограничения c_h, кроме [2(c)iii], степень набора ограничений уменьшается в результате выполнения соответствующей итерации. Случай [2(c)iii] *необходимо* влечёт за собой выполнение [1b] на следующей итерации, в результате чего степень набора также уменьшается. Пару итераций [2(c)iii] и [1b] назовём *полной итерацией*. Все отдельные итерации, кроме [2(c)iii], так же будем считать полными. Тогда алгоритм унификации представляется последовательностью полных итераций. В результате выполнения каждой полной итерации алгоритма выполняется одно из двух условий:

1. Уменьшается степень (sc, m, n) множества ограничений (при этом величина sc_0 ограничена: $sc_0 \leq sc$).
2. Степень множества ограничений не меняется, но уменьшается величина sc_0.

Исходное множество ограничений конечно, а значит конечны его степень и величина sc_0. Процесс уменьшения степени и sc_0 не может продолжаться бесконечно, следовательно, наступит одна из двух ситуаций:

1. $(sc, m, n) = (0, 0, 0) \iff$ очередь ограничений пуста ($C = \varnothing$). Алгоритм $unify(\varnothing)$ завершается.
2. $(sc, m, n) = (a, b, c) \mid a > 0,\ b > 0,\ c > 0$, а $sc_0 = 0 \iff$ очередь ограничений не содержит не просмотренных ограничений подтипирования второго рода. Поэтому для очередного c_h атрибут c_h.`viewCount` $=$ `1`, и алгоритм $unify$ завершается.

   Таким образом, алгоритм всегда завершается.

Замечание. Уменьшение sc происходит при обработке ограничения подтипирования. Если очередь не содержит ограничений подтипирования, $sc = 0$ и дальнейшее уменьшение невозможно (то есть sc не может стать отрицательной). То же касается и величин m и n — они не могут стать отрицательными. Поскольку n описывает размер типов в наборе ограничений, а типовые переменные по определению имеют ненулевой размер, ситуация $n = 0, m > 0$ невозможна — $n = 0$ всегда влечёт $m = 0$ (обратное неверно).

2.1.4 Вычислительная сложность

Унификация используется не только для реконструкции типов, но и, например, в средствах автоматического и полуавтоматического доказательства теорем. Самый простой вид унификации — унификация термов *первого порядка*, именно к этой категории относится алгоритм унификации ограничений Хиндли-Милнера, взятый нами за основу. В контексте рассматриваемой задачи работы с *типами* «первый порядок» означает, что типовыми переменными могут быть только переменные типов, но не конструкторов типов. Например, типовая переменная `X` может быть равна `List<String>` или `List<Y>` (`Y` — типовая переменная), но не может быть равна `List<>` (`List<>` — пример конструктор типа).

Один из первых алгоритмов унификации первого порядка — рекурсивный алгоритм Робинсона [21] — в худшем случае имеет экспоненциальную сложность относительно размера входных данных (в случае унификации ограничений Хиндли-Милнера это суммарный размер типов множества ограничений). Для него доказаны завершимость и корректность, а сам алгоритм достаточно нагляден. Позже были найдены и другие алгоритмы унификации термов первого порядка [23, 24], име-

ющие полиномиальную и даже линейную сложность (например, [22]). Уменьшение сложности достигается, в частности, за счёт выбора «подходящих» структур данных.

Базовый алгоритм унификации ограничений Хиндли-Милнера основан на алгоритме Робинсона и в худшем случае имеет экспоненциальную сложность. Рекурсивный алгоритм унификации .NET концептов, рассмотренный выше, по структуре аналогичен алгоритму Хиндли-Милнера: на каждом шаге рассматривается ограничение, и в результате анализа либо обнаруживается противоречие, либо модифицируется набор ограничений. Модификация происходит применением подстановки или добавлением новых ограничений равенства. Дополнительные ограничения равенства могут добавляться в набор в двух ситуациях:

1. При анализе равенства $\{\mathtt{S} = \mathtt{T}\}$, если оба аргумента `S` и `T` являются сконструированными обобщёнными классами, и хотя бы один из них — открытый.
2. При анализе отношения подтипирования $\{\mathtt{S} <: \mathtt{T}\}$, если оба аргумента представляют .NET классы, причём `T` — открытый сконструированный обобщённый класс ($\mathtt{T} : genvartype$). В этом случае рассматривается цепочка надтипов `S`, и если она содержит тип $\mathtt{S}' : \{genstype \cup genvartype\} \mid \mathtt{S}'.genericDef = \mathtt{T}.genericDef$, типовые аргументы $\mathtt{S}'$ и `T` приравниваются, то есть вместо исходного $\{\mathtt{S} <: \mathtt{T}\}$ рассматривается ограничение $\{\mathtt{S}' = \mathtt{T}\}$.

В базовом алгоритме унификации возможен только случай 1, но 2 сводится к 1: ограничение подтипирования, фактически, заменяется ограничением равенства, где второй аргумент обязательно остаётся тем же, а первый либо остаётся, либо заменяется на некоторый тип из цепочки надтипирования.

Обозначим через $||\mathtt{T}||$ максимальный размер типа в цепочке надтипирования $\mathtt{T}$. Величину $||\mathtt{T}||$ ($|\mathtt{T}| \leq ||\mathtt{T}||$) будем называть *родовым размером* типа:

$$||\mathtt{T}|| = \begin{cases} max\{|\mathtt{T}_i|,\ \mathtt{T} = \mathtt{T}_1 : \mathtt{T}_2 : \ldots : \mathtt{T}_m\}, & \mathtt{T} : typeval \\ 1 + ||\mathtt{T.maxLT}|| + ||\mathtt{T.maxUT}||, & \mathtt{T} : typevar \end{cases}$$

Обозначим через $||C||$ суммарный *родовой размер* типов в наборе ограничений C. Тогда с точки зрения вычислительной сложности унификация набора C алгоритмом унификации .NET концептов аналогична унификации набора C', $|C'| = ||C||$ базовым алгоритмом.

2.2 Трансляция

В этом разделе мы рассмотрим трансляцию (перевод) кода на C# с .NET концептами в базовый C#. Прежде всего будет представлена общая идея перевода, затем более подробно остановимся на трансляции отдельных конструкций, вовлечённых в механизм концептов.

2.2.1 Обзор

Конструкции языка с концептами	**Конструкции базового языка**
Концепт	Абстрактный класс
Параметр концепта	Типовый параметр
Ассоциированный тип	Типовый параметр
Уточнение концепта	Подтипирование
Вложенное концепт-требование	Типовый параметр
Концепт-требование в обобщённом коде	Типовый параметр
Модель	Класс

Рис. 2.3: Трансляция расширенного кода

```
concept CBad[S, T]
{
    require S == T;
}
```

Рис. 2.4: Некорректный концепт `CBad`: параметры соответствуют одному типу

Таблица на Рис. 2.3 иллюстрирует соответствие между основными конструкциями механизма .NET концептов и их представлением в базовом языке.

Во Введении мы обсуждали необходимость сохранения информации об исходных конструкциях при переводе с расширенного языка. Сделать это без дополнительной метаинформации невозможно. Так, например, в целевом коде трансляции нужно уметь различать типовые параметры, соответствующие параметрам концепта, его ассоциированным типам и вложенным концепт-требованиям. А обычные абстрактные классы, определённые пользователем, нужно отличать от абстрактных классов концептов. Чтобы разрешить подобные неоднозначности, будем использовать атрибуты.

2.2.2 Трансляция концептов

Перед трансляцией концепт должен быть подвергнут семантическому анализу. Один из аспектов анализа — непротиворечивость набора ограничений — мы уже обсуждали выше. Для проверки непротиворечивости используется алгоритм унификации (Гл. 2.1). Набор ограничений формируется по непосредственному определению концепта, а также вложенным концепт-требованиям.

В случае успешного завершения алгоритма унификации к телу и параметрам концепта применяется полученная подстановка типов. Ещё одно требование корректности концепта заключается в том, что его ти-

повые параметры должны соответствовать различным типам. Рис. 2.4 демонстрирует пример концепта с такой ошибкой.

После окончания семантического анализа определение концепта включает следующую информацию:

- Список типовых параметров, представленных типовыми переменными.
- Уточняемый концепт (если есть).
- Список ассоциированных типов (также представляются типовыми переменными).
- Набор ограничений подтипирования второго рода, полученный в результате унификации.
- Список сигнатур функций и функций с реализацией по умолчанию.
- Список вложенных концепт-требований.

Трансляция. Элементы концептов транслируются следующим образом:

- Концепт отображается в абстрактный класс с атрибутом `[Concept]` или `[ExplicitConcept]` (если концепт явный). Уточняющий концепт переводится в абстрактный класс-наследник соответствующего класса уточняемого концепта.
- Параметры концепта переводятся в типовые параметры абстрактного класса, они сопровождаются атрибутом `[IsConceptParameter]`. Ассоциированные типы отображаются в типовые параметры того же класса с атрибутом `[IsAssociatedType]`.
- Вложенные концепт-требования отображаются в типовые параметры с атрибутом `[IsNestedConceptRequirement]`, при этом в секцию `where` добавляются требования подтипирования (типовый па-

раметр, соответствующий концепт-требованию, должен быть наследником соответствующего абстрактного класса).

Ассоциированные типы и концепт-требования непосредственно вложенных концепт-требований текущего концепта обрабатываются рекурсивно, но уже без добавления атрибутов.

- Для каждой типовой переменной концепта: если верхняя граница отлична от Top, соответствующее требование подтипирования добавляется в секцию `where`; если нижняя граница отлична от $Bottom$, абстрактный класс концепта дополняется специальным атрибутом `[SupertypingConstraint]`[2] (их может быть несколько).
- Каждому ограничению подтипирования второго рода соответствует ограничение подтипирования в секции `where`.
- Сигнатуры функций переводятся абстрактные методы-члены абстрактного класса концепта. Функции с реализацией по умолчанию отображаются в виртуальные методы.

Рис. 2.6 демонстрирует код, соответствующий результату трансляции кода на Рис. 2.5. Все концепты переведены в абстрактные классы, помеченные атрибутом `[Concept]`. Типовые параметры этих классов представляют параметры концептов, ассоциированные типы и вложенные концепт-требования. Например, класс `CTransferFunction` содержит типовый параметр `CSemilattice_Data`, указывающий наличие вложенного концепт-требования, а в секции `where` зафиксирован класс концепта `CSemilattice<Data, CEquatible_T>`, соответствующий этому требованию. Параметры вложенных концепт-требований также сопровождаются ограничением `new()`, мы обсудим его смысл ниже.

[2].NET не поддерживает ограничения *над*типирования для параметров обобщённого кода, только подтипирования

```
concept CEquatible[T]
{
    bool Equal(T x, T y);
    bool NotEqual(T x, T y) { ... }
}
concept CComparable[T] refines CEquatible[T]
{ bool Less(T x, T y); }

concept CTransferFunction[TF]
{
    type Data;
    require CSemilattice[Data];
    Data Apply(TF tf, Data d);
    TF Compose(TF tfa, TF tfb);
}

class EncryptedData<T> {...}
class PlainData {...}
class Opq { }
...
class C<T> : B<T>{ }

concept C1[S]
{
    type Key;
    require CEquatible[Key];
    type Data <: EncryptedData<Key>;
    type Abc >: C<Opq>;
    Data GetData(S x);
}
concept C2[S, T]
{
    require C1[S];
    type Data == C1[S].Data;
    type Efg >: C<Data>;
    ...
}
```

Рис. 2.5: Трансляция концептов: исходный код

Уточняющий концепт `CComparable` представлен наследником класса `CEquatible`. В отличие от перевода концептов в G [15], наследование в нашем случае необходимо, поскольку должна быть обеспечена возможность использования уточняющих концептов вместо уточняемых.

```
[Concept]
abstract class CEquatible<[IsConceptParameter]T>
{
    public abstract bool Equal(T x, T y);
    public virtual bool NotEqual(T x, T y) {...}
}

[Concept]
abstract class CComparable<[IsConceptParameter]T> : CEquatible<T>
{ public abstract bool Less(T x, T y); }

[Concept]
abstract class CTransferFunction<[IsConceptParameter]TF,
        [IsAssociatedType]Data,
        [IsNestedConceptRequirement]CSemilattice_Data>
    where CSemilattice_Data : CSemilattice<Data>, new()
{
    public abstract Data Apply(TF tf, Data d);
    public abstract TF Compose(TF tfa, TF tfb);
}

class EncryptedData<T> {...}
class PlainData {...}
class Opq { }
...
class C<T> : B<T>{ }

[Concept]
[SupertypingConstraint("Abc", typeof(C<Opq>))]
abstract class C1<[IsConceptParameter]S,
        [IsAssociatedType]Key, [IsAssociatedType]Data, [IsAssociatedType]Abc,
        [IsNestedConceptRequirement]CEquatible_Key>
    where Data : EncryptedData<Key>
    where CEquatible_Key : CEquatible<Key>, new()
{
    public abstract Data GetData(S x);
}

[TypeVariableRepresenter]
class TypeVar01 { }

[Concept]
[SupertypingConstraint("Efg", typeof(C<TypeVar01>))]
[TypevarClassMap(typeof(TypeVar01), "Data")]
abstract class C2<[IsConceptParameter]S, [IsConceptParameter]T,
        [IsAssociatedType]Data, [IsAssociatedType]Efg,
        [IsNestedConceptRequirement]C1_S,
        C1_S__Key, C1_S__Abc, C1_S__CEquatible_Key>
    where C1_S : C1<S, C1_S__Key, Data, C1_S__Abc,
                    C1_S__CEquatible_Key>, new()
    where Data : EncryptedData<C1_S__Key>
    where C1_S__CEquatible_Key : CEquatible<C1_S__Key>, new()
{...}
```

Рис. 2.6: Трансляция концептов: итоговый код

Равные типы представляются одним типовым параметром. Например, в классе `C2` типовый параметр `Data`, соответствующий ассоциированному типу концепта, также представляет ассоциированный тип `Data` вложенного концепт-требования `C1`, поскольку от них требуется равенство. Алиасы типов могут быть выражены с помощью атрибутов (например, `[TypeAlias(Name1, Name2)]`), но в целевом коде все эти имена заменяются одним типовым параметром, поэтому подобная информация оказывается бесполезной.

Ограничения подтипирования вида `S <: T`, где `S` — типовая переменная, а `T` — произвольный тип, включаются в секцию `where`: это ограничения подтипирования второго рода, а также ограничения, описывающие верхние границы типовых переменных. Ограничения надтипирования, описывающие нижние границы типовых переменных (`S >: T`, где `S` — типовая переменная, а `T` — нет), не могут быть непосредственно выражены в базовом языке. Мы используем атрибут `[SupertypingConstraint(...)]`, чтобы решить эту проблему, он содержит два аргумента: имя типового параметра (в определении класса типовые параметры уникальны) и соответствующий ему объект `System.Type`. Средства .NET не позволяют получить информацию `System.Type` об открытых сконструированных классах (как `C<Data>` в концепте `C2`), поэтому генерируется вспомогательный класс `TypeVar01`: он играет роль типовой переменной в `C<TypeVar01>`, а атрибут `[TypevarClassMap(...)]` задаёт имя соответствующего ему типового параметра.

Типовые параметры и ассоциированные типы вложенных концепт-требований рекурсивно включаются в определение абстрактного класса внешнего концепта, но уже без атрибутов (как, например, `C1_S__Key`, `C1_S__Abc` или `C1_S__CEquatible_Key` в концепте `C2`). Все ограничения подтипирования из секции `where` вложенного концепт-требования также

дублируются, поскольку C# не поддерживает распространение ограничений. Таким образом, определения концептов в целевом коде оказываются достаточно объёмными и громоздкими. В Гл. 1.1 мы обсуждали проблемы низкоуровневого моделирования концептов на простом примере алгоритма `Sort`. Чем более сложным является концепт, тем «хуже» его низкоуровневое представление — работать с таким кодом непосредственно крайне неудобно.

2.2.3 Трансляция обобщённого кода

Семантический анализ обобщённых классов и методов обязательно включает унификацию набора ограничений, задаваемых концепт-требованиями. Если унификация прошла успешно, подстановка типов применяется к контексту обобщённого кода, заданного концепт-требованиями — набору сигнатур функций, а также всему обобщённому коду. После этого обобщённый код подвергается проверке типов.

Трансляция. Обобщённые классы переводятся в обобщённые классы, обобщённые методы — в методы, однако результирующий обобщённый код содержит бóльшее число типовых параметров, чем исходный. Типовые параметры обобщённого кода отображаются в типовые параметры. Но концепт-требования также отображаются в типовые параметры вместе со всеми ассоциированными типами и вложенными концепт-требованиями. Как и в случае с трансляцией концептов, все ограничения подтипирования из секций `where` концепт-требований дублируются в секцию `where` обобщённой конструкции.

```
public static class ConceptSingleton<T>
    where T : class, new()
{
    private static readonly T instance = new T();
    public static T Instance
    {
        get { return instance; }
    }
}
```

Рис. 2.7: Класс объектов-синглтонов концептов

```
class BinarySearchTree<T>
    // концепт требование с алиасом
    where CComparable[T] using cCmp
{
    private BinTreeNode<T> root;
    ...
    private bool AddNested(T x, ref BinTreeNode<T> root)
    {   ...
        // ссылка на концепт по имени
        if (cCmp.Equal(x, root.data))
            return false;
        else if (Less(x, root.data))
            return AddNested(x, ref root.left);
        else
            return AddNested(x, ref root.right);
    }
    ...
}
public class BSTSet<T>
{
    // требование CComparable[T] распространено
    private BinarySearchTree<T> bst;
    ...
    public bool FindMaxLower(T x, out T maxLow)
    {
        ...
            // ссылка на слово через ключевое слово 'reqs'
            if (reqs.Comparable[T].Less(curr.data, x))
        ...
    }
}
```

Рис. 2.8: Трансляция обобщённых классов: исходный код

Рис. 2.10 иллюстрирует результат трансляции кода, представлен-

```
static class GraphMethods
{
    public static void TraverseGraph<G>
            (G graph, Action<Data> act)
        where CDataGraph[G]
    {
        var look = new HashSet<Vertex>();
        foreach (Vertex v in GetSources(graph))
            TraverseVertex(v, look, act);
    }

    public static void TraverseVertex<V>(V vertex,
            HashSet<V> look, Action<cVert.Data> act)
        where CDataVertex[V] using cVert
    {
        if (look.Contains(vertex))
            return;
        look.Add(vertex);
        act(cVert.GetValue(vertex));
        var childs = cVert.GetSuccessors(vertex);
        foreach (V child in childs)
            TraverseVertex(child, look, act);
    }
}
```

Рис. 2.9: Трансляция обобщённых методов: исходный код

ного на Рис. 2.8 и Рис. 2.9.

Статический обобщённый класс `ConceptSingleton<T>`, приведённый на Рис. 2.7, позволяет работать с моделями концептов, представленными в единственном экземпляре — объект модели однозначно определяется типом соответствующего класса. Ограничение на конструктор по умолчанию `new()` необходимо для создания соответствующего объекта. Это ограничение приходится указывать для всех типовых параметров, описывающих концепт-требования, иначе обратиться к нужному объекту-синглтону через статический класс `ConceptSingleton<T>` из обобщённого кода нельзя. Так, например, `CSemilattice_Data` концепта `CTransferFunction` (Рис. 2.6) и `CComparable_T` класса `BSTSet<T>` (Рис. 2.10) снабжены ограничением `new()` именно поэтому.

Вызов функции, предоставляемой концептом, отображается в обращение к методу соответствующего абстрактного класса концепта. Например, непосредственный вызов функции `Less` в методе `AddNested` (Рис. 2.8) транслируется в `ConceptSingleton<CComparable_T>.Instance.Less` так же, как и её вызов через `reqs` в `FindMaxLower`.

При переводе обобщённые классы и методы, содержащие концепт-требования, получают новые типовые параметры. Поэтому все обращения к таким конструкциям должны быть исправлены, как, например, тип поля `bst` в классе `BSTSet`.

Замечание. В классе `GraphMethods` используются упрощённые версии концептов `DataVertex` и `DataGraph`: вместо дополнительного ассоциированного типа множества вершин `VertexSet` используется `HashSet<Vertex>`.

```
[GenericClassWithConcepts]
class BinarySearchTree<[IsGenericParameter]T, [IsConceptRequirement]CComparable_T>
    where CComparable_T : CComparable<T>, new()
{
    ...
    private bool AddNested(T x, ref BinTreeNode<T> root)
    {
        CComparable_T cCmp = ConceptSingleton<CComparable_T>.Instance;
        if (cCmp.Equal(x, root.data))
            return false;
        else if (ConceptSingleton<CComparable_T>.Instance.Less(x, root.data))
            return AddNested(x, ref root.left);
        else
            return AddNested(x, ref root.right);
    }
    ...
}
[GenericClassWithConcepts]
class BSTSet<[IsGenericParameter]T, [IsPropagatedConceptRequirement]CComparable_T>
    where CComparable_T : CComparable<T>, new()
{
    private BinarySearchTree<T, CComparable_T> bst;

    public bool FindMaxLower(T x, out T maxLow)
    {   ...
            if (ConceptSingleton<CComparable_T>.Instance.Less(curr.data, x))
        ...
    }
}
public static class GraphMethods
{
    [GenericMethodWithConcepts]
    public static void TraverseGraph<[IsGenericParameter]G,
    [IsNestedConceptRequirement]CDataGraph_G, Vertex, Data, CDataVertex_Vertex>
        (G graph, Action<Data> act)
        where CDataGraph_G : CDataGraph<G, Vertex, Data, CDataVertex_Vertex>, new()
        where CDataVertex_Vertex : CDataVertex<Vertex, Data>, new()
    {
        var look = new HashSet<Vertex>();
        foreach (Vertex v in ConceptSingleton<CDataGraph_G>.Instance.GetSources(graph))
            TraverseVertex<Vertex, CDataVertex_Vertex, Data>(v, look, act);
    }

    [GenericMethodWithConcepts]
    public static void TraverseVertex<[IsGenericParameter]V,
            [IsNestedConceptRequirement]CDataVertex_V, Data>
        (V vertex, HashSet<V> look, Action<Data> act)
        where CDataVertex_V : CDataVertex<V, Data>, new()
    {
        CDataVertex_V cVert = ConceptSingleton<CDataVertex_V>.Instance;
        if (look.Contains(vertex))
            return;
        look.Add(vertex);
        act(cVert.GetValue(vertex));
        var childs = cVert.GetSuccessors(vertex);
        foreach (V child in childs)
            TraverseVertex<V, CDataVertex_V, Data>(child, look, act);
    }
}
```

Рис. 2.10: Трансляция обобщённого кода: итоговый код

2.2.4 Трансляция моделей

Семантический анализ моделей включает несколько аспектов:

- Модель должна определять все функции и ассоциированные типы соответствующего концепта.
- Набор ограничений моделируемого концепта в объединении с ограничениями равенства для типов-аргументов модели и параметров концепта должен быть непротиворечив. Если модель обобщённая, набор ограничений также должен включать вспомогательные концепт-требования к параметрам модели.
- Для функций должна быть выполнена проверка типов.
- Должна быть проверены корректность множества моделей (безымянная модель может быть только одна, не должно быть моделей разного уровня специализации, и т. д.).

Трансляция. Определение модели отображается в класс-наследник соответствующего абстрактного класса концепта. Обобщённая модель отображается в обобщённый класс. Определения функций переопределяют соответствующие абстрактные и виртуальные методы предка.

Рис. 2.12 иллюстрирует трансляцию моделей, представленных на Рис. 2.11.

Модели вложенных концепт-требований концепта выражаются соответствующими типовыми аргументами: например, концепт-требование `CSemilattice[Data]` в `CTransferFunction` соответствует типовому параметру `CSemilattice_Data` класса концепта (Рис. 2.6) и заменяется типом модели `MUnionSemilattice<ReachDef, CComparable_ReachDef_Default>` в определении `CTransferFunction_ReachDefs_Default`.

Специальным образом транслируются модели уточняющих кон-

```
model<T> CEquatible[BSTSet<T>]
{
    bool Equal(BSTSet<T> x, BSTSet<T> y)
    {
        return x.EqualsTo(y);
    }
}

model MUnionSemilattice<T> of CSemilattice[BSTSet<T>]
{
    BSTSet<T> Join(BSTSet<T> x, BSTSet<T> y) {...}
}

class ReachDefsTF {...}
model CTransferFunction[ReachDefsTF]
{
    type Data = BSTSet<ReachDef>;
    using MUnionSemilattice<ReachDef>;
    ...
}
```

Рис. 2.11: Трансляция моделей: исходный код

цептов. Необходимо как-то задать связь между моделями уточняющего и уточняемого концептов, ведь класс уточняющей модели наследует класс уточняющего концепта, но не класс уточняемой модели. Например, `MUnionSemilattice` должна «уточнять» модель `CEquatible_BSTSet_T_Default`. Мы добавляем специальное поле `refinedModel` соответствующего типа в класс уточняющей модели, чтобы решить эту проблему. В уточняющей модели должны быть определены все методы «уточняемой» — они просто вызывают соответствующие методы объекта `refinedModel`.

Заметим, что если некоторый обобщённый метод содержит концепт-требование `CEquatible[BSTSet<T>]`, при вызове этого метода может быть использована модель уточняющего концепта `CComparable[BSTSet<T>]`, поскольку `MUnionSemilattice<T, CComparable_T>`
<: `CSemilattice<BSTSet<T, CComparable_T>>`
<: `CEquatible<BSTSet<T, CComparable_T>>`.

```
[ExplicitModel]
public class CEquatible_BSTSet_T_Default<
[IsModelGenericParameter]T, CComparable_T>
    : CEquatible<BSTSet<T, CComparable_T>>
    where CComparable_T : CComparable<T>, new()
{
    public override bool Equal(BSTSet<T, CComparable_T> x,
        BSTSet<T, CComparable_T> y)
    {...}
}

[ExplicitModel]
public class MUnionSemilattice<[IsModelGenericParameter]T,
                               CComparable_T>
    : CSemilattice<BSTSet<T, CComparable_T>>
    where CComparable_T : CComparable<T>, new()
{
    [RefinedModel]
    private CEquatible_BSTSet_T_Default<T, CComparable_T>
    refinedModel =
        ConceptSingleton<CEquatible_BSTSet_T_Default<
        T, CComparable_T>>.Instance;
    ...
    public override bool Equal(BSTSet<T, CComparable_T> x,
        BSTSet<T, CComparable_T> y)
    { return refinedModel.Equal(x, y); }
}

[ExplicitModel]
public class CTransferFunction_ReachDefs_Default
: CTransferFunction<ReachDefsTF,
    BSTSet<ReachDef, CComparable_ReachDef_Default>,
    MUnionSemilattice<ReachDef, CComparable_ReachDef_Default>>
{...}
```

Рис. 2.12: Трансляция моделей: итоговый код

Трансляция обобщённых вызовов

Вызовы обобщённых методов и классов с конкретными типами корректны, если в области видимости однозначно определены модели всех требуемых концептов. Типы классов, соответствующих этим моделям, становятся значениями типовых параметров, представляющих концепт-требования обобщённого кода. Например, ниже приведённое использова-

ние класса

```
BSTSet<Rational> rations = new BSTSet<Rational>();
```

отображается в

```
BSTSet<Rational, CComparable_Rational_Default> rations
    = new BSTSet<Rational, CComparable_Rational_Default>();
```

Вызовы обобщённых конструкций из обобщённого кода транслируются аналогичным образом: примером является вызов метода `TraverseVertex` из `TraverseGraph` (Рис. 2.10).

Глава 3

Программная реализация алгоритма унификации

Данная глава посвящена приложению, представляющему реализацию алгоритма унификации для набора ограничений равенства и подтипирования. Необходимая теоретическая информация, обзор алгоритма и доказательство его завершимости были рассмотрены в Гл. 2.1.

3.1 Обзор приложения

Оконное приложение **UnifyProject** предназначено для выполнения алгоритма унификации: оно предоставляет пользователю средства задания и выполнения унификации набора ограничений. Алгоритм унификации проверяет непротиворечивость набора ограничений равенства и подтипирования, соответствующих дизайну .NET концептов (представлен в Гл. 1.2), находит подстановку типов и набор выполнимых ограничений подтипирования.

Приложение представляет собой «мини-компилятор»: исходный текст «программы» задаёт набор ограничений, «компиляция» программы выполняет анализ и унификацию ограничений, а итогом компиляции явля-

ется информация, описывающая результат унификации.

Интерфейс. Интерфейс приложения представлен на Рис. 3.1:

- Левое верхнее окно предназначено для ввода текста программы, задающей набор ограничений.
- В нижнее левое окно выводится информация о результатах унификации.
- В разделе «Подключенные библиотеки» находится список dll-библиотек, подключенных к программе. Они подключается командной меню «Программа» → «Подключить библиотеку». Отключить библиотеки можно, отметив их в списке и нажав кнопку «Отключить выбранные библиотеки».
- Меню «Файл» содержит стандартные команды для работы с текстовым файлом: «Открыть» (`Ctrl+O`), «Сохранить» (`Ctrl+S`), «Сохранить как».
- Меню «Программа» содержит две команды:
  - «Компилировать» (`F5`): компиляция текущей программы.
  - «Подключить библиотеку» (`Ctrl+L`): подключение к программе dll-библиотеки.
- Меню «Настройки» содержит две команды для настройки шрифтов в окнах текста программы и итогов компиляции.

Формат исходной программы. Программа представляет собой код на подмножестве языка C# с .NET концептами. Она может содержать секцию *подключения пространств имён* (директива `using`) и *секцию определения концептов* — последовательность из нуля или более определений концептов.

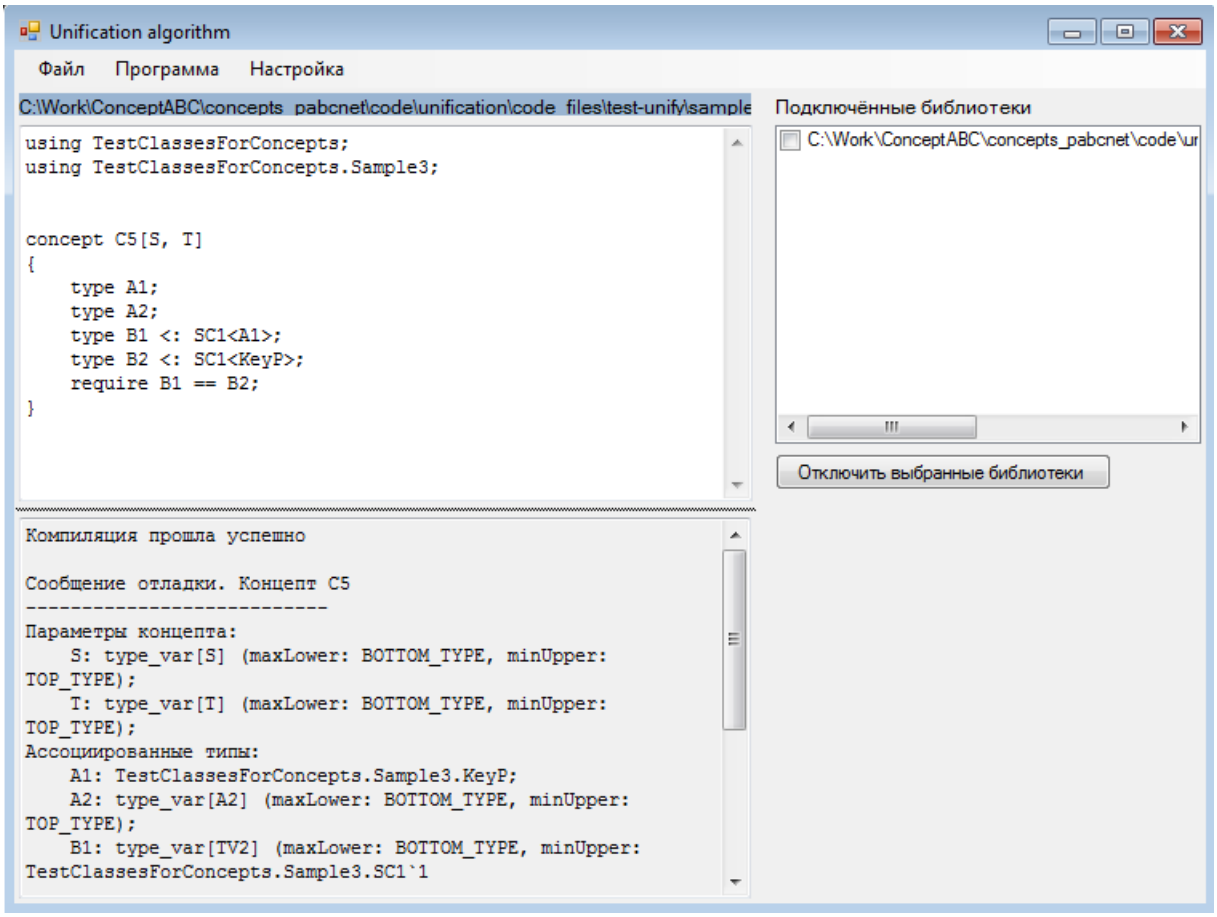

Рис. 3.1: Интерфейс приложения

Формат описания концепта соответствует дизайну, рассмотренному в Гл. 1.2, но концепт включает только конструкции, непосредственно связанные с заданием типовых ограничений: ассоциированные типы, алиасы типов, ограничения равенства и ограничения под-/надтипирования (но не сигнатуры функций или вложенные концепт-требования). Доступны однострочные комментарии (`//`).

Кроме типовых параметров концепта и ассоциированных типов в ограничениях могут использоваться типы-классы (обычные и обобщённые), доступные в области видимости. Это могут быть стандартные классы из библиотеки `mscorlib.dll`, или пользовательские классы из подключённых dll-библиотек (в программе должны быть подключены соответствующие пространства имён).

```
public class PA { }

public class KA { }
public class KB : KA { }
public class KC : KB { }
public class KD1<T> : KC { }
public class KE1<T> : KD1<QB> { }
public class KF1<S, T> : KE1<T> { }

// ------------
public class KeyS { }
...
public class SE1<S, T> : SD1<S, T> { }
public class SF1<S, T> : SE1<S, T> { }
public class SG1<S> : SF1<S, KeyS> { }
```

Рис. 3.2: Классы, используемые в примерах

Рис. 3.3 иллюстрирует пример корректной программы. К программе подключена библиотека `TestClassesForConcepts.dll`, содержащая тестовые классы, представленные на Рис. 3.2.

Результаты компиляции. Итогом компиляции программы является либо сообщение об ошибке (использование неизвестного пространства имён или типа, противоречивый набор ограничений, ...), либо описание результатов унификации для каждого концепта: подстановка типов и набор ограничений подтипирования второго рода.

Рис. 3.4 демонстрирует результаты компиляции программы, приведённой на Рис. 3.3.

На Рис. 3.5 представлен пример программы с противоречивым набором ограничений, на Рис. 3.6 — соответствующие результаты компиляции. Класс `SE1<,>` в цепочке подтипирования находится выше класса `SG1<>`, поэтому типовая переменная `A2`, совпадающая с `A3`, не может иметь границы `maxLT` = `SE1`<...>, `minUT` = `SG1`<...>.

```
// подключение пространства имён
using System.Collections.Generic;
using TestClassesForConcepts.Sample3;

// определение концепта C1 с одним параметром T
concept C1[T]
{
    // ассоциированный тип
    type A1;
    // ассоциированный тип с ограничением
    type A2 <: List<List<A1>>;
}

// определение концепта C2 с тремя параметрами
concept C2[S, T, U]
{
    type A1 >: KD1<U>;
    // ограничение подтипирования
    require A1 <: KA;

    type B1 <: KB;
    require KF1<T, B1> <: KE1<KD1<U>>;
    require B1 >: A1;

    type B2 <: PA;
    //require B2 <: B1;
}
```

Рис. 3.3: Пример корректной программы

```
Компиляция прошла успешно

Сообщение отладки. Концепт C1
---------------------------
Параметры концепта:
    T: type_var[T] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
Ассоциированные типы:
    A1: type_var[A1] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
    A2: type_var[TV1] (maxLower: BOTTOM_TYPE,
        minUpper: System.Collections.Generic.List`1[
                  System.Collections.Generic.List`1[A1]]);
Ограничения:
{}
Подстановка: [ type_var[TV1]/type_var[A2] ]

Сообщение отладки. Концепт C2
---------------------------
Параметры концепта:
    S: type_var[S] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
    T: type_var[T] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
    U: type_var[U] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
Ассоциированные типы:
    A1: TestClassesForConcepts.Sample3.KD1`1[U];
    B1: TestClassesForConcepts.Sample3.KD1`1[U];
    B2: type_var[TV2] (maxLower: BOTTOM_TYPE, minUpper:
        TestClassesForConcepts.Sample3.PA);
Ограничения:
{}
Подстановка: [ TestClassesForConcepts.Sample3.KD1`1[U]/type_var[TV3],
    TestClassesForConcepts.Sample3.KD1`1[U]/type_var[TV5],
    TestClassesForConcepts.Sample3.KD1`1[U]/type_var[TV4],
    TestClassesForConcepts.Sample3.KD1`1[U]/type_var[B1],
    TestClassesForConcepts.Sample3.KD1`1[U]/type_var[TV1],
    type_var[TV2]/type_var[B2],
    TestClassesForConcepts.Sample3.KD1`1[U]/type_var[A1] ]
```

Рис. 3.4: Результаты компиляции программы на Рис. 3.3

```
using TestClassesForConcepts.Sample3;

concept C1[T, S]
{
    type A1;
    type A2 >: SE1<A1, KeyS>;
    type A3 == A2;
    require A3 <: SG1<A1>;
}
```

Рис. 3.5: Пример некорректной программы

```
Обнаружены ошибки:

Ошибка унификации. Концепт C1:
    Тип TestClassesForConcepts.Sample3.SE1`2[
        A1,TestClassesForConcepts.Sample3.KeyS]
    не может быть подтипом типовой переменной type_var[TV2],
        так как является надтипом верхней границы
        TestClassesForConcepts.Sample3.SG1`1[A1]

Сообщение отладки. Концепт C1
--------------------------
Параметры концепта:
    T: type_var[T] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
    S: type_var[S] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
Ассоциированные типы:
    A1: type_var[A1] (maxLower: BOTTOM_TYPE, minUpper: TOP_TYPE);
    A2: type_var[TV2] (maxLower: BOTTOM_TYPE, minUpper:
        TestClassesForConcepts.Sample3.SG1`1[A1]);
    A3: type_var[TV2] (maxLower: BOTTOM_TYPE, minUpper:
        TestClassesForConcepts.Sample3.SG1`1[A1]);
Подстановка: [ type_var[TV2]/type_var[TV1],
    type_var[TV2]/type_var[A3],
    type_var[TV2]/type_var[A2] ]
```

Рис. 3.6: Результаты компиляции программы на Рис. 3.5

3.2 Особенности реализации

Программная реализация выполнена в форме проекта оконного приложения на языке C#, .NET Framework 4.5 (решение *UnifyProject*). В структуре приложения можно выделить несколько компонент:

1. Синтаксический анализатор.
2. Семантический анализатор.
3. «Компилятор», включающий синтаксический и семантический анализаторы, окружение анализа и обработчик ошибок компиляции.
4. Интерфейс.

Синтаксический анализатор реализован с помощью генераторов сканера (Gardens Point LEX [26]) и парсера (Gardens Point Parser Generator [27]). Грамматика сканера представлена в файле `ConceptParser/ParserSources/ConceptsLex.lex`, парсера — в `ConceptParser/ParserSources/ConceptsYacc.y`. Классы синтаксическое дерева программы определены в файле `ConceptParser/SyntaxTree.cs`.

Семантический анализ. Семантический анализ выполняется в процессе обхода синтаксического дерева программы с помощью визиторов: визитор `Compiler/SyntaxTreeAnalysis/GeneralSemanticVisitor` анализирует структуру программы в целом, а визитор `Compiler/SyntaxTreeAnalysis/ConceptVisitor` используется для анализа концептов. В начале семантического анализа инициализируется окружение компиляции `Compiler/CompilerAPI/CompilationEnvironment`, содержащее стек контекстов (в глубине стека находится глобальный контекст программ, выше — контекст концепта), таблицу символов (класс `Compiler/TableOfSymbols/SymbolTable`) и обработчик ошибок компи-

ляции.

В процессе семантического анализа активно используются средства рефлексии платформы .NET. Они необходимы для разбора dll-библиотек, подключённых к программе, а также при работе с объектами `System.Type`, представляющими типы в .NET.

Для реализации алгоритма унификации (`Compiler/Unification`) возможностей `System.Type` недостаточно. В частности, не предусмотрена работа с открытыми сконструированными обобщёнными типами: нельзя получить объект `System.Type`, представляющий инстанцию обобщённого класса типовыми переменными (а можно лишь полностью конкретную инстанцию, либо определение обобщённого класса). При этом, например, в процессе унификации нужно уметь получать базовый класс для таких типов. Чтобы решить эту проблему, мы определили класс `Compiler/TableOfSymbols/SymbolTable/NETTypeVariable` — наследник `System.Type`, он моделирует типовую переменную. Этот класс носит вспомогательный характер, он не является полноценным наследником: переопределены лишь те методы и свойства, которые необходимы для работы алгоритма унификации. Имея этот класс, можно «вручную» создавать объекты `System.Type` для открытых сконструированных обобщённых типов, используя средства рефлексии.

В каталоге `Compiler/Unification` также можно найти классы ограничений и подстановки типов, необходимые для работы алгоритма унификации. Процесс унификации запускается визитором при анализе концепта: сначала обходится заголовок и тело концепта, формируются типовые переменные и набор ограничений, затем отрабатывает алгоритм унификации. Подстановка, полученная в результате унификации, применяется к концепту.

Заключение

В данной работе представлен дизайн концептов для императивных объектно-ориентированных .NET-языков со статической типизацией (на примере языка C#). Рассмотрен возможный способ их реализации методом трансляции в базовый язык.

.NET концепты обеспечивают все возможности обобщённого программирования, предоставляемые интерфейсами, и лишены таких недостатков, как отсутствие ограничений на несколько типов и невозможность адаптации типов, которые существенны для удобства обобщённого программирования [17].

Главное отличие .NET концептов от других дизайнов (C++, G) — поддержка ограничений *подтипирования*. Это усложняет семантический анализ, так как требуется дополнительная проверка непротиворечивости набора ограничений. Проверка непротиворечивости выполняется с помощью алгоритма унификации: в работе представлен обзор алгоритма, доказана его завершимость и выполнена соответствующая программная реализация. Таблица на Рис. 3.7 представляет сравнение .NET концептов с другими дизайнами концептов (ext. C# соответствует C# с расширенными интерфейсами [16]).

Трансляция кода с .NET концептами, представленная в работе, возможна благодаря особенностям MSIL (Microsoft Intermediate Language) — он хранит обобщённый код. В отличие от трансляции G [15], концепт-

Характеристика	**G**	**C++**	**ext. C#**	**.NET концепты**
концепты с несколькими параметрами	+	+	−	+
ассоциированные типы	+	+	+	+
ограничения равенства	+	+	+	+
ограничения подтипирования	−	−	+	+
адаптация типов	+	+	−	+
перегрузка на основе концептов	+	+	−	−
раздельная проверка типов	+	+	+	+
раздельная компиляция	+	+	+	+
модульность	±	?	±	+

Рис. 3.7: Сравнение дизайнов концептов

требования обобщённого кода (как и вложенные концепт-требования в концептах) представляются дополнительными *типовыми параметрами*, а не дополнительными полями классов или параметрами методов. Благодаря этому стоимость этапа выполнения снижается по сравнению с языком G. Кроме того, трансляция сохраняет информацию об исходном коде. Это позволяет обеспечить для расширенного языка полностью раздельную компиляцию и модульность.

Литература

[1] M. H. Austern. Generic Programming and the STL: Using and Extending the C++ Standard Template Library. Addison-Wesley, 1998.

[2] Bjarne Stroustrup, Gabriel Dos Reis. Concepts — design choices for template argument checking. Technical Report N1522=03-0105, ISO/IEC JTC1/SC22/WG21. C++ Standards Committee Papers, October 2003

[3] Gabriel Dos Reis, Bjarne Stroustrup. Specifying C++ concepts. In POPL '06: Conference record of the 33rd ACM SIGPLAN-SIGACT symposium on Principles of programming languages, pages 295–308. ACM Press, 2006.

[4] Programming languages — C++. International Organization for Standardization. ISO/IEC 14882:1998. Geneva, Switzerland, September 1998.

[5] H. Sutter, T. Plum. Why We Can't Afford Export. Technical Report N1426: J16/03-0008, ISO/IEC JTC1/SC22/WG21. C++ Standards Committee Papers, March 3, 2003.

[6] Jaakko Järvi, Douglas Gregor, Jeremiah Willcock, Andrew Lumsdaine, Jeremy Siek. Algorithm specialization in generic programming: challenges of constrained generics in C++. In PLDI '06 Proceedings of the 2006 ACM SIGPLAN conference on Programming language design and implementation, 41(6):272–282. ACM Press, June 2006.

[7] Douglas Gregor, Jaakko Järvi, Jeremy Siek, Bjarne Stroustrup, Gabriel Dos Reis, Andrew Lumsdaine. Concepts: linguistic support for generic programming in C++. In OOPSLA '06 Proceedings of the 21st annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications, pages 291–310. ACM Press, October 2006.

[8] Gabriel Dos Reis, Bjarne Stroustrup, Alisdair Meredith. Axioms: Semantics Aspects of C++ Concepts. Technical Report N2887=09-0077, ISO/IEC JTC1/SC22/WG21. C++ Standards Committee Papers, June 2009.

[9] Bjarne Stroustrup. The C++0x "Remove Concepts"Decision. Dr. Dobb's, July 22, 2009. URL http://www.drdobbs.com/cpp/the-c0x-remove-concepts-decision/218600111

[10] Danny Kalev, Bjarne Stroustrup. Bjarne Stroustrup Expounds on Concepts and the Future of C++. Dev^x, August 6, 2009. URL http://www.devx.com/cplus/Article/42448

[11] Larisse Voufo, Marcin Zalewski, and Andrew Lumsdaine. ConceptClang: An Implementation of C++ Concepts in Clang. In WGP '11: Proceedings of the seventh ACM SIGPLAN workshop on Generic programming, pages 71–82. ACM Press, 2011.

[12] Bjarne Stroustrup, Andrew Sutton. A Concept Design for the STL. Technical Report N3351=12-0041, ISO/IEC JTC1/SC22/WG21. C++ Standards Committee Papers, January 2012.

[13] Andrew Sutton, Bjarne Stroustrup, Gabriel Dos Reis. Concepts Lite: Constraining Templates with Predicates. Technical Report N3580, ISO/IEC JTC1/SC22/WG21. C++ Standards Committee Papers, October 2013.

[14] Bruno C. d. S. Oliveira, Adriaan Moors, Martin Odersky. Type classes as objects and implicits. In OOPSLA '10 Proceedings of the ACM international conference on Object oriented programming systems languages and applications, 45(10):341–360. ACM Press, October 2010.

[15] Jeremy J. Siek. A Language for Generic Programming. Doctotal Dissertation. Indiana University, Computer Science, 2005.

[16] Jaakko Järvi, Jeremiah Willcock, Andrew Lumsdaine. Associated Types and Constraint Propagation for Mainstream Object-Oriented Generics. In OOPSLA '05 Proceedings of the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications, 40(10):1–19. ACM Press, October 2005.

[17] Ronald Garcia, Jaakko Järvi, Andrew Lumsdaine, Jeremy Siek, Jeremiah Willcock. An Extended Comparative Study of Language Support for Generic Programming. In Journal of Functional Programming, 17(2):145–205, March 2007.

[18] Jean-Philippe Bernardy, Patrik Jansson, Marcin Zalewski, Sibylle Schupp, Andreas Priesnitz. A comparison of C++ concepts and Haskell type classes. In WGP '08: Proceedings of the ACM SIGPLAN workshop on Generic programming, pages 37–48. ACM Press, 2008.

[19] `Array.Sort` Method. URL http://msdn.microsoft.com/library/system.array.sort.aspx

[20] Benjamin C. Pierce. Types and Programming Languages, 22 Type Reconstruction. The MIT Press, 2002.

[21] John Alan Robinson. A machine-oriented logic based on the resolution principle. J. ACM, 12(1):23–41, 1965.

[22] M. S. Paterson and M. N. Wegman. Linear unification. In STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing, pages 181–186. ACM Press, 1976.

[23] Franz Baader, Wayne Snyder. Unification Theory. In J.A. Robinson and A. Voronkov, editors, Handbook of Automated Reasoning, pages 447–533. Elsevier Science Publishers, 2001.

[24] Kryštof Hoder, Andrei Voronkov. Comparing unification algorithms in first-order theorem proving. In KI'09 Proceedings of the 32nd annual German conference on Advances in artificial intelligence, pages 435–443. Springer-Verlag Berlin, Heidelberg, 2009.

[25] Dmitriy Traytel, Stefan Berghofer, Tobias Nipkow. Extending Hindley-Milner Type Inference with Coercive Structural Subtyping. In APLAS'11 Proceedings of the 9th Asian conference on Programming Languages and Systems, pages 89–104. Springer-Verlag Berlin, Heidelberg, 2011.

[26] Gardens Point LEX. URL http://gplex.codeplex.com

[27] Gardens Point Parser Generator. URL http://gppg.codeplex.com