Московский государственный университет имени М. В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра системного программирования

Дипломная работа на тему «восстановление объектных структур данных при декомпиляции»

Выполнил:
студент 527 группы
Фокин А. П.

Научный руководитель:
к.ф.-м.н., доцент Чернов А. В.

Москва, 2009

Аннотация

В работе рассматриваются методы автоматического восстановления объектных структур данных из низкоуровневого представления программ на языке Си++.

Содержание

Список иллюстраций

Введение

Программное обеспечение требует постоянных изменений — для исправления ошибок, добавления новой функциональности, адаптации к новым требованиям, или новым возможностям аппаратуры. При этом исходный код программы является ключом к ее пониманию, и если он недоступен, то понимание принципов ее работы и внесение изменений оказывается чрезвычайно затруднено. Так как не редки случаи отсутствия исходного кода используемого приложения или компонента, то актуальной становится задача *обратного проектирования*[1)]. В применении к программному обеспечению целью обратного проектирования является повышение уровня абстракции представления программ. В процессе повышения уровня абстракции принято выделять несколько этапов [4]:

1. Получение ассемблерного представления программы из бинарного модуля, называемое дизассемблированием.

2. Восстановление программы на языке высокого уровня из ассемблерного представления, называемое декомпиляцией.

3. Последующее применение средств повышения уровня абстракции программ на языке высокого уровня в зависимости от целей анализа.

В настоящее время для разработки большого числа программных систем применяется объектно-ориентированный подход с использованием языка Си++, и потому весьма актуальны соответствующие задачи обратного проектирования. Дизассемблирование написанных на языке Си++ программ не имеет каких-либо специфичных особенностей, и потому может быть произведено стандартными методами. В настоящее время в основном рассматриваются задачи декомпиляции в язык Си, однако язык Си++ привносит новые концепции, не имеющие прямого отражения в Си — в частности, концепции объектно-ориентированного проектирования, такие как наследование и полиморфизм. Поэтому при декомпиляции кода, изначально написанного на языке Си++, необходимо как можно более полно восстанавливать эти концепции.

[1)] Здесь и далее определения всех терминов, выделенных *курсивом*, даны в приложении А.

1 Постановка задачи

Цель данной работы — разработка и реализация методов восстановления объектных структур данных из низкоуровневого представления программ, написанных на языке Си++, для компиляторов GCC[2)] и MSVC[3)] для платформы x86. При этом предполагается, что программа на языке Си++ не была подвергнута обфускации, и рассматриваемое низкоуровневое представление было получено с использованием одного из перечисленных компиляторов без произведения каких-либо последующих преобразований. Под низкоуровневым представлением понимается программа на языке ассемблера, полученная дизассемблированием исполняемого файла. В дальнейшем будем называть низкоуровневое представление ассемблерным.

Необходимо восстановить иерархию использованных в программе *полиморфных классов*. Дадим определение корректно восстановленной иерархии полиморфных классов. Пусть N — это некоторый компилятор с языка Си++ в ассемблер, а N_n — компилятор N с набором опций n. Тогда для некоторой программы $\mathbf{P}$ на языке Си++, $\mathbf{A} = N_n(\mathbf{P})$ является программой на языке ассемблера, полученной путем компиляции программы $\mathbf{P}$ компилятором N с набором опций n. Будем говорить, что множество полиморфных классов $\mathfrak{C}_\mathbf{A}$ с определенным на нем бинарным отношением непосредственного наследования $\leftarrow \subseteq \mathfrak{C}_\mathbf{A} \times \mathfrak{C}_\mathbf{A}$ является результатом корректного восстановления иерархии полиморфных классов ассемблерной программы $\mathbf{A}$, если существует $\mathbf{P}'$ — программа на языке Си++, такая, что для множества ее полиморфных классов $\mathfrak{C}_{\mathbf{P}'}$ выполнено $\mathfrak{C}_{\mathbf{P}'} = \mathfrak{C}_\mathbf{A}$, для отношения непосредственного наследования $\leftarrow_{\mathbf{P}'}$ на множестве $\mathfrak{C}_{\mathbf{P}'}$ выполнено $\leftarrow_{\mathbf{P}'} = \leftarrow$, и существует компилятор N' с набором опций n' такой, что ассемблерные программы $\mathbf{A}' = N'_{n'}(\mathbf{P}')$ и $\mathbf{A}$ эквивалентны в некотором смысле.

Результатом работы разрабатываемых методов должна быть корректно восстановленная иерархия полиморфных классов для входной ассемблерной программы $\mathbf{A}$, а также соответствие между восстановленными классами и принадлежащими им виртуальными функциями.

[2)] GNU Compiler Collection — набор компиляторов для различных языков программирования и платформ, разработанный в рамках проекта GNU и распространяемый на условиях GNU General Public License.

[3)] Microsoft Visual C++ — интегрированная среда разработки приложений на языке Си++, разработанная фирмой Microsoft и поставляемая как часть комплекта Microsoft Visual Studio.

2 Обзор существующих решений

В общей задаче декомпиляции до сих пор существует много нерешенных проблем. Несмотря на то, что проблема декомпиляции известна давно, было проведено лишь небольшое количество исследований в области восстановления объектных структур данных для языка Си++. Для таких языков, как C#, восстановление объектных структур зачастую не представляет сложностей, и существует множество успешных декомпиляторов для этого языка, способных восстановить код в практически изначальном виде [14].

Наиболее близки к проблеме восстановления объектных структур данных исследования в области восстановления типов данных при декомпиляции, например [7]. Однако такие исследования фокусируются на восстановлении внутренней структуры типов данных, а не отношений между ними.

Интересны с точки зрения рассматриваемой проблемы работы [3] и [4]. Первая из этих работ посвящена обзору основных проблем, возникших при использовании декомпилятора Boomerang [19] для восстановления алгоритмов, использованных в системе анализа речи. Система представляла собой большое приложение для операционной системы Windows, написанное на языках Си и Си++, и авторам были доступны исходные коды прототипа системы. В этой работе авторы описали возникшие при декомпиляции проблемы и использованные для их решения методы, а также некоторые «неожиданности», обнаруженные в ходе работы. Одной из таких «неожиданностей» стало наличие в исполнимом модуле *информации о типах времени выполнения*, с использованием которой авторам удалось восстановить полную иерархию полиморфных классов системы со всеми именами. Однако технические детали восстановления иерархии классов авторами предоставлены не были. Во второй работе также упоминается возможность использования информации о типах времени выполнения при декомпиляции, но детали не приводятся.

С точки зрения восстановления объектных структур данных наиболее интересна работа [9]. Авторы этой работы являются специалистами в области анализа вредоносных программ, и имеют большой опыт изучения принципов работы программ при отсутствии исходного кода. В данной работе авторами описаны приемы, с использованием которых возможно восстановление иерархий полиморфных классов как в случае присутствия информации о типах времени выполнения в исполнимом файле, так и в случае ее отсутствия.

Для случая присутствия информации о типах времени выполнения в работе представлен метод извлечения этой информации из исполнимого файла

и ее анализа. Также приведены все используемые для хранения этой информации компилятором MSVC структуры. Для случая отсутствия информации о типах времени выполнения авторами предложен метод восстановления отношений между классами, основанный на анализе конструкторов, деструкторов, и таблиц виртуальных функций. Описанные алгоритмы анализа конструкторов, деструкторов, и виртуальных функций, были реализованы авторами данной работы, и протестированы на коде реально существующих вредоносных программ. Также авторами были предложены способы восстановления полей классов и распознавания методов, не являющихся виртуальными.

Основная проблема методов, предложенных в работе [9] — это их неточность. Во многих случаях предложенные методы работают, но существует также множество примеров, на которых они не полностью восстанавливают требуемую информацию об объектных структурах данных, требуя дополнительного ручного анализа, или восстанавливают ее не точно.

Как видно, задаче восстановления объектных структур данных для ассемблерного представления программ, написанных на языке Си++, на данный момент было уделено мало внимания, и было проведено лишь небольшое количество исследований в этой области. Несмотря на то, что в рассмотренных работах было представлено несколько методов анализа объектных структур данных, эти методы опираются либо на использования информации о типах времени выполнения, которая может отсутствовать, либо на соображения, которые в общем случае не выполняются, и потому могут привести к неверным результатам.

3 Восстановление объектных структур данных при декомпиляции с использованием информации о типах времени выполнения

3.1 Информация о типах времени выполнения в Си++

Обзор существующих решений проблемы восстановления объектных структур данных показал, что с использованием информации о типах времени выполнения возможно восстановление иерархии полиморфных классов в программах на языке Си++ из ассемблерного представления. Информация о типах времени выполнения, или RTTI, используется в языке Си++ для реализации следующих средств [25]:

- Оператор dynamic_cast<>. Этот оператор используется для приведения типов, однако в отличие от приведения типов в стиле Си, проверка на корректность производится во время выполнения программы, а не во время компиляции. Обычно этот оператор используется для «нисходящего» приведения типов в иерархии классов — от базового класса к производному. При этом выполняется проверка на то, что переданный объект действительно является экземпляром нужного производного класса. Для реализации такой проверки необходимо на этапе времени выполнения обладать информацией о структуре иерархии классов.

  Еще один способ использования оператора dynamic_cast<> — это преобразование к типу void*. При этом, в отличие от обычного приведения типов, возвращается указатель на начало объемлющего объекта, который в случае множественного наследования может не совпадать с указателем на текущий объект.

- Оператор typeid. Этот оператор можно применять к выражениям любого типа, но чаще всего он используется для определения фактического класса переданного объекта во время выполнения. Он возвращает объект типа type_info, с использованием которого можно, в частности, получить строковое представление имени рассматриваемого класса.

- Исключительные ситуации. При обработке исключительных ситуаций необходимо сравнивать тип возбужденной исключительной ситуации с типом, указанным в catch-блоке. Стандарт Си++ не оговаривает детали реализации обработки исключительных ситуаций, однако так как для таких

проверок на этапе времени выполнения также может потребоваться информация о структуре иерархии классов, то современные компиляторы зачастую используют приемы, схожие с используемыми в реализации информации о типах времени выполнения. К примеру, Компилятор MSVC для каждого catch-блока хранит объекты типа type_info, соответствующие всем типам исключительных ситуаций, которые могут быть обработаны данным catch-блоком, а согласно стандарту Си++, эти типы находятся в одной иерархии наследования [6].

В стандарте Си++ описан интерфейс этих средств, но ничего не сказано о том, как они должны быть реализованы. Их реализация зависит от используемого компилятором *бинарного интерфейса приложений*.

Бинарный интерфейс приложений для Си++ кроме формата информации и типах времени выполнения также определяет используемые *соглашения о вызовах*, схему *декорирования имен* и некоторые другие детали. В частности, бинарный интерфейс приложений определяет способ реализации виртуальных функций. В современных компиляторах для Си++ этот механизм реализован следующим образом: для каждого полиморфного класса генерируется *таблица виртуальных функций*, содержащая указатели на все виртуальные методы класса. Каждый объект такого класса хранит указатель на соответствующую таблицу, и все вызовы виртуальных функций производятся косвенно, с использованием этого указателя, что называется *поздним связыванием* [5, 26].

Рассмотрим бинарный интерфейс приложений, используемый в компиляторах GCC и MSVC для платформы x86.

Как компилятор GCC, так и компилятор MSVC генерируют информацию о типах времени выполнения для всех используемых в приложении полиморфных классов, добавляя указатель на соответствующую классу структуру, содержащую информацию о типах времени выполнения, перед его таблицей виртуальных функций [11, 26]. Так как каждый экземпляр полиморфного класса и так содержит указатель на таблицу виртуальных функций, то при такой организации компиляция с поддержкой информации о типах времени выполнения не приводит к появлению дополнительных издержек для кода, который информацию о типах времени выполнения не использует.

Рассмотрим формат структур, содержащих информацию о типах времени выполнения, используемый в компиляторах GCC и MSVC.

3.1.1 Формат информации о типах времени выполнения, используемый в компиляторе MSVC

Компилятор MSVC при использовании информации о типах времени выполнения для каждого полиморфного класса генерирует несколько структур, которые впоследствии используется реализацией механизма обработки исключений и оператором dynamic_cast<>. Эти структуры не документированы компанией Microsoft, и их формат был восстановлен с использованием методов обратного проектирования. Описание этих структур можно, к примеру, посмотреть в исходных кодах системы Wine, позволяющей пользователям UNIX-подобных операционных систем исполнять приложения, созданные для операционной системы Microsoft Windows [32].

Компилятор MSVC для каждого полиморфного класса перед таблицей виртуальных функций помещает указатель на RTTICompleteObjectLocator, содержащий в частности указатель на структуру RTTIClassHierarchyDescriptor, описание которой представлено на рис. 1.

```
struct RTTIClassHierarchyDescriptor {
  /** Всегда ноль. */
  unsigned long signature;

  /** Атрибуты, отвечающие за множественное и
   * виртуальное наследование. */
  unsigned long attributes;

  /** Количество классов в pBaseClassArray. */
  unsigned long numBaseClasses;

  /** Указатель на массив структур, описывающих
   * базовые классы. */
  RTTIBaseClassDescriptor** pBaseClassArray;
};
```

Рис. 1: Описание структуры RTTIClassHierarchyDescriptor.

Базовые классы задаются структурами типа RTTIBaseClassDescriptor, описание которой представлено на рис. 2.

Описанные выше структуры помещаются в сегмент данных только для чтения. Зная точное расположение этих структур, можно восстановить полную иерархию полиморфных классов приложения. Кроме того, для каждого полиморфного класса генерируется соответствующий объект типа std::type_info, описание которого представлено на рис. 3.

```
struct RTTIBaseClassDescriptor {
  /** Указатель на соответствующий объект type_info. */
  std::type_info* pTypeInfo;

  /** Количество последующих записей в массиве,
   * соответствующих классам, являющимся предками
   * данного. */
  unsigned long numContainedBases;

  /** Информация для создания указателей на член класса. */
  struct PMD {
    /** Смещение члена класса. */
    int mdisp;

    /** Смещение указателя на таблицу виртуальных базовых
     * классов. */
    int pdisp;

    /** Смещение внутри таблицы виртуальных базовых
     * классов. */
    int vdisp;
  } where;

  /** Атрибуты, обычно равны нулю. */
  unsigned long attributes;
};
```

Рис. 2: Описание структуры RTTIBaseClassDescriptor.

```
class type_info {
public:
  /* ... */
private:
  void *_m_data;
  char _m_d_name[1];
  /* ... */
};
```

Рис. 3: Описание класса std::type_info, используемого компилятором MSVC.

В поле _m_d_name записано *декорированное* имя класса, соответствующего данному объекту класса std::type_info. В компиляторе MSVC используется разработанная компанией Microsoft и не опубликованная схема декорирования имен. Как и в случае с форматом структур, содержащих информацию о типах времени выполнения, алгоритм декорирования был восстановлен с использованием методов обратного проектирования [10]. Таким образом, используя

информацию о типах времени выполнения, можно восстановить полную иерархию полиморфных классов вместе с их именами. Согласно поставленным в главе 1 условиям, рассматриваемая программа перед компиляцией не была подвергнута обфускации, то есть восстановленные имена классов соответствуют их реальным именам в исходной программе на языке Си++. При проектировании объектно-ориентированных систем классам стараются давать осмысленные имена, соответствующие их функциональности, и потому знание этих имен может существенно упростить понимание принципов работы системы.

3.1.2 Формат информации о типах времени выполнения, используемый в компиляторе GCC

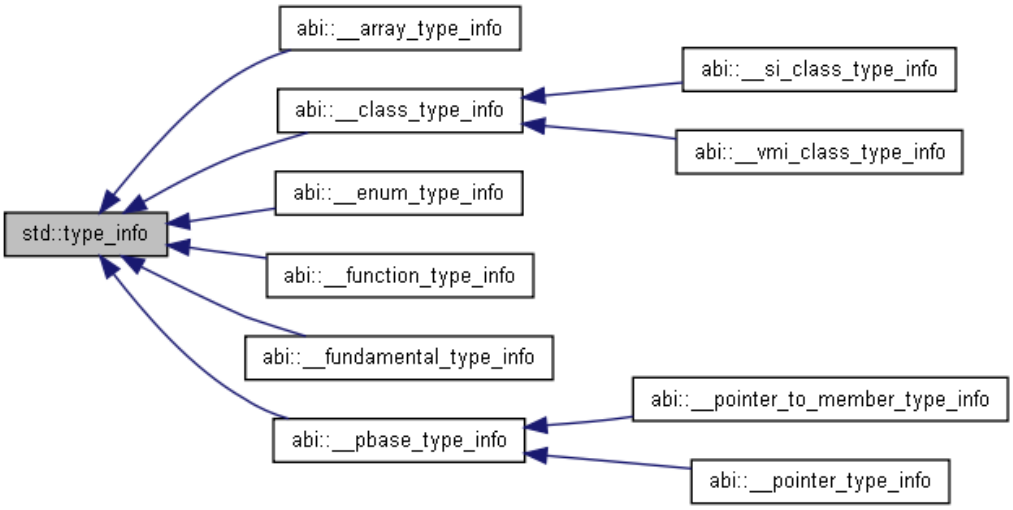

Рис. 4: Иерархия производных от std::type_info классов, используемая в компиляторе GCC.

Компилятор GCC, в отличие от MSVC, для хранения информации о типах времени выполнения использует иерархию производных от std::type_info классов. Описание иерархии этих классов приводится в заголовочном файле <cxxabi.h>, поставляемом с компилятором GCC, и доступно для использования из любой программы, написанной на Си++. Организация доступа к информации о типах времени выполнения в компиляторе GCC схожа с используемой в MSVC — для каждого полиморфного класса перед таблицей виртуальных функций помещается указатель на соответствующий ему объект производного от std::type_info класса, хранящий информацию о типе. Иерархия производных от std::type_info классов представлена на рис. 4.

Для классов, не имеющих базовых, используются тип abi::__class_type_info,

не содержащий никакой дополнительной информации по сравнению с std::type_info. Он является базовым для двух других классов — abi::__si_class_type_info и abi::__vmi_class_type_info. Первый используется для классов, имеющих ровно один невиртуальный публичный базовый класс[4]. По сравнению с std::type_info, он содержит только одно дополнительное поле - указатель на abi::__class_type_info соответствующего базового класса. Для всех прочих классов используется тип abi::__vmi_class_type_info. Описание этого класса представлено на рис. 5.

```
class __vmi_class_type_info: public __class_type_info {
public:
  /** Атрибуты, отвечающие за множественное и
   * виртуальное наследование. */
  unsigned int __flags;

  /** Количество базовых классов. */
  unsigned int __base_count;

  /** Массив структур, описывающих базовые классы. */
  __base_class_type_info __base_info[1];
};
```

Рис. 5: Описание класса abi::__vmi_class_type_info.

Для хранения информации о базовых классах используется массив структур типа abi::__base_class_type_info. Описание этой структуры представлено на рис. 6. Используя информацию, предоставляемую этими структурами, можно восстановить полную иерархию полиморфных классов приложения.

```
struct __base_class_type_info {
public:
  /** Указатель на __class_type_info базового класса. */
  const __class_type_info *__base_type;

  /** Поле, хранящее смещение объекта данного класса
   * внутри экземпляра производного. */
  long __offset_flags;
};
```

Рис. 6: Описание класса abi::__base_class_type_info.

[4] англ. non-virtual public base class — базовый класс, унаследованный с модификатором public и без модификатора virtual.

```
class type_info {
public:
  /* ... */
private:
  const char *__type_name;
  /* ... */
};
```

Рис. 7: Описание класса std::type_info, используемого компилятором GCC.

Так же, как и в случае с компилятором MSVC, возможно восстановление имен классов. Рассмотрим описание класса std::type_info, представленное на рис. 7. В поле __type_name хранится указатель на декорированное имя класса, соответствующего данному объекту класса std::type_info. Декодирование осуществляется с помощью функции abi::__cxa_demangle, объявленной в <cxxabi.h>. Таким образом, компилятор GCC предоставляет возможности по восстановлению иерархий полиморфных классов как минимум такие же, как компилятор MSVC.

3.2 Получение информации о типах времени выполнения

Зная описанный выше формат использованных в программе структур, содержащих информацию о типах времени выполнения, можно путем анализа ассемблерного представления программы извлечь из них информацию обо всех полиморфных классах этой программы. С использованием извлеченной информации возможно восстановление иерархии полиморфных классов. Для этого необходимо:

- Выяснить используемый формат информации о типах времени выполнения.
- Локализовать структуры, содержащие информацию о типах времени выполнения.
- Извлечь предоставляемую этими структурами информацию о полиморфных классах.
- Провести анализ извлеченной информации.

Формат структур, содержащих информацию о типах времени выполнения, определяется использованным в компиляторе бинарным интерфейсом приложений. Так как различных бинарных интерфейсов приложений для архитектуры

не так много, то, локализовав структуры, содержащие информацию о типах времени выполнения, можно попытаться разобрать их с использованием всевозможных форматов. В случае, если рассматриваются только компиляторы GCC и MSVC, такой способ всегда дает результат в силу того, что форматы информации о типах времени выполнения, используемые этими компиляторами, сильно отличаются.

Также, если определить компилятор, с использованием которого была получена анализируемая программа, и узнать используемый этим компилятором бинарный интерфейс приложений, то можно определить и используемый в анализируемой программе формат структур, содержащих информацию о типах времени выполнения. Компилятор в большинстве случаев можно определить по используемой приложением стандартной библиотеке. Если приложение использует динамически подключаемую стандартную библиотеку, то компилятор определяется по файлу этой библиотеки. Если же используется статически связанная с программой стандартная библиотека, то ее версию и производителя можно выявить с использованием сравнения сигнатур функций. В случае же, если стандартная библиотека приложением не используется, что является редким случаем, можно применить описанный выше переборный метод.

Извлечение информации о полиморфных классах при известном формате и положении структур, содержащих информацию о типах времени выполнения, не представляет проблем. Основная сложность на этом этапе заключается в локализации этих структур.

Так как указатель на структуру, содержащую информацию о типах времени выполнения, всегда предшествует соответствующей таблице виртуальных функций (см. главу 3.1), то задача локализации этих структур сводится к задаче поиска таблиц виртуальных функций.

3.2.1 Локализация информации о типах времени выполнения

Как было сказано в главе 3.2, задача локализации структур, содержащих информацию о типах времени выполнения, может быть сведена к задаче поиска таблиц виртуальных функций. На основе следующих соображений можно построить алгоритм поиска таблиц виртуальных функций:

- Таблицы виртуальных функций хранятся в сегменте данных.
- Каждая таблица виртуальных функций представляет собой массив из указателей в сегмент кода.

- Каждый такой указатель ссылается на начало функции или *адаптера*.
- Из сегмента кода существуют ссылки на первый элемент таблицы виртуальных функций — адрес первого элемента таблицы используется в конструкторе и деструкторе соответствующего класса.
- На остальные элементы таблицы виртуальных функций ссылок нет.

```
1:  p ← DataSegment_start
2:  while p < DataSegment_end do
3:      vTableSize ← getVTableSize(p)
4:      if vTableSize ≠ 0 then
5:          if isPointerToRtti(*(p − sizeof(void*))) then
6:              dump RTTI structure at *(p − sizeof(void*))
7:          end if
8:          p ← p + sizeof(void*) · vTableSize
9:      else
10:         p ← p + sizeof(void*)
11:     end if
12: end while
```

Рис. 8: Алгоритм поиска таблиц виртуальных функций.

```
1:  function getVTableSize(p)
2:  if not isReferenced(p) or not isPointerToFunction(*p) then
3:      return 0
4:  else
5:      vTableSize ← 1
6:      loop
7:          if isReferenced(p) or not isPointerToFunction(*p) then
8:              return vTableSize
9:          end if
10:         vTableSize ← vTableSize + 1
11:         p ← p + sizeof(void*)
12:     end loop
13: end if
```

Рис. 9: Описание функции вычисления размера таблицы виртуальных функций.

Перед таблицей виртуальных функций должен находиться указатель на соответствующую структуру, содержащую информацию о типах времени выполнения, формат которой известен (см. главы 3.1.2 и 3.1.1) и накладывает некоторые ограничения на возможные значения ее двоичного представления. Поэтому

после локализации таблицы виртуальных функций необходимо также проверить выполнение этих ограничений.

Описание алгоритма поиска структур, содержащих информацию о типах времени выполнения, представлен на рис. 8. Этот алгоритм использует функцию *getVTableSize*, описание которой приведено на рис. 9. При описании алгоритмов также были использованы следующие функции:

isPointerToRtti — определяет, может ли данный указатель ссылаться на структуру, содержащую информацию о типах времени выполнения.

isReferenced — определяет, существуют ли ссылки на данную локацию из сегмента кода.

isPointerToFunction — определяет, является ли данный указатель указателем на функцию.

Согласно описанным в главах 3.1.1 и 3.1.2 форматам информации о типах времени выполнения, путем разбора структуры, содержащей информацию о типах времени выполнения для некоторого класса A, можно узнать имена всех непосредственные базовых классов для A. Тогда, применяя представленный на рис. 8 алгоритм поиска структур, содержащих информацию о типах времени выполнения, можно путем разбора этих структур восстановить полную иерархию полиморфных классов программы, причем такое восстановление будет корректным.

4 Восстановление объектных структур данных при декомпиляции при отсутствии информации о типах времени выполнения

В связи с тем, что использование информации о типах времени выполнения в программах на языке Си++ считается «дурным тоном», и зачастую используется неправильно [12], сегодня существуют программные продукты, не использующие информацию о типах времени выполнения. В случае, если программа была скомпилирована без информации о типах времени выполнения, для каждого полиморфного класса по-прежнему генерируется таблица виртуальных функций. Вообще говоря, уникальность таблицы для каждого класса не гарантируется, и компилятор может объединить таблицы для двух полиморфных классов Base и Derived, если Base является предком Derived и при этом Derived не *перекрывает* ни одну из виртуальных функций, объявленных в Base. Однако в случае, если хотя бы одна из добавленных в Derived виртуальных функций не являются *чисто виртуальной*, компиляторы GCC и MSVC создают для Derived отдельную таблицу виртуальных функций.

Как было отмечено в главе 1, для корректного восстановления иерархии классов некоторой ассемблерной программы $\mathbf{A}$, полученной путем компиляции программы на языке Си++ $\mathbf{P}$, необходимо построить множество классов $\mathfrak{C}_{\mathbf{A}}$, и восстановить отношение непосредственного наследования $\leftarrow \subseteq \mathfrak{C}_{\mathbf{A}} \times \mathfrak{C}_{\mathbf{A}}$. В дальнейшем будем рассматривать отношение[5)] $\lhd = \leftarrow^{+}$. $\lhd$ является отношением строгого частичного порядка, так как оно:

- Антирефлексивно: $\forall A \in \mathfrak{C}_{\mathbf{A}} : \neg(A \lhd A)$;
- Ассиметрично: $\forall B, D \in \mathfrak{C}_{\mathbf{A}} : B \lhd D \Longrightarrow \neg(D \lhd B)$;
- Транзитивно: $\forall B, C, D \in \mathfrak{C}_{\mathbf{A}} : B \lhd C \wedge C \lhd D \Longrightarrow B \lhd D$.

Это в частности означает, что отношение $\leftarrow$ восстанавливается по отношению $\lhd$ следующим образом: $\leftarrow = \lhd \setminus \lhd \circ \lhd$.

В дальнейшем будем предполагать, что компилятор, с помощью которого была скомпилирована исследуемая программа, известен. Информация о компиляторе могла быть получена как с использованием методов, описанных в главе 3.2, так и из внешних источников.

[5)] Здесь и далее определения используемых обозначений даны в приложении А.

4.1 Обработка множественного наследования

Так как для классов, имеющих несколько непосредственных базовых полиморфных классов, компиляторы GCC и MSVC генерируют несколько таблиц виртуальных функций — по одной таблице на каждую таблицу виртуальных функций каждого непосредственного базового полиморфного класса [5, 26], то в общем случае отличить множественное наследование от одиночного оказывается затруднительно. На рис. 11 представлены сгенерированные компилятором MSVC таблицы виртуальных функций для иерархии классов, представленной на рис. 10.

```
class A {
public:
    virtual void a() {/* ... */}
};

class B {
public:
    virtual void b() {/* ... */}
};

class C: public A, public B {
public:
    virtual void a() {/* ... */}
    virtual void b() {/* ... */}
};
```

Рис. 10: Пример использования множественного наследования в Си++.

```
const A::`vftable'
&A::a(void)

const B::`vftable'
&B::b(void)

const C::`vftable'{for `A'}
&C::a(void)

const C::`vftable'{for `B'}
&C::b(void)
```

Рис. 11: Таблицы виртуальных функций, сгенерированные компилятором MSVC для иерархии классов, представленной на рис. 10.

```
class B1: public B {
public:
    virtual void b() {/* ... */}
};

class C: public A {
private:
    B1 b;
public:
    virtual void a() {/* ... */}
};
```

Рис. 12: Модифицированная иерархия наследования.

Сравним приведенное на рис. 12 альтернативное описание класса C с приведенным на рис. 10. Как видно, альтернативное описание было получено путем переноса метода C::b(void) из класса C в производный от B класс B1, удаления класса B из списка базовых классов для класса C, и добавления нового поля типа B1 в класс C. При такой организации иерархии множественное наследование было заменено одиночным наследованием с добавлением поля. В результате проведения экспериментов было выяснено, что для иерархий, представленных на рис. 10 и рис. 12, компилятор MSVC генерирует таблицы виртуальных функций, имеющие одинаковую структуру — такую же, как на рис. 11. Дальнейшее исследование полученных в результате компиляции ассемблерных программ показало, что они эквивалентны в смысле, определенном в [2]. Такие же результаты были получены и для компилятора GCC.

В некоторых случаях использование множественного наследования можно выявить. Один из таких случаев — наличие обращения по отрицательному смещению относительно указателя this в одной из виртуальных функций. Рассмотрим пример, приведенный на рис. 13. Согласно используемым в компиляторах GCC и MSVC бинарным интерфейсам приложений [5, 26], объект класса D будет иметь структуру, представленную на рис. 14.

При вызове виртуальной функции D::b2 в качестве указателя this ей будет передан не указатель на начало объекта класса D, а указатель на начало экземпляра класса B2 внутри этого объекта. Функция D::b2 работает с полями, унаследованными классом D от B1, и так как экземпляр класса B1 предшествует экземпляру класса B2 внутри объекта класса D, как это изображено на рис. 14, то обращение к этим полям будет производиться по отрицательному смещению относительно переданного в функцию D::b2 указателя this. Заметим, что в

```
class B1 {
    /* ... */
public:
    virtual b1() {/* ... */}
};

class B2 {
    /* ... */
public:
    virtual b2() {/* ... */}
};

class D: public B1, public B2 {
    /* ... */
public:
    virtual b2() {/* обращение к полям B1 */}
};
```

Рис. 13: Пример иерархии классов, в виртуальных функциях классов которой возможно обращение по отрицательному смещению относительно переданного в функцию указателя this.

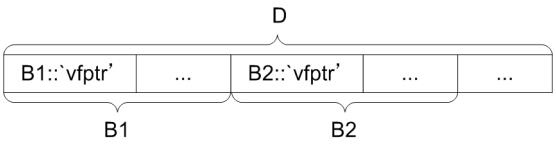

Рис. 14: Структура объекта класса D, описанного на рис. 13.

случае одиночного наследования такая ситуация невозможна.

Еще один случай, в котором можно выявить множественное наследование, связан с использованием виртуальных деструкторов. Рассмотрим пример, приведенный на рис. 15.

В данном примере для класса D генерируется две таблицы виртуальных функций — по одной для каждого полиморфного базового класса. При этом обе таблицы должны содержать указатель на виртуальный деструктор класса D. Однако при вызове виртуального деструктора для класса B2, ему передается указатель не на объект класса D, а на экземпляр класса B2 внутри объекта класса D. Поэтому в таблице виртуальных функций класса D, соответствующей классу B2, хранится указатель не на виртуальный деструктор класса D, а на *адаптер*, производящий выравнивание переданного указателя this таким образом, чтобы он указывал на начало объекта класса D, и передающий управление виртуальному деструктору. Размер полиморфного класса всегда больше нуля, так как объекты такого класса хранят указатель на таблицу виртуальных

```
class B1 {
public:
    virtual ~B1() {/* ... */}
};

class B2 {
public:
    virtual ~B2() {/* ... */}
};

class D: public B1, public B2 {
public:
    virtual ~D() {/* ... */}
};
```

Рис. 15: Пример использования виртуальных деструкторов в иерархии с множественным наследованием.

функций, и поэтому различным полиморфным базовым классам с виртуальными деструкторами соответствуют различные адаптеры. По коду такого адаптера также можно определить смещение базового класса внутри объемлющего. Метод выявления виртуальных деструкторов описан в главе 4.2.4.

Так как большинство иерархий классов используют виртуальные деструкторы, то изложенные выше факты в большинстве случаев дают возможность выявить использование множественного наследования, и определить, какие таблицы виртуальных функций принадлежат одному и тому же классу. В случае отсутствия виртуальных деструкторов можно выявить факт использования множественного наследования, но определить, какие таблицы виртуальных функций принадлежат одному и тому же классу, оказывается затруднительно. Также не смотря на то, что обращение по отрицательному относительно указателя this смещению подразумевает использование множественного наследования, такое обращение возможно и при использовании одиночного наследования при наложении дополнительных ограничений на параметры функции. В результате проведения экспериментов было выяснено, что в случае отсутствия виртуальных деструкторов, для программы на языке Си++ $\mathbf{P}$, использующей множественное наследование, и соответствующей ей программы на языке ассемблера $\mathbf{A}$, возможно построить программу $\mathbf{P}'$, использующую лишь одиночное наследование, при компиляции которой получается ассемблерная программа $\mathbf{A}'$, эквивалентная $\mathbf{A}$. Метод построения такой программы основан на тех же приемах, что были приведены выше в комментариях к рис. 10 и 12.

Таким образом, восстановление иерархии полиморфных классов можно раз-

бить на два этапа:

1. Восстановление отношения наследования на множестве таблиц виртуальных функций. Наследование на этом множестве будет одиночным, так как каждой таблице виртуальных функций некоторого класса, использующего множественное наследование, соответствует только одна таблица виртуальных функций одного из базовых классов.

2. Путем анализа виртуальных деструкторов выяснить, какие таблицы виртуальных функций принадлежат одним и тем же классам, и, таким образом, восстановить отношение наследования на множестве полиморфных классов.

На первом этапе восстановления будем считать, что каждой таблице виртуальных функций соответствует отдельный класс, то есть множество всех таблиц виртуальных функций программы на языке ассемблера $\mathbf{A}$ задает множество всех классов $\mathfrak{C}_{\mathbf{A}}$. На втором этапе восстановления иерархии некоторые из классов из $\mathfrak{C}_{\mathbf{A}}$ будут объединены в один. Рассмотрим первый этап восстановления.

4.2 Получение информации об отношении наследования

Пусть поиск таблиц виртуальных функций выполнен так, как это описано в главе 3.2.1, но с поправкой на отсутствие информации о типах времени выполнения. Пусть $\mathfrak{V}^R_{\mathbf{A}}$ — множество всех найденных таблиц виртуальных функций. Так как алгоритм поиска таблиц виртуальных функций не отличает массивы из указателей на функции от сгенерированных компилятором таблиц виртуальных функций, то в $\mathfrak{V}^R_{\mathbf{A}}$ могут присутствовать элементы, которым не соответствует таблица виртуальных функций, и для которых не существует соответствующего класса в $\mathfrak{C}_{\mathbf{P}}$. Пусть $\mathfrak{V}_{\mathbf{A}} \subseteq \mathfrak{V}^R_{\mathbf{A}}$ — реальное множество таблиц виртуальных функций. При компиляции разные классы могут разделять одну таблицу виртуальных функций, как это описано в начале главы 4, и потому каждому элементу из $\mathfrak{V}_{\mathbf{A}}$ может соответствовать более одного класса из $\mathfrak{C}_{\mathbf{P}}$. Для программы на языке Си++ $\mathbf{P}$ расширим отношение непосредственного наследования $\leftarrow_{\mathbf{P}}$ на множество $\mathfrak{V}_{\mathbf{A}}$. Пусть $\forall b, d \in \mathfrak{V}_{\mathbf{A}} : b \leftarrow_{\mathbf{P}} d \iff$ для некоторых соответствующих b и d классов $B, D \in \mathfrak{C}_{\mathbf{P}}$ выполнено $B \leftarrow_{\mathbf{P}} D$ и таблица виртуальных функций d класса D соответствует таблице виртуальных функций b класса B.

На первом этапе восстановления будем считать, что каждой таблице виртуальных функций соответствует отдельный полиморфный класс, и будем отождествлять понятия «таблица виртуальных функций» и «класс». Тогда мно-

жество полиморфных классов $\mathfrak{C}_\mathbf{A}$ однозначно задается множеством восстановленных таблиц виртуальных функций $\mathfrak{V}^R_\mathbf{A}$, и наследование на множестве $\mathfrak{C}_\mathbf{A}$ является одиночным, как это описано в конце главы 4.1. Пусть функция $\mathcal{M}_\mathbf{A} \in \mathfrak{C}_\mathbf{A} \to \mathfrak{V}_\mathbf{A} \cup \nu$ для класса из $\mathfrak{C}_\mathbf{A}$ возвращает специальное значение ν, если в $\mathfrak{V}_\mathbf{A}$ не существует соответствующей ему таблицы виртуальных функций, и соответствующую ему таблицу виртуальных функций в противном случае. Расширим функцию $\mathcal{M}_\mathbf{A}$ на множество $2^{\mathfrak{C}_\mathbf{A}}$ следующим образом:

$$\forall \mathfrak{f} \subseteq \mathfrak{C}_\mathbf{A} : \mathcal{M}_\mathbf{A}(\mathfrak{f}) = \bigcup_{C \in \mathfrak{f}} \mathcal{M}_\mathbf{A}(C). \tag{1}$$

Для восстановления связей между классами рассмотрим бинарное отношение наследования $\lhd$, определенное в главе 4. Для удобства восстановления разложим отношение $\lhd$ на два отношения $<$ и $\sim$ следующим образом:

$$< = \mathfrak{C}^2_\mathbf{A} \setminus \lhd^{-1},$$
$$\sim = \lhd^{=} \cup \lhd^{-1}.$$

Иначе говоря:

$$\forall B, D \in \mathfrak{C}_\mathbf{A} : (B, D) \in < \iff D \text{ не является предком } B,$$
$$\forall B, D \in \mathfrak{C}_\mathbf{A} : (B, D) \in \sim \iff D \text{ и } B \text{ связаны наследованием.}$$

Не сложно проверить, что $\lhd = < \cap \sim^{\neq}$. Также заметим, что оба введенных отношения рефлексивны, и при этом $\sim$ симметрично, то есть:

$$\forall A \in \mathfrak{C}_\mathbf{A} : A < A \wedge A \sim A,$$
$$\forall B, D \in \mathfrak{C}_\mathbf{A} : B \sim D \Longrightarrow D \sim B.$$

Информация о структуре иерархии классов может быть записана в виде множества ограничений на отношение $\leftarrow$. При этом каждое ограничение представимо в виде некоторого булева выражения:

$$\mathcal{R}(\leftarrow, C_1, \ldots, C_n), \text{ где } C_i \in \mathfrak{C}_\mathbf{A}, \text{ или} \tag{2}$$
$$\mathcal{R}(\leftarrow, \mathfrak{f}_1, \ldots, \mathfrak{f}_n), \text{ где } \mathfrak{f}_i \subseteq \mathfrak{C}_\mathbf{A}. \tag{3}$$

Будем рассматривать такие ограничения $\mathcal{R}$, для которых выполнено:

$$\begin{gathered}\mathcal{R}(\leftarrow_{\mathbf{P}}, \mathcal{M}_{\mathbf{A}}(C_1), \ldots, \mathcal{M}_{\mathbf{A}}(C_n)), \text{ для ограничений вида (2), и} \\ \mathcal{R}(\leftarrow_{\mathbf{P}}, \mathcal{M}_{\mathbf{A}}(\mathfrak{f}_1), \ldots, \mathcal{M}_{\mathbf{A}}(\mathfrak{f}_n)), \text{ для ограничений вида (3).}\end{gathered} \tag{4}$$

Так как в область значений функции $\mathcal{M}_{\mathbf{A}}$ входит $\nu \notin \mathfrak{V}_{\mathbf{A}}$, то будем также считать, что выражения вида (4), в которых встречается ν, всегда истинны.

Далее будут рассмотрены способы построения ограничений, для которых выполнено (4). Перед тем, как перейти к описанию способов построения таких ограничений, введем некоторые обозначения. Пусть

VT_A	—	таблица виртуальных функций класса $A \in \mathfrak{C}_{\mathbf{A}}$.
$\lvert VT_A \rvert$	—	размер таблицы виртуальных функций класса $A \in \mathfrak{C}_{\mathbf{A}}$.
VT_A^i	—	i-я виртуальная функция класса $A \in \mathfrak{C}_{\mathbf{A}}$.
$params(f)$	—	параметры функции f.
$\lvert params(f) \rvert$	—	суммарный размер параметров функции f в байтах.
$pure$	—	обработчик вызова чисто виртуальной функции.
$executes$	—	бинарное отношение «выполняет» между множествами функций и выражений.

Будем считать, что если для некоторой функции f и выражения e выполнено f *executes* e, и некоторая функция g вызывает f, используя статическое связывание, то есть выполнено g *executes* f(/* ... */), то также выполнено и g *executes* e.

4.2.1 Анализ таблиц виртуальных функций

Утверждение 1. *Пусть* $B, D \in \mathfrak{C}_{\mathbf{A}} : \lvert VT_B \rvert < \lvert VT_D \rvert$. *Тогда для ограничения* $\mathcal{R}_1^{BD} = B < D$ *выполнено (4).*

В соответствии с правилами наследования Си++ [25], если для D виртуальных функций определено больше, чем для B, то D не может являться предком B.

Утверждение 2. *Пусть* $B, D \in \mathfrak{C}_{\mathbf{A}}$ *и* $\exists\, i : VT_B^i = pure \wedge VT_D^i \neq pure$. *Тогда для ограничения* $\mathcal{R}_2^{BD} = B < D$ *выполнено (4).*

В Си++ невозможно при наследовании *перекрыть* некоторую уже определенную в базовом классе виртуальную функцию как чисто виртуальную. Поэтому в указанной выше ситуации D не может быть предком B.

Утверждение 3. *Пусть* $A, C \in \mathfrak{C}_{\mathbf{A}}$ *и* $\exists i : VT_A^i = VT_C^i \wedge VT_A^i \neq pure$. *Если при компиляции каждое объявление виртуальной функции в* $\mathbf{P}$ *породило отдельную функцию в* $\mathbf{A}$, *то для ограничения* $\mathcal{R}_3^{AC} = A \sim^+ C$ *выполнено (4).*

Если в таблицах виртуальных функций двух разных классов записан указатель на одну и ту же функцию, то это означает, что или эта функция была унаследована одним из этих классов от другого, или оба эти класса унаследовали ее от общего базового класса, или в результате оптимизаций эта функция оказалась записанной в таблицах виртуальных функций двух связанных наследованием классов. Пример для последнего случая рассмотрен в комментарии к утверждению 5.

Это утверждение опирается на то, что при компиляции каждое объявление функции в исходном файле порождает новую функцию в ассемблерном представлении. Ни компилятор MSVC, ни компилятор GCC не проводят попарного сравнения кода всех функций при сборке исполнимого файла, но при компиляции отдельного исходного файла компилятор MSVC объединяет простые функции с пустым телом или с телом вида return /* *const* */ в одну. Пусть $isPrimitive(f)$ принимает значение «истина» только в том случае, когда функция f ассемблерной программы $\mathbf{A}$ является «элементарной» — достаточно простой для того, чтобы f могла быть получена путем объединения нескольких функций из $\mathbf{P}$ с одинаковым телом. Путем проведения экспериментов было выяснено, что «элементарность» функции f определяется длиной ее ассемблерного представления — компилятор MSVC не объединяет достаточно большие функции в одну. С использованием *isPrimitive*, утверждение 3 можно переписать следующим образом.

Утверждение 4. *Пусть* $A, C \in \mathfrak{C}_{\mathbf{A}} : \exists i : VT_A^i = VT_C^i \wedge VT_A^i \neq pure \wedge \neg isPrimitive(VT_A^i)$. *Тогда для ограничения* $\mathcal{R}_4^{AC} = A \sim^+ C$ *выполнено (4).*

Верно и более общее утверждение.

Утверждение 5. *Пусть для* $\mathfrak{f} \subseteq \mathfrak{C}_{\mathbf{A}}$, *и для некоторых* $i \in \mathbb{N}$ *и функции* $f \neq pure : \neg isPrimitive(f)$ *выполнено* $\forall C \in \mathfrak{f} : VT_C^i = f \wedge \forall C \in \mathfrak{C}_{\mathbf{A}} \setminus \mathfrak{f} : VT_C^i \neq f$. *Тогда, если при компиляции не возникает ситуации, когда однажды перекрытая в базовом классе виртуальная функция появляется в таблице виртуальных функций производного класса, то для ограничения* $\mathcal{R}_5^{\mathfrak{f}} = (\mathfrak{f}, \leftarrow[\mathfrak{f}]) \in \mathbb{T}(\mathfrak{f})$, *выполнено (4).*

Заметим, что ограничение $\mathcal{R}_5^{\mathfrak{f}}$ означает, что сужение заданной отношением $\leftarrow$ иерархии наследования на множество $\mathfrak{f}$ является деревом.

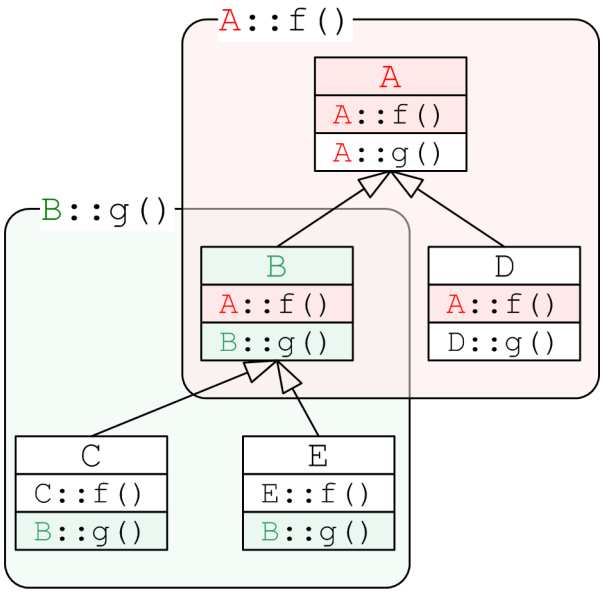

Рис. 16: Пример применения утверждения 5 к иерархии наследования.

```
class A {
public:
    virtual void f() { /* .1. */ }
};

class B: public A {
public:
    virtual void f() { /* .2. */ }
};

class C: public B {
public:
    virtual void f() { A::f(); }
};
```

Рис. 17: Пример иерархии наследования, для которой дополнительное требование утверждения 5 может быть существенным.

На рисунке 16 приведен пример применения утверждения 5 — все классы из множества $\{\mathrm{A}, \mathrm{B}, \mathrm{D}\}$ имеют функцию A::f на первой позиции в таблице виртуальных функций, и поэтому сужение иерархии наследования на множество $\{\mathrm{A}, \mathrm{B}, \mathrm{D}\}$ является деревом. Утверждение 5 справедливо и для множества классов $\{\mathrm{B}, \mathrm{C}, \mathrm{E}\}$, имеющих функцию B::g на второй позиции в таблице виртуальных функций.

Утверждение 5 опирается на то, что при компиляции не возникает ситуации, когда однажды перекрытая в базовом классе виртуальная функция появляется

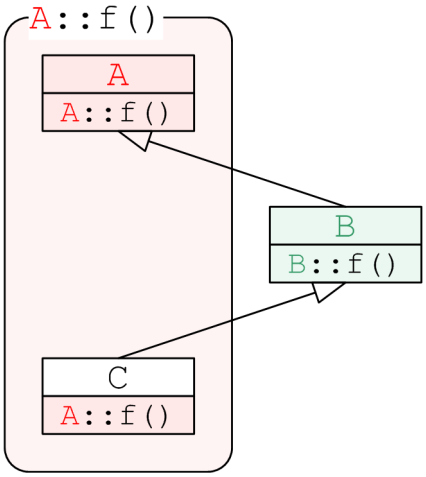

Рис. 18: Пример иерархии наследования, для которой дополнительное требование утверждения 5 существенно.

в таблице виртуальных функций производного класса. Пример такой ситуации представлен на рис. 17. При включении оптимизаций компиляторы GCC и MSVC не генерируют тело для функции C::f, а просто записывают указатель на функцию A::f в соответствующем месте в таблице виртуальных функций класса C. Рис. 18 иллюстрирует применение утверждения 5 к иерархии классов, представленной на рис. 17. Как видно, применение утверждения в данном случае породило бы ограничение, для которого не выполнено (4).

Если в утверждении 5 не требовать отсутствия ситуаций, подобных изображенной на рис. 17, то не для всех полученных с использованием этого утверждения ограничений будет выполнено (4). Однако так как ситуации, подобные изображенной на рис. 17, встречаются достаточно редко, то ограничения, для которых не выполняется (4), можно в случае возникновения конфликтов обработать отдельно.

4.2.2 Анализ параметров виртуальных функций

Утверждение 6. *Пусть* $A, C \in \mathfrak{C}_{\mathbf{A}} : \exists i : VT_i^A \neq pure \wedge VT_i^C \neq pure \wedge |params(VT_i^A)| \neq |params(VT_i^C)|$. *Тогда для ограничения* $\mathcal{R}_6^{AC} = \neg(A \sim C)$ *выполнено (4).*

Согласно правилам наследования Си++, виртуальная функция производного класса должна иметь тот же список параметров, что и перекрываемая ею виртуальная функция базового класса [25].

Используя методы статического анализа типов можно улучшить точность ограничений, порождаемых приведенным выше утверждением, сравнивая так-

же восстановленные типы параметров, а не только их размер. Однако это требует реализации статического анализа типов, и может привести к проблемам в случае, когда в исходной программе используются структуры данных union, или оператор reinterpret_cast<>.

Также проблемы могут возникнуть, если пытаться восстановить суммарный размер параметров, анализируя обращения к стеку. Рассмотрим две функции, представленные на рисунке 19.

```
int a(int p0, int p1) {
    return p0;
}

int b(int p0, int p1) {
    return p1;
}
```

Рис. 19: Пример функций, для которых нельзя восстановить суммарный размер параметров путем анализа обращений к стеку.

Так как в каждой из функций один из аргументов не используется, то независимо от порядка передачи параметров, одна из функций будет обращаться в стек по большему смещению относительно `esp`, чем другая. То есть, даже вычислив максимальное смещение относительно `esp`, по которому происходило обращение к стеку внутри функции, нельзя с уверенностью сказать, что аргументов с большим смещением нет.

Если используется соглашение о вызовах, в котором очисткой стека занимается вызываемая сторона, то суммарный размер параметров определяется по эпилогу функции. В компиляторе MSVC для виртуальных функций по умолчанию используется соглашение thiscall, обладающее этим свойством. В компиляторе GCC по умолчанию используется стандартное соглашение о вызовах Си, в котором очисткой стека занимается вызывающая сторона, и поэтому для программ, скомпилированных компилятором GCC, такой метод в большинстве случаев неприменим. Также стоит отметить, что для функций, не имеющих аргументов, как при использовании соглашения о вызовах thiscall, так и при использовании стандартного соглашения о вызовах Си, генерируется один и тот же эпилог. В случае же наличия аргументов эпилоги отличаются, и используемый тип соглашения о вызовах можно определить.

С учетом всего вышесказанного, будем считать суммарный размер параметров определенным только для функций, использующих соглашение о вызовах,

в котором очисткой стека занимается вызываемая функция.

4.2.3 Анализ вызовов виртуальных функций

Утверждение 7. *Пусть* $B, D \in \mathfrak{C}_{\mathbf{A}} : \exists i, j : VT_D^i$ *executes* this$->VT_B^j$ *(...)* $\wedge$ $VT_D^j \neq VT_B^j$. *Тогда если в программе* P *для приведения типов, не связанных наследованием, не используется преобразование* reinterpret_cast<>, *или подобное ему, то для ограничения* $\mathcal{R}_7^{BD} = B \lhd D$ *выполнено (4).*

Если метод m_D класса D вызывает метод m_B класса B для того же объекта, для которого он был вызван сам (то есть для this), то m_B должен быть определен либо в самом классе D, либо в одном из его предков. Конечно, возможен также вариант с использованием оператора reinterpret_cast<> — например, m_D может вызывать метод m_B следующим образом: reinterpret_cast<$B*$>(this)$->m_B$, однако такие конструкции зачастую лишены смысла и в реальном коде встречаются редко. Поэтому аналогично с утверждением 5, требование отсутствия в программе **P** приведения типов, не связанных наследованием, с использованием преобразования reinterpret_cast<>, или подобного ему, можно опустить, и в случае возникновения конфликтов обработать отдельно ограничения, для которых не выполняется (4).

Если m_B является виртуальной функцией, перекрытой в классе D, то остается только один вариант — B является предком D. При этом подразумевается, что при вызове виртуальной функции m_B используется *статическое связывание*, потому как лишь в этом случае на этапе статического анализа известно, какая функция будет вызвана.

Для того, чтобы определить, действительно ли происходит вызов функции VT_B^j для текущего объекта, необходимо знать способ передачи указателя this в функции VT_D^i и VT_B^j. Зная способ передачи указателя this в функцию VT_D^i, можно провести анализ потока данных и для каждой инструкции в теле функции определить множество регистров и локаций памяти, в которых хранятся копии указателя this. Используя построенные множества, и зная способ передачи указателя this в функцию VT_B^j, можно определить, действительно ли она вызывается для текущего объекта.

Способ передачи указателя this зависит от используемого функцией соглашения о вызовах. Используемые компиляторами GCC и MSVC соглашения о вызовах описаны в таблице 20. По умолчанию и компилятор GCC, и компилятор MSVC используют соглашение о вызовах thiscall .

Из таблицы 20 видно, что определение используемого функцией соглаше-

Соглашение о вызовах	Очистка стека	Передача указателя **this**
MSVC cdecl GCC cdecl	Вызывающая функция	Первый параметр в стеке
MSVC fastcall GCC fastcall	Вызывающая функция	регистр `ECX`
MSVC stdcall GCC stdcall	Вызываемая функция	Первый параметр в стеке
MSVC thiscall	Вызываемая функция	регистр `ECX`
GCC thiscall	Вызывающая функция	Первый параметр в стеке
GCC regparm		регистр `EAX`

Рис. 20: Соглашения о вызовах, используемые компиляторами GCC и MSVC.

ния о вызовах по ее телу представляет некоторые сложности. Конечно, можно считать, что используется соглашение о вызовах thiscall, что в большинстве случаев является правдой, однако такое предположение верно не всегда, и это может привести к неверным результатам. Однозначно выявить можно только использование соглашения о вызовах stdcall, которое определяется по эпилогу функции. Однако на практике соглашение о вызовах stdcall в программах на Си++ используется довольно редко.

В случае использования компилятора MSVC, виртуальной функции в регистре `ECX` нельзя передать ничего кроме указателя this, и поэтому если внутри функции происходит косвенное обращение к памяти с использованием значения, переданного в регистре `ECX`, то это означает, что в `ECX` был передан указатель this. Так как компилятор MSVC по умолчанию использует соглашение о вызовах thiscall, передающее указатель this в регистре `ECX`, то во многих случаях используемый способ передачи указателя this можно определить.

Будем применять утверждение 7 только в случаях, когда для функций VT_D^i и VT_B^j, с использованием приведенных выше соображений, был восстановлен способ передачи указателя this.

4.2.4 Анализ виртуальных деструкторов

Сначала рассмотрим, какие действия должен производить деструктор:

- Выполнить заданный программистом в теле деструктора код.
- Вызвать деструкторы для всех неунаследованных полей.
- Вызвать деструкторы для всех базовых классов. При вызове деструктора какого-либо базового класса должен использоваться указатель на таблицу виртуальных функций этого класса. Это означает, что в деструкторе

указатель на таблицу виртуальных функций будет несколько раз перезаписываться разными значениями, соответствующими различным базовым классам, как это представлено на рис. 21.

То есть реальный деструктор — это функция, сгенерированная компилятором, и написанный программистом код тела деструктора является лишь ее частью.


```
D::`destructor'() {
    this->`vfptr' = &D::`vftable';
    this->~D();

    /* ... */

    this->`vfptr' = &B::`vftable';
    this->~B();
}
```

Рис. 21: Общая схема работы деструктора.

При создании иерархий классов практически во всех случаях используется виртуальный деструктор — он дает возможность единообразно уничтожать созданные объекты, при этом не беспокоясь о совместимости существующего кода с новыми классами, которые возможно будут добавлены в будущем. Более того, компилятор GCC даже выдает предупреждение в случае отсутствия виртуального деструктора в полиморфном базовом классе. При использовании виртуального деструктора объекты иерархии обычно создаются с помощью оператора new, и уничтожаются с использованием оператора delete. Но использование оператора delete несет в себе некоторые проблемы. Рассмотрим пример, представленный на рис. 22.

```
struct V {
    virtual ~V();
};

struct W: V {
    void operator delete(void*);
};
```

Рис. 22: Пример переопределения оператора delete для класса с виртуальным деструктором.

Согласно правилам Си++ [25], при вызове оператора delete для объекта класса V память должна быть освобождена с помощью глобального оператора ::operator delete(void*), в то время как для объекта класса W должен использоваться W::operator delete(void*). Так как в месте вызова виртуального деструктора фактический класс уничтожаемого объекта неизвестен, то невозможно определить, какой оператор delete необходимо вызывать. Поэтому как компилятор GCC, так и компилятор MSVC вместо указателя на деструктор хранят в таблице виртуальных функций указатель на так называемый «освобождающий деструктор»[6)], который в случае необходимости освобождает память с помощью соответствующего оператора delete.

И в компиляторе GCC, и в компиляторе MSVC код освобождающего деструктора имеет вполне определенную структуру, и может быть идентифицирован с помощью сравнения сигнатур. Теперь, когда место освобождающего деструктора в таблице виртуальных функций определено, можно сформулировать следующее утверждение:

Утверждение 8. *Пусть* $B, D \in \mathfrak{C}_\mathbf{A} : D::\tilde{}D$ *executes* this->'vfptr' = &VT_B, *где поле* 'vfptr' *хранит указатель на таблицу виртуальных функций. Тогда для ограничения* $\mathcal{R}_8^{BD} = B \lhd D$ *выполнено (4).*

Казалось бы, это утверждение будет выполнено для всех классов, являющихся предками D, однако из-за примененных компилятором оптимизаций это может оказаться не так. Эксперименты с компиляторами GCC и MSVC показали, что в случае использования пустых деструкторов в результате оптимизаций компилятор удаляет последовательные перезаписи поля this->'vfptr', оставляя лишь последнюю. Так как в последнюю очередь всегда вызывается деструктор базового класса всей иерархии, то как минимум для него утверждение будет выполнено. Если же у D нет базовых классов, то перезапись поля this->'vfptr' будет произведена только один раз — указателем на VT_D.

Для того, чтобы определить, действительно ли происходит присваивание вида this->'vfptr' = &VT_B, необходимо знать способ передачи указателя this освобождающему деструктору, и провести анализ потока данных. Как было сказано в главе 4.2.3, способ передачи указателя this зависит от используемого освобождающим деструктором соглашения о вызовах.

В компиляторе MSVC, независимо от установленного соглашения о вызовах по умолчанию, для освобождающего деструктора всегда используется соглашение о вызовах thiscall. В компиляторе GCC используемое для освобождающих

[6)] англ. deleting destructor.

деструкторов соглашение о вызовах зависит от установленных по умолчанию атрибутов функций. Так как параметры и структура освобождающего деструктора известны, и различных соглашений о вызовах, используемых компилятором GCC, не так много (см. главу 4.2.3), то используемое соглашение о вызовах можно определить на этапе сравнения сигнатур.

4.2.5 Ненадежные источники информации

Отличительная особенность ограничений, получаемых с использованием описанных выше утверждений — это их «надежность». Даже при ослабленных требованиях, для большинства встречающихся на практике программных продуктов вероятность получить с их использованием ограничения, для которых не выполнено (4), близка к нулю. Но существуют и другие источники информации, которые дают ограничения, для которых (4) не выполняется лишь в небольшом проценте случаев. Приведем несколько примеров таких источников информации.

Пусть $MaxThisAccessOffset_A$ — максимальное смещение относительно указателя this, по которому происходит обращение к памяти в виртуальных функциях класса A, а также в функциях, вызываемых из виртуальных. Пусть $B, D \in \mathfrak{C}_{\mathbf{A}} : MaxThisAccessOffset_B < MaxThisAccessOffset_D$. Тогда для ограничения $\mathcal{R}_?^{BD} = B < D$ с большой вероятностью выполнено (4).

Действительно, при наследовании в большинстве случаев добавляется дополнительная функциональность, и для реализации этой функциональности зачастую требуется хранение некоторой дополнительной информации. Эта дополнительная информация добавляется к классу в форме полей, которые затем используются в виртуальных функциях. Из того факта, что добавленные в производном классе поля всегда имеют большее смещение, чем унаследованные [5, 26], напрямую следует приведенное выше соображение. Однако в силу того, что для встречающихся на практике программ это соображение не всегда выполнено, его нельзя напрямую использовать для восстановления отношения $<$.

Еще одним примером ненадежного источника информации может служить утверждение 7, применяемое не только к функциям, для которых используемый метод передачи указателя this может быть однозначно восстановлен, но и для всех остальных в предположении, что используется соглашение о вызовах по умолчанию.

4.3 Восстановление одиночного наследования

Как было сказано в главе 4.2, наследование на множестве классов $\mathfrak{C}_\mathbf{A}$ является одиночным. В случае использования только одиночного наследования, любое множество классов может быть представлено в виде нескольких деревьев наследования. Каждое дерево наследования является корневым деревом, причем ребра этого дерева заданы отношением непосредственного наследования.

В главе 4.2.4 было отмечено, что для каждого класса D, принадлежащего некоторому дереву наследования, корнем которого является класс B, имеющий виртуальный деструктор, в результате применения утверждения 8 будет получено ограничение $\mathcal{R}_8^{BD} = B \lhd D$. Таким образом, с использованием утверждения 8 можно восстановить множество вершин любого дерева наследования с виртуальным деструктором, а таковых, как было отмечено в главе 4.2.4, большинство. В случае же отсутствия виртуального деструктора, множество вершин дерева наследования можно восстановить с использованием утверждения 3. Таким образом, множество $\mathfrak{C}_\mathbf{A}$ можно разбить на несколько непересекающихся подмножеств, каждое из которых будет задавать отдельное дерево наследования. С каждым из таких подмножеств можно работать отдельно, и поэтому, для упрощения дальнейших рассуждений, будем считать, что все классы из множества $\mathfrak{C}_\mathbf{A}$ принадлежат одному дереву наследования.

В этом случае восстановление иерархии классов подразумевает построение дерева наследования $T \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A})$, а перечисленные выше источники информации о структуре иерархии определяют ограничения, которым это дерево должно удовлетворять. Пусть $\mathfrak{R} \subset \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) \to \{0, 1\}$ — множество всех ограничений, полученных из утверждений 1, 2, 5, 6, 7 и 8, а $\mathfrak{R}_? \subset \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) \to \{0, 1\}$ — множество всех ограничений, полученных из ненадежных источников информации, как это описано в главе 4.2.5. В $\mathfrak{R}$ также входят ограничения, полученные из утверждений 5 и 7 в предположении о том, что дополнительные требования этих утверждений всегда выполнены. Это означает, что некоторые из ограничений из $\mathfrak{R}$ могут не выполняться.

Тогда задачу построения необходимого корневого дерева можно представить как задачу максимизации функции $\mathcal{F} \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) \to \mathbb{R} \times \mathbb{R}$, определенной следующим образом:

$$\forall T \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) : \mathcal{F}(T) = (\sum_{\mathcal{R} \in \mathfrak{R}} \mathcal{R}(T), \sum_{\mathcal{R} \in \mathfrak{R}_?} \mathcal{R}(T)), \tag{5}$$

где отношение $<$ на $\mathbb{R} \times \mathbb{R}$ определено следующим образом:

$$\forall (a_1, b_1), (a_2, b_2) \in \mathbb{R} \times \mathbb{R} : (a_1, b_1) < (a_2, b_2) \Longleftrightarrow a_1 < a_2 \vee (a_1 = a_2 \wedge b_1 < b_2). \quad (6)$$

Так как различные ограничения из $\mathfrak{R}_?$ могут иметь различную «степень надежности», то при вычислении значения функции $\mathcal{F}(T)$ возможно применение весовых коэффициентов.

При такой постановке задачи можно использовать ограничения произвольной формы, но при этом точное решение будет возможно получить лишь переборными методами. Приближенное же решение для задачи, поставленной в такой форме, можно получить, использовав метод генетического программирования. В работе [8] было предложено представление деревьев, которое дает хорошие результаты в применении к методам генетического программирования. Подобное представление может быть применено и для поставленной выше задачи.

Метод генетического программирования не всегда находит лучшее решение. С другой стороны, известно, что с большой вероятностью существует дерево наследования, удовлетворяющее всем ограничениям из $\mathfrak{R}$. Введем обозначение:

$$\begin{aligned} &\mathit{isSolution} \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) \times 2^{\mathbb{T}^R(\mathfrak{C}_\mathbf{A}) \to \{0,1\}} \to \{0, 1\}, \\ &\forall T \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}), \mathfrak{R} \in 2^{\mathbb{T}^R(\mathfrak{C}_\mathbf{A}) \to \{0,1\}} : \mathit{isSolution}(T, \mathfrak{R}) \Longleftrightarrow \forall \mathcal{R} \in \mathfrak{R} : \mathcal{R}(T). \end{aligned} \quad (7)$$

Тогда, если не рассматривать ограничения из $\mathfrak{R}_?$, то в предположении, что все ограничения из $\mathfrak{R}$ могут быть выполнены, поставленная выше задача преобразуется в:

$$\text{Найти } T \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) : \mathit{isSolution}(T, \mathfrak{R}) \quad (8)$$

Так как возможна ситуация, что некоторые из ограничений из $\mathfrak{R}$ могут быть невыполнимы, и потому решения задачи 8 может не существовать, необходимо построить алгоритм, который будет строить дерево наследования, удовлетворяющее как можно большему количеству ограничений из $\mathfrak{R}$, то есть множеству ограничений $\mathfrak{R}' \subseteq \mathfrak{R}$ такому, что:

$$\mathit{isSolution}(T, \mathfrak{R}') \wedge \forall \mathcal{R} \in \mathfrak{R} \setminus \mathfrak{R}' : \nexists T \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) : \mathit{isSolution}(T, \mathfrak{R}' \cup \{\mathcal{R}\}) \quad (9)$$

То есть представленный алгоритм должен решать следующую задачу:

$$\text{Найти } \mathfrak{R}' \subseteq \mathfrak{R}, T \in \mathbb{T}^R(\mathfrak{C}_\mathbf{A}) : \text{ для } \mathfrak{R}' \text{ выполнено } 9 \quad (10)$$

4.3.1 Алгоритм построения частичного решения задачи восстановления одиночного наследования

Для каждой функции f такой, что $\exists A \in \mathfrak{C}_{\mathbf{A}}, i \in \mathbb{N} : VT_A^i = f$ построим множество $\mathfrak{f} \subseteq \mathfrak{C}_{\mathbf{A}} : \forall C \in \mathfrak{C}_{\mathbf{A}} : C \in \mathfrak{f} \iff VT_C^i = f$. Применяя к этому множеству утверждение 5, и учитывая то, что все ограничения, порожденные этим утверждением, принадлежат множеству $\mathfrak{R}$, получаем:

$$\forall T \in \mathbb{T}^R(\mathfrak{C}_{\mathbf{A}}) : isSolution(T, \mathfrak{R}) \Longrightarrow T[\mathbb{T}][\mathfrak{f}] \in \mathbb{T}(\mathfrak{f}), \tag{11}$$

то есть для того, чтобы дерево T являлось решением задачи (8), необходимо, чтобы его сужение на множество $\mathfrak{f}$ также являлось деревом. Пусть $\mathfrak{F}$ — множество всех таких множеств $\mathfrak{f}$. Введем обозначение:

$$\begin{aligned} &isPartialSolution \in \mathbb{T}(\mathfrak{C}_{\mathbf{A}}) \times 2^{2^{\mathfrak{C}_{\mathbf{A}}}} \to \{0, 1\}, \\ &\forall T \in \mathbb{T}(\mathfrak{C}_{\mathbf{A}}), \mathfrak{F} \in 2^{2^{\mathfrak{C}_{\mathbf{A}}}} : isPartialSolution(T, \mathfrak{F}) \iff \forall \mathfrak{f} \in \mathfrak{F} : T[\mathfrak{f}] \in \mathbb{T}(\mathfrak{f}). \end{aligned} \tag{12}$$

Из (11) и (12) следует:

$$\forall T \in \mathbb{T}^R(\mathfrak{C}_{\mathbf{A}}) : isSolution(T, \mathfrak{R}) \Longrightarrow isPartialSolution(T[\mathbb{T}], \mathfrak{F}), \tag{13}$$

то есть $isPartialSolution(T[\mathbb{T}], \mathfrak{F})$ является необходимым условием того, что дерево T является решением задачи (8). Будем говорить, что если выполнено $isPartialSolution(T[\mathbb{T}], \mathfrak{F})$, то дерево T является частичным решением задачи (8).

Пусть

$$\mathfrak{F}^{\cap} = \bigcup_{\mathfrak{F}' \subseteq \mathfrak{F}} \left\{ \bigcap_{\mathfrak{f} \in \mathfrak{F}'} \mathfrak{f} \right\}, \tag{14}$$

то есть $\mathfrak{F}^{\cap}$ — множество всевозможных пересечений множеств из $\mathfrak{F}$. Оценим количество элементов в множестве $\mathfrak{F}^{\cap}$. Для этого сначала докажем следующую лемму:

Лемма 1. *Для любого конечного множества X, в любом дереве $T \in \mathbb{T}(X)$ можно выбрать вершину $x \in X$ таким образом, что при удалении этой вершины и всех инцидентных ей ребер из дерева T, это дерево распадется на несколько деревьев, в каждом из которых будет не более $\left\lfloor \frac{|X|}{2} \right\rfloor$ вершин.*

Доказательство. Пусть для некоторого дерева T такую вершину выбрать нельзя, то есть для любой вершины $x \in X$ при ее удалении из дерева T, это

дерево распадается на несколько поддеревьев, в одном из которых содержится более $\left\lfloor \frac{|X|}{2} \right\rfloor$ вершин. Таких поддеревьев не может быть два, так как иначе в них суммарно содержалось бы более $|X|$ вершин. Обозначим множество вершин такого поддерева, соответствующего вершине x, через V_x. В множестве V_x содержится ровно одна вершина, инцидентная x в дереве T. Обозначим эту вершину x'. Тогда возможны два варианта:

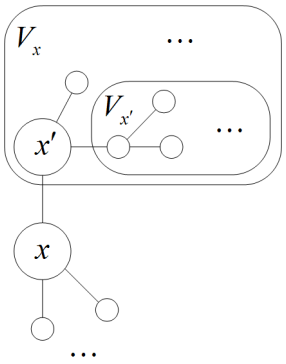

Рис. 23: Вариант $x \notin V_{x'}$.

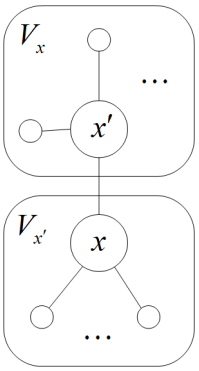

Рис. 24: Вариант $x \in V_{x'}$.

1. $x \notin V_{x'}$, рис. 23. В этом случае можно взять $x = x'$ и повторить приведенные выше рассуждения. В силу конечности множества $V_{x'}$, когда-нибудь будет верно, что:

2. $x \in V_{x'}$, рис. 24. В этом случае $V_{x'} \cap V_x = \emptyset$, и $|V_{x'}| + |V_x| > |X|$, что невозможно.

Полученное противоречие завершает доказательство леммы. □

С использованием леммы 1 оценим количество элементов в множестве $\mathfrak{F}^{\cap}$.

Лемма 2. *Если k — максимальное количество функций в таблицах виртуальных функций классов из некоторого множества классов $\mathfrak{C}$, $\mathfrak{F} \subseteq 2^{\mathfrak{C}}$, $n = |\mathfrak{C}|$, и $\exists T \in \mathbb{T}(\mathfrak{C}) : isPartialSolution(T, \mathfrak{F})$, то $|\mathfrak{F}^{\cap}| \leq 2n^{k+\log_2 3}$.*

Доказательство. Пусть $\psi(\mathfrak{F}) = |\mathfrak{F}^{\cap}|$, а $\mathbb{F}_k(\mathfrak{C}) \subseteq 2^{2^{\mathfrak{C}}}$ — множество всевозможных $\mathfrak{F} \subseteq 2^{\mathfrak{C}}$ таких, что каждый элемент из $\mathfrak{C}$ принадлежит не более чем k различным множествам из $\mathfrak{F}$, и $\exists T \in \mathbb{T}(\mathfrak{C}) : isPartialSolution(T, \mathfrak{F})$. Тогда определим

функцию ϕ следующим образом:

$$\phi(n,k) = \max_{\mathfrak{F}\in\mathbb{F}_k(\mathfrak{C})\wedge|\mathfrak{C}|=n} \psi(\mathfrak{F}). \tag{15}$$

Для функции ϕ выполнено:

$$\phi(n_1,k) + \phi(n_2,k) \leq \phi(n_1+n_2,k). \tag{16}$$

Действительно, для любых двух множеств

$$\begin{aligned} &\mathfrak{F}_1 \in \mathbb{F}_k(\mathfrak{C}_1) : |\mathfrak{C}_1| = n_1, \\ &\mathfrak{F}_2 \in \mathbb{F}_k(\mathfrak{C}_2) : |\mathfrak{C}_2| = n_2 \wedge \mathfrak{C}_1 \cap \mathfrak{C}_2 = \emptyset, \end{aligned} \tag{17}$$

и деревьев

$$\begin{aligned} &T_1 \in \mathbb{T}(\mathfrak{C}_1) : \mathit{isPartialSolution}(T_1, \mathfrak{F}_1), \\ &T_2 \in \mathbb{T}(\mathfrak{C}_2) : \mathit{isPartialSolution}(T_2, \mathfrak{F}_2), \end{aligned} \tag{18}$$

можно взять T — дерево, полученное из деревьев T_1 и T_2 добавлением ребра, соединяющего вершину A дерева T_1 с вершиной C дерева T_2, и $\mathfrak{F} = \mathfrak{F}_1 \cup \mathfrak{F}_2 \cup \{A, C\}$. В силу $\mathfrak{C}_1 \cap \mathfrak{C}_2 = \emptyset$, выполнено $\mathfrak{F}_1 \cap \mathfrak{F}_2 = \emptyset$, и потому единственным общим элементом множеств $\mathfrak{F}_1^\cap$ и $\mathfrak{F}_2^\cap$ может быть пустое множество. Так как $\mathfrak{F}_1^\cap \in \mathfrak{F}^\cap$ и $\mathfrak{F}_2^\cap \in \mathfrak{F}^\cap$, и возможное наличие общего элемента в $\mathfrak{F}_1^\cap$ и $\mathfrak{F}_2^\cap$ скомпенсировано наличием элемента $\{A, C\}$ в $\mathfrak{F}^\cap$, отсутствующего в $\mathfrak{F}_1^\cap$ и $\mathfrak{F}_2^\cap$, то будет справедливо $|\mathfrak{F}_1^\cap| + |\mathfrak{F}_2^\cap| \leq |\mathfrak{F}^\cap|$.

В силу 18 и 12, по построению для T будет выполнено $\mathit{isPartialSolution}(T, \mathfrak{F})$. Следовательно, $\mathfrak{F}^\cap \in \mathbb{F}_k(\mathfrak{C}_1 \cup \mathfrak{C}_2)$, и поэтому, согласно (15):

$$\psi(\mathfrak{F}_1) + \psi(\mathfrak{F}_2) \leq \psi(\mathfrak{F}) \leq \phi(n_1+n_2, k). \tag{19}$$

В силу того, что $\mathfrak{F}_1$ и $\mathfrak{F}_2$ были выбраны произвольно из множеств $\mathbb{F}_k(\mathfrak{C}_1)$ и $\mathbb{F}_k(\mathfrak{C}_2)$ соответственно, выполнено и неравенство (16). Из этого неравенства также следует то, что функция $\phi(n,k)$ является неубывающей по n.

Рассмотрим некоторые фиксированные множество $\mathfrak{F}$ и соответствующее ему дерево T такие, что $\mathit{isPartialSolution}(T, \mathfrak{F})$. В силу леммы 1, в этом дереве можно выделить вершину A так, что при удалении этой вершины и всех инцидентных ей ребер это дерево распадется на несколько деревьев T_i, каждое из которых будет состоять не более, чем из $\left\lfloor \frac{n}{2} \right\rfloor$ вершин. Пусть m — количество деревьев, на которые распалось дерево T, $\mathfrak{F}_i$ — подмножество $\mathfrak{F}$, элементы которого целиком лежат в множестве вершин T_i, а $\mathfrak{F}_A$ — подмножество $\mathfrak{F}$, элементы которого со-

держат A. В силу того, что элементы любых двух множеств $\mathfrak{F}_i$ не пересекаются, выполнено неравенство:

$$\psi(\mathfrak{F}) \leq \psi(\mathfrak{F}_A) \sum_{i=1}^{m} \psi(\mathfrak{F}_i). \tag{20}$$

Так как A принадлежит не более чем k множествам из $\mathfrak{F}$, всевозможных пересечений которых не более чем 2^k, то в силу (14) выполнено $\psi(\mathfrak{F}_A) = |\mathfrak{F}_A^{\cap}| \leq 2^k$, и неравенство (20) преобразуется в:

$$\psi(\mathfrak{F}) \leq 2^k \sum_{i=1}^{m} \psi(\mathfrak{F}_i). \tag{21}$$

Пусть n_i — количество вершин в i-м дереве. Тогда полученные m деревьев можно разделить на три группы так, как это показано на рисунке 25. При этом в каждой группе будет суммарно не более $\left\lfloor \frac{n}{2} \right\rfloor$ вершин.

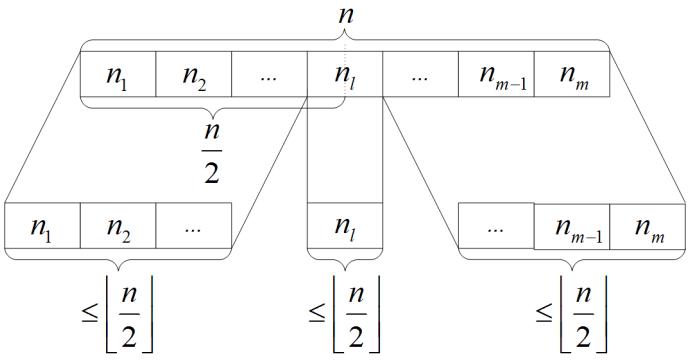

Рис. 25: Разбиение множества деревьев $\{T_i\}$ на три группы, в каждой из которых суммарно не более $\left\lfloor \frac{n}{2} \right\rfloor$ вершин.

Тогда неравенство (21) преобразуется в:

$$\begin{aligned}
\psi(\mathfrak{F}) &\leq 2^k \sum_{i=1}^{m} \psi(\mathfrak{F}_i) \\
&= 2^k \left(\sum_{i=1}^{l-1} \psi(\mathfrak{F}_i) + \psi(\mathfrak{F}_l) + \sum_{i=l+1}^{m} \psi(\mathfrak{F}_i) \right) \\
&\leq 2^k \left(\sum_{i=1}^{l-1} \phi(n_i, k) + \phi(n_l, k) + \sum_{i=l+1}^{m} \phi(n_i, k) \right) \\
&\{\text{согласно (16)}\} \\
&\leq 2^k \left(\phi(\sum_{i=1}^{l-1} n_i, k) + \phi(n_l, k) + \phi(\sum_{i=l+1}^{m} n_i, k) \right) \\
&\{\phi \text{ — неубывает по } n\} \\
&\leq 3 \cdot 2^k \phi(\left\lfloor \frac{n}{2} \right\rfloor, k).
\end{aligned} \tag{22}$$

Так как $\phi(1, k) = 2$, то согласно (22), для функции ϕ справедлива следующая оценка:

$$\phi(n, k) \leq 2 \cdot 2^{k \log_2 n} 3^{\log_2 n} = 2n^{k+\log_2 3}, \tag{23}$$

что завершает доказательство леммы. □

Полученное неравенство дает верхнюю оценку размера множества $\mathfrak{F}^{\cap}$. В результате проведения экспериментов было выяснено, что в задачах, встречающихся на практике, размер множества $\mathfrak{F}^{\cap}$ в большинстве случаев не превышает n^2.

Для множества $\mathfrak{F}^{\cap}$ выполнена следующая лемма.

Лемма 3. *Для произвольного множества классов* $\mathfrak{C}$ *выполнено* $\forall T \in \mathbb{T}(\mathfrak{C}), \mathfrak{F} \subseteq 2^{\mathfrak{C}} : isPartialSolution(T, \mathfrak{F}) \Longrightarrow isPartialSolution(T, \mathfrak{F}^{\cap})$.

Доказательство. Пусть утверждение леммы не верно. Тогда, согласно (12), $\exists\ \mathfrak{f}' :\ T[\mathfrak{f}'] \notin \mathbb{T}(\mathfrak{f}')$. Множество $\mathfrak{f}'$ не может быть пустым, потому как сужение дерева на пустое множество является деревом, и не может содержать только одну вершину, так как сужение дерева на множество из одной вершины также всегда является деревом. Следовательно, в множестве $\mathfrak{f}'$ содержится минимум две вершины.

В $T[\mathfrak{f}']$ не может быть циклов, так как их нет в T, и при сужении не добавляются новые ребра. Следовательно, $T[\mathfrak{f}']$ может не быть деревом только если оно не связно. Не нарушая общности, будем считать, что в $T[\mathfrak{f}']$ — две компоненты связности. Пусть вершина A лежит в первой компоненте, а B — во второй. По определению множества $\mathfrak{F}^{\cap}$, $\mathfrak{f}'$ было получено пересечением некоторого количества множеств из $\mathfrak{F}$, то есть:

$$\mathfrak{f}' = \bigcap_{\mathfrak{f} \in \mathfrak{F}'} \mathfrak{f}, \tag{24}$$

где $\mathfrak{F}' \subseteq \mathfrak{F}$. Значит, вершины A и B принадлежат всем множествам из $\mathfrak{F}'$. Так как по условию леммы $\forall \mathfrak{f} \in \mathfrak{F}' : T[\mathfrak{f}] \in \mathbb{T}(\mathfrak{f})$, то если бы в T между A и B существовал только один простой путь, то он принадлежал бы всем деревьям $T[\mathfrak{f}]$, где $\mathfrak{f} \in \mathfrak{F}'$, и, следовательно, все вершины этого простого пути принадлежали бы пересечению множеств из $\mathfrak{F}'$, и потому этот простой путь также принадлежал бы дереву $T[\mathfrak{f}']$, что противоречит предположению об отсутствии пути между вершинами A и B в $T[\mathfrak{f}']$. Следовательно, в T существует несколько простых путей между A и B. Но T — это дерево, и потому в нем существует единственный простой путь между вершинами A и B. Полученное противоречие завершает доказательство леммы. □

Рассмотрим множество $\mathfrak{F}^*$, определенное следующим образом:

$$\mathfrak{F}^* = \mathfrak{F}^{\cap} \cup \{\mathfrak{C}_{\mathbf{A}}\} \cup \bigcup_{C \in \mathfrak{C}_{\mathbf{A}}} \{\{C\}\} \setminus \emptyset. \tag{25}$$

В силу того, что $T[\mathfrak{C}_{\mathbf{A}}] = T$, и сужение любого дерева на множество из одного элемента также является деревом, выполнено:

$$\mathit{isPartialSolution}(T, \mathfrak{F}^{\cap}) \iff \mathit{isPartialSolution}(T, \mathfrak{F}^*). \tag{26}$$

По построению также выполнено, что любое пересечение множеств из $\mathfrak{F}^*$ содержится в $\mathfrak{F}^*$. Тогда из (13), (26) и леммы 3 следует:

$$\forall T \in \mathbb{T}^R(\mathfrak{C}_{\mathbf{A}}) : \mathit{isSolution}(T.\mathfrak{R}) \Longrightarrow \mathit{isPartialSolution}(T[\mathbb{T}], \mathfrak{F}^*), \tag{27}$$

то есть $\mathit{isPartialSolution}(T[\mathbb{T}], \mathfrak{F}^*)$ является необходимым условием того, что дерево T является решением задачи (8).

Рассмотрим отношение $\subset$ на множестве $\mathfrak{F}^*$. Это отношение является отно-

шением частичного порядка, и поэтому для него можно построить диаграмму Хассе.

Диаграмма Хассе[7)] — графическое представление частично упорядоченного множества $(X, \leq)$, в котором с каждым элементом X сопоставляется точка плоскости таким образом, что меньшая точка всегда располагается ниже большей точки. Две точки x и y в диаграмме Хассе соединены тогда и только тогда, когда $x \leq y$, и не существует такой точки z, что $x \leq z \leq y$.

Пусть диаграмма Хассе $\mathfrak{H} = (\mathfrak{F}^*, \subset)$ для множества $\mathfrak{F}^*$ построена. Каждой вершине $\mathfrak{H}$ соответствует множество из $\mathfrak{F}^*$. Будем обозначать вершины $\mathfrak{H}$ также, как и элементы множества $\mathfrak{F}^*$, то есть если $\mathfrak{f} \in \mathfrak{F}^*$, то будем говорить, что $\mathfrak{f}$ является вершиной $\mathfrak{H}$. В силу (27), если дерево T является решением задачи (8), то каждой вершине $\mathfrak{f}$ диаграммы Хассе $\mathfrak{H}$ соответствует поддерево $T[\mathfrak{f}]$ дерева T.

Диаграмму Хассе $\mathfrak{H}$ можно рассматривать как неориентированный граф. Будем обозначать множество ребер этого графа $\mathfrak{H}_E$.

Рассмотрим алгоритм построения дерева T такого, что выполнено $\mathit{isPartialSolution}(T, \mathfrak{F}^*)$, представленный на рис. 26.

1: $T_E = \emptyset$
2: **for all** $\mathfrak{f} \in \mathfrak{F}^*$ **do**
3: $\quad \mathfrak{D} \leftarrow \{\mathfrak{f}' \in \mathfrak{F}^* | \mathfrak{f}' \subset \mathfrak{f} \wedge \{\mathfrak{f}, \mathfrak{f}'\} \in \mathfrak{H}_E\}$
4: $\quad \mathfrak{K} \leftarrow \mathfrak{D}$
5: $\quad$ **while** $\exists \mathfrak{f}_1, \mathfrak{f}_2 \in \mathfrak{K} : \mathfrak{f}_1 \cap \mathfrak{f}_2 \neq \emptyset$ **do**
6: $\quad\quad \mathfrak{K} \leftarrow \mathfrak{K} \setminus \{\mathfrak{f}_1, \mathfrak{f}_2\} \cup \{\mathfrak{f}_1 \cup \mathfrak{f}_2\}$
7: $\quad$ **end while**
8: $\quad t \leftarrow \mathit{any}(\mathbb{T}(\mathfrak{K}))$
9: $\quad$ **for all** $\{\mathfrak{f}_1, \mathfrak{f}_2\} \in t_E$ **do**
10: $\quad\quad T_E \leftarrow T_E \cup \{\mathit{any}(\mathfrak{f}_1), \mathit{any}(\mathfrak{f}_2)\}$
11: $\quad$ **end for**
12: **end for**
13: **return** $T = (\mathfrak{C}_{\mathbf{A}}, T_E)$

Рис. 26: Алгоритм построения дерева T по множеству $\mathfrak{F}^*$.

Представленный на рис. 26 алгоритм использует функцию *any*, которая возвращает произвольный элемент данного множества. Для этого алгоритма справедлива следующая теорема.

[7)] англ. Hasse diagram.

Теорема 1. *Если* $\exists T' \in \mathbb{T}(\mathfrak{C}_{\mathbf{A}}) : isPartialSolution(T', \mathfrak{F}^*)$*, то алгоритм, представленный на рис. 26, строит дерево* $T \in \mathbb{T}(\mathfrak{C}_{\mathbf{A}}) : isPartialSolution(T, \mathfrak{F}^*)$ *независимо от того, какие значения были возвращены функциями any.*

Доказательство. Покажем, что в результате применения алгоритма будет выполнено $\forall \mathfrak{f} \in \mathfrak{F}^* : T[\mathfrak{f}] \in \mathbb{T}(\mathfrak{f})$. Так как $\mathfrak{C}_{\mathbf{A}} \in \mathfrak{F}^*$, то из этого будет следовать справедливость теоремы.

Будем рассматривать вершины диаграммы Хассе $\mathfrak{H}$ в порядке снизу вверх. Такой порядок просмотра гарантирует, что при рассмотрения очередного множества $\mathfrak{f} \in \mathfrak{F}^*$, все множества $\mathfrak{f}' \in \mathfrak{F}^* : \mathfrak{f}' \subset \mathfrak{f}$ будут уже рассмотрены. Пронумеруем элементы множества $\mathfrak{F}^*$ в соответствии с этим порядком: $\mathfrak{F}^* = \{\mathfrak{f}_1, \mathfrak{f}_2, \ldots, \mathfrak{f}_{|\mathfrak{F}^*|}\}$. В этом случае будет справедливо

$$\mathfrak{f}_i \subset \mathfrak{f}_j \Longrightarrow i < j. \tag{28}$$

Пусть

$$\forall \mathfrak{f}_i : 1 \leq i \leq n-1 : T[\mathfrak{f}_i] \in \mathbb{T}(\mathfrak{f}_i), \tag{29}$$

где $1 \leq n \leq |\mathfrak{F}^*|$. Покажем, что в этом случае будет выполнено и $T[\mathfrak{f}_n] \in \mathbb{T}(\mathfrak{f}_n)$.

В случае, если $\mathfrak{f}_n$ состоит из одной вершины, $T[\mathfrak{f}_n]$ является деревом.

Рассмотрим случай $|\mathfrak{f}_n| > 1$. В процессе выполнения алгоритма к $\mathfrak{f}_n$ было применено тело основного цикла. В результате было построено соответствующее $\mathfrak{f}_n$ множество $\mathfrak{D}$. Согласно определению множества $\mathfrak{D}$, в нем содержатся все вершины диаграммы Хассе $\mathfrak{H}$, находящиеся ниже $\mathfrak{f}_n$, и соединенные с $\mathfrak{f}_n$ ребром. Так как $|\mathfrak{f}_n| > 1$, то согласно (25), выполнено $\forall C \in \mathfrak{f}_n : \{C\} \in \mathfrak{F}^*$, и потому множество $\mathfrak{D}$ не пусто.

Рассмотрим множество $\mathfrak{K}$, построение которого происходит в строках 4-7 алгоритма. По построению выполнено:

$$\begin{aligned} &\forall \mathfrak{f}'_1, \mathfrak{f}'_2 \in \mathfrak{K} : \mathfrak{f}'_1 \neq \mathfrak{f}'_2 \Longrightarrow \mathfrak{f}'_1 \cap \mathfrak{f}'_2 = \emptyset \\ &\mathfrak{f}_n = \bigcup_{\mathfrak{f}' \in \mathfrak{K}} \mathfrak{f}' \end{aligned} \tag{30}$$

В строках 8-11 над множеством $\mathfrak{K}$ строится произвольное дерево, и в соответствии со структурой этого дерева добавляются ребра к дереву T. Заметим, что если выполнено $\forall \mathfrak{f}' \in \mathfrak{K} : T[\mathfrak{f}'] \in \mathbb{T}(\mathfrak{f}')$, то будет также выполнено и $T[\mathfrak{f}_n] \in \mathbb{T}(\mathfrak{f}_n)$. Действительно, если над каждым из множеств из $\mathfrak{K}$ уже построено дерево, то в силу (30), в результате добавления ребер в строках 8-11 алгоритма $T[\mathfrak{f}_n]$ будет связным графом с $\sum_{\mathfrak{f}' \in \mathfrak{K}} |\mathfrak{f}'|$ вершин и $\sum_{\mathfrak{f}' \in \mathfrak{K}} (|\mathfrak{f}'| - 1) + |\mathfrak{K}| - 1 = \sum_{\mathfrak{f}' \in \mathfrak{K}} |\mathfrak{f}'| - 1$

ребер, то есть деревом.

Покажем, что для всех $\mathfrak{f}' \in \mathfrak{K}$, $T[\mathfrak{f}']$ является деревом. Проведем доказательство для некоторого отдельного множества $\mathfrak{f}' \in \mathfrak{K}$. По построению, каждый элемент множества $\mathfrak{K}$ является объединением некоторого количества элементов множества $\mathfrak{D}$, то есть:

$$\mathfrak{f}' = \bigcup_{\mathfrak{f} \in \mathfrak{D}} \mathfrak{f}, \tag{31}$$

где $\mathfrak{D} \subseteq \mathfrak{D}$. Так как для всех $\mathfrak{f} \in \mathfrak{D}$ выполнено $\mathfrak{f} \subset \mathfrak{f}_n$, то в силу (28) и (29), также выполнено:

$$\forall \mathfrak{f} \in \mathfrak{D} : T[\mathfrak{f}_i] \in \mathbb{T}(\mathfrak{f}_i). \tag{32}$$

Из (31) и (32) следует, что $\forall \mathfrak{f} \in \mathfrak{D} : T[\mathfrak{f}_i] \in \mathbb{T}(\mathfrak{f}_i)$. То есть $T[\mathfrak{f}']$ является «объединением» некоторого количества деревьев, причем граф $T[\mathfrak{f}']$ связен. Покажем, что $T[\mathfrak{f}']$ — дерево.

Пусть это не так. Так как граф $T[\mathfrak{f}']$ связен, и не является деревом, то в нем должен присутствовать один или несколько простых циклов, то есть циклов, в которых вершины и ребра не повторяются.

Пусть отношение $\subseteq$ на множестве неориентированных графов над некоторым множеством X определено как $\forall G = (G_V, G_E), G' = (G'_V, G'_E) \in \mathbb{G}(X) : G \subseteq G' \iff G_V \subseteq G'_V \wedge G_E \subseteq G'_E$, то есть G является подграфом G'. Тогда зададим отношение $\subset$: $\forall G, G' \in \mathbb{G}(X) : G \subset G' \iff G \subseteq G' \wedge G \neq G'$. Аналогично вводятся операции $\cup$ и $\cap$ над множеством $\mathbb{G}(X)$.

Рассмотрим некоторый простой цикл $S \subseteq T[\mathfrak{f}']$. Тогда для любого $\mathfrak{f} \in \mathfrak{D}$ пересечение $S \cap T[\mathfrak{f}]$ пусто или состоит из одного или нескольких отрезков из S. Каждый такой отрезок может быть рассмотрен как простой путь в S, то есть путь, в котором вершины и ребра не повторяются. Пусть $\mathfrak{S}_\mathfrak{f}$ — множество таких отрезков для $\mathfrak{f}$. По построению для этого множества выполнено:

$$\forall_{\mathfrak{s} \in \mathfrak{S}_\mathfrak{f}, \mathfrak{s}' \subseteq S \cap T[\mathfrak{f}]} \mathfrak{s}' \text{ — отрезок} \Longrightarrow \mathfrak{s} \cap \mathfrak{s}' = \emptyset \vee \mathfrak{s}' \subseteq \mathfrak{s} \tag{33}$$

Будем считать, что каждый отрезок из $\mathfrak{S}_\mathfrak{f}$ помечен множеством $\mathfrak{f}$. Пусть $\mathfrak{S}$ является объединением всех таких множеств $\mathfrak{S}_\mathfrak{f}$, то есть:

$$\mathfrak{S} = \bigcup_{\mathfrak{f} \in \mathfrak{D}} \mathfrak{S}_\mathfrak{f}. \tag{34}$$

По построению для $\mathfrak{S}$ выполнено:

$$\bigcup_{\mathfrak{s}\in\mathfrak{S}} \mathfrak{s} = S \tag{35}$$

Так как каждый отрезок из $\mathfrak{S}$ помечен множеством из $\mathfrak{D}$, то пусть функция $\mathcal{I}$ для каждого отрезка возвращает соответствующее ему множество из $\mathfrak{D}$.

Заметим, что если $\exists \mathfrak{s}_i, \mathfrak{s}_j \in \mathfrak{S} : \mathcal{I}(\mathfrak{s}_i) = \mathcal{I}(\mathfrak{s}_j)$, то между некоторыми двумя вершинами отрезков $\mathfrak{s}_i$ и $\mathfrak{s}_j$ существует простой путь $\mathfrak{p}_{ij}$ в $T[\mathcal{I}(\mathfrak{s}_i)]$. Соответствующий пример представлен на рис. 27. Этот путь не может целиком лежать в S, так как тогда было бы нарушено (33) — пересечение $T[\mathcal{I}(\mathfrak{s}_i)]$ с S состояло бы из одного или нескольких отрезков, одним из которых являлся бы $\mathfrak{s}_i \cup \mathfrak{p}_{ij} \cup \mathfrak{s}_j$, и потому отрезки $\mathfrak{s}_i$ и $\mathfrak{s}_j$ не попали бы в множество $\mathfrak{S}_{\mathcal{I}(\mathfrak{s}_i)}$.

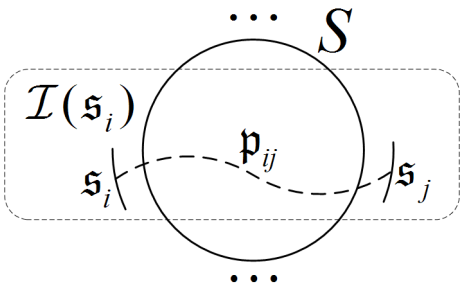

Рис. 27: Пример существования в простом цикле нескольких отрезков, которым соответствует одно и то же множество.

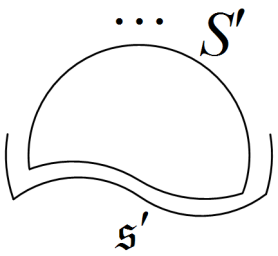

Рис. 28: Результат обработки случая, представленного на рис. 27.

Таким образом, между некоторыми двумя вершинами, лежащими на отрезках $\mathfrak{s}_i$ и $\mathfrak{s}_j$, существуют два простых пути — один лежащий в S, другой — лежащий в $\mathfrak{p}_{ij}$, причем ни один из этих простых путей не содержится в другом. Но тогда это означает, что в графе, построенном путем объединения этих двух простых путей, существует простой цикл. Обозначим этот цикл $S^{(1)}$. Пусть $\mathfrak{s}^{(1)} = (\mathfrak{s}_i \cup \mathfrak{p}_{ij} \cup \mathfrak{s}_j) \cap S^{(1)}$, тогда простой цикл $S^{(1)}$ можно покрыть множеством отрезков:

$$\mathfrak{S}^{(1)} \subseteq \mathfrak{S} \cup \{\mathfrak{s}^{(1)}\} \setminus \{\mathfrak{s}_i, \mathfrak{s}_j\} \tag{36}$$

Будем считать, что $\mathfrak{S}^{(1)}$ выбрано так, что все отрезки из $\mathfrak{S}^{(1)}$ имеют непустое пересечение с $S^{(1)}$. Так как в процессе построения структура цикла была изменена, то для некоторых отрезков из $\mathfrak{S}^{(1)}$ может больше не выполняться

(33), примененное к циклу $S^{(1)}$, то есть не обязательно выполняется:

$$\forall_{\mathfrak{s}\in\mathfrak{S}^{(1)},\mathfrak{s}'\subseteq S^{(1)}\cap T[\mathcal{I}(\mathfrak{s})]}\ \mathfrak{s}' \text{ — отрезок} \Longrightarrow \mathfrak{s}\cap\mathfrak{s}'=\emptyset\vee\mathfrak{s}'\subseteq\mathfrak{s} \tag{37}$$

Заменим каждый отрезок $\mathfrak{s}\in\mathfrak{S}^{(1)}$ на $\mathfrak{s}_+$ такой, что для $\mathfrak{s}_+$ выполняется (37), и при этом $\mathfrak{s}\cap\mathfrak{s}_+\neq\emptyset$. Полученное таким образом новое множество отрезков обозначим $\mathfrak{S}^{(1)}_+$. Будем считать, что если в результате замены в $\mathfrak{S}^{(1)}_+$ были добавлены два равных отрезка, которым соответствует одно и то же множество из $\mathfrak{D}$, то такие отрезки представлены в множестве $\mathfrak{S}^{(1)}_+$ только один раз. Так как множество $\mathfrak{S}^{(1)}$ покрывало цикл $S^{(1)}$, то и построенное множество $\mathfrak{S}^{(1)}_+$ будет его покрывать.

Заметим, что в силу (36), выполнено $|\mathfrak{S}^{(1)}|<|\mathfrak{S}|$, и потому также выполнено $|\mathfrak{S}^{(1)}_+|<|\mathfrak{S}|$. Так как количество отрезков в множестве $\mathfrak{S}$ конечно, то повторяя несколько раз приведенные выше рассуждения, на k-м шаге будет выполнено $\nexists\mathfrak{s}_i,\mathfrak{s}_j\in\mathfrak{S}^{(k)}_+ : \mathcal{I}(\mathfrak{s}_i)=\mathcal{I}(\mathfrak{s}_j)$.

Выделим минимальное подмножество отрезков $\mathfrak{S}_{\min}\subseteq\mathfrak{S}^{(k)}_+$, покрывающих весь простой цикл $S^{(k)}$, то есть:

$$\begin{gathered}\bigcup_{\mathfrak{s}\in\mathfrak{S}_{\min}}\mathfrak{s}=S^{(k)}\\ \forall\mathfrak{s}'\in\mathfrak{S}_{\min}\ \bigcup_{\mathfrak{s}\in\mathfrak{S}_{\min}\setminus\{\mathfrak{s}'\}}\mathfrak{s}\neq S^{(k)}\end{gathered} \tag{38}$$

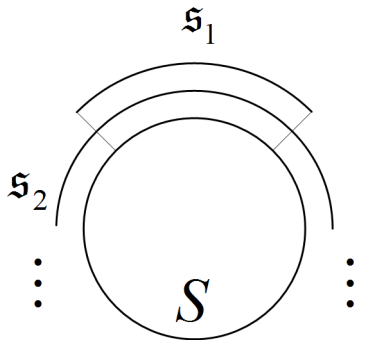

Рис. 29: Пример отрезка, лежащего целиком внутри другого отрезка.

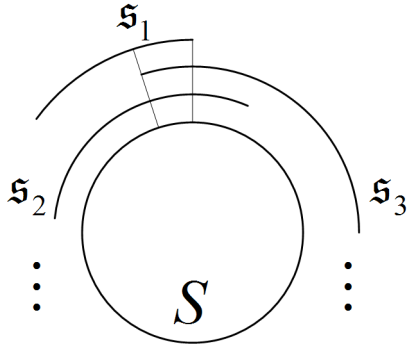

Рис. 30: Пример пересечения отрезков кратности три.

Для этого подмножества выполнено:

- В $\mathfrak{S}_{\min}$ нет отрезков, целиком лежащих внутри других. Если $\exists\mathfrak{s}_1,\mathfrak{s}_2\in\mathfrak{S}_{\min}:\mathfrak{s}_1\subseteq\mathfrak{s}_2$, то $\mathfrak{S}'_{\min}=\mathfrak{S}_{\min}\setminus\{\mathfrak{s}_1\}$ также покрывает весь простой цикл

$S^{(k)}$, что противоречит определению множества $\mathfrak{S}_{\min}$. Пример для данного случая представлен на рис. 29.

- Среди пересечений отрезков из $\mathfrak{S}_{\min}$ нет пересечений кратности три. На рис. 30 представлен пример пересечения кратности три. Как видно, отрезок $\mathfrak{s}_1$ не может содержаться в $\mathfrak{S}_{\min}$, так как отрезки $\mathfrak{s}_2$ и $\mathfrak{s}_3$ и так покрывают соответствующие $\mathfrak{s}_1$ вершины и ребра. В общем случае, если существует пересечение кратности три, то среди отрезков, его образующих, существует отрезок $\mathfrak{s}_1$ такой, что $\mathfrak{s}_1 \subseteq \mathfrak{s}_2 \cup \mathfrak{s}_3$, где $\mathfrak{s}_2$ и $\mathfrak{s}_3$ — два оставшихся отрезка в этом пересечении. Тогда $\mathfrak{S}'_{\min} = \mathfrak{S}_{\min} \setminus \{\mathfrak{s}_1\}$ также покрывает весь простой цикл $S^{(k)}$, что противоречит определению множества $\mathfrak{S}_{\min}$.

Таким образом, среди пересечений отрезков из $\mathfrak{S}_{\min}$ существуют только пересечения кратности два, и простой цикл $S^{(k)}$ можно представить как последовательность отрезков $S^{(k)} = (\mathfrak{s}_1, \ldots, \mathfrak{s}_n)$, в которой только соседние отрезки имеют непустое пересечение. Выбрав по точке на каждом пересечении соседних отрезков, получим последовательность точек $(C_1, \ldots, C_n)$ такую, что любые две соседние точки C_i и C_{i+1} лежат в отрезке $\mathfrak{s}_i$, и потому лежат и в $\mathcal{I}(\mathfrak{s}_i)$.

По условию теоремы, $\exists T' \in \mathbb{T}(\mathfrak{C}_{\mathbf{A}}) : \mathit{isPartialSolution}(T', \mathfrak{F}^*)$, то есть сужение дерева T' на любое множество из $\mathfrak{F}^*$ является деревом. Тогда сужение T' на любое из множеств $\mathcal{I}(\mathfrak{s}_i)$ также является деревом, и потому в каждом из таких деревьев существует единственный простой путь между соответствующими вершинами C_i и C_{i+1}. Обозначим этот простой путь $\mathfrak{p}_i$. По построению эти простые пути образуют цикл $S_{T'} = (\mathfrak{p}_1, \ldots, \mathfrak{p}_n)$. Так как T' является деревом, то этот цикл не может быть простым, и каждая вершина из этого цикла принадлежит как минимум двум простым путям. Это означает, что в этом цикле существуют два простых пути $\mathfrak{p}_i$ и $\mathfrak{p}_{i+1}$, образующих «поворот», как это изображено на рис. 31.

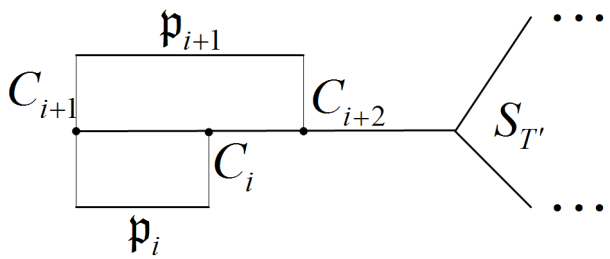

Рис. 31: Пример «поворота», образованного двумя простыми путями в цикле, не являющимся простым.

Но тогда один из простых путей $\mathfrak{p}_i$ и $\mathfrak{p}_{i+1}$ содержится в другом. Не нарушая

общности, будем считать, что $\mathfrak{p}_i$ содержится в $\mathfrak{p}_{i+1}$. Это означает, что C_i содержится в $\mathfrak{p}_{i+1}$, и так как по построению множество вершин простого пути $\mathfrak{p}_{i+1}$ является подмножеством $\mathcal{I}(\mathfrak{s}_{i+1})$, то $C_i \in \mathcal{I}(\mathfrak{s}_{i+1})$.

Вернемся к дереву T. По построению для $\mathcal{I}(\mathfrak{s}_{i+1})$ выполнено $C_{i+1}, C_{i+2} \in \mathcal{I}(\mathfrak{s}_{i+1})$. $T[\mathcal{I}(\mathfrak{s}_{i+1})]$ является деревом, и поэтому в нем существует единственный простой путь из C_i в C_{i+1}, и единственный простой путь из C_{i+1} в C_{i+2}. Эти простые пути должны принадлежать циклу $S^{(k)}$, и потому в силу (37) они также должны принадлежать отрезку $\mathfrak{s}_{i+1}$. Но тогда среди отрезков из $\mathfrak{S}_{\min}$ существует пересечение кратности три — $C_i \in \mathfrak{s}_{i+1}$, $C_i \in \mathfrak{s}_i$, $C_i \in \mathfrak{s}_{i-1}$, что противоречит определению множества $\mathfrak{S}_{\min}$. Следовательно, изначальное предположение о том, что в $T[\mathfrak{f}']$ существует цикл, неверно, и $T[\mathfrak{f}']$ является деревом.

Полученное противоречие доказывает, что если выполнено (29), то выполнено и $T[\mathfrak{f}_n] \in \mathbb{T}(\mathfrak{f}_n)$. Тогда применение метода математической индукции для индукционного перехода (29) и для базы индукции $n = 1$, дает, что (29) верно для всех $1 \leq n \leq |\mathfrak{F}^*|$, что завершает доказательство теоремы. □

Следствием доказанной теоремы является то, что любое решение задачи (8) может быть получено с помощью алгоритма 26, а также то, что если в результате применение алгоритма 26 не было построено дерево, то невозможно одновременно удовлетворить все ограничения вида $\mathcal{R}_5$, порожденные утверждениями 5. В этом случае путем автоматического анализа имеющейся информации затруднительно выяснить, какое из ограничений вида $\mathcal{R}_5$ следует удалить из множества $\mathfrak{R}$. Поэтому будем считать, что в этом случае удаление вызывающего конфликт ограничения выполняется вручную. Как было отмечено в комментарии к утверждению 5, такая ситуация маловероятна. В дальнейшем будем считать, что все ограничения, порожденные утверждением 5, выполнены. Для этого случая модифицируем алгоритм 26 так, чтобы он строил только решения задачи (10).

4.3.2 Алгоритм построения решения задачи восстановления одиночного наследования

Рассмотрим диаграмму Хассе $\mathfrak{H}$. Алгоритм 26 для каждой вершины $\mathfrak{f}$ этой диаграммы строит множество $\mathfrak{D}$. Будем называть элементы этого множества дочерними вершинами для $\mathfrak{f}$ в диаграмме Хассе $\mathfrak{H}$. По диаграмме Хассе $\mathfrak{H}$ построим граф $\mathfrak{G}$, вершинами которого также являются множества $\mathfrak{f} \subset \mathfrak{C}_{\mathbf{A}}$, но

разделенные на два типа — *any* и *fixed*. Алгоритм построения этого графа представлен на рис. 32. Этот алгоритм использует следующие функции:

children — возвращает множество дочерних вершин $\mathfrak{D}$ для данного множества $\mathfrak{f}$. Построение множества $\mathfrak{D}$ происходит также, как и в алгоритме 26;

merge — возвращает множество $\mathfrak{K}$ для данного множества $\mathfrak{D}$. Построение множества $\mathfrak{K}$ происходит также, как и в алгоритме 26.

1: $\mathfrak{G} \leftarrow (\emptyset, \emptyset)$
2: **for all** $\mathfrak{f}_1 \in \mathfrak{F}^*$ **do**
3: $\quad\mathfrak{D} \leftarrow \mathit{children}(\mathfrak{f}_1)$
4: $\quad\mathfrak{K} \leftarrow \mathit{merge}(\mathfrak{D})$
5: $\quad\mathit{type}(\mathfrak{f}_1) \leftarrow \mathit{any}$
6: $\quad\mathfrak{G}_V \leftarrow \mathfrak{G}_V \cup \{\mathfrak{f}_1\}$
7: $\quad$**for all** $\mathfrak{f}_2 \in \mathfrak{K}$ **do**
8: $\quad\quad\mathfrak{G}_E \leftarrow \mathfrak{G}_E \cup \{\{\mathfrak{f}_1, \mathfrak{f}_2\}\}$
9: $\quad\quad$**if** $\mathfrak{f}_2 \notin \mathfrak{D}$ **then**
10: $\quad\quad\quad$ {Это значит, что $\mathfrak{f}_2$ не является вершиной диаграммы Хассе $\mathfrak{H}$, и потому $\mathfrak{f}_2$ еще не было добавлено в $\mathfrak{G}_V$}
11: $\quad\quad\quad\mathit{type}(\mathfrak{f}_2) \leftarrow \mathit{fixed}$
12: $\quad\quad\quad\mathfrak{G}_V \leftarrow \mathfrak{G}_V \cup \{\mathfrak{f}_2\}$
13: $\quad\quad\quad$**for all** $\mathfrak{f}_3 \in \mathfrak{D} : \mathfrak{f}_3 \subseteq \mathfrak{f}_2$ **do**
14: $\quad\quad\quad\quad\mathfrak{G}_E \leftarrow \mathfrak{G}_E \cup \{\{\mathfrak{f}_2, \mathfrak{f}_3\}\}$
15: $\quad\quad\quad$**end for**
16: $\quad\quad$**end if**
17: $\quad$**end for**
18: **end for**

Рис. 32: Алгоритм построения графа $\mathfrak{G}$ по диаграмме Хассе $\mathfrak{H}$.

Пример работы алгоритма 32 приведен на рис. 33 и рис. 34. На рис. 33 представлена часть исходной диаграммы Хассе $\mathfrak{H}$, а на рис. 34 — соответствующая ей часть графа $\mathfrak{G}$. Так как вершины $\mathfrak{f}_1$ и $\mathfrak{f}_2$ диаграммы Хассе $\mathfrak{H}$ лежат в одном множестве $\mathfrak{f}'_1 \in \mathfrak{K}$, то для них была добавлена *fixed*-вершина $\mathfrak{f}'_1$. Для вершины же $\mathfrak{f}_3$ выполнено $\mathfrak{f}_3 = \mathfrak{f}'_2$, и потому $\mathfrak{f}'_2 \in \mathfrak{D}$, и для $\mathfrak{f}_3$ дополнительных *fixed*-вершин добавлено не было.

По построению, граф $\mathfrak{G}$ является диаграммой Хассе для множества, подмножеством которого является $\mathfrak{F}^*$, и потому на $\mathfrak{G}$ определен порядок расположения вершин на плоскости, и аналогично $\mathfrak{H}$ для каждой вершины определено множество дочерних вершин. Также по построению каждой *any*-вершине в $\mathfrak{G}$ соответствует вершина в $\mathfrak{H}$, и наоборот. Тогда, согласно теореме 1, выбирая для каждой *any*-вершины дерево на множестве ее дочерних вершин, и добавляя по-

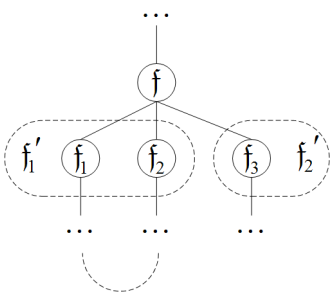

Рис. 33: Пример части диаграммы Хассе, обрабатываемой алгоритмом 32.

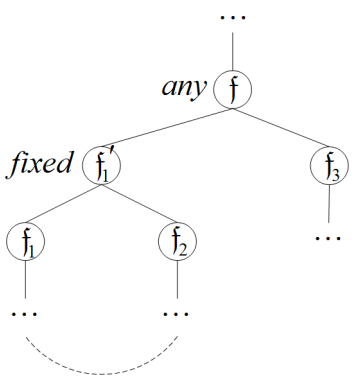

Рис. 34: Результат обработки алгоритмом 32 части диаграммы Хассе, представленной на рис. 33.

сле этого ребра к дереву $T \in \mathbb{T}(\mathfrak{C}_\mathbf{A})$ так, как в алгоритме 26, можно получить произвольное дерево, являющееся частичным решением задачи 8.

С помощью графа $\mathfrak{G}$ построим полное решение задачи 10.

Заметим, что в графе $\mathfrak{G}$ нет циклов, проходящих только по *any*-вершинам — для существования цикла, проходящего через вершину $\mathfrak{f}$, и две ее дочерние вершины $\mathfrak{f}_1$ и $\mathfrak{f}_2$ необходимо, чтобы в $\mathfrak{H}$ существовал путь между вершинами $\mathfrak{f}_1$ и $\mathfrak{f}_2$. Но существование такого пути означает, что пересечение множеств $\mathfrak{f}_1$ и $\mathfrak{f}_2$ не пусто, и потому на этапе построения графа $\mathfrak{G}$ две эти вершины были отнесены к одному множеству $\mathfrak{f}'_1$, как это показано на рис. 33. Следовательно, для этих вершин была добавлена *fixed*-вершина, как это показано на рис. 34, и потому они не могут являться дочерними для $\mathfrak{f}$. Это означает, что невозможно появление цикла, проходящего только по *any*-вершинам, и содержащего вершины $\mathfrak{f}$, $\mathfrak{f}_1$ и $\mathfrak{f}_2$. Также для каждой *any*-вершины $\mathfrak{f}$ графа $\mathfrak{G}$ выполнено:

$$|children(\mathfrak{f})| \leq |\mathfrak{C}_\mathbf{A}|, \tag{39}$$

то есть $\mathfrak{f}$ имеет не более $|\mathfrak{C}_\mathbf{A}|$ дочерних вершин. Если бы *any*-вершина $\mathfrak{f}$ имела более $|\mathfrak{C}_\mathbf{A}|$ дочерних вершин, то пересечение некоторых двух из них было бы не пусто, что, как было только что доказано, невозможно.

Определим несколько функций. Пусть:

type — возвращает тип данной вершины, то есть значение из множества $\{\mathit{any}, \mathit{fixed}\}$;

class — если для данной вершины $\mathfrak{f}$ выполнено $|\mathfrak{f}| = 1$, то есть $\mathfrak{f} = \{C\}$, то возвращает C, иначе возвращает $\emptyset$;

adjacent — для данной вершины возвращает множество смежных с ней вершин в графе $\mathfrak{G}$;

parents — для данной вершины $\mathfrak{f}$ возвращает множество вершин, для которых $\mathfrak{f}$ является дочерней в $\mathfrak{G}$, то есть $\mathit{parents}(\mathfrak{f}) = \mathit{adjacent}(\mathfrak{f}) \setminus \mathit{children}(\mathfrak{f})$. По построению, если $\mathit{type}(\mathfrak{f}) = \mathit{fixed}$, то в этом множестве содержится ровно одна вершина.

Всевозможным парам вершин $(\mathfrak{f}_1, \mathfrak{f}_2)$ графа $\mathfrak{G}$ поставим в соответствие множества достижимых из $\mathfrak{f}_1$ по направлению на $\mathfrak{f}_2$ классов $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$. Если две вершины $\mathfrak{f}_1$ и $\mathfrak{f}_2$ инцидентны в графе $\mathfrak{G}$, то $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ для них определяется согласно следующим правилам:

$$\mathit{type}(\mathfrak{f}_1) = \mathit{type}(\mathfrak{f}_2) = \mathit{any} \wedge \mathit{class}(\mathfrak{f}_2) \neq \emptyset \Longrightarrow \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \mathit{class}(\mathfrak{f}_2), \tag{40}$$

Иначе, если выполнено:

$$\begin{aligned} &(\mathit{type}(\mathfrak{f}_1) = \mathit{type}(\mathfrak{f}_2) = \mathit{any}) \vee \\ &(\exists\{\mathfrak{f}'_1, \mathfrak{f}'_2\} = \{\mathfrak{f}_1, \mathfrak{f}_2\} : \mathit{type}(\mathfrak{f}'_1) = \mathit{any} \wedge \mathit{type}(\mathfrak{f}'_2) = \mathit{fixed} \wedge \mathit{parents}(\mathfrak{f}'_2) = \{\mathfrak{f}'_1\}), \end{aligned} \tag{41}$$

то есть $\mathfrak{f}_1$ и $\mathfrak{f}_2$ являются *any*-вершинами, или одна из вершин $\mathfrak{f}_1$ и $\mathfrak{f}_2$ является *fixed*-вершиной, а другая — *any*-вершиной, причем *fixed*-вершина является дочерней для *any*-вершины, то:

$$\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \bigcup_{\mathfrak{f} \in \mathit{adjacent}(\mathfrak{f}_2) \setminus \mathfrak{f}_1} \mathfrak{C}_{(\mathfrak{f}_2,\mathfrak{f})}. \tag{42}$$

В случае, если не выполнено ни (40), ни (41), множество $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ пусто.

Так как для двух вершин $\mathfrak{f}_1$ и $\mathfrak{f}_2$ таких, что $\mathfrak{f}_1$ является *fixed*-вершиной, $\mathfrak{f}_2$ — *any*-вершиной, и $\mathfrak{f}_2$ является дочерней для $\mathfrak{f}_1$, множество $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ пусто, то в силу отсутствия циклов в графе $\mathfrak{G}$, проходящих только по *any*-вершинам, рекурсивное определение (42) множества $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ корректно.

Изменим множества $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ для таких пар вершин, что $\mathfrak{f}_1$ является *any*-вершиной, $\mathfrak{f}_2$ — *fixed*-вершиной, и между $\mathfrak{f}_2$ и $\mathfrak{f}_1$ в $\mathfrak{G}$ существует путь, содержащий только одну *fixed*-вершину $\mathfrak{f}_2$, и такой, что каждая вершина в этом пути является дочерней для предыдущей. Это означает, что $\mathfrak{f}_2$ является «первой»

fixed-вершиной, находящейся над $\mathfrak{f}_1$, и связанной с ней. В этом случае будем считать, что

$$\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} = \mathfrak{C}_{\mathbf{A}} \setminus \left(class(\mathfrak{f}_1) \cup \bigcup_{\mathfrak{f} \in \mathfrak{C}_{\mathbf{A}}} \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f})} \right) \tag{43}$$

Тогда для каждой вершины $\mathfrak{f}$, по построению выполнено:

$$\begin{gathered} \bigcup_{\mathfrak{f}' \in \mathfrak{G}_V} \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}')} = \mathfrak{C}_{\mathbf{A}} \setminus class(\mathfrak{f}), \\ \forall_{\mathfrak{f}_1,\mathfrak{f}_2 \in \mathfrak{G}_V} \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}_1)} \cap \mathfrak{C}_{(\mathfrak{f},\mathfrak{f}_2)} = \emptyset. \end{gathered} \tag{44}$$

Будем строить дерево, являющееся решением задачи (10) как дерево $T \in \mathbb{T}(\mathfrak{C}_{\mathbf{A}})$, для каждого ребра которого определено направление наследования. Для этого:

1. Для каждых двух дочерних вершин $\mathfrak{f}_1$ и $\mathfrak{f}_2$ каждой *any*-вершины $\mathfrak{f}$ построим множество $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ — множество классов, которые должен унаследовать корень $T[\mathfrak{f}_1]$ в случае, если существует класс в $\mathfrak{f}_2$, являющийся предком этого корня, и множество $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ — множество классов, которые корень $T[\mathfrak{f}_1]$ не должен унаследовать в том же случае.

2. Для каждой *any*-вершины $\mathfrak{f}$ построим корневое дерево $T_{\mathfrak{f}} \in \mathbb{T}(children(\mathfrak{f}))$ на множестве дочерних для нее вершин в $\mathfrak{G}$.

Согласно определению, для множеств $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ и $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ выполнено:

$$\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}, \mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \subseteq \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \tag{45}$$

Для построения корневых деревьев $T_{\mathfrak{f}}$ в каждой *any*-вершине будем восстанавливать отношение $\lhd$ на множестве дочерних для нее вершин. Отношение $\lhd$ для вершины $\mathfrak{f}$ будем задавать множеством ограничений $\mathfrak{R}_{\mathfrak{f}}$. Будем считать, что в процессе восстановления это отношение задано ограничениями следующего вида:

$$\begin{aligned} &\mathcal{R} = B < D, \quad \text{или} \\ &\mathcal{R} = B \lhd D, \quad \text{или} \\ &\mathcal{R} = A \sim C. \end{aligned} \tag{46}$$

Ограничения вида $\mathcal{R} = \neg(A \sim C)$ можно заменить двумя ограничениями $\mathcal{R}_1 = A < C$ и $\mathcal{R}_2 = A > C$ и рассматривать как ограничения вида (46).

Все ограничения из $\mathfrak{R}$, за исключением ограничений вида $\mathcal{R}_5$, полученных применением утверждения 5, представимы в виде 46. Согласно предположе-

нию, высказанному в конце главы 4.3.1, все ограничения вида $\mathcal{R}_5$ выполнимы, и согласно теореме 1, любое дерево, построенное с помощью алгоритма 26, удовлетворяет этим ограничениям. Тогда соблюдение ограничений вида $\mathcal{R}_5$ обеспечивается структурой построенного графа $\mathfrak{G}$ и выбранным способом построения корневого дерева, являющегося решением задачи (10).

Пусть $\mathfrak{R}_s \subseteq \mathfrak{R}$ — множество всех ограничений из $\mathfrak{R}$, представимых в виде (46). Тогда в $\mathfrak{R}_s$ содержатся все ограничения из $\mathfrak{R}$, кроме ограничений вида $\mathcal{R}_5$. В начале главы 4.3 было отмечено, что некоторые из ограничений из $\mathfrak{R}$ могут оказаться ложными, и вызвать конфликты в ходе восстановления иерархии. Будем строить иерархию, рассматривая ограничения из $\mathfrak{R}_s$ в порядке увеличения вероятности получения конфликта. Ограничения вида $\mathcal{R}_8$, определенные в утверждении 8, всегда выполняются, и являются наиболее информативными. В результате проведения экспериментов было выяснено, что во многих случаях только ограничений вида $\mathcal{R}_8$ и $\mathcal{R}_5$ достаточно, чтобы полностью восстановить иерархию классов. Поэтому будем рассматривать ограничения начиная с ограничений вида $\mathcal{R}_8$. В случае, если на очередном шаге обнаружится конфликт в структуре построенных множеств, в результате чего построение корневого дерева, являющегося решением, станет невозможным, будем возвращаться к предыдущему шагу, и продолжить обработку, добавив ограничение, вызвавшее конфликт, в изначально пустое множество конфликтных ограничений $\mathfrak{R}_\times$. Тогда после завершения обработки всех ограничений из $\mathfrak{R}_s$, множество ограничений $\mathfrak{R} \setminus \mathfrak{R}_\times$ будет соответствовать множеству ограничений $\mathfrak{R}'$ из определения задачи (10).

1: **let** $\mathcal{R} \in \mathfrak{R}_s, A, C \in \mathfrak{C}_\mathbf{A} : A \neq C \wedge \mathcal{R} = A \sim C$
2: **let** $\mathfrak{f}_1 : \mathit{class}(\mathfrak{f}_1) = \{C\}$
3: **while** $\mathit{class}(\mathfrak{f}_1) \neq \{A\}$ **do**
4: $\{\mathfrak{f}_2$ существует и единственно в силу (44)$\}$
5: **let** $\mathfrak{f}_2 \in \mathit{adjacent}(\mathfrak{f}_1) : A \in \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$
6: **if** $\mathit{type}(\mathfrak{f}_2) = \mathit{any} \wedge \mathfrak{f}_1 \in \mathit{children}(\mathfrak{f}_2) \wedge \exists \mathfrak{f}_3 \in \mathit{children}(\mathfrak{f}_2) : B \in \mathfrak{C}_{(\mathfrak{f}_2,\mathfrak{f}_3)}$ **then**
7: $\mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \leftarrow \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \cup \{C\}$
8: $\mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \leftarrow \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \cup \{A\}$
9: **else if** $\mathit{type}(\mathfrak{f}_2) = \mathit{fixed} \wedge \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}_1)$ **then**
10: **break**
11: **end if**
12: $\mathfrak{f}_1 \leftarrow \mathfrak{f}_2$
13: **end while**

Рис. 35: Алгоритм обработки ограничения вида $A \sim C$.

1: **let** $\mathcal{R} \in \mathfrak{R}_s, B, D \in \mathfrak{C}_{\mathbf{A}} : B \neq C \wedge \mathcal{R} = B < D$
2: **let** $\mathfrak{f}_1 : \mathit{class}(\mathfrak{f}_1) = \{D\}$
3: **while** $\mathit{class}(\mathfrak{f}_1) \neq \{B\}$ **do**
4: $\quad$ {$\mathfrak{f}_2$ существует и единственно в силу (44)}
5: $\quad$ **let** $\mathfrak{f}_2 \in \mathit{adjacent}(\mathfrak{f}_1) : B \in \mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$
6: $\quad$ **if** $\mathit{type}(\mathfrak{f}_2) = \mathit{any} \wedge \mathfrak{f}_1 \in \mathit{children}(\mathfrak{f}_2) \wedge \exists \mathfrak{f}_3 \in \mathit{children}(\mathfrak{f}_2) : B \in \mathfrak{C}_{(\mathfrak{f}_2,\mathfrak{f}_3)}$ **then**
7: $\quad\quad \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \leftarrow \mathfrak{N}^{\mathfrak{f}_2}_{(\mathfrak{f}_1,\mathfrak{f}_3)} \cup \{B\}$
8: $\quad\quad \mathfrak{P}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \leftarrow \mathfrak{P}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)} \cup \{D\}$
9: $\quad$ **else if** $\mathit{type}(\mathfrak{f}_2) = \mathit{fixed} \wedge \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}_1)$ **then**
10: $\quad\quad$ **break**
11: $\quad$ **end if**
12: $\quad \mathfrak{f}_1 \leftarrow \mathfrak{f}_2$
13: **end while**

Рис. 36: Алгоритм обработки ограничения вида $B < D$.

Будем считать, что изначально множества ограничений $\mathfrak{R}_\mathfrak{f}$, а также множества $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ и $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ пусты. Сначала рассмотрим все ограничения $\mathcal{R}_8$, затем — все прочие ограничения вида $B \lhd D$. Так как отношение $\lhd$ можно выразить через отношения $<$ и $\sim$, то приведем алгоритмы модификации множеств $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$, $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ и $\mathfrak{R}_\mathfrak{f}$ только для ограничений вида $B < D$ и $A \sim C$. Алгоритм отработки отдельного ограничения вида $A \sim C$ представлен на рис. 35, а ограничения вида $B < D$ — на рис. 36.

После работы этих алгоритмов из измененных множеств $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_2,\mathfrak{f}_1)}$ и $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ можно извлечь дополнительную информацию об ограничениях из $\mathfrak{R}_\mathfrak{f}$ с использованием правил поддержания непротиворечивости для множеств $\mathfrak{R}_\mathfrak{f}$. Из определения множеств $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ и отношения $\sim$ следует:

$$\begin{aligned} &\forall \mathfrak{f} \in \mathfrak{G}_V : \mathit{type}(\mathfrak{f}) = \mathit{any}, \mathfrak{f}_1, \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}) : \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \neq \emptyset \wedge \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_2,\mathfrak{f}_1)} \neq \emptyset \\ &\quad \Longrightarrow (\mathfrak{f}_1 \sim \mathfrak{f}_2) \in \mathfrak{R}_\mathfrak{f} \end{aligned} \tag{47}$$

Множества $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ и $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ не должны конфликтовать, то есть должно быть выполнено:

$$\begin{aligned} &\forall \mathfrak{f} \in \mathfrak{G}_V : \mathit{type}(\mathfrak{f}) = \mathit{any}, \mathfrak{f}_1, \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}) : \mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \cap \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)} \neq \emptyset \\ &\quad \Longrightarrow (\mathfrak{f}_1 < \mathfrak{f}_2) \in \mathfrak{R}_\mathfrak{f} \end{aligned} \tag{48}$$

Отношение $\lhd$ транзитивно, и потому выполнено:

$$\begin{aligned}&\exists \mathfrak{f} \in \mathfrak{G}_V : \mathit{type}(\mathfrak{f}) = \mathit{any}, \mathfrak{f}_1, \mathfrak{f}_2, \mathfrak{f}_3 \in \mathit{children}(\mathfrak{f}) : \{\mathfrak{f}_1 \rhd \mathfrak{f}_2, \mathfrak{f}_2 \rhd \mathfrak{f}_3\} \subseteq \mathfrak{R}_\mathfrak{f} \\ &\quad \Longrightarrow (\mathfrak{f}_1 \rhd \mathfrak{f}_3) \in \mathfrak{R}_\mathfrak{f}\end{aligned} \tag{49}$$

После применения правил поддержания непротиворечивости (47), (48) и (49) в множества $\mathfrak{R}_\mathfrak{f}$ могут быть добавлены новые ограничения. Для элементов этих множеств используются следующие правила поддержания непротиворечивости:

$$\begin{gathered}b \lhd d \Longleftrightarrow b < d \wedge b \sim d \\ b \lhd c \wedge c \lhd d \Longrightarrow b \lhd d \\ d_1 \sim d_2, \ldots, d_{n-1} \sim d_n, d_n \sim d_1\} \Longrightarrow \forall a, c \in \{d_1, \ldots, d_n\} : a \sim c\end{gathered} \tag{50}$$

Эти правила опираются на свойства отношений $<$, $\lhd$ и $\sim$, а также на то, что эти отношения должны задавать деревья на множествах $\mathit{children}(\mathfrak{f})$. Также во множествах $\mathfrak{R}_\mathfrak{f}$ не должно существовать конфликтов, то есть не должно быть выполнено:

$$b \sim d \wedge b > d \wedge b < d \tag{51}$$

Из измененных множеств $\mathfrak{R}_\mathfrak{f}$ может быть извлечена дополнительная информация об отношении наследования на множестве $\mathfrak{C}_\mathbf{A}$ с использованием правил поддержания непротиворечивости для множества $\mathfrak{C}_\mathbf{A}$. По одному ребру в дереве наследования не может быть унаследовано два класса, не связанных наследованием, и потому выполнено:

$$\begin{aligned}&\exists \mathfrak{f} \in \mathfrak{G}_V : \mathit{type}(\mathfrak{f}) = \mathit{any}, \mathfrak{f}_1, \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}) : \{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}, A, C \in \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1, \mathfrak{f}_2)} : A \neq C \\ &\quad \Longrightarrow \text{ выполнено } A \sim C\end{aligned} \tag{52}$$

В силу того, что в $\mathfrak{G}$ не существует цикла, проходящего только по *any*-вершинам, выполнено:

$$\begin{aligned}&\exists \mathfrak{f} \in \mathfrak{G}_V : \mathit{type}(\mathfrak{f}) = \mathit{any}, \mathfrak{f}_1, \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}) : \{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}, B \in \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1, \mathfrak{f}_2)} \\ &\quad \Longrightarrow \forall D \in \mathfrak{C}_{(\mathfrak{f}, \mathfrak{f}_1)} : \text{ выполнено } D \rhd B\end{aligned} \tag{53}$$

$$\begin{aligned}&\exists \mathfrak{f} \in \mathfrak{G}_V : \mathit{type}(\mathfrak{f}) = \mathit{any}, \mathfrak{f}_1, \mathfrak{f}_2 \in \mathit{children}(\mathfrak{f}) : \{\mathfrak{f}_1 \rhd \mathfrak{f}_2\} \subseteq \mathfrak{R}_\mathfrak{f}, B \in \mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1, \mathfrak{f}_2)} \\ &\quad \Longrightarrow \forall D \in \mathfrak{P}_{(\mathfrak{f}_1, \mathfrak{f}_2)} : \text{ выполнено } D > B\end{aligned} \tag{54}$$

Таким образом, применение алгоритмов 35 и 36 добавляет новые элементы

в множества $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_2,\mathfrak{f}_1)}$ и $\mathfrak{P}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$, применение правил поддержания непротиворечивости (47), (48) и (49) на основе добавленных элементов добавляет ограничения в множества $\mathfrak{R}_{\mathfrak{f}}$, применение правил (50) также добавляет новые ограничения в множества $\mathfrak{R}_{\mathfrak{f}}$, а применение правил поддержания непротиворечивости (52) и (54) добавляет новые ограничения на классы из $\mathfrak{C}_{\mathbf{A}}$, к которым снова можно применить алгоритмы 35 и 36. Так как все рассматриваемые множества конечны, то такой цикл применения алгоритмов 35 и 36 и правил поддержания непротиворечивости (47), (48), (49), (50), (52) и (54) не может быть бесконечен. Этот цикл необходимо применить к каждому из ограничений из $\mathfrak{R}_s$.

Если в результате применения цикла к очередному ограничению $\mathcal{R}$ из множества $\mathfrak{R}_s$ было нарушено (51) в одном из множеств $\mathfrak{R}_{\mathfrak{f}}$, или выяснилось, что не существует корневого дерева $T_{\mathfrak{f}} \in \mathbb{T}^R(\mathit{children}(\mathfrak{f}))$, удовлетворяющего ограничениям $\mathfrak{R}_{\mathfrak{f}}$, или (51) было нарушено для восстанавливаемого отношения наследования на множестве $\mathfrak{C}_{\mathbf{A}}$, то следует вернуться к предыдущему шагу, добавив ограничение $\mathcal{R}$ в множество конфликтных ограничений $\mathfrak{R}_{\times}$.

Обработав все ограничения, построим дерево T. Для этого для каждой *any*-вершины $\mathfrak{f}$ построим корневое дерево $T_{\mathfrak{f}} \in \mathbb{T}^R(\mathit{children}(\mathfrak{f}))$ на множестве дочерних для нее вершин в $\mathfrak{G}$. По построению такое дерево существует. Будем рассматривать *any*-вершины из $\mathfrak{G}$ в порядке снизу вверх и строить для них соответствующие деревья $T[\mathfrak{f}]$. Пусть $\mathfrak{f}$ — очередная такая вершина, и для всех ее дочерних вершин соответствующие деревья уже построены, то есть $\forall \mathfrak{f}' \in \mathit{children}(\mathfrak{f}) : T[\mathfrak{f}'] \in \mathbb{T}^R(\mathfrak{f}')$. Будем рассматривать такие деревья как ориентированные графы, в которых все дуги имеют направление от листьев к корню дерева. Для каждой дуги $(\mathfrak{f}_1, \mathfrak{f}_2)$ из $T_{\mathfrak{f}}$ добавим в T дугу, соединяющую корень $T[\mathfrak{f}_1]$, и вершину из $\mathfrak{f}_2$ такую, что по дугам достижимы все вершины из $\mathfrak{N}^{\mathfrak{f}}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$, и не достижимы вершины из $\mathfrak{P}^{\mathfrak{f}_2}_{(\mathfrak{f}_3,\mathfrak{f}_1)}$. Существование такой вершины следует из выполнения (54). Добавив все такие дуги, выберем корень дерева $T[\mathfrak{f}_r]$ корнем построенного дерева $T[\mathfrak{f}]$, где $\mathfrak{f}_r$ — корень $T_{\mathfrak{f}}$. Так как такое построение полностью соответствует описанному в алгоритме 26, то согласно теореме 1, для построенного корневого дерева T выполнены все ограничения, порожденные утверждением 5. По построению для этого дерева также выполнено множество ограничений $\mathfrak{R} \setminus \mathfrak{R}_{\times}$, и потому построенное дерево является решением задачи 10.

Оценим сложность представленного алгоритма. Пусть n — количество классов в множестве $\mathfrak{C}_{\mathbf{A}}$. Как было отмечено в главе 4.3.1, в задачах, встречающихся на практике, размер множества $\mathfrak{F}^{\cap}$ в большинстве случаев не превышает n^2. Пусть N — количество *any*-вершин в графе $\mathfrak{G}$. Так как каждой *any*-вершине

графа $\mathfrak{G}$ соответствует элемент множества $\mathfrak{F}^*$, то согласно (25), можно считать, что $N = O(n^2)$. Тогда в силу (44), суммарный размер множеств $\mathfrak{C}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ не превосходит nN.

Дочерними вершинами *any*-вершин могут быть другие *any*-вершины, и *fixed*-вершины. Так как в $\mathfrak{G}$ нет циклов, проходящих целиком по *any*-вершинам, то количество ребер, соединяющих *any*-вершины, не превосходит N. По построению графа $\mathfrak{G}$, у каждой *fixed*-вершины есть дочерние вершины, причем дочерними вершинами *fixed*-вершины могут быть только *any*-вершины. Следовательно, в графе $\mathfrak{G}$ не более N *fixed*-вершин. Это в частности означает, что:

$$|\mathfrak{G}_V| \leq 2N. \tag{55}$$

Так как предком каждой *fixed*-вершины является единственная *any*-вершина, то суммарное количество дочерних вершин всех *any*-вершин не превосходит $2N$, то есть:

$$\sum_{\mathfrak{f} \in \mathfrak{G}_V : type(\mathfrak{f}) = any} |children(\mathfrak{f})| \leq 2N. \tag{56}$$

Тогда из (55) следует, что суммарный размер всех множеств, соответствующих вершинам графа $\mathfrak{G}$, не превосходит $2nN$. Из (39) и (56) также следует, что суммарное количество ограничений в множествах $\mathfrak{R}_\mathfrak{f}$ не превосходит $6nN$. Также из (45), (44) и (56) следует, что суммарный размер всех множеств $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ и $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$ не превосходит $4Nn$. Число же различных ограничений на множестве $\mathfrak{C}_\mathbf{A}$ не превосходит $3n^2$.

Это означает, что затраты памяти предложенного алгоритма составляют $O(nN) = O(n^3)$. Оценим вычислительные затраты предложенного алгоритма. Проверка выполнения условий (47)-(54) производится только в случае, если множество, для которого производится проверка, было изменено. В результате анализа проверок (47)-(54) было выяснено, что для выполнения каждой их них требуется не более $O(n^2)$ операций. Так как в процессе работы алгоритма элементы только добавляются в множества $\mathfrak{N}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$, $\mathfrak{P}^\mathfrak{f}_{(\mathfrak{f}_1,\mathfrak{f}_2)}$, $\mathfrak{R}_\mathfrak{f}$, и множество ограничений над $\mathfrak{C}_\mathbf{A}$, то проверки (47)-(54) будут применены не более одного раза к каждому из элементов этих множеств. Следовательно, суммарная сложность предложенного алгоритма составляет $O(n^5)$.

Не смотря на высокую сложность, предложенный алгоритм применим для большинства возникающих на практике задач, связанных с восстановлением иерархий классов при отсутствии информации о типах времени выполнения. Это объясняется тем, что деревья наследования, состоящие более чем из 100

классов, встречаются в крупных библиотеках, таких как Qt [31] или MFC [27], которые в большинстве случаев используют ту или иную форму информации о типах времени выполнения, и поэтому для таких деревьев наследования применение предложенного алгоритма не требуется.

4.4 Восстановление множественного наследования

Как было сказано в главе 4.1, так как большинство иерархий классов используют виртуальные деструкторы, то путем анализа деструкторов в большинстве случаев можно выявить использование множественного наследования и определить, какие таблицы виртуальных функций принадлежат одному и тому же классу. В главе 4.3.2 был описан алгоритм восстановления отношения одиночного наследования на множестве таблиц виртуальных функций.

Результатом последовательного применения описанного в главе 4.3.2 алгоритма восстановления одиночного наследования на множестве таблиц виртуальных функций является одно или несколько деревьев наследования $T_1, \ldots, T_n$. Будем предполагать, что деревья наследования из $T_1, \ldots, T_n$ восстановлены корректно в смысле, определенном в главе 1. Результатом анализа деструкторов на предмет использования множественного наследования является набор множеств таблиц виртуальных функций $\mathfrak{v}_1, \ldots, \mathfrak{v}_m$ такой, что все таблицы виртуальных функций, принадлежащие одному множеству, принадлежат одному классу. Тогда, каждое из множеств $\mathfrak{v}_i$ задает отдельный класс, использующий множественное наследование. Объединив для каждого из множеств $\mathfrak{v}_1, \ldots, \mathfrak{v}_m$ соответствующие вершины в деревьях $T_1, \ldots, T_n$, можно получить иерархию, учитывающую множественное наследование. При этом следует учитывать, что, вообще говоря, два множества $\mathfrak{v}_i$ и $\mathfrak{v}_j$ могут иметь непустое пересечение — это будет означать, что классы, соответствующие $\mathfrak{v}_i$ и $\mathfrak{v}_j$, используют один и тот же деструктор, и разделяют некоторые из таблиц виртуальных функций. Тогда исходя из предположения о том, что деревья наследования $T_1, \ldots, T_n$ были восстановлены корректно, согласно правилам наследования Си++ [25], отношение непосредственного наследования между таблицами виртуальных функций из $(\mathfrak{v}_i \cup \mathfrak{v}_j) \setminus (\mathfrak{v}_i \cap \mathfrak{v}_j)$ задает отношение непосредственного наследования между классами, соответствующими $\mathfrak{v}_i$ и $\mathfrak{v}_j$. Следовательно, можно заменить в соответствии с этим отношением непосредственного наследования каждую из вершин в деревьях $T_1, \ldots, T_n$, соответствующую таблицам виртуальных функций из $\mathfrak{v}_i \cap \mathfrak{v}_j$, на две, одна из которых будет соответствовать множеству $\mathfrak{v}_i$, а другая — $\mathfrak{v}_j$, после чего произвести объединение вершин деревьев $T_1, \ldots, T_n$ также, как

и в случае отсутствия пересечений между $\mathfrak{v}_i$ и $\mathfrak{v}_j$. Аналогично рассматривается случай существования более двух множеств из $\mathfrak{v}_1, \ldots, \mathfrak{v}_m$, имеющих непустое пересечение.

Таким образом, восстановив одиночное наследования на множестве таблиц виртуальных функций, и выяснив путем анализа деструкторов, какие таблицы виртуальных функций принадлежат одним и тем же классам, можно также восстановить множественное наследование. В главе 4.1 было также отмечено, что в случае отсутствия виртуальных деструкторов множественное наследование может быть заменено одиночным без нарушения корректности. Поэтому в предположении о том, что все деревья наследования $T_1, \ldots, T_n$ были восстановлены корректно, итоговая иерархия классов, учитывающая множественное наследование, также будет восстановлена корректно.

5 Реализация.

Описанные к главах 3 и 4 алгоритмы были рассмотрены в применении к версии 9.0.30729.1 компилятора MSVC, и версии 4.2.4 компилятора GCC.

Как было сказано в главе 1, целью данной работы является разработка и реализация методов восстановления объектных структур данных из низкоуровневого представления программ, написанных на языке Си++, для компиляторов GCC и MSVC. Для того, чтобы применить разработанные в главах 3 и 4 методы к исполняемому файлу, этот файл необходимо дизассемблировать. К настоящему времени проблема дизассемблирования является достаточно хорошо исследованной, и в данной работе не рассматривается. Существует множество различных инструментов для решения этой проблемы, и для поэтому для дизассемблирования исполняемых файлов был использован один из таких инструментов — интерактивный дизассемблер IDA Pro [22]. Описание основных возможностей этого дизассемблера и сравнение его с другими аналогичными по функциональности программными продуктами приведено ниже.

5.1 Интерактивный дизассемблер IDA Pro

IDA Pro, Interactive Disassembler Pro — интерактивный дизассемблер, который широко используется для решения задач обратного проектирования. IDA Pro до определенной степени умеет автоматически выполнять анализ, разделяя код и данные, выделяя функции и *адаптеры* в потоке инструкций и распознавая стандартные библиотечные функции. Отличительной особенностью IDA Pro является возможность интерактивного взаимодействия с пользователем. В начале исследования дизассемблер выполняет автоматический анализ программы, а затем пользователь с помощью интерактивных средств начинает давать осмысленные имена, комментировать, создавать сложные структуры данных и другим образом добавлять информацию в листинг, генерируемый дизассемблером, пока не станет ясно, что именно делает исследуемая программа.

Дизассемблер имеет консольную и графическую версии, поддерживает большое количество форматов исполняемых файлов, в том числе PE-COFF[8)], ELF[9)], Mach-O[10)] и другие [34], позволяет анализировать код для более чем пятидесяти семейств процессоров, а также байт-код виртуальных машин Java и .NET [33].

[8)] Portable Executable and Common Object File Format, используется в Windows.

[9)] Executable and Linkable Format, используется в Linux и большинстве операционных систем семейства BSD.

[10)] Mach Object, используется в Mac OS X.

IDA Pro имеет модульную архитектуру, что позволяет добавлять в дизассемблер новую функциональность с помощью созданных пользователями плагинов. Примером такого плагина является система Hex-Rays, строящая схему исследуемой программы на Си-подобном языке. Для решения задач меньшего масштаба в IDA Pro встроен похожий на Си язык программирования IDC. Также, воспользовавшись плагинами IDAPython [23] или IDARub [24], возможно написание скриптов на python и ruby соответственно. Для скриптов, написанных на языках IDC, python, или ruby, IDA Pro предоставляет большой набор средств для доступа к анализируемому файлу и функциональности дизассемблера. К таким средствам в частности относятся:

- Простая навигация по анализируемому файлу с использованием виртуальных адресов, а не смещений относительно начала файла.
- Удобный доступ как к бинарному, так и к разобранному на инструкции представлению программы.
- Возможность вносить изменения в результаты автоматического анализа.
- Возможность дизассемблировать произвольную последовательность байт, с проверкой на корректность возможного выполнения полученных инструкций.
- Доступ к встроенным в IDA Pro средствам анализа. В частности, с использованием стандартных функций IDC можно выделять подпрограммы в потоке инструкций и определять размер *стекового фрейма* функции и его составляющих — параметров, сохраненных регистров, и локальных переменных.
- Возможность использования поддерживаемой IDA Pro базы перекрестных ссылок для получения адресов всех локаций, каким-либо образом ссылающихся на данную.

Использование некоторых из этих средств существенно упрощает решение описанных в главах 3 и 4 задач анализа ассемблерного представления программы, так как:

- Для поиска таблиц виртуальных функций необходимо иметь возможность находить адреса всех локаций, каким-либо образом ссылающихся на данную, как это описано в главе 3.2.1.

- Для применения утверждений 6 и **??**, описанных в главе 4.2.2, необходимо иметь возможность определять суммарный размер параметров функции по ее эпилогу.

- Информация о типах времени выполнения имеет сложную структуру, как это описано в главах 3.1.2 и 3.1.1, и потому для извлечения этой информации из ассемблерного листинга необходимо иметь возможность навигации по нему с использованием виртуальных адресов.

В качестве альтернатив интерактивному дизассемблеру IDA Pro были рассмотрены программные продукты SoftICE и OllyDbg.

SoftICE — отладчик режима ядра для Microsoft Windows. Программа предназначена для работы на низком уровне операционной системы Windows так, чтобы операционная система не распознавала работу отладчика [1]. В отличие от отладчиков, работающих в режиме пользователя, SoftICE может остановить все операции в Windows, что очень важно для отладки драйверов. Поддержка SoftICE была приостановлена разработчиками в 2007 году, но не смотря на это, он все еще широко используется. Существует набор средств для создания плагинов для SoftICE. Однако из-за того, что SoftICE работает в режиме ядра, написание плагинов для него становится чрезвычайно сложной задачей, сравнимой с написанием драйверов, работающих в режиме ядра. И если драйвера, работающие в режиме ядра, можно отлаживать с помощью SoftICE, то плагины для SoftICE отлаживать таким способом нельзя. Существующие же плагины для SoftICE, позволяющие использовать скриптовые языки программирования для реализации дополнительной функциональности, имеют весьма ограниченные возможности [21].

OllyDbg — бесплатный 32-битный отладчик для операционных систем семейства Windows, предназначенный для анализа и модификации исполняемых файлов и библиотек, работающих в режиме пользователя. Работа в режиме пользователя в частности означает, что с помощью OllyDbg невозможно отлаживать драйвера, работающие в привилегированном режиме. OllyDbg отличается простым интерфейсом, интуитивной подсветкой специфических структур кода, простотой в установке и запуске. Из особенностей OllyDbg следует выделить [28]:

- Поддержку процессоров семейства x86. Расширения SSE2, SSE3, SSE4 не поддерживаются.

- Анализатор кода, распознающий процедуры, циклы, ветвления, таблицы, константы и текстовые строки.
- Развернутую система поиска: поиск всех возможных констант, команд, последовательностей команд, текстовых строк и ссылок в коде на данный адрес.
- Эвристический анализ стека, распознавание адресов возврата в родительскую процедуру.
- Возможность написания плагинов для расширения имеющейся функциональности. В частности, существует плагин OllyScript, позволяющий писать расширения для OllyDbg на похожем на ассемблер языке программирования.

Одним из недостатков системы плагинов OllyDbg является невозможность работы в коде плагинов с подсистемой поиска. Это означает, что поиск всех ссылок в коде на данный адрес необходимо реализовывать самостоятельно. Также несмотря на то, что кроме OllyScript существуют плагины, поддерживающие другие скриптовые языки программирования, такие как python [29], многие из них недостаточно документированы или уже не поддерживаются авторами.

В результате сравнения возможностей перечисленных программных продуктов было выяснено, что предоставляемые интерактивным дизассемблером IDA Pro средства расширения функциональности являются более простыми в использовании, более документированными, и предоставляющими большие возможности, чем аналогичные средства отладчиков OllyDbg и SoftICE. Написание необходимых алгоритмов с использованием OllyDbg или SoftICE возможно, однако это потребовало бы реализации части той функциональности, которую предоставляет IDA Pro для расширений, реализованных на IDC или python.

5.2 Программная реализация алгоритмов анализа, использующих информацию о типах времени выполнения

Было разработано приложение crec[11)], в котором были реализованы описанные в главах 3 и 4 методы восстановления объектных структур данных. Часть этого приложения была реализована на встроенном в интерактивный дизассемблер IDA Pro языке программирования IDC, часть — на языке Си++.

[11)] от англ. Class hierarchy REConstruction

В случае присутствия информации о типах времени выполнения часть crec, реализованная на языке IDC, анализирует ассемблерный листинг, полученный путем дизассемблирования исполняемого файла дизассемблером IDA Pro, и извлекает из него структуры, содержащие эту информацию. Поиск этих структур производится с использованием алгоритма, описанного в главе 3.2.1. Извлеченные структуры сохраняются в файл в формате XML.

Этот файл является входным для реализованной на языке Си++ части crec. По извлеченной информации о типах времени выполнения эта часть восстанавливает иерархию полиморфных классов программы. Результатом работы этой части является входной файл для утилиты dot из пакета GraphViz [20] с описанием графа, заданного восстановленной иерархией классов. Утилита dot по входному описанию графа строит его графическое представление. Таким образом, с использованием разработанного приложения crec, по ассемблерному листингу исходной Си++-программы, использующей информацию о типах времени выполнения, можно получить графическое представление ее иерархии классов.

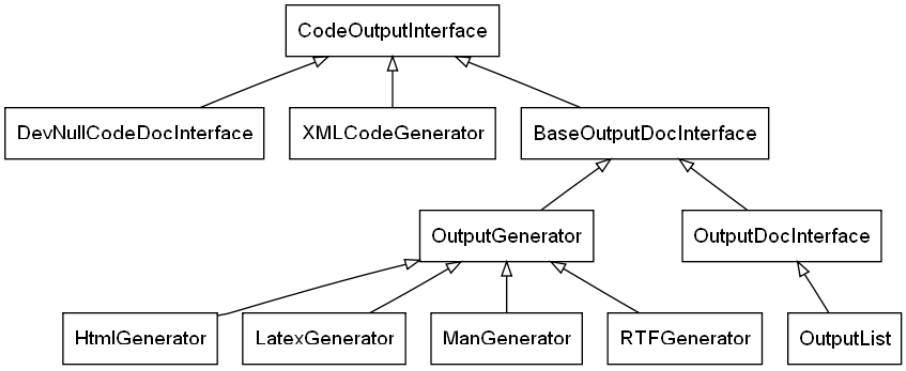

Рис. 37: Графическое представление фрагмента иерархии полиморфных классов программы doxygen, полученное с использованием приложения crec путем анализа информации о типах времени выполнения.

Пример графического представления иерархии полиморфных классов, восстановленной с использованием приложения crec, приведен на рис. 37. В качестве анализируемой программы использовалась система документирования исходных кодов doxygen. Одной из причин использования doxygen в качестве анализируемой программы является то, что doxygen распространяется с открытыми исходными кодами, что упрощает проверку корректности результатов восстановления. Также изучение исходных кодов doxygen показало, что в некоторых методах используется оператор dynamic_cast<>, и поэтому для правильной работы этой программы она должна была быть скомпилирована с поддержкой

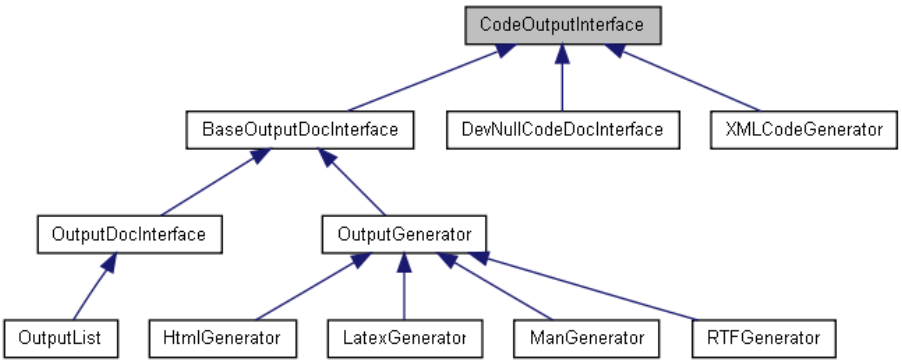

Рис. 38: Графическое представление фрагмента иерархии полиморфных классов doxygen, полученное путем анализа исходного кода с помощью программы doxygen.

информации о типах времени выполнения. Целью анализа являлось восстановление иерархии полиморфных классов. Для этого необходима информация о:

- Компиляторе, с использованием которого был получен исполняемый файл;
- Формате RTTI структур, который используется этим компилятором.

В результате применения методов, описанных в главе 3.2, было выяснено, что doxygen 1.5.8 для Windows скомпилирован компилятором MSVC, формат RTTI структур для которого был описан в главе 3.1.1. С использованием написанной на языке IDC части crec был проанализирован ассемблерный листинг. С использованием написанной на Си++ части crec была восстановлена полная иерархия полиморфных классов программы doxygen, состоящая из 444 классов. На рис. 37 приведено графическое представление фрагмента восстановленной иерархии полиморфных классов, а на рис. 38 — графическое представление того же фрагмента, полученное путем анализа исходного кода. Как видно, фрагменты иерархии совпадают.

В результате сравнения иерархии полиморфных классов doxygen, построенной путем анализа исходного кода, с иерархией, восстановленной с использованием приложения crec, было выяснено, что эти иерархии совпадают.

5.3 Программная реализация алгоритмов анализа, не использующих информацию о типах времени выполнения

В приложении crec также были реализованы методы восстановления иерархий полиморфных классов для случая отсутствия информации о типах времени выполнения, описанные в главе 4. Извлечение информации, необходимой для построения ограничений с использованием утверждений 1-8, производится в части crec, реализованной на языке IDC. Анализ извлеченной информации с использованием описанных в главе 4 методов производится в части crec, реализованной на языке Си++. Результатом работы приложения crec является входной файл для утилиты dot из пакета GraphViz, как это описано в главе 5.2.

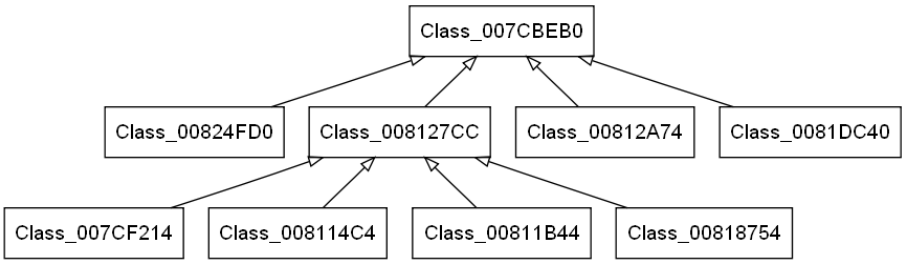

Рис. 39: Графическое представление фрагмента иерархии полиморфных классов программы doxygen, полученное с использованием приложения crec без использования информации о типах времени выполнения.

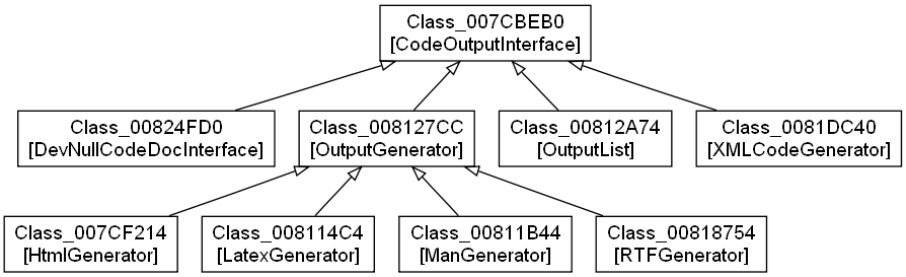

Рис. 40: Графическое представление иерархии классов, представленной на рис. 39, с сопоставленными классам реальными именами.

Пример графического представления иерархии полиморфных классов, восстановленной с использованием приложения crec, представлен на рис. 39. Также, как и в главе 5.2, в качестве анализируемой программы использовалась система документирования исходных кодов doxygen. Анализируемый исполнимый файл был получен путем компиляции doxygen без поддержки информации

о типах времени выполнения компилятором MSVC. При компиляции без информации о типах времени выполнения имена классов не попадает в исполнимый файл, и поэтому все классы на рис. 39 имеют автоматически сгенерированные имена.

Для проверки корректности восстановления система doxygen была скомпилирована еще раз с поддержкой информации о типах времени выполнения. Для исполнимого файла, полученного путем компиляции с использованием информации о типах времени выполнения, иерархия полиморфных классов была восстановлена так, как это описано в главе 5.2. Путем сравнения виртуальных функций каждой таблице виртуальных функций из ассемблерного листинга исполнимого файла, полученного компиляцией без поддержки информации о типах времени выполнения, была поставлена в соответствие таблица виртуальных функций из ассемблерного листинга исполнимого файла, полученного компиляцией с поддержкой информации о типах времени выполнения. Таким образом, для каждой таблицы виртуальных функций из ассемблерного листинга исполнимого файла, полученного компиляцией без поддержки информации о типах времени выполнения, было получено имя соответствующего ей класса. На рис. 40 представлен фрагмент иерархии, соответствующий представленному на рис. 39 фрагменту иерархии, но с сопоставленными классам реальными именами.

Представленный на рис. 40 восстановленный фрагмент иерархии соответствует фрагменту, изображенному на рис. 38. Как видно, фрагменты иерархии отличаются. Это объясняется тем, что классы BaseOutputDocInterface и OutputDocInterface, присутствующие в фрагменте иерархии, изображенном на рис. 38, не перекрывают унаследованные ими виртуальные функции, а лишь добавляют новые чисто виртуальные функции, и потому компилятором для этих классов не была сгенерирована таблица виртуальных функций.

В результате сравнения иерархии полиморфных классов doxygen, восстановленной с помощью приложения crec без использования информации о типах времени выполнения, с иерархией, построенной путем анализа исходного кода, было выяснено, что восстановление иерархии было произведено корректно.

Заключение

В данной работе были представлены методы автоматического восстановления объектных структур данных из низкоуровневого представления программ на языке Си++. Так как программа на языке Си++ может быть скомпилирована как с поддержкой, так и без поддержки информации о типах времени выполнения, то были рассмотрены оба случая.

Формат информации о типах времени выполнения зависит от используемого компилятора. Были рассмотрены форматы, используемые компиляторами GCC и MSVC. Для этих компиляторов также были рассмотрены основные положения используемых ими бинарных интерфейсов приложений, связанные с реализацией виртуальных функций. Анализ бинарных интерфейсов приложений показал, что задача поиска информации о типах времени выполнения сводится к задаче поиска таблиц виртуальных функций. Был представлен алгоритм поиска таблиц виртуальных функций, и метод восстановления иерархий классов, использующий этот алгоритм для локализации информации о типах времени выполнения.

Для случая отсутствия информации о типах времени выполнения был предложен метод поэтапного восстановления отношения наследования. Были рассмотрены правила языка программирования Си++ и положения бинарных интерфейсов приложений, используемых компиляторами GCC и MSVC, связанные с множественным наследованием. Анализ этих правил показал, что при отсутствии виртуальных деструкторов, множественное наследование может быть корректно заменено одиночным.

Было показано, что отношение наследования на множестве классов можно расширить на множество таблиц виртуальных функций, и что при этом наследование на множестве таблиц виртуальных функций является одиночным. Для восстановления одиночного наследования был разработан метод, основывающийся на анализе:

- Таблиц виртуальных функций;
- Параметров виртуальных функций;
- Вызовов виртуальных функций;
- Виртуальных деструкторов.

Для восстановления множественного наследования был предложен метод объединения таблиц виртуальных функций, принадлежащих одному классу, и

модификации деревьев одиночного наследования на множестве таблиц виртуальных функций, основывающийся на анализе виртуальных деструкторов.

Было разработано приложение crec, в котором были реализованы предложенные методы восстановления объектных структур данных как для случая присутствия информации о типах времени выполнения, так и для случая ее отсутствия. Приложение crec было протестировано на системе документирования исходных кодов doxygen, и были получены положительные результаты. В случае присутствия информации о типах времени выполнения иерархия полиморфных классов и таблицы виртуальных функций были восстановлены точно и полностью. В случае отсутствия информации о типах времени выполнения иерархия полиморфных классов была восстановлено корректно, с точностью до замены множественного наследования одиночным и удаления некоторых классов, для которых компилятором не были сгенерированы таблицы виртуальных функций.

Приложение А. Список используемых терминов и обозначений

К сожалению, термины в области программирования не устоялись. В разных переводах, и в различных работах на русском языке, наблюдаются существенные рассогласования. Поэтому здесь приведены значения используемых в данной работе терминов. Все термины, имеющие отношения к языкам программирования, такие как *класс*, *объект*, или *функция*, рассматриваются в данной работе в применении к языку программирования Си++.

- *Абстрактный класс* (англ. *abstract class*) — класс, содержащий хотя бы одну *чисто виртуальную функцию*.
- *Адаптер* (англ. *thunk*) — дополнительный код, сгенерированный компилятором, выполняющий преобразование (например, типов), и передающий управление какой-либо функции с использованием инструкции безусловного перехода.
- *Базовый класс* (англ. *base class*). Класс B является базовым классом для класса D, если он является *непосредственным базовым классом* для класса D, или *непосредственным базовым классом* для одного из базовых классов класса D [25].
- *Бинарный интерфейс приложений* (англ. *application binary interface*) — набор конфигураций среды разработки и компилятора, гарантирующих бинарную совместимость разрабатываемых приложений. Бинарный интерфейс приложений определяет взаимодействие на низком уровне между приложениями, между компонентами приложения, между приложением и библиотеками, и между приложением и операционной системой на используемой платформе.
- *Виртуальная функция* (англ. *virtual function*) — метод класса, который может быть переопределён в производных классах так, что конкретная реализация метода для вызова будет определяться во время выполнения программы. В языке программирования Си++ виртуальные функции определяются с ключевым словом virtual [16].
- *Виртуальный метод* (англ. *virtual method*) — см. *виртуальная функция*.

- *Встраивание* (англ. *inline expansion*) — оптимизация, при которой вызов функции заменяется на подстановку копии ее тела.

- *Декорирование имен* (англ. *name mangling*) — декорирование имен добавляет дополнительную информацию к имени функций, структур, классов, и других типов данных, чтобы избежать конфликтов на этапе *компоновки* программы. Разные компиляторы используют разные схемы декорирования имен.

- *Динамический полиморфизм* (англ. *dynamic polymorphism*) — форма *полиморфизма*, подразумевающая выбор необходимой операции обработки данных на этапе выполнения программы, то есть с использованием *позднего связывания*. В языке Си++ динамический полиморфизм реализуется системой *виртуальных функций*.

- *Динамическое связывание* (англ. *dynamic binding*) — определение реализации вызываемого метода во время выполнения программы на основании фактического типа объекта (или объектов), для которого был вызван этот метод.

- *Информация о типах времени выполнения* (англ. *run-time type information*)— стандартизованный интерфейс, с помощью которого программы на языке Си++, использующим указатели или ссылки на базовые классы, могут выяснять фактические типы объектов производных классов, к которым относятся эти указатели или ссылки [15]. Реализации информации о типах времени выполнения в разных компиляторах существенно отличаются.

- *Компоновка* (англ. *linking*) — процесс создания исполняемого модуля программы из одного или нескольких объектных модулей, полученных в результате компиляции исходных файлов программы.

- *Наследник* (англ. *child class*) — см. *производный класс*.

- *Непосредственный базовый класс* (англ. *direct base class*). Класс B является непосредственным базовым классом для класса D, если в соответствующей программе на языке Си++ класс B присутствует в списке базовых классов для класса D [25].

- *Обратное проектирование* (англ. *reverse engeneering*) — процесс извлечения информации о строении и принципах работы системы путем анали-

за ее функций, структуры, и поведения. В применении к программному обеспечению целью обратного проектирования является повышение уровня абстракции представления программ.

- *Перегрузка функций* (англ. *function overloading*) — возможность языка программирования поддерживать одновременное существование в одной области видимости нескольких различных вариантов функции, имеющих одно и то же имя, но различающихся типами параметров, к которым она применяется.

- *Перекрытие методов* (англ. *method overriding*) — особенность языка программирования, позволяющая производному классу заменить реализацию *виртуального метода* базового класса.

- *Позднее связывание* (англ. *late binding*) — см. *динамическое связывание*.

- *Полиморфизм* (англ. *polymorphism*) — в применении к языкам программирования, это возможность работать с данными разных типов с использованием одного и того же интерфейса.

- *Полиморфный класс* (англ. *polymorphic class*) — класс, имеющий хотя бы одну *виртуальную функцию* [16].

- *Предок* (англ. parent class) — см. *базовый класс*.

- *Производный класс* (англ. *derived class*). Класс D является производным от класса B классом, тогда и только тогда, когда класс B является *базовым классом* для класса D.

- *Соглашение о вызовах* (англ. *calling conventions*) — часть *бинарного интерфейса приложений*, которая регламентирует технические особенности вызова подпрограмм, передачи параметров, возврата из подпрограмм и передачи результата вычислений в точку вызова.

- *Статический полиморфизм* (англ. *static polymorphism*) — форма *полиморфизма*, подразумевающая выбор необходимой операции обработки данных на этапе компиляции программы, то есть с использованием *статического связывания*. Примерами статического полиморфизма в языке Си++ являются *шаблоны* и *перегруженные функции*.

- *Статическое связывание* (англ. *static binding*) — в применении к вызовам функций в программе на языке Си++, применение статического связывания подразумевает, что реализация вызываемой функции известна во время компиляции программы. Это в частности означает, что вызываемая функция может быть *встроена* в код вызывающей функции.

- *Стековый фрейм* (англ. *stack frame*) — зависимая от используемой платформы и компилятора структура данных, хранящая информацию о состоянии выполняемой подпрограммы.

- *Таблица виртуальных функций* (англ. *virtual function table*) — система, используемая в языках программирования для реализации *динамического связывания*. При использовании таблицы виртуальных функций, каждый объект *полиморфного класса* содержит указатель на соответствующую классу таблицу виртуальных функций, включающую адреса всех *виртуальных методов* класса [17].

- *Чисто виртуальная функция* (англ. *pure virtual function*) — *виртуальная функция* без реализации. Реализации ее, возможно, различные, могут содержаться в производных классах.

Ниже перечислены основные обозначения, используемые в данной работе.

- R^{-1} — отношение, обратное к $R \subseteq X \times X$. $R^{-1} = \{(y, x) | (x, y) \in R\}$.

- R^{+} — транзитивное замыкание бинарного отношения $R \subseteq X \times X$. R^{+} является наименьшим транзитивным отношением на множестве X, включающим R.

- $R^{=}$ — рефлексивное замыкание бинарного отношения $R \subseteq X \times X$. $R^{=} = R \cup \{(x, x) | x \in X\}$.

- $R^{\neq}$ — рефлексивное сужение бинарного отношения $R \subseteq X \times X$. $R^{\neq} = R \setminus \{(x, x) | x \in X\}$.

- $S \circ R$ — композиция бинарных отношений $R, S \subseteq X \times X$, определенная как $S \circ R = \{(x, z) | \exists y \in X : (x, y) \in R \wedge (y, z) \in S\}$.

- $|X|$ — мощность множества X.

- $\mathbb{G}(X)$ — множество всевозможных графов $G = (X, E)$.

- $\mathbb{T}(X)$ — множество всевозможных деревьев $T = (X, E)$.

- $\mathbb{T}^R(X)$ — множество всевозможных корневых деревьев $T = (X, E, r)$, где $r \in X$ — корень дерева.

- $R[Y]$ — сужение отношения $R \subseteq X \times X$ на множество $Y \subseteq X$, то есть $R[Y] = R \cap (Y \times Y)$.

- $T[Y]$ — сужение дерева $T = (X, E)$ на множество вершин $Y \subseteq X$, то есть $T[Y] = (Y, E[Y])$.

- $T[\mathbb{T}]$ — дерево $T' = (X, E)$, соответствующее корневому дереву $T = (X, E, r)$.

- $\mathfrak{C}_{\mathbf{P}}$ — множество всех полиморфных классов программы на языке Си++ **P**.

- $\leftarrow_{\mathbf{P}}$ — бинарное отношение непосредственного наследования, определенное на множестве $\mathfrak{C}_{\mathbf{P}}$ для программы на языке Си++ **P**. Для любых двух классов $B, D \in \mathfrak{C}_{\mathbf{P}}$, $B \leftarrow_{\mathbf{P}} D \iff B$ является *непосредственным базовым классом* для D.

- $\lhd_{\mathbf{P}} = \leftarrow_{\mathbf{P}}^{+}$ — бинарное отношение наследования, определенное на множестве $\mathfrak{C}_{\mathbf{P}}$ для программы на языке Си++ **P**. Для любых двух классов $B, D \in \mathfrak{C}_{\mathbf{P}}$, $B \lhd_{\mathbf{P}} D \iff B$ является *базовым классом* для D. Это обозначение соответствует определению базового класса, данному выше.

Список литературы

[1] Boldewin F. The big SoftICE how-to [PDF] (http://www.reconstructer.org/papers/The%20big%20SoftICE%20howto.pdf).

[2] Dolgova K., Chernov A. Automatic Type Reconstruction in Disassembled C Programs // 15th Working Conference on Reverse Engineering. 2008. P. 202-206.

[3] Emmerik M. V., Waddington T. Using a Decompiler for Real-World Source Recovery // 11th Working Conference on Reverse Engineering. 2004. P. 27-36.

[4] Emmerik M. V. Static Single Assignment for Decompilation [PDF] (http://vanemmerikfamily.com/mike/master.pdf).

[5] Gray J. C++: 水面下の仕組み [HTML] (http://www.microsoft.com/japan/msdn/vs_previous/visualc/techmat/feature/jangrayhood/).

[6] Kochhar V. How a C++ compiler implements exception handling [HTML] (http://www.codeproject.com/KB/cpp/exceptionhandler.aspx).

[7] Mycroft A. Type-Based Decompilation // 8th European Symposium on Programming Languages and Systems. 1999. P. 208-223.

[8] Palmer C., Kershenbaum A. Representing trees in genetic algorithms // First IEEE Conference on Evolutionary Computation. 1994. P. 27-29.

[9] Sabanal P., Yason M. Reversing C++ // Black Hat DC. 2007.

[10] Seonghoon K. Microsoft C++ Name Mangling Scheme [HTML] (http://mearie.org/documents/mscmangle).

[11] Skochinsky I. Reversing Microsoft Visual C++ Part II: Classes, Methods and RTTI [HTML] (http://www.openrce.org/articles/full_view/23).

[12] Stroustrup B. A history of C++: 1979–1991 // The second ACM SIGPLAN conference on History of programming languages. 1993. P. 271-297.

[13] Stroustrup B. The C++ programming language 3rd edition. Massachusetts: Addison-Wesley, 1997. 1022 p.

[14] Wiener R. Obfuscation and .NET // Journal of Object Technology. 2005. 4. N 4. P. 73-92.

[15] Липпман С., Му Б., Лажойе Ж. Язык программирования C++, вводный курс, 4-е издание. М.: Вильямс, 2007. 896 с.

[16] Строуструп Б. Язык программирования C++, специальное издание. М.: Бином, 2004. 1104 с.

[17] Эллис М., Строуструп Б. Справочное руководство по языку программирования C++ с комментариями. Проект стандарта ANSI. М.: Мир, 1992. 445 с.

[18] Application Binary Interface [HTML] (`http://en.wikipedia.org/wiki/Application_binary_interface`).

[19] Boomerang. BSD licensed software [HTML] (`http://boomerang.sourceforge.net/`).

[20] GraphViz. CPL licensed software [HTML] (`http://www.graphviz.org/`).

[21] IceX. A free SoftICE Plugin [HTML] (`http://mamaich.uni.cc/`).

[22] IDA Pro Disassembler. Proprietary software [HTML] (`http://www.hex-rays.com/idapro/`).

[23] IDAPython. A free IDA Pro plugin [HTML] (`http://www.d-dome.net/idapython`).

[24] IDARub. A free IDA Pro plugin [HTML] (`http://www.metasploit.com/users/spoonm/idarub/`).

[25] ISO/IEC Standard 14882:2003. Programming languages - C++. New York: American National Standards Institute, 2003. 786 p.

[26] Itanium C++ ABI, revision 1.86 [HTML] (`http://www.codesourcery.com/public/cxx-abi/abi.html`).

[27] Microsoft Foundation Class Library Reference [HTML] (`http://msdn.microsoft.com/en-us/library/d06h2x6e.aspx`).

[28] OllyDbg. Proprietary software [HTML] (`http://www.ollydbg.de/`).

[29] OllyPython. A free OllyDbg plugin [HTML] (`http://code.google.com/p/ollypython/`).

[30] Polymorphism in object-oriented programming [HTML] (http://en.wikipedia.org/wiki/Polymorphism_in_object-oriented_programming).

[31] Qt — A cross-platform application and UI framework [HTML] (http://www.qtsoftware.com/products/).

[32] Wine. LGPL licensed software [HTML] (http://www.winehq.org).

[33] Поддерживаемые типы процессоров IDA Pro - Interactive Disassembler [HTML] (http://www.idapro.ru/description/proc/).

[34] Поддерживаемые форматы входных файлов IDA Pro - Interactive Disassembler [HTML] (http://www.idapro.ru/description/format/).