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

На правах рукописи

Трошина Екатерина Николаевна

ИССЛЕДОВАНИЕ И РАЗРАБОТКА МЕТОДОВ ДЕКОМПИЛЯЦИИ ПРОГРАММ

Специальность 05.13.11 – математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей

АВТОРЕФЕРАТ
диссертации на соискание ученой степени
кандидата физико-математических наук

Москва – 2009

Работа выполнена на кафедре системного программирования факультета вычислительной математики и кибернетики Московского государственного университета имени М. В. Ломоносова.

Научный руководитель: академик РАН, профессор
Иванников Виктор Петрович

Официальные оппоненты: доктор физико-математических наук
профессор
Смелянский Руслан Леонидович

кандидат физико-математических наук
Волконский Владимир Юрьевич

Ведущая организация: Вычислительный центр имени
А. А. Дородницина РАН

Защита диссертации состоится «20» ноября 2009 г. в 11:00 на заседании диссертационного совета Д 501.001.44 в Московском государственном университете имени М. В. Ломоносова по адресу: 119991, ГСП-1, Москва, Ленинские горы, МГУ, 2-й учебный корпус, факультет вычислительной математики и кибернетики, ауд. 685.

С диссертацией можно ознакомиться в библиотеке факультета вычислительной математики и кибернетики МГУ имени М. В. Ломоносова. С текстом автореферата можно ознакомиться на официальном сайте ВМК МГУ имени М. В. Ломоносова: http://cs.msu.su в разделе «Наука» – «Работа диссертационных советов» – «Д 501.001.44».

Автореферат разослан « __ » октября 2009 г.

Ученый секретарь
диссертационного совета
профессор Н.П. Трифонов

Общая характеристика работы

Актуальность.

Создание и разработка сложных программных систем различного назначения часто ведется посредством интеграции отдельных компонент, выполненных как собственными, так и сторонними разработчиками. Это позволяет значительно сократить стоимость и время разработки программного обеспечения. Внешние модули могут поставляться без исходного кода. Наличие таких модулей в системе уменьшает уровень надежности разрабатываемого приложения с точки зрения информационной безопасности. В частности, сторонние модули могут содержать закладки или уязвимости, способствующие утечке информации и успешным атакам на информационную систему. Кроме того, программные модули от внешних разработчиков могут содержать ошибки, исправление которых оказывается затруднительным. Следовательно, весь сторонний код должен подвергаться аудиту с точки зрения безопасности его внедрения и использования.

Программные компоненты, представленные в виде исполняемых файлов или на языке ассемблера, сложны для анализа специалистами в области информационной безопасности. Для более качественного и продуктивного анализа их лучше предоставлять специалистам на более высоком уровне представления, например, на языке высокого уровня, в частности на языке программирования Си. Ассемблерный код и, тем более, исполняемые файлы не позволяют с приемлемыми трудозатратами оценить взаимосвязь элементов программы, а также идентифицировать в программе различные алгоритмические конструкции, в то время как наличие восстановленной программы на языке высокого уровня дает возможность преодолеть указанные выше трудности. В качестве одного из средств для повышения уровня абстракции представления программы может использоваться декомпиляция.

Декомпиляция – это процесс автоматического восстановления программы на языке высокого уровня из программы на языке низкого уровня. Под декомпилятором мы будем понимать инструментальное средство, получающее на вход программу на языке ассемблера или другое аналогичное низкоуровневое представление и выдающее на выход эквивалентную ей программу на некотором языке высокого уровня.

Также декомпиляция может использоваться для обеспечения совместимости программных приложений, а именно для анализа протоколов взаимодействия в случае, когда они описаны недостаточно полно или не описаны вообще. Декомпиляция позволяет упростить восстановление состояний и структур данных протокола взаимодействия.

В настоящее время из широко используемых компилируемых языков программирования высокого уровня распространены языки Си и Си++, поскольку именно они наиболее часто используются при разработке прикладного и системного программного обеспечения для платформ Windows, MacOS и Unix. Поэтому декомпиляторы с этих языков имеют наибольшую практическую

значимость. Язык Си++ можно считать расширением языка Си, добавляющим в него концепции более высокого уровня относительно языка Си. Поскольку при обратной инженерии в целом и при декомпиляции в частности уровень абстракции представления программы повышается, можно считать, что программы на языке Си являются промежуточным уровнем при переходе от программы на языке ассемблера к программе на языке Си++. Дальше повысить уровень абстракции представления программы можно посредством широко известных методов рефакторинга, позволяющих, например, выделять объектные иерархии из процедурного кода.

Из-за ряда трудностей задача декомпиляции не решена в полной мере до сих пор, хотя была поставлена еще в 60-е годы прошлого века. С теоретической точки зрения задачи построения полностью автоматического универсального дизассемблера и декомпилятора относят к алгоритмически неразрешимым. Неразрешимость задач следует из того, что задача автоматического разделения кода и данных является алгоритмически неразрешимой.

Помимо функциональной эквивалентности исходной программе декомпилированная программа должна быть восстановлена наиболее полно. Полнота восстановления определяется степенью использования высокоуровневых конструкций целевого языка, то есть программы, полученные на выходе декомпилятора, должны как можно больше и полнее использовать высокоуровневые конструкции языка.

Для оценки полноты восстановления в работе вводится новое требование к результату декомпиляции программ – *качество*, а также вводится *мера качества*, которая позволяет количественно сравнивать программы, восстановленные различными инструментальными средствами декомпиляции.

Представляемая работа посвящена разработке алгоритмов и методов автоматической декомпиляции программ на языке ассемблера в программы на языке Си.

Низкоуровневые конструкции языка Си, такие как явные приведения типов, неестественные циклы, организованные с использования операторов `goto` или с использованием операторов `continue` или `break`, замена оператора множественного выбора `switch` операторами `if` и другие, а также ассемблерные вставки кода, который не удалось восстановить, крайне нежелательны в восстановленной программе. Автоматически восстановленная программа декомпилирована качественно, если она содержит минимальное количество низкоуровневых конструкций и наиболее полно использует высокоуровневые конструкции языка Си. Например, вместо оператора цикла `while`, если возможно, используется оператор цикла `for`, вместо операций с адресной арифметикой используются операции доступа к элементам массива.

Отображение конструкций языка высокого уровня в конструкции языка низкого уровня – это отображение «многие ко многим». Если зафиксировать компилятор и значений опций, управляющих генерацией кода, то это отображение становится «многие к одному», что несколько упрощает задачу восстановления программ, даже несмотря на то, что распознавание компилятора,

которым была скомпилирована высокоуровневая программа – отдельная нетривиальная задача.

Декомпилятор, который восстанавливает низкоуровневую программу, изначально написанную на неизвестном заранее языке программирования, в качественную программу на фиксированном языке высокого уровня, назовем универсальным.

Разные языки высокого уровня одного типа (например, процедурные, компилируемые) предоставляют примерно одинаковый набор возможностей, однако, некоторые возможности одного языка могут не иметь прямых аналогов в другом языке. Например, указатели языка Си не имеют прямого аналога в языке Фортран. В таких случаях трансляторы программ из одного языка высокого уровня в другой на выходе вынуждены генерировать код, моделирующий особенности исходного языка средствами целевого языка. Сгенерированная на выходе программа содержит *артефакты трансляции*, то есть фрагменты кода, появившиеся вследствие различия понятийного аппарата исходного и целевого языков. Поэтому, хотя существующие в настоящее время трансляторы между языками высокого уровня и дают на выходе синтаксически и семантически корректный код, он зачастую труден для анализа и модификации человеком. Представляется, что задача построения универсального декомпилятора не проще задачи построения универсального транслятора в фиксированный язык высокого уровня, поэтому в настоящее время нет оснований рассчитывать на возможность успешного построения универсального декомпилятора.

В силу вышесказанного является актуальной разработка методов восстановления программ в предположении, что исходная программа была реализована на конкретном языке программирования. В представляемой работе в качестве такого языка используется язык Си. *Декомпилятор* определяется как транслятор с языка низкого уровня в язык высокого уровня, который минимизирует количество артефактов трансляции в результирующей программе и выполняет восстановление высокоуровневых конструкций целевого языка максимально полно в предположении, что исходная программа была написана на этом языке высокого уровня.

Целью диссертационной работы является разработка алгоритмов и методов декомпиляции программ, применимых к широкому спектру целевых низкоуровневых программ и позволяющих автоматически полно восстановить все высокоуровневые конструкции целевого языка высокого уровня. Помимо этого целью работы является разработка прототипа экспериментальной диалоговой инструментальной среды для декомпиляции программ в язык Си на основе предложенных методов и алгоритмов декомпиляции.

Научная новизна. В представляемой работе предложено новое требование к методам декомпиляции – качество восстановления программ, которое определяется полнотой восстановления высокоуровневых конструкций целевого

языка. Также предложен метод, позволяющий выполнять количественное сравнение качества восстановления программ различными декомпиляторами.

В работе представлен новый подход к восстановлению типов данных на основе методов статического и динамического анализа входной программы, позволяющий существенно повысить качество восстановления программ. Кроме того, для обеспечения восстановления всех высокоуровневых конструкций целевого языка существующие методы структурного анализа программ расширены новыми методами, позволяющими восстанавливать оператор множественного выбора `switch` и оператор цикла `for`.

Основные результаты диссертационной работы. В диссертации получены следующие результаты, характеризующиеся научной новизной.

1. Предложен новый подход к декомпиляции программ, позволяющий автоматически восстанавливать все высокоуровневые конструкции целевого языка. Предложен метод количественной оценки полноты восстановления.
2. Предложена модель представления типов данных, позволяющая восстанавливать как базовые, так и производные типы данных, и разработаны алгоритмы восстановления типов данных посредством статического и динамического анализа программы.
3. Для обеспечения полноты восстановления программ расширены существующие методы восстановления структурных конструкций.
4. На основе предложенных методов разработан прототип инструментальной среды декомпиляции TyDec, позволяющий целостно восстанавливать низкоуровневые программы в программы на языке Си, максимально полно использующие высокоуровневые конструкции языка. Проведена экспериментальная проверка прототипной версии инструментальной среды декомпиляции TyDec на наборе программ с открытыми исходными кодами, которая показала полноту восстановления высокоуровневых конструкций целевого языка разработанными алгоритмами и методами декомпиляции.

Все разработанные в рамках работы методы реализованы в инструментальной среде *TyDec*. Инструментальная среда для декомпиляции программ *TyDec* позволяет автоматически корректно и качественно восстанавливать широкий класс программ в низкоуровневом представлении.

Инструментальная среда *TyDec*, являясь диалоговой, позволяет аналитику управлять процессом декомпиляции вручную в случаях, когда не удается автоматически однозначно восстановить высокоуровневые конструкции. Также, являясь расширяемой, инструментальная среда *TyDec* позволяет добавлять новые методы и эвристики, позволяющие разрешать конфликты восстановления, и повышающие качество восстановленной программы. Прототипная версия среды *TyDec* является базовым средством для проведения исследований.

Практическая значимость. Разработанные алгоритмы восстановления типов данных реализованы в инструментальной среде декомпиляции *TyDec*, что позволило восстанавливать все высокоуровневые конструкции языка Си, включая как базовые, так и производные типы данных, оператор множественного выбора `switch`, оператор цикла `for`. Предложенные уточнения правил отображения шаблонов графа потока управления и деревьев доминирования программ на языке низкого уровня в конструкции языка Си позволили в рамках системы *TyDec* реализовать инструмент, который позволяет восстанавливать структурные конструкции языка Си, даже если в программе на низком уровне имелись неструктурные операторы выхода из середины цикла (оператор `break`) и перехода на следующую итерацию цикла (оператор `continue`). Предложены методы восстановления типов данных на основе статического и динамического анализа, позволяющие существенно повысить качество декомпиляции программ. Прототип инструментальной среды декомпиляции *TyDec*, разработанный в рамках представленной работы, позволяет поддерживать унаследованное ПО с утраченными исходными кодами, позволяет восстанавливать структуры данных обмена и протоколы взаимодействия между модулями ПО, а также существенно облегчает задачу исследования ПО с точки зрения информационной безопасности его использования.

Апробация работы. Основные результаты диссертационной работы опубликованы в статьях [1] – [12], в том числе две [1] и [2] – в изданиях рекомендованных ВАК для публикации результатов кандидатских и докторских диссертаций, и представлены в докладах на следующих конференциях и семинарах.

1. Общероссийская научно-техническая конференция «Методы и технические средства обеспечения безопасности информации», Санкт-Петербург, Россия, 7-11 июля 2008 г.
2. Международная конференция «15th Working Conference on Reverse Engineering» (WCRE 2008). Антверпен, Бельгия, 15-18 октября 2008 г.
3. Конференция «РусКрипто», г. Звенигород, Россия, 3-5 апреля 2009 г.
4. Международная конференция «17th International Conference on Program Comprehension» (ICPC 2009). Ванкувер, Канада, 17-19 мая 2009 г.
5. Международная конференция «7th International Andrei Ershov Memorial Conference Perspectives of System Informatics» (PSI 2009). Новосибирск, Россия. 15-19 июня 2009 г.
6. Международный семинар «International Workshop on Program Understanding» (PU 2009). Алтай, Россия. 19-23 июня 2009 г.

Структура и объем диссертации. Диссертация состоит из введения, пяти глав, заключения, списка литературы и приложения. Объем диссертации составляет 134 страницы. Диссертация содержит 17 таблиц и 66 иллюстраций. Список литературы состоит из 81 наименования.

Краткое содержание работы.

Во введении дается неформальное определение декомпиляции программы, приводятся примеры некорректной декомпиляции программ и декомпиляции программ с низким качеством. Формулируются цели и задачи представляемой работы, обосновывается ее актуальность, обсуждается научная новизна и практическая значимость. Приводятся краткий обзор работы.

Глава 1 посвящена рассмотрению декомпиляции как задачи обратной инженерии. К наиболее трудным задачам обратной инженерии относятся задачи анализа программ, исходные тексты, на языке высокого уровня которых отсутствуют, а доступно только низкоуровневое представление, обычно это исполняемый модуль.

Получение ассемблерного представления программы по бинарному модулю называется *дизассемблированием*. При дизассемблировании выполняется трансляция исполняемого файла, содержащего машинные команды и служебную информацию, в программу на языке ассемблера. Обзор методов задачи дизассемблирования представлен в первой главе, однако решение задачи дизассемблирования не является предметом рассмотрения представляемой работы.

Декомпиляция – это процесс *автоматического* восстановления программы на языке высокого уровня из программы на языке низкого уровня. С практической точки зрения задача декомпиляции чрезвычайно сложна в силу того, что при компиляции утрачивается много информации о высокоуровневой программе. Часть информации, например, имена неэкспортируемых объектов, утрачивается безвозвратно. Также утрачивается информация о высокоуровневых конструкциях, используемых типах данных и др., однако эту информацию частично или полностью можно восстановить. На практике задача восстановления программы из низкоуровневого представления в программу на языке высокого уровня зачастую решается посредством привлечения специалиста. Такой подход к восстановлению программ высоко трудозатратный и малоэффективный по времени.

В данной работе восстановление выполняется в язык программирования Си. Программы на языке ассемблера, корректно работающие со стеком и не содержащие данных, интерпретируемых как код и наоборот, являются *правильно построенными* программами. Так как язык Си сочетает в себе высокоуровневые и низкоуровневые возможности программирования, то можно утверждать, что существует отображение, которое любую *правильно построенную* ассемблерную программу непосредственно переводит в программу на языке Си. Однако такое отображение является трансляцией, но не декомпиляцией. Восстановленная таким образом программа будет содержать много низкоуровневых конструкций языка Си, таких как операторы перехода `goto`, явного приведения типов `(type)`, прерывания цикла `break`, прерывания витка цикла `continue`, ассемблерные

вставки и другие. Такое восстановление является корректным, однако уровень представления программы не повышается.

В работе вводится новое требование к результату декомпиляции программы – *качество*. Качество декомпиляции определяет полноту восстановления высокоуровневых конструкций целевого языка. Также в работе вводится понятие *меры качества* декомпиляции, которая позволяет количественно сравнивать полноту восстановления высокоуровневых конструкций целевого языка разными инструментальными средствами декомпиляции.

В таблице 1 представлены штрафы, назначаемые за низкоуровневые конструкции в декомпилированной программе, а также за неполноту восстановления высокоуровневых конструкций языка Си.

Мера качества декомпиляции рассчитывается следующим образом. Пусть исходная программа на языке высокого уровня содержала `K` штрафных баллов. Пусть восстановленная программа содержит `K'` штрафных баллов. Качество восстановления оценивается на некотором тестовом наборе программ, который обозначим `TS`. Пусть `prog` – это исходная программа. Пусть `KLOC(prog)` – это количество тысяч значимых строк кода программы `prog`. Мера качества декомпиляции C_{decomp} вычисляется согласно следующей формуле:

$$C_{decomp} = \sum_{prog \in TS} \frac{\max(0, K' - K)}{KLOC(prog)} \text{ [1].}$$

Чем ближе значение меры к нулю, тем выше качество декомпиляции.

Таблица 1. Штрафы за артефакты трансляции и неполноту восстановления.

конструкции программы	назначаемые штрафы
операция явного приведения типов `(type)`	2
оператор перехода `goto`	3
оператор выхода из середины цикла `break`	1
оператор прерывания витка цикла `continue`	1
тип данных объединение `union`	3
ассемблерная инструкция	4
невосстановление оператора `switch`	2
невосстановление оператора `for`	1
невосстановление производного типа данных	4
использование адресной арифметики вместо оператора доступа к элементу массива	2

Задачу декомпиляции предлагается рассматривать как совокупность трех задач:

1. восстановление функционального интерфейса,
2. восстановление управляющих конструкций,
3. восстановление переменных и типов данных.

В первой главе рассмотрен класс входных ассемблерных программ, для которых все задачи декомпиляции решаемы с приемлемым уровнем качества. Такой класс ассемблерных программ определяется наличием программ-прообразов во множестве Си-программ, строго удовлетворяющих стандарту,

отображаемых в программы на языке низкого уровня. В качестве такого отображения рассматривается компиляция.

В заключении первой главы представлена постановка задачи диссертационной работы.

В **главе 2** в первой части представлен обзор методов решения всех составляющих задачи декомпиляции. Обзор задачи восстановления функционального интерфейса программы начинается с обзора методов обнаружения точки входа пользовательского кода, то есть функции `main`. Далее рассматривается восстановление библиотечных функций и системных вызовов для программ, скомпонованных как динамически, так и статически. Также рассматривается восстановление функций с учетом возможных оптимизаций компилятора и использования разных стандартных соглашений о вызовах.

Также во второй главе рассматриваются методы решения задачи восстановления управляющих конструкций языка высокого уровня. Анализ графа потока управления для выявления структурных конструкций выполняется и при компиляции программ для оптимизации циклов, распараллеливания и т.д. Такие методы можно использовать как основу при восстановлении структурных конструкций при декомпиляции. Однако задача структурного анализа при компиляции проще, нежели при декомпиляции. Наличие конструкций, нарушающих структурность программы, таких как операторы выхода из середины цикла, операторы прерывания витка цикла в задаче компиляции только препятствуют выполнению некоторых оптимизационных преобразований, тогда как при декомпиляции они могут повлечь затруднения при восстановлении структурной конструкции в целом.

Анализ графа потока управления при декомпиляции описан в работах К. Цифуентес (C. Cifuentes). Предложенный ею метод основан на выделении в графе потока управления регионов, где регион – это набор базовых блоков, имеющий только одну входную и только одну выходную дуги. После чего размеченный на регионы граф потока управления перестраивается в модифицированный граф, где, в отличие от исходного графа, узлами являются не базовые блоки, а регионы. После того, как в графе выделены регионы, вычисляется отношение доминирования, а дуги классифицируются на прямые, обратные и косые. Далее на модифицированный граф потока управления накладываются шаблоны[1], соответствующие восстанавливаемым структурным конструкциям программы: `if-then`, `if-then-else`, `while`, `do-while`. Циклом считается множество вершин, доступных из данной по обратному ребру и удовлетворяющих дополнительным условиям, определяющим тип цикла. Метод работает итеративно. Восстановленная структурная конструкция преобразовывается в новый регион, после чего на обновленный граф снова накладываются шаблоны структурных конструкций. Выполнение завершается, когда на очередном этапе ни один из структурных шаблонов уже не подходит.

[1] Шаблон – это подграф графа потока управления, соответствующий представлению управляющих конструкций целевого языка.

Учитывая требования полноты восстановления высокоуровневых конструкций, которое предъявляется к результату декомпиляции, предложенный К. Цифуентес метод не позволяет решить задачу структурного анализа с приемлемым уровнем качества.

Программы на языке высокого уровня оперируют с типизированными переменными, а не с ячейками памяти, адресуемыми по абсолютным или относительным адресам. При декомпиляции инструкции работы с ячейками памяти и регистрами процессора должны быть отображены в выражения над переменными.

Существующие декомпиляторы зачастую ограничиваются отображением ячеек памяти и регистров в символические имена, но не восстанавливают типы данных. По сути, низкоуровневая программа декомпилируется в нетипизированный или слаботипизированный язык. Если типы аргументов могут быть восстановлены по инструкциям процессора, то в декомпилированной программе в соответствующее место добавляются операции преобразования типов. Поэтому при декомпиляции не происходит качественного перехода от нетипизированного языка ассемблера, в котором типы аргументов определяются выполняемой инструкцией, к типизированному языку высокого уровня, в котором типы задаются при определении переменных.

Можно сказать, что задача автоматического восстановления типов данных на настоящее время – одна из наименее проработанных с теоретической точки зрения задач в области декомпиляции. Полнота восстановления типов данных существенно повышает качество декомпиляции. Задачу восстановления типов данных условно можно разделить на подзадачи восстановления

1. **базовых типов данных** языка, таких как `char`, `unsigned long` и т. п., и
2. **производных типов данных**, таких как типы структур, массивов и указателей.

Во второй главе работы рассматриваются некоторые существующие теоретические подходы к восстановлению типов данных.

Один из подходов к восстановлению типов заключается в использовании методов математической логики. Подход впервые был предложен А. Майкрофтом (A. Mycroft). Программа рассматривается как константное выражение над термами типов данных. Компилятор в процессе семантического анализа программы вычисляет это выражение. Предложенный метод основан на построении термов, описывающих типы переменных, и применении к ним правил преобразования термов (унификации) в соответствии с текстом программы. В случае, когда одному шаблону соответствует несколько возможных вариантов реконструкции типов, в системе типов используется дизъюнктивное ограничение.

Алгоритм опускает из рассмотрения восстановление локальных переменных, сохраненных на стеке и регистрах. Также алгоритм не поддерживает восстановление знаковости типов. В случае конфликтов создаются объединения (union). Как следствие, программа восстанавливается с низким уровнем качества. Конфликтующие термы не удаляются из рабочего множества, что может приводить к его быстрому росту, делая алгоритм неэффективным по времени и

памяти. Размер рабочего множества не ограничен и, как следствие, во многих случаях алгоритм не сходится. Следовательно, предложенный А. Майкрофтом метод не применим для автоматического восстановления типов данных.

Другой подход к задаче анализа бинарной программы представлен в работах Г. Балакришнана (G. Balacrishnan), которые посвящены методам статического выявления значений переменных в исполняемых файлах. Предложенные им методы основаны на интервальном анализе. Для необходимых переменных восстанавливаются размер производных типов данных и размещение в памяти полей структурных типов. Алгоритм базируется на интуитивном предположении, что шаблоны обращения к памяти в программе дают информацию о том, как располагаются данные. Следует заметить, что, поскольку задачей алгоритма является поиск уязвимостей, а не восстановление типов данных, в нем не предусмотрено объединение раздельных использований одного и того же структурного типа, что неприемлемо для задачи восстановления типов данных. Кроме того, в методе не предусмотрено восстановление базовых типов полей структур и элементов массивов.

Еще один подход к восстановлению типов данных в задаче декомпиляции предложен в работе М. Ю. Гусенко. В качестве операций над типизированными данными рассматриваются инструкции процессора, а в качестве значений типизированных данных рассматриваются значения примитивов, которыми они оперируют. Каждый тип характеризуется именем и длиной. В работе предложены методы восстановления следующих типов данных: числовой, указатель, структура (struct), объединение (union) и массив.

Представленный подход достаточно эффективен в реализации, однако имеет ряд существенных недостатков, которые приводят к тому, что типы данных восстанавливаются неполно. Следствием этого является то, что восстанавливаемая программа имеет низкое качество декомпиляции. В представленном методе не уделено внимание проблеме восстановления знаковости целочисленных типов данных. Можно утверждать, что данный метод не позволит восстановить указатели, имеющие более одного уровня косвенности, то есть указателей на указатель, так как за один проход восстановить всю глубину косвенности затруднительно. Также можно утверждать и то, что представленный метод не позволит восстановить рекурсивно-определенный структурный тип. Следовательно, представленный метод также не позволяет решать задачу восстановления типов данных с высоким уровнем качества.

Следует также отметить, что в работах К. Цифуентес восстановление типов переменных практически не рассматривается.

На практике все декомпиляторы, кроме расширения Hex-Rays к интерактивному дизассемблеру Ida Pro, вообще не восстанавливают даже базовые типы переменных, а в выражениях используют явное приведение типов, что делает декомпилированные программы сложными для понимания и модификации.

Существуют случаи, для которых статический анализ не позволяет восстановить типы данных однозначно. Один из возможных вариантов

разрешения такого рода неоднозначностей – запросить дополнительную информацию у пользователя, другой – использовать информацию, собранную в процессе наблюдения за работой декомпилируемой программы. Во второй главе представлен обзор методов и инструментальных средств, которые собирают и анализируют информацию времени выполнения программы, в частности представлен подробный обзор системы Valgrind. Система Valgrind позволяет писать расширения на основе общего ядра, позволяющие анализировать бинарные приложения во время их работы. Особенностью системы являются развитая функциональность, устойчивость приложений к ошибкам времени выполнения, высокая производительность за счет минимизации количества операций.

Во второй части второй главы дается краткое описание существующих на данный момент декомпиляторов. Это – декомпилятор Boomerang, декомпилятор DCC, декомпилятор REC и плагин Hex-Rays к интерактивному дизассемблеру Ida Pro. Все рассматриваемые декомпиляторы, кроме плагина Hex-Rays, на вход принимают исполняемый файл, а на выходе выдают программу на языке Си. Для сравнения функциональных возможностей декомпиляторов был разработан тестовый набор на языке Си, покрывающий основные языковые конструкции языка.

Ни один существующий на данный момент декомпилятор не поддерживает в достаточной мере восстановление типов данных, как базовых, так и производных. Существующие декомпиляторы испытывают сложности с восстановлением цикла `for` и оператора `switch`. Наиболее развитым в настоящее время является декомпилятор Hex-Rays, который, в отличие от других декомпиляторов, поддерживает распознавание массивов и распознавание библиотечных функций, хотя даже декомпилятор Hex-Rays имеет много слабых сторон. Декомпилятор Hex-Rays является коммерческим и с закрытыми исходными кодами, поэтому его доработка невозможна.

Глава 3 посвящена описанию моделей восстановления типов данных в задаче декомпиляции, разработанных в рамках диссертационной работы и построенным на их основе алгоритмам.

Модели восстановления типов данных оперируют объектами. Объекты строятся из конструкций ассемблерной программы, в которые отображаются переменные исходной программы, а именно из:

1. регистров общего назначения центрального процессора,
2. непосредственно адресуемых областей памяти:
    1) ячеек памяти по абсолютным адресам, которые соответствуют глобальным переменным в исходной программе,
    2) ячеек памяти по фиксированным смещениям относительно стекового кадра, соответствующим локальным параметрам,
    3) ячеек памяти по фиксированному смещению относительно вершины стека, а также занесенных на стек соответствующими инструкциями. Они соответствуют фактическим параметрам в вызываемых функциях.

3. косвенно-адресуемых областей памяти, что соответствует разыменованию других объектов, которые возникают в результате выполнения операций доступа к памяти в ассемблерном коде.

Для каждого регистра общего назначения центрального процессора, локальной переменной и параметра предварительно строится двудольный граф «определения-использования», где в левой доле находятся определения (то есть присваивание значений) соответствующих регистров, локальных переменных или параметров, а в правой – использования. Для каждой компоненты связности графа строится один объект, таким образом, одному регистру, локальной переменной или параметру могут соответствовать несколько объектов.

Для восстановления типов данных в представляемой работе разработано две модели: модель восстановления базовых типов данных (`char`, `unsigned char`, `short int`, `int`,...,`float`) и модель восстановления производных типов данных (массивы, структуры, указатели произвольного уровня косвенности).

Основная задача, которая должна быть решена при восстановлении **базовых типов данных**, это определить,

1. имеет ли объект указательный, вещественный или целый тип,
2. для объектов целого или вещественного типа определить размер типа в байтах,
3. для объектов целого типа определить, знаковый ли это тип данных или беззнаковый.

Все **базовые типы** языка Си отображаются в модельные типы, которые представляются в виде тройки <*core, size, sign* >. *Ядро(core)* может принимать значение указательный (pointer), целый (int) или вещественный (float). *Размер (size)* может принимать значение 1, 2 или 4 (для 32-битной архитектуры). *Знак (sign)* может принимать значение «знаковый» (signed) или «беззнаковый» (unsigned). Множество модельных типов – это множество троек, полученных отображением базовых типов языка Си, следовательно, оно не содержит троек, которые не соответствуют ни одному базовому типу языка. В процессе восстановления типов данных для каждого объекта строится модельный тип. Для этого вводится понятие *обобщенного модельного типа* данных. О*бобщенный модельный тип* данных – это тройка множеств <*CORE, SIZE, SIGN*>, где

$$\text{CORE} \subseteq \{\text{pointer, int, float}\},$$
$$\text{SIZE} \subseteq \{1, 2, 4\} \text{ и}$$
$$\text{SIGN} \subseteq \{\text{signed, unsigned}\}.$$

Изначально каждый объект характеризуется обобщенным модельным типом, каждый атрибут которого – полное множество, то есть:

<{pointer, int, float}, {1,2,4}, {signed, unsigned}>

Далее по ассемблерной программе строятся три системы уравнений типов для каждого из атрибутов модельного типа. Например, по ассемблерной инструкции $add\ r_1, r_2, r_3$, которая складывает регистры r_1, r_2 и помещает результат в регистр r_3, для атрибута *sign* строится уравнение $obj_3 :< sign_3 >= obj_1 < sign_1 > + obj_2 < sign_2 >$, где

объекты obj_1, obj_2, obj_3 построены по регистрам r_1, r_2, r_3 соответственно, а $< sign_1 >, < sign_2 >, < sign_3 >$ – это атрибуты обобщенных восстанавливаемых типов объектов obj_1, obj_2, obj_3 соответственно. Каждая система решается итеративным алгоритмом, который основан на продвижении значений. Сбор информации по всем атрибутам выполняется независимо от значений остальных атрибутов тройки.

Начальные значения для каждого атрибута объекта уточняются ограничениями, накладываемыми регистрами процессора, ассемблерными инструкциями и т.д.

Определены пять типов ограничений:

- *регистровое ограничение*, оно влияет на атрибут *core* и атрибут *size* типа объекта,
- *командное ограничение*, оно влияет на все атрибуты типа объекта,
- *флаговое ограничение*, оно влияет на атрибут *sign* соответствующего типа объкта и
- *ограничение окружения*, оно влияет на все три атрибута типа объекта. Это ограничение накладывается, если в исходной программе использовались стандартные функции. Типы параметров и возвращаемых значений стандартных библиотечных функций предполагаются известными.
- *ограничение профиля* является дополнительным. Это ограничение предоставляет информацию о том, что тип объекта не является указательным или не является знаковым. Ограничения профиля строятся на основе результатов динамического анализа программы, описание которого представлено в главе 4 работы.

Продвижение значений атрибутов выполняется на основании правил работы с типами данных, зафиксированных в стандарте языка Си. Для вычисления атрибутов используется решетка свойств с монотонной невозрастающей функцией слияния (пересечение множеств). На рис. 1 представлен пример уравнения системы уравнений типов для атрибута *sign*.

$$(1) < sign_p > \{unsigned\} + \qquad (2) < sign_v > \{signed, unsigned\} +$$
$$(1) < sign_q > \{signed, unsigned\} = \qquad (2) < sign_q > \{signed\} =$$
$$(1) < sign_r > \{signed, unsigned\} \qquad (2) < sign_r > \{signed, unsigned\}$$

Рис. 1. Пример операции объединения для атрибута `знак`.

На рис. 1 показаны 2 уравнения. В уравнении (1) тип объекта obj_p беззнаковый, о знаковости типа объекта obj_q нет информации, он может быть как знаковый, так и беззнаковый, но согласно стандарту языка Си тип объекта obj_r должен быть беззнаковый. В уравнении (2) из уравнения (1) уже известно, что тип объекта obj_r

беззнаковый, следовательно, согласно стандарту, тип объекта obj_v тоже должен быть беззнаковый, так как тип объекта obj_q знаковый согласно уравнению (2).

Нахождение решения всех систем уравнения выполняется до достижения неподвижной точки. Далее по найденному обобщенному модельному типу выполняется построение модельного типа, в свою очередь по которому находится соответствующий базовый тип языка Си.

Модель восстановления **производных типов данных** основывается на представлении адресов обращения к памяти в канонической форме: $(base + offset + \sum_{j=1}^{n} C_j x_j)$, где *base* – это база, то есть базовый адрес, который является объектом при восстановлении базовых типов, *offset* – это константное смещение, значение которого известно при статическом анализе, $\sum_{j=1}^{n} C_j x_j$ – мультипликативная составляющая. При условии, что исходная Си программа строго удовлетворяет стандарту, *base* имеет указательный тип. Для каждой инструкции обращения к памяти во входной программе на языке ассемблера, за исключением инструкций обращения к локальным переменным, глобальным переменным и параметрам функций, строится локальное адресное выражение адреса обращения к памяти в данной инструкции. Например, инструкции `movl 12(%ebx), %ecx` соответствует локальное адресное выражение `%ebx + 12`. Для аргументов локального адресного выражения выполняется прослеживание значений в обратном направлении либо до загрузки из памяти значения, либо до вычисления, отличного от сложения или умножения, либо до границы базового блока. Найденные выражения для аргументов подставляются в локальное адресное выражение, и после алгебраических преобразований строится полное адресное выражение, которое отображается в терм адресного выражения. Таким образом, полным адресным выражениям доступа к памяти во входной программе соответствуют термы адресных выражений в модели восстановления производных типов.

Все множество термов адресных выражений разделяется на множества термов с эквивалентными базами. Каждое такое множество соответствует одному производному типу данных. Для разделения множеств термов на множества с эквивалентными базами используется алгоритм прямого продвижения: сначала, каждый объект, соответствующий базе, помечается меткой. Далее выполняется прямое продвижение меток по входной программе в соответствии с правилами переписывания термов. Все термы с одинаковыми метками объединяются в одно множество, которое соответствует классу эквивалентности. В результате может оказаться, что количество полученных классов эквивалентности больше количества производных типов в исходной Си-программе. Это означает, что для статического анализа было недостаточно информации для точного определения классов эквивалентности баз. Однако количество различных множеств эквивалентных баз производных типов данных не меньше количества

производных типов в исходной программе. Неточности восстановления статического анализа частично устраняются с помощью динамического анализа, который представлен в четвертой главе диссертационной работы.

После того, как выполнено разделение множества термов адресных выражений на классы эквивалентных баз, для каждого класса эквивалентности собирается множество смещений. Обратное отображение модельного представления производного типа в тип языка Си выполняется посредством построения скелета производного типа для каждого класса эквивалентности. Смещения по всем базам отображаются в поля скелета типа. По множеству смещений в одном классе эквивалентности строится скелет структурного типа, в котором типы полей вычисляются с помощью алгоритма восстановления базовых типов. На рис. 2 представлена демонстрация работы алгоритма восстановления производных типов данных.

```
[1] struct t {
[2]   int of1;
[3]   short of2;
[4]   double of3;
[5] }
[6]
[7] void f(void) {
[8]   struct t * tmp1, *tmp2, *tmp3;
[9]   tmp1=tmp2;
[10]  tmp1->of1;
[11]  tmp2->of2;
[12]  tmp3=tmp2;
[13]  tmp3->of3;
[14] }
```

Рис. 2. Пример работы алгоритма восстановления производных типов данных.

Объекты, соответствующие переменным `struct t * tmp1, *tmp2` и `*tmp3`, помечаются метками L1, L2 и L3 соответственно. Все три метки оказываются в одном классе эквивалентности баз после применения прямого продвижения меток. Операция присваивания в строке 9 определяет эквивалентность объектов, помеченных метками L1 и L2. Операция присваивания в строке 12 определяет эквивалентность объектов, помеченных метками L2 и L3. Далее строится множество смещений для нового структурного типа `S`. В соответствии со строками 10, 11 и 13 множество смещений для структуры `S` следующее: `{of1, of2, of3}`. На рис. 3 представлен скелет восстанавливаемой структуры `S`.

```
struct S{
      type1 of1;
      type2 of2;
      type3 of3;
}
```

Рис. 3. Пример скелета восстанавливаемой структуры `S`.

Массивы восстанавливаются аналогичным способом. Если все смещения по эквивалентным базам имеют одну и ту же мультипликативную составляющую, то восстанавливаемый производный тип объявляется массивом. Для дальнейшего восстановления строится объект для первого элемента массива. Размер элемента массива вычисляется как НОД всех мультипликативных составляющих. В некоторых случаях оказывается возможным восстановить размер самого массива. На рис. 4 представлен пример восстановления массива структур.

```
arr[i].f2 => arr + 4 + 8 * i;
arr[j].f1 => arr + 0 + 8 * j;
```

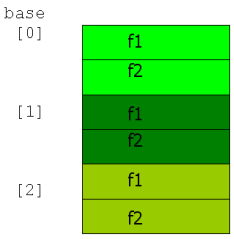

Рис. 4. Пример восстановления массива структур.

Массив `arr` состоит из элементов, тип которых – структура, состоящая из полей `f1` и `f2`. В исходной программе были инструкции обращения к полю `f2` `i`-го элемента массива и полю `f1` `j`-го элемента массива. На рис. 4 представлены выражения в канонической форме, соответствующие этим инструкциям. Размер элемента массива составляет 8, однако разность между смещениями `f2` элемента `arr[i]` и `f1` элемента `arr[j]` составляет $|4 - 0| = 4 < 8$, что меньше размера элемента массива, и, следовательно, тип элементов массива `arr` – структура.

В **главе 4** представлены разработанные методы повышения точности статического восстановления типов данных посредством анализа информации времени выполнения программы. Повышение точности восстановления типов данных способствует повышению качества декомпиляции программы. Информация времени выполнения может использоваться в статических алгоритмах восстановления типов данных с целью более точного и качественного выполнения восстановления в тех случаях, когда только статической информации недостаточно. Например, при декомпиляции неполной программы переменная указательного типа может не разыменовываться в восстанавливаемом фрагменте программы, в этом случае только статического анализа недостаточно, чтобы отличить указательный тип от целого в силу того, что сгенерированный код на языке ассемблера в обоих случаях идентичный. Однако различать указательный тип и целый важно для понимания программы. Аналогичная трудность возникает со знаковостью типа. Также информация времени выполнения используется для повышения точности восстановления производных типов данных. Профилирование кучи позволяет уточнять разделение баз адресных выражений на множества эквивалентности, что позволяет избежать появления клонов структурных типов данных в восстановленной программе.

В корректной программе на языке Си указатели, как правило, хранят адреса из множества отображенных виртуальных адресов (стек, куча, статические

данные, код), либо специальное значение *null*. Значения, которые принимают знаковые целочисленные переменные, сгруппированы около нуля. То есть, например, значение -2 знаковая целочисленная переменная принимает значительно чаще, нежели значение $2*10^9$. С другой стороны беззнаковые целочисленные переменные для 32-х битной архитектуры редко принимают значение, соответствующее -2 в дополнительном коде. Значение -1 необходимо рассматривать специальным образом, так как оно соответствует максимальному беззнаковому целому значению и часто выступает в роли индикатора ошибки[2].

Для 32-х битной архитектуры достаточно профилировать значения только 32-битных ячеек памяти, так как указатели имеют размер 32-бита. Непосредственно профилирование значений переменных реализовать затруднительно, так как:

- значения разных переменных могут храниться по одним и тем же адресам. Ячейки стека, как и кучи, постоянно повторно используются,
- одни и те же локальные переменные могут иметь различный адрес в разные моменты времени.

Однако на многих архитектурах, в частности, на IA32, для того чтобы использовать значение ячейки памяти, ее необходимо загрузить на регистр. Следовательно, вместо профилирования непосредственно значений ячеек памяти, можно собирать профили значений, которые загружаются на регистры и выгружаются из регистров соответствующими инструкциями процессора.

Отображение *VP* – это отображение из множества *Addr* виртуальных адресов сегмента кода программы во множество 32-битных значений *Val*, которые записывались или считывались инструкцией по этому адресу. Множество адресов, которые не принадлежат сегменту кода программы, исключается. Отображение *DM* – это отображение из множества *Addr* виртуальных адресов памяти во множество объектов *Loc*, соответствующих этим адресам памяти в дизассемблированной программе. Отображение *IVP* из множества ассемблерных инструкций во множество 32-битных значений, которые загружаются на регистр или выгружаются в память соответствующими инструкциями, строится посредством комбинирования отображений *VP* и *DM:*

$$\begin{aligned} &VP : Addr \to 2^{Val} \\ &DM : Addr \to Loc \\ &IVP : Loc \to 2^{Val} \\ &IVP(x) = VP(DM^{-1}(x)) \end{aligned} .$$

В результате каждой инструкции дизассемблированной программы ставится в соответствие множество 32-битных значений, которое она загружает из памяти или выгружает в память. Инструкции, работающие не с 32-битными значениями, или которые не выполнялись при профилирующих запусках программы, остаются неаннотированными.

[2] Эвристики составлены на основании результатов исследования значений, принимаемых переменными различного типа данных, которые опубликованы в работе Я. Сазейда (Y. Sazeides) и Дж. Смита (J. Smith) «Предсказуемые значения данных»//Материалы конференции Micro-3, 1997, стр. 749-754.

Множество *VP(obj)* – это множество значений, принимаемых объектом `obj` за время выполнения программы. *ValidAddr* – это множество значений, которое может принимать переменная указательного типа. Для разделения целочисленных и указательных типов предложена метрика *PC(obj)*, для уточнения знаковости типа предложена метрика *UC(obj)*. На рис. 5 представлены формулы для вычисления этих метрик.

$$PC(obj_i) = \frac{|VP(obj_i) \cap ValidAddr|}{|VP(obj_i) - \{null\}|} \quad [2]$$

$$UC(obj_i) = 1 - \sum_{x=-2^{31}+1}^{-2} \frac{\sigma_{obj_i}(x)}{2^{2*\lfloor \log_2|x+1| \rfloor + 1}} \quad [3]$$

$$\sigma_{obj_i}(x) = 1, \text{ если } x \in VP(obj_i), \text{ и } \sigma_{obj_i}(x) = 0, \text{ если } x \notin VP(obj_i)$$

Рис. 5. Метрики для повышения точности восстановления базовых типов данных.

В этой же главе представлены алгоритмы сбора профиля анализируемой программы и обработки собранного профиля. Представленные алгоритмы реализованы в утилите *TDTrace*, особенности реализации которой представлены в главе 5. Утилита позволяет анализировать адреса и значения, используемые во время выполнения программы, и выдаёт результат в формате XML-документа, который подгружается по запросу инструментальной среды декомпиляции TyDec.

Для повышения точности восстановления производных типов данных используется динамический анализ состояния кучи программы, который позволяет собрать информацию о связях между блоками динамической памяти.

Для выделения памяти в куче под структуры данных используются функции семейства malloc, которые перехватываются и отслеживаются во время исполнения анализируемой программы. Для каждого блока памяти строится множество адресов в коде программы, из которых производился доступ к нему. В это множество не включаются инструкции обнуления и копирования памяти. После завершения работы программы проводится анализ собранной информации для выделения множеств блоков памяти, содержащих структуры данных одного типа.

В главе представлены алгоритм сбора профиля кучи программы, алгоритм анализа собранного профиля, который включает алгоритм распознавания структурных типов, массивов и связей между блоками памяти. По наличию связей между блоками памяти можно восстановить информацию о том, что у структурного типа полями были указательные типы, также алгоритм поддерживает распознавание рекурсивных типов данных. Представленные алгоритмы реализованы в утилите *TDHeap*, которая реализована с помощью среды Valgrind. Описание реализации утилиты представлено в главе 5.

Результат работы утилиты *TDHeap* предоставляется в формате XML-документа, который подгружается по запросу инструментальной среды декомпиляции TyDec.

Глава 5 посвящена описанию инструментальной диалоговой среды декомпиляции программ в язык Си – декомпилятору TyDec. Декомпилятор поддерживает несколько входных форматов: программы на языке ассемблера для архитектуры IA32 в синтаксисе AT&T и Intel, исполняемые модули, бинарные трассы выполнения, собранные на симуляторе процессора. Декомпилятор имеет развитый графический интерфейс аналитика для управления процессом декомпиляции. Результатом работы декомпилятора является типизированная структурная программа на языке Си. Схема архитектуры декомпилятора представлена на рис. 6.

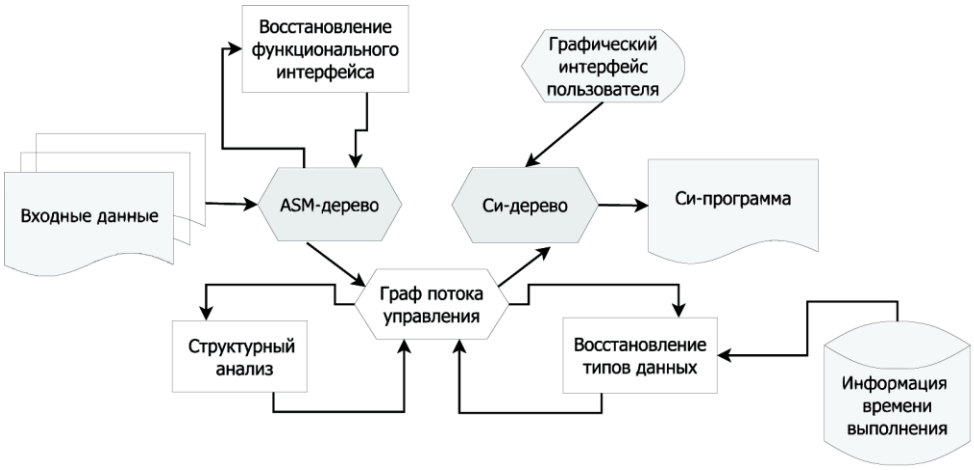

Рис. 6. Схема архитектуры декомпилятора TyDec.

Многоугольниками представлены компоненты декомпилятора, а стрелками отображается поток данных между ними.

Множество входных форматов: текстовый файл с ассемблерными инструкциями в нотации AT&T или Intel, исполняемые файлы, трассы, снятые симулятором AMD SimNow или другими инструментальными средствами.

ASMTree – это внутреннее представление входной программы в виде последовательности инструкций. Модуль *ASMTree* отвечает за построение и хранение дерева внутреннего представления программы и предоставляет интерфейс для работы с ним другим компонентам системы.

Модуль восстановления функций преобразует внутреннее представление программы, выявляя в нем функции.

Модуль CFG выполняет построение графа потока управления.

Модуль восстановления структурных конструкций программы преобразует граф потока управления, восстанавливая управляющие конструкции программы.

CTree – это компонент, обеспечивающий работу декомпилятора с внутренним представлением восстанавливаемой программы на языке Си в виде синтаксического дерева. Модуль строит текст программы на языке Си по внутреннему представлению с использованием результатов работы модулей восстановления типов данных и структурных конструкций.

Модуль GUI реализует графический интерфейс декомпилятора и поддержку диалогового режима работы эксперта.

Модуль восстановления типов данных отвечает за восстановление типов данных декомпилируемой программы. В качестве входных данных используется внутреннее представление программы (граф потока управления). По нему строится система уравнений типов данных. Далее применяется алгоритм восстановления типов данных. Информация, полученная при динамическом анализе программы, подгружается непосредственно в модуль восстановления типов данных из файлов, в которые сохраняется при профилировании программы дополнительными утилитами, встроенными в инструментальную среду декомпиляции TyDec.

Представленные в работе методы позволяют решать задачу декомпиляции целостным образом от начала до конца, так как разработанные и реализованные методы позволяют решить все подзадачи декомпиляции. В **главе 5** представлены особенности реализации *восстановления функционального интерфейса программы* на основе стандартных соглашений о вызовах, также описаны методы обнаружения точки входа в пользовательскую программу (функции main). Методы структурного анализа, традиционно используемые при компиляции, существенно расширены и адаптированы с учетом особенностей декомпиляции как преобразования, обратного к компиляции. Поддерживается полное восстановление управляющих конструкций языка Си, включая не только операторы цикла `while` и `do-while` и оператор ветвления `if`, но и оператор множественного выбора `switch`, а также оператор цикла `for`. Восстановление типов данных выполнено на основе представленных моделей.

На рис. 7 представлены примеры восстановления программ расширением Hex-Rays к интерактивному дизассемблеру IdaPro. Меры качества декомпиляции для Hex-Rays, рассчитанные по формуле (1) с учетом назначаемых штрафов, представленных в таблице 1, для примера 1 и для примера 2 соответственно равны:

$$C_{decomp} = \frac{\max(0,11*4-0)}{6*10^{-3}} = \frac{44}{6*10^{-3}} = 7333; \text{ и } C_{decomp} = \frac{\max(0,9*2-0+1)}{7*10^{-3}} = \frac{19}{7*10^{-3}} = 2714;.$$

Соответствующие меры качества декомпиляции для декомпилятора *TyDec* для тех же примеров нулевые.

Тестирование работы инструментальной среды TyDec было произведено более чем на 100 примерах, пятая часть которых это программы с открытым исходным кодом. На всех примерах инструментальная среда показала высокое качество восстановления низкоуровневых программ. На рис. 8 представлен пример работы инструментальной среды TyDec для примера 1 и примера 2, которые представлены на рис. 7.

Экспериментальное сравнение качества восстановления программ декомпилятором *TyDec* и декомпилятором *HexRays* проводилось на 7 программах, 6 из которых – это утилиты с открытым исходным кодом, а последняя программа – это утилита, которая вычисляет значение производной многочлена в точке.

Пример 1	Результат декомпиляции	Пример 2	Результат декомпиляции

include <stdio.h>
 	
void f
(float a, float b)
{
  if (a < b) {
   printf("a < b\n");
  }
  if (a > b) {
   printf("a > b\n");
  }
return;
}
	__int16 __cdecl
	f(float a1, float a2)
	{
	  __int16 result; // ax@3
	  __asm
	{
	  fld [ebp+arg_0]
	  fld [ebp+arg_4]
	  fucompp
	  fnstsw ax
	  sahf
	}
	if (!(_CF | _ZF))
	  puts("a < b");
	  __asm
	{
	  fld [ebp+arg_0]
	  fld [ebp+arg_4]
	  fxch st(1)
	  fucompp
	  fnstsw ax
	  sahf
	}
	if (!(_CF | _ZF))
	  result = puts("a > b");
	return result;
	}
		void f2
		(signed short x,
		 unsigned short y)
		{
		unsigned short
		  i = 0;
		 for (i = 0;
		  i < x;
		  i++)
		 {
		  y += 3;
		  g(y);
		 }
		return;
		}
			int __cdecl
			sub_401314
			(int a1, int a2)
			{
			int result; // eax@2
			int v3;
			// [sp+14h] [bp-4h]@1
			signed __int16 v4;
			// [sp+12h] [bp-6h]@1
			HIWORD(v3) = a1;
			LOWORD(v3) = a2;
			v4 = 0;
			while (1){
			 result = SHIWORD(v3);
			 if((unsigned __int16)v4>=
			  (signed int)SHIWORD(v3))
			 break;
			 LOWORD(v3)=(_WORD)v3 + 3;
			 sub_401290((unsigned
			__int16)v3);
			 ++v4;
			}
			return result;
			}

Рис. 7. Примеры восстановления программ с невысоким уровнем качества.

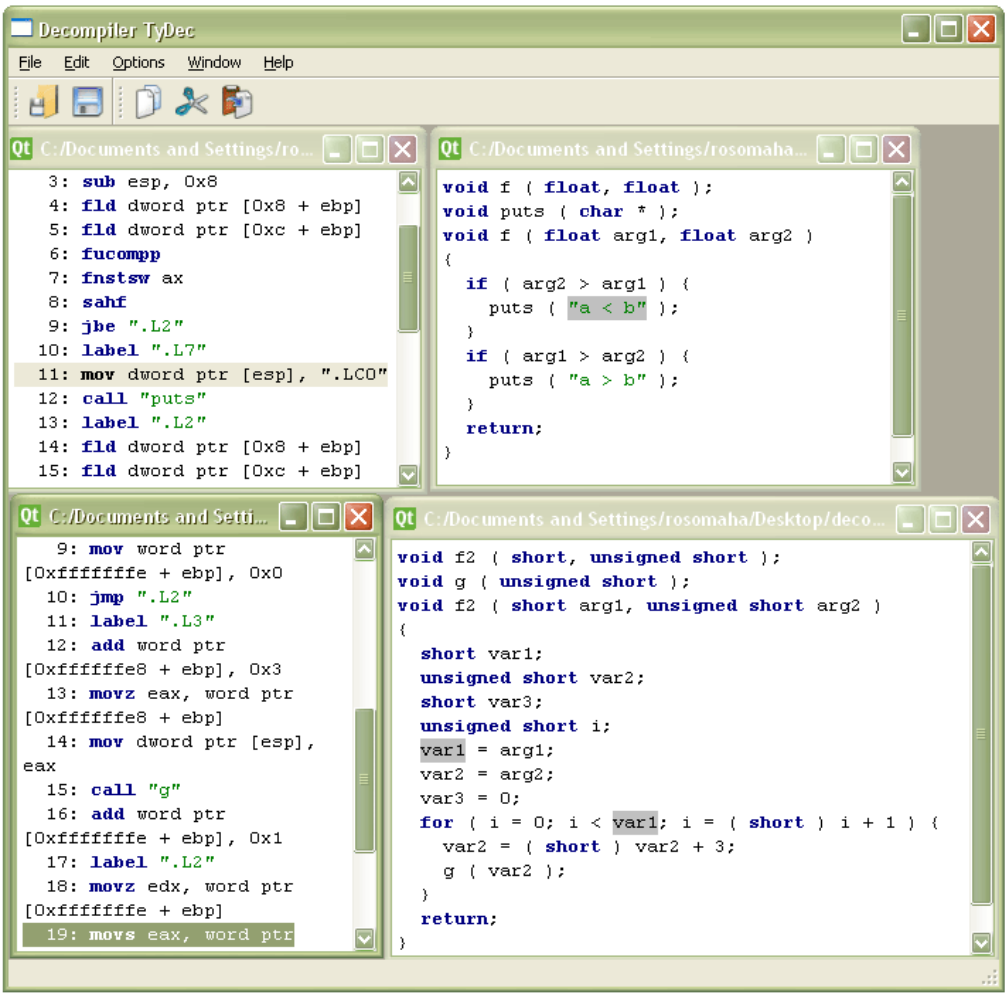

Рис. 8. Пример работы инструментальной среды TyDec.

В таблице 2 представлены характеристики программ, для которых выполнялось сравнение качества восстановления. Колонка «CLOC» содержит количество строк кода в исходной программе, колонка «ALOC» содержит количество строк кода в ассемблерной программе.

Таблица 2. Описание программ, для которых выполнялось сравнение качества восстановления.

Название	CLOC	ALOC	Описание
35_wc	241	465	Утилита wc, файл «wc.c»
36_cat	262	618	Утилита cat, файл «cat.c»
37_execute	726	1837	Утилита bc, файл «execute.c»
38_day	503	1383	Утилита calendar, файл «day.c»
39_deflate	403	669	Утилита gzip, файл «deflate.c»
59_lalr	674	1664	Утилита yacc, файл «lalr.c»
84_derive	71	255	Утилита derive, файл «derive.c»

Таблица 3 показывает процент восстановления исходного кода для декомпилятора *TyDec* и декомпилятора *HexRays* соответственно. Следует отметить, что программа `36_cat` и программа `37_execute` не были полностью восстановлены декомпилятором *HexRays* из-за наличия оператора `switch`, восстановление которого он не поддерживает в полном объеме. В таблице 4 представлен подсчет количества штрафных баллов за использование низкоуровневых конструкций при восстановлении, а также за невосстановление высокоуровневых конструкций языка Си в соответствии с таблицей 1. Колонки «OR» содержат количество штрафных балов у исходных программ, колонки «TD» – количество штрафных балов у программ, восстановленных декомпилятором *TyDec*, и колонки «HR» – количество штрафных балов у программ, восстановленных декомпилятором *HexRays*.

Таблица 3. Процент восстановления программы декомпиляторами.

Название	TyDec	HexRays
35_wc	100%	100%
36_cat	100%	**84%**
37_execute	100%	**0%**
38_day	100%	100%
39_deflate	100%	100%
59_lalr	100%	100%
84_derive	100%	100%

Вычисление качества восстановления программ обоими декомпиляторами в соответствии с формулой [1] представлено в таблице 5. Как можно заметить, на всех примерах качество восстановления программ декомпилятора *TyDec* существенно выше, чем качество восстановления программ декомпилятора *HexRays*.

Таблица 4. Вычисление количества штрафных баллов.

	35_wc			36_cat			38_day			39_deflate			59_lalr			84_derive		
	OR	TD	HR	OR	TD	HR	OR	TD	HR	OR	TD	HR	OR	TD	HR	OR	TD	HR
(type)	1	1	32	0	0	9	7	6	102	23	6	66	5	0	332	0	0	0
goto	1	4	2	0	3	7	0	5	0	0	0	1	0	1	0	0	0	2
break	0	0	5	8	8	10	0	0	0	1	3	0	5	3	9	0	1	1
continue	0	0	0	5	0	0	0	0	0	1	0	0	0	1	0	0	1	0
union	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
asm	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	48
switch	-	0	1	-	0	1	-	0	1	-	0	0	-	0	0	-	0	0
for	-	1	2	-	1	2	-	6	14	-	32	37	-	2	36	-	1	5
struct	-	0	1	-	0	0	-	0	1	-	0	0	-	0	1	-	0	0
[]	-	0	0	-	0	3	-	0	3	-	0	16	-	16	79	-	0	0
total	**5**	**15**	**52**	**13**	**18**	**59**	**14**	**33**	**230**	**48**	**44**	**201**	**15**	**41**	**539**	**0**	**3**	**204**

Таблица 5. Вычисление меры качества.

Название	TyDec	HexRays
35_wc	41	195
36_cat	19	156
38_day	37	429
39_deflate	0	389
59_lalr	38	777
84_derive	42	2873

В заключении формулируются результаты диссертационной работы и намечаются направления дальнейших исследований. В будущем планируется обобщить восстановление типов данных на межпроцедурный анализ, адаптировать алгоритм восстановления типов данных для восстановления типов данных других языков программирования, разработать методы восстановления иерархии наследования и использования исключений для языка Си++, а также расширить инструментальное средство модулями, позволяющими восстанавливать программы в язык Си++ непосредственно.

По теме диссертации опубликованы следующие работы:

В изданиях рекомендованных ВАК для публикации результатов кандидатских и докторских диссертаций:

1. К. Н. Долгова, А. В. Чернов. Автоматическое восстановление типов в задаче декомпиляции. Программирование, номер 2, журнал Российской академии наук. Март - апрель 2009, стр. 63-80.
2. К. Н. Долгова, А. В. Чернов, Е. О. Деревенец. Методы и алгоритмы восстановления программ на языке ассемблера в программы на языке высокого уровня. Проблемы информационной безопасности, Компьютерные системы, № 3. 2008 год, стр. 48-62.

Прочие публикации:

3. Katerina Troshina, Alexander Chernov and Yegor Derevenets. C Decompilation: Is It Possible? In Proceedings of International Workshop on Program Understanding. Altai Mountains, Russia. 19-23 June 2009, pp. 18-27.
4. Katerina Troshina and Alexander Chernov. High-Level Composite Type Reconstruction During Decompilation from Assembly Programs. In Proceedings of 7th Perspectives of System Informatics. Akademgorodok, Novosibirsk, Russia, 15-19 June 2009, pp. 292-299.
5. Katerina Troshina, Alexander Chernov and Alexander Fokin. Profile-Based Type Reconstruction for Decompilation. In Proceedings of IEEE 17th International Conference on Program Comprehension. Vancouver, Canada, 17-19 May 2009, pp. 263-267.
6. Katerina Dolgova, Alexander Chernov. Automatic Type Reconstruction in Disassembled C Programs. In Proceedings of IEEE 15th Working Conference on Reverse Engineering 2008. Antwerp, Belgium, October 2008, pp. 202-206.
7. Е. О. Деревенец, К. Н. Долгова. Структурный анализ в задаче декомпиляции. Прикладная информатика №4 Август 2009 г., Москва МаркетДС, стр. 87-99.
8. Е. О. Деревенец, К. Н. Долгова. Восстановление управляющих конструкций языка высокого уровня из исходной программы на языке ассемблера. Сборник статей молодых ученых факультета ВМиК МГУ № 6 2009 г., Москва, стр. 69-80.
9. В. Ю. Антонов, К. Н. Долгова. Восстановление типов данных с использованием информации о выполнении программы. Сборник статей молодых ученых факультета ВМиК МГУ № 6 2009 г., Москва, стр. 6-16.
10. К. Н. Долгова, А. В. Чернов. О некоторых задачах обратной инженерии. Труды Института системного программирования, Том 15, Москва 2008, стр. 119-134.
11. К. Н. Долгова, А. В. Чернов. Методы и алгоритмы восстановления программ на языке ассемблера в программы на языке высокого уровня. Материалы XVII Общероссийской научно-технической конференции «Методы и технические средства обеспечения безопасности информации», С.-Петербург, июль 2008, стр. 101.
12. К. Н. Долгова, В. Ю. Антонов. Введение в задачу декомпиляции. Сборник статей молодых ученых факультета ВМК МГУ № 5 2008 г., Москва, стр. 5-15.