УДК № 681.3.06

К.Н. Долгова, А.В. Чернов, Е.О. Деревенец
г. Москва, ИСП РАН

МЕТОДЫ И АЛГОРИТМЫ ВОССТАНОВЛЕНИЯ ПРОГРАММ НА ЯЗЫКЕ АССЕМБЛЕРА В ПРОГРАММЫ НА ЯЗЫКЕ ВЫСОКОГО УРОВНЯ

В работе рассматриваются методы и алгоритмы декомпиляции, то есть восстановления программ с языка ассемблера в язык высокого уровня. Также в работе представлен сравнительный анализ декомпиляторов. В работе предложены новые алгоритмы восстановления типов данных. Описана экспериментальная среда декомпиляции программ, разрабатываемая в ИСП РАН.

K.N. Dolgova, A.V. Chernov, E.O. Derevenets

METHODS AND ALGORITHMS FOR RECONSTRUCTING PROGRAMS FROM ASSEMBLY TO HIGH LEVEL LANGUAGE

The paper presents methods and algorithms for reconstructing programs form assembly to high level language. A comparative study of several existing decompilers is performed. New algorithms for high level language type recovery are proposed. An experimental decompilation environment being developing by the authors is described as well.

Введение.

На сегодняшний день широко распространена практика построения информационных систем из отдельных модулей, выполненных как собственными, так и сторонними разработчиками. Это позволяет значительно сократить стоимость и время разработки программного обеспечения. Обычно внешние разработанные модули поставляются 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. Также программные модули от внешних разработчиков могут содержать скрытые ошибки, а, значит, внедрение такого модуля может приводить к нестабильной работе всей информационной системы. Следовательно, весь сторонний код должен подвергаться аудиту специалистами с точки зрения безопасности его внедрения и использования.

Однако программные приложения, представленные в виде исполняемых файлов или на языке ассемблера, сложны для анализа специалистами в области информационной безопасности. Для более качественного и продуктивного анализа программные приложения лучше предоставлять специалистам на более высоком уровне представления, например, на языке высокого уровня, в частности Си. Ассемблерный листинг не позволяет с приемлемыми трудозатратами оценить взаимосвязь элементов программы, а также идентифицировать в программе стандартные 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.

Также декомпиляция может использоваться для обеспечения совместимости программных приложений, а именно, ее можно использовать для анализа протоколов взаимодействия. Зачастую протоколы взаимодействия описаны недостаточно полно или не описаны вообще. Декомпиляция позволяет упростить восстановление машины состояний и структур данных протокола обмена.

Декомпиляция – это процесс 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 и выдающее на выход эквивалентную ей программу на некотором языке высокого уровня.

В настоящее время из промышленно используемых компилируемых языков программирования высокого уровня широко распространены языки Си и Си++, поскольку именно на них разрабатывается основная масса 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 абстракции более высокого уровня, нежели те, которые поддерживаются языком Си. Так как процесс декомпиляции – это процесс повышения уровня абстракции представления программы, можно считать, что программы на Си являются промежуточным уровнем при переходе от программы на ассемблере к программе на языке Си++. Дальше повысить уровень абстракции представления программы можно посредством широко известных методов рефакторинга, позволяющих выделять объектно-ориентированные абстракции из процедурного кода [1].

Поэтому в предлагаемой работе ограничим множество рассматриваемых декомпиляторов декомпиляторами, восстанавливающими программы, представленные либо на языке ассемблера, либо в виде исполняемых файлов, в язык Си.

Задача декомпиляции была поставлена в 60-е годы XX века сразу же, когда стали широко применяться компиляторы с языков высокого уровня, но не утратила своей актуальности и по сей день [2]. Эта задача не решена в полной мере до сих пор в силу ряда трудностей принципиального характера. В частности, при компиляции программы из языка высокого уровня в язык ассемблера характерно отображение «многие к одному» концепций языка высокого уровня в концепции языка ассемблера, и, как следствие, однозначное восстановление программы на языке высокого уровня становится зачастую невозможным.

В силу указанных выше причин полностью автоматический декомпилятор реализовать принципиально невозможно. Поэтому системы декомпиляции программ должны работать во взаимодействии с аналитиком, который (зачастую методом проб и ошибок) управляет процессом декомпиляции.

В данной работе в качестве процессорной архитектуры, с которой ведется декомпиляция, выбрана архитектура ia32, наиболее распространенная в настоящее время. В листингах фрагментов программ на языке ассемблера используется синтаксис AT&T [3].

Предлагаемая работа имеет следующую структуру. Во втором разделе статьи дается формальное определение задачи декомпиляции. Третий раздел посвящен описанию существующих декомпиляторов, а также в нем представлены результаты сравнительного тестирования декомпиляторов на разработанном наборе тестовых примеров. Пятый раздел работы посвящен описанию декомпилятора ИСП РАН, который разрабатывается авторами. В заключении сформулированы выводы работы и направления дальнейших исследований.

2. Определение декомпиляции.

Декомпиляцию назовем корректной, если выполняются следующее условия. Пусть N – это некоторый компилятор с языка высокого уровня в ассемблер, N_n – это компилятор N с набором опций n. Пусть A – это некоторая программа на языке высокого уровня, тогда $N_n(A) = C$, если C – это программа на языке ассемблера, полученная в результате компиляции программы A компилятором N с набором опций n.

Пусть C и D – это некоторые программы на языке ассемблера. Пусть f – это преобразование переименования ассемблерных программ, то есть, $f(C)=D$, если

существует переименование именованных объектов (регистров, меток и т.д.) программы *C*, сохраняющее семантику программы *C*, такое, что программы *f(C)* и *D* синтаксически неразличимы.

Пусть *g* – это преобразование перестановки инструкций в ассемблерных программах, то есть, *g(C)=D*, если существует такая перестановка ассемблерных инструкций программы *C*, сохраняющая семантику программы *C*, что программы *f(C)* и *D* синтаксически неразличимы.

Для каждого компилятора множество инструкций процессора, используемых в сгенерированной программе, ограничено и может быть существенно меньше множества всех инструкций процессора (если не используются ассемблерные вставки), однако подмножества используемых ассемблерных инструкций процессора у разных компиляторов различаются. Набор используемых компилятором ассемблерных инструкций мы предполагаем известным и считаем свойством компилятора. Тогда зафиксируем некоторый набор инструкций, достаточный для отображения языка Си в язык ассемблера, и назовем такой набор инструкций *каноническим*. В качестве такого канонического набора мы выбрали набор инструкций, используемых компилятором GCC, в силу того, что код этого компилятора открыт. Так как и исходный язык высокого уровня, и целевая архитектура совпадают, то для любого компилятора существует отображение инструкций, специфичных для него, в инструкции канонического набора. Будем считать такое отображение известным и назовем его *h*.

Введем следующее отношение эквивалентности на классе ассемблерных программ. Будем считать, что программа *C* эквивалентна программе *D*, если $[f \circ g \circ h](C) = D$. Таким образом, мы рассматриваем программы на языке ассемблера с точностью до переименования регистров и меток и перестановки инструкций, не влияющих на семантику программы, а также с точностью до семантически эквивалентных замен одного набора инструкций другим.

Назовем результатом корректной декомпиляции программы *C* (на языке ассемблера) программу *B* (на языке высокого уровня), если существует компилятор *N* с набором опций *n*, такой что $N_n(B) = D$, и *D, C* – эквивалентные программы. Тогда в дальнейшем будем понимать под декомпилятором программу, выполняющую корректную декомпиляцию с языка ассемблера в язык высокого уровня.

В данной работе предполагается, что декомпилируемые в язык Си программы изначально были написаны на языке Си. Хотя существует много компилируемых в машинный код языков (Pascal, Fortran и т.д.), на практике различить по откомпилированной программе то, на каком языке она была изначально написана и даже каким компилятором собрана, не составляет особого труда. Для этого можно использовать названия библиотечных функций, способы передачи параметров подпрограмм и другие отличительные черты компиляторов. Например, подобный модуль встроен в дизассемблер Ida Pro [4].

Каждый язык программирования имеет свой набор понятий, которые определенным образом отображаются в язык низкого уровня. Для разных компиляторов одного и того же языка эти правила отображения близки, а зачастую даже идентичны, поскольку семантика входного и выходного языка фиксирована, а, следовательно, разработчики компиляторов имеют ограниченные возможности для маневра. С другой стороны, разные языки определяют разный, а зачастую даже ортогональный набор понятий. Например, COMMON блоки языка Fortran не имеют прямого аналога в языке Си, как и указатели языка Си не имеют прямых аналогов в языке Fortran. Другим примером являются различия в организации хранения многомерных массивов в памяти и т.д. На практике при трансляции с одного языка в другой возникают сложности при отображении понятий входного языка, отсутствующих в целевом языке. Все существующие в настоящее время трансляторы из одного языка высокого уровня в другой генерируют на

выходе трудночитаемый код, насыщенный артефактами трансляции, возникающими из-за несовпадения понятийного аппарата языков.

Поэтому можно сказать, что задача построения декомпилятора, который корректно декомпилирует исполняемый модуль, написанный на произвольном языке программирования высокого уровня в программу на языке Си, чтобы она была «читабельна» и близка к коду, написанному программистом, нереалистична.

3. Исследование возможностей существующих Си-декомпиляторов.

На настоящий момент не известны декомпиляторы, восстанавливающие программы в язык Си, которые позволяют полностью автоматически восстановить любую программу в «читабельном» виде, то есть достаточно похожую на код, написанный программистом. Для практического использования декомпиляторов требуется хорошо представлять их возможности, и для достижения наилучшего результата, возможно, потребуется использовать набор декомпиляторов в некотором сочетании.

В работе рассматриваются следующие декомпиляторы в язык Си: декомпилятор Boomerang [5], декомпилятор DCC [6], декомпилятор REC [7] и плагин Hex-Rays [8] к дизассемблеру IdaPro [4]. Рассматриваемые декомпиляторы сравниваются на разработанной системе тестов. Критерием качества разработки системы тестов была полнота покрытия всех составляющих языка от управляющих конструкций до типов данных. Кроме того, декомпиляторы сравнивались на чувствительность к различным опциям компиляции исходной программы и возможность восстановления функций стандартной библиотеки, а также обнаружение функции *main*.

Все рассматриваемые декомпиляторы, кроме плагина Hex-Rays, на вход принимают исполняемый файл, а на выходе выдают программу на языке Си. Плагин Hex-Rays принимает на вход программу, являющуюся результатом работы дизассемблера Ida Pro, то есть схему программы на ассемблеро-подобном языке программирования. В качестве результата плагин Hex-Rays выдает восстановленную программу в виде схемы на Си-подобном языке программирования. Тем не менее, для простоты мы в дальнейшем объединим процесс дизассемблирования с использованием Ida Pro и последующей декомпиляции.

В том случае, когда декомпилятор оказывается не в состоянии восстановить некоторый фрагмент исходной программы в язык Си, этот фрагмент сохраняется в виде ассемблерной вставки в восстановленной программе. Надо заметить, что даже небольшие исходные программы после декомпиляции зачастую содержат очень много ассемблерных вставок, что практически сводит на нет эффект от декомпиляции. Более того, ассемблерные вставки серьезно затрудняют решение задачи повышения уровня абстракции представления программы.

3.1 Современные декомпиляторы.

Boomerang. Декомпилятор Boomerang [5] является программным обеспечением с открытым исходным кодом (open source). Разработка этого декомпилятора началась 2002 году. Изначально задачей проекта было разработать такой декомпилятор, который восстанавливает исходный код из исполняемых файлов вне зависимости от того, каким компилятором и с какими опциями исполняемый файл был получен. Для этого в качестве внутреннего представления используется представление программы со статическими одиночными присваиваниями (SSA). Однако, несмотря на это декомпилятор не сильно адаптирован под различные компиляторы и чувствителен к применению различных опций, в частности, опций оптимизации. Кроме того, в нем не поддерживается распознавание функций стандартной библиотеки Си.

DCC. Проект по разработке этого декомпилятора [6] был открыт в 1991 году и закрыт в 1994 году с получением главным разработчиком степени PhD. В качестве входных данных декомпилятор DCC принимает 16-битные исполняемые файлы в формате DOS EXE. В рамках данного проекта было разработано несколько базовых алгоритмов

восстановления структурных конструкций программы в язык высокого уровня, которые описаны в публикациях автора. Исходный код декомпилятора доступен на официальном сайте [6]. Предложенные алгоритмы декомпиляции основаны на теории графов (анализ потока данных и потока управления). Для распознавания библиотечных функций используется сигнатурный поиск, для которого была разработана библиотека сигнатур.

REC. Этот проект [7] был открыт в 1997 году компанией BackerStreet Software, но вскоре закрылся из-за ухода ведущего разработчика проекта. Позднее разработка декомпилятора продолжилась его автором в статусе собственного продукта. Сейчас декомпилятор распространяется свободно, но без исходного кода. Публикаций по данному проекту не известно. REC восстанавливает исполняемые файлы в различных форматах, в частности ELF [9] и PE [10]. Также декомпилятор REC можно использовать на различных платформах. В ходе тестирования этого декомпилятора было выявлено, что наиболее успешно декомпилятор восстанавливает исполняемые файлы, полученные при компиляции с отключенной оптимизацией и добавлении отладочной информации.

Hex-Rays. Как сказано выше, инструмент Hex-Rays [8] не является самостоятельным программным продуктом, а распространяется как плагин к дизассемблеру IdaPro [4]. Это – самое новое из рассматриваемых средств декомпиляции: плагин появился на рынке в 2007 году. Особенностью данного инструмента является то, что он восстанавливает программы, полученные на выходе дизассемблера Ida Pro. Из алгоритмов, реализованных в нем, особого внимания заслуживает алгоритм сигнатурного поиска FLIRT [11] и алгоритм поиска параметров в стеке PIT (Parameter Identification and Tracking).

В таблице 1 представлена сводная характеристика всех рассматриваемых декомпиляторов.

Таблица 1. Сравнительный анализ декомпиляторов.

	Boomerang	*DCC*	*REC*	*Hex-Rays*
распознавание библиотечных функций	нет	заявлено	нет	да
активность разработки	заявлено	нет	да	да
переносимость	нет	да	да	да
open source	да	да	нет	нет

3.2 Исследование возможностей декомпиляторов.

В этом разделе приведены результаты тестирования возможностей рассмотренных декомпиляторов. Для тестирования был разработан тестовый набор программ на языке Си, покрывающий основные языковые конструкции языка Си.

Тестирование проводилось по следующей методике. Исходный код программы на языке Си компилировался в исполняемый файл компилятором gcc 3.4.5 в системе Debian Linux и компилятором Borland C++ 3.1 в системе Windows XP. В первом случае результатом работы компилятора являлся файл формата ELF [10] для архитектуры ia32, во втором случае – исполняемый файл DOS для 16-битного режима процессора. Исполняемый файл формата ELF подавался на вход декомпиляторам Boomerang, REC и Hex-Rays, работающим в системе Windows XP. Исполняемый файл формата DOS EXE подавался на вход декомпилятору DCC. Результат декомпиляции сравнивался с исходным текстом.

Такая комбинация инструментальных и целевых сред была выбрана по следующим причинам. Во-первых, декомпилятор DCC поддерживает только 16-битные исполняемые модули DOS, поэтому для оценки качества работы декомпилятора был использован компилятор 16-битного режима. Декомпиляторы Boomerang и REC наоборот не поддерживают 16-битный режим DOS. Исполняемый модуль подавался на вход декомпиляторам в формате ELF, а не в естественном для Windows формате PE [11], так

как декомпиляторы Boomerang и REC, как оказалось, некорректно обнаруживали точку начала программы на языке Си в файлах формата PE.

Качество работы каждого декомпилятора для каждого теста оценивалось по 4-х бальной экспертной шкале, приведенной в таблице 2.

Так, оценка «3» выставлялась в случаях, когда декомпилированная программа использовала адресную арифметику вместо массивов или приведения типов для получения указательных значений вместо корректного объявления типов переменных. Кроме того, оценка «3» выставлялась, если в результате декомпиляции цикл `for` оказывался преобразованным в цикл `while`.

Таблица 2. Шкала оценки декомпиляторов.

Количество баллов за тест	*Комментарий*
0	Декомпилятор закончил работу с ошибкой выполнения или пустым результатом.
1	Декомпилятор выдал ассемблерный код. Программа на Си не была получена.
2	Декомпилятор выдал программу на языке Си, которая либо не компилируется, либо работает неверно (неэквивалентна исходной), либо содержит ассемблерные вставки, то есть недекомпилированные фрагменты программы.
3	Декомпилятор выдал корректную программу, которая эквивалентна исходной, но использует конструкции, отличные от конструкций в исходной программе.
4	Декомпилятор выдал программу, которая эквивалентна исходной и использует те же конструкции, которые использовались в исходной программе.

3.2.1 Система тестов. Тестовый набор содержал следующие основные группы.

1. **Типы.** В тестах этой группы проверялась корректность восстановления типов переменных и параметров функций. Язык Си поддерживает богатый набор базовых целочисленных типов от типа *char* до типа *unsigned long long*. Декомпилятор должен по возможности точно восстановить как размер, так и знаковость переменной.

   Также рассматривались типы указателей на базовые типы. Проверялся факт обнаружения того, что переменная является указательным, а не целым типом, и корректность восстановления целевого типа указателя.

   Для массивов проверялся факт обнаружения того, что переменная является локальным или глобальным массивом, точность восстановления типа элементов массива, точность восстановления размера массива как для одномерных, так и для многомерных массивов.

   Для структурных типов проверялся факт распознавания использования структурного типа и точность восстановления полей структур.

   Кроме того, были рассмотрены разные комбинации указательных, массивовых и структурных типов и оценена корректность восстановления таких составных типов. В частности, рассматривались массивы структур, указатели на структуры, структуры, содержащие массивы, структуры, содержащие указатели на самих себя.
2. **Языковые конструкции.** В тестах этой группы проверялась корректность восстановления управляющих структур программы. Проверялась корректность восстановления оператора `if` с простым условием, в том числе и с отсутствующей частью `else`, операторов цикла `while` и `do while` с простыми условиями.

В другой группе тестов проверялась корректность восстановления логических операций `&&` (логическое «и»), `||` (логическое «или») в условиях операторов `if` и циклов. Согласно семантике языка Си эти операторы транслируются в условные и безусловные переходы, то есть являются конструкциями, задающими поток управления, а не вычисления значений. Декомпиляторы должны по возможности восстанавливать сложные условия в операторах языка.

Отдельно проверялась корректность восстановления структурных операторов передачи управления, таких как `break`, `continue` и `return`.

Оператор `switch` рассматривался отдельно, так как в большинстве компиляторов он транслируется в косвенный безусловный переход, где адрес перехода выбирается из таблицы в соответствии с вычисленным в заголовке оператора значением. Декомпиляторы должны распознавать использование этого оператора в программе.

3. **Функции.** В тестах этой группы проверялась корректность выделения параметров функций и локальных переменных в условиях разных соглашений о вызовах. Кроме того, проверялась корректность обработки рекурсивных функций.
4. **Оптимизации.** В тестах этой группы проверялась корректность работы декомпиляторов при условии, что при компиляции были использованы некоторые оптимизационные преобразования, такие как открытая вставка функций (inlining) и оптимизации вызовов функций (tail call optimization, tail recursion optimization, sibling call optimization).
5. **Взаимодействие с окружением.** В данной группе находился тест, проверяющий корректность обнаружения функции main в исполняемых файлах формата PE. Как известно, выполнение программы на языке Си начинается с функции main, которой передается определенный список параметров. Однако в исполняемых файлах вызову функции main предшествует выполнение специального кода, задача которого настроить окружение программы на Си, что заключается, в частности, в создании стандартных потоков ввода-вывода, инициализации служебных структур данных управления динамической памятью и т. п. Этот код частично написан на языке ассемблера, кроме того, он не представляет интереса, так как стандартный для всех программ. Поэтому декомпиляторы должны игнорировать этот инициализационный код и начинать декомпиляцию непосредственно с функции main.

   Кроме того, в тестах этой группы проверялось распознавание стандартных библиотечных функций языка Си (например, *strlen* и т. п.).

3.2.2 Результаты тестирования.

В таблице 3 приводятся результаты работы декомпиляторов на выбранном наборе тестов в соответствии с системой оценок, указной в таблице 2. Каждый столбец таблицы соответствует декомпилятору, а каждая строка – тесту. Общий результат для каждого декомпилятора получен суммированием оценок по всем тестам.

Из всех рассмотренных декомпиляторов только Boomerang поддерживает декомпиляцию оператора `switch`. Остальные декомпиляторы генерируют в этом случае некорректный код на языке ассемблера. Только декомпилятор REC сумел восстановить цикл `for`, в то время как остальные декомпиляторы в этом случае генерируют программу, использующую цикл `while`.

Таблица 3. Результаты тестирования декомпиляторов.

		Bommerang	Rec	DCC	Hex-Rays
типы данных	struct	3	2	3	3
	массивы	4	3	3	4
	unsigned int	4	3	3	3
	unsigned short	3	3	3	3
структурные конструкции языка	логические операции	4	4	4	4
	циклы **for**	3	4	3	3
	циклы **while**	4	3	4	4
	циклы **do while**	4	4	4	4
	оператор **switch**	4	2	2	2
функции	рекурсия	4	4	4	4
оптимизация	inlining	2	2	–	2
	tail recursion	2	2	–	2
обнаружение функции *main* в PE файлах		1	1	–	4
обнаружение функций стандартной библиотеки		2	2	3	4
сумма баллов		48	43	40	50

Наиболее развитым в настоящее время является декомпилятор Hex-Rays, который, в отличие от других декомпиляторов, поддерживает распознавание массивов и распознавание библиотечных функций, хотя даже и Hex-Rays имеет много слабых сторон.

Все рассмотренные Си-декомпиляторы по входной программе выдают Си-программу с восстановленными структурными конструкциями и распознанными функциями, но они не восстанавливают полностью типы данных, и получаемая на выходе Си-программа содержит множество операций явного приведения типов. Такая программа сложна для анализа человеком.

4. Декомпилятор ИСП РАН.

В Институте системного программирования РАН ведется разработка экспериментальной среды декомпиляции. Декомпилятор восстанавливает все высокоуровневые конструкции языка Си, поддерживает множество входных форматов: программы на языке ассемблера, реализованные на диалектах AT&T и Intel, исполняемые модули, а также бинарные трассы выполнения, собранные на симуляторе процессора. Также декомпилятор имеет развитый графический интерфейс аналитика для управления процессом декомпиляции. Результатом работы декомпилятора является типизированная Си-программа. Схема архитектуры декомпилятора представлена на рис. 1.

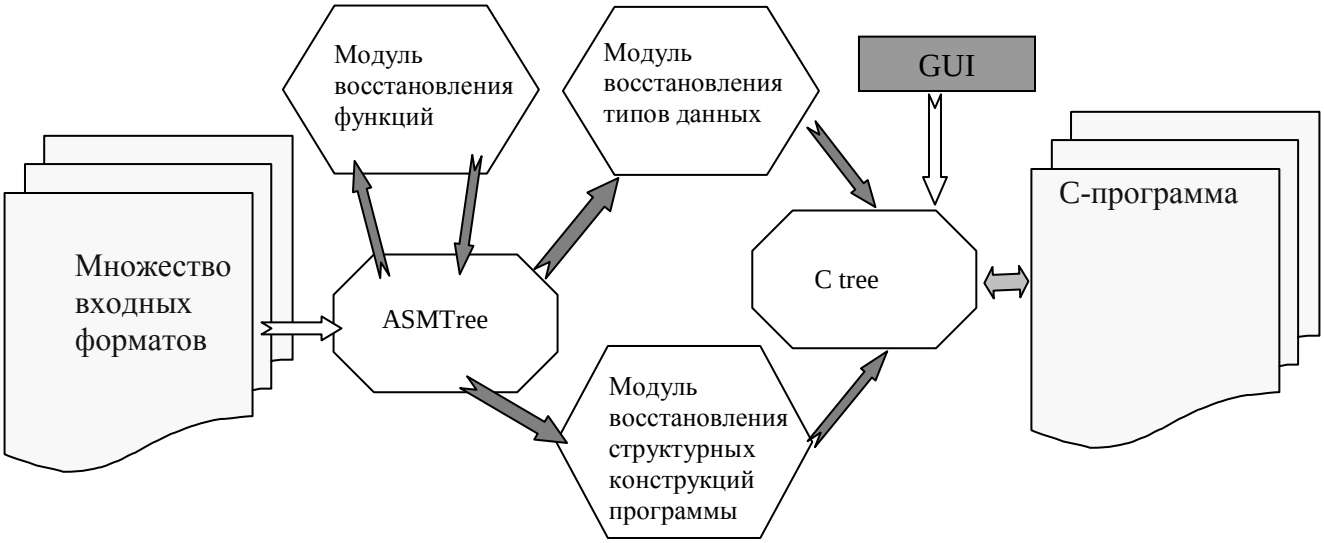

Рис. 1. Архитектура декомпилятора ИСП РАН.

Многоугольниками на рис. 1 представлены компоненты декомпилятора, а стрелками отображается поток данных между ними.

Множество входных форматов. Декомпилятор поддерживает следующие входные форматы данных: текстовый файл с ассемблерными инструкциями в нотации AT&T или Intel, исполняемые файлы, трассы, снятые симулятором AMD SimNow [12] или другими инструментальными средствами.

ASMTree. *ASMTree* – это внутреннее представление входной программы в виде вектора инструкций. Каждая инструкция содержит код соответствующей машинной инструкции и набор аргументов. Аргументы могут быть одного из следующих типов: регистр, константа, строка (используется для хранения символьных меток в инструкциях перехода), адрес в памяти, сумма двух аргументов, произведение двух аргументов. Для сложных аргументов, таких как сложение и умножение строится поддерево. Модуль *ASMTree* отвечает за построение и хранение дерева внутреннего представления программы и предоставляет интерфейс для работы с деревом другим компонентам системы.

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

Модуль восстановления типов данных. Этот модуль отвечает за восстановление типов данных декомпилируемой программы. В качестве входных данных используется внутреннее представление программы. По нему строится дерево зависимостей использования типов данных. Далее применяется алгоритм восстановления типов данных.

Модуль восстановления структурных конструкций программы. Этот модуль восстанавливает управляющие конструкции программы. В нем выполняется построение графа потока управления. Граф представляет собой совокупность вершин и направленных дуг. Для каждой вершины хранится список входящих ребер, список исходящих дуг, номера первой и последней инструкции, образующих линейный участок, представляемый этим базовым блоком. Дуга хранит указатели на вершину, из которой она исходит и в которую она входит. Класс, реализующий граф потока управления, хранит список указателей на все вершины и все дуги, составляющие этот граф. Реализованы методы для создания графа потока управления из внутреннего представления программы на языке ассемблера и для отладочной печати графа потока управления в формате для программы dot, что делает возможным дальнейшую визуализацию графа. Также в этом модуле реализованы методы работы с графом потока управления.

CTree. CTree – это компонент, обеспечивающий работу с внутренним представлением дерева Си-программы, а также позволяющий хранить и обрабатывать вспомогательные структуры данных. Этот модуль по внутреннему представлению с использованием результатов работы модулей восстановления типов данных и структурных конструкций на выходе строит программу на языке Си. Внутреннее представление CTree предназначено для: хранения выходной программы на языке Си в памяти в удобной для анализа и трансформации форме; обеспечения возможности управления генерацией текстового представления со стороны пользователя; хранения информации о восстановленных типах и т. п. в процессе декомпиляции; хранения восстанавливаемых фрагментов программы в процессе декомпиляции. Представление CTree устроено как дерево с переменным числом листьев. Управление памятью реализовано с помощью подсчета ссылок на узлы дерева как извне дерева, так и внутри него. Также реализована возможность получать дерево программы в текстовом представлении.

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

Рассмотрим подробнее основной тракт процесса декомпиляции. В ходе декомпиляции программы решаются следующие задачи: выделение структурных единиц программы, в частности, подпрограмм в однородном ассемблерном листинге, выявление параметров подпрограмм и возвращаемых ими значений, структурный анализ, то есть восстановление операторов циклов, ветвлений и т. п., восстановление типов данных как базовых, так и производных и другие.

4.1 Выделение функций в потоке инструкций. Одной из основных структурных единиц программ на языке Си являются функции, которые могут принимать параметры и возвращать значения. Откомпилированная программа, однако, состоит из потока инструкций, функции в котором никак структурно не выделяются. Как правило, компиляторы генерируют код с одной точкой входа в функцию и одной точкой выхода из функции. При этом в начало кода для функции помещается последовательность машинных инструкций, называемая прологом функции, а в конец кода, генерируемого для функции, помещается эпилог функции. И пролог, и эпилог функции, как правило, стандартны для каждой архитектуры, и претерпевают на ней незначительные вариации. Например, стандартный пролог и эпилог функции для архитектуры i386 показаны ниже:

```
Пролог:
        pushl   %ebp
        movl    %esp, %ebp

Эпилог:
        movl    %ebp, %esp
        popl    %ebp
        ret
```

Декомпилятор ИСП РАН поддерживает несколько распространенных форм прологов и эпилогов для разных соглашений о передаче параметров. При обработке трасс, кроме того, предполагается, что инструкции, получающие управление в результате выполнения инструкции `call`, также являются точками входа в функции, а инструкции `ret` завершают функции.

Кроме того, в декомпиляторе реализован ряд эвристик для обработки некоторых специальных случаев. Например, опция компилятора GCC *–fomit-frame-pointer* подавляет использование регистра %ebp в качестве указателя на текущий стековый кадр в случаях, когда это возможно. В этом случае пролог и эпилог функции будут, как таковые, отсутствовать. Для обработки таких функций вводится фиктивный регистр стекового

фрейма, и смещения относительно регистра %esp внутри функции пересчитываются в смещения относительно %ebp. Пересчет выполняется с помощью абстракного выполнения операций над регистром %esp в теле функции.

Если в процессе компиляции было выполнено встраивание тела функции в точку вызова, такие встроенные функции не могут быть автоматически выявлены. Задача выделения встроенных функций возлагается на аналитика, который может использовать средства рефакторинга Си-программ, предоставляемые интегрированной средой декомпиляции.

Существуют оптимизирующие преобразования, которые приводят к появлению в машинном коде конструкций, принципиально невозможных в языках высокого уровня. Таким оптимизирующим преобразованием является, например, sibling call optimization. Если список параметров двух функций идентичен, и первая функция вызывает вторую с этими параметрами, то инструкция вызова подпрограммы `call` может быть преобразована в инструкцию безусловного перехода `jmp` в середину тела второй функции. Результатом такого рода «неструктурных» оптимизаций будет появление переходов из одной функции в другую, появление функций с несколькими точками входа или несколькими точками выхода. Для функций с несколькими точками входа или несколькими точками выхода выполняется клонирование тела функции, в результате чего получается несколько функций, каждая из которых имеет единственную точку входа и единственную точку выхода.

Тем не менее, возможна и ситуация, когда даже в результате всех описанных выше преобразований не может быть получена функция с одной точкой входа и одной точкой выхода и одним из стандартных прологов и эпилогов. В этом случае распознавание функции не выполняется, соответствующий фрагмент ассемблерного кода помечается и передается для анализа аналитику.

4.2 Выявление параметров и возвращаемых значений функций. Языки высокого уровня, в частности, Си поддерживают передачу параметров в функции и возврат значений. В языке Си существует только передача параметров по значению, в других языках могут поддерживаться и другие механизмы. Заметим, что здесь мы рассматриваем только механизмы передачи параметров, отображаемые в генерируемый машинный код. Передача параметров по имени, передача параметров в шаблоны и другие механизмы периода компиляции программы здесь не рассматриваются.

Способы передачи параметров и возврата значений для каждой платформы специфицированы и являются составной частью так называемого ABI (application binary interface). Под платформой здесь понимается, как обычно, тип процессора и тип операционной системы, например, Win32/i386 или Linux/x86_64. Одной из задач ABI является обеспечение совместимости по вызовам приложений и библиотек, скомпилированных разными компиляторами одного языка или написанных на разных языках.

Так, для платформы win32/i386 используется несколько соглашений о передаче параметров. Соглашение о передаче параметров *_cdecl* используется по умолчанию в программах на Си и Си++ и имеет следующие особенности [13]:

1. Параметры передаются в стеке и заносятся в стек справа налево (то есть, первый в списке параметр заносится в стек последним).
2. Параметры выравниваются в стеке по границе 4 байт, и адреса всех параметров кратны 4. То есть параметры типа *char* и *short* передаются как *int*, но и дополнительное выравнивание для размещения, например, *double* не производится.
3. Очистку стека производит вызывающая функция.
4. Регистры %eax, %ecx, %edx и %st(0) – %st(7) могут свободно использоваться (не должны сохраняться при входе в функцию и восстанавливаться при выходе из нее).
5. Регистры %ebx, %esi, %edi, %ebp не должны модифицироваться в процессе работы функции.

6. Значения целых типов, размер которых не превосходит 32 бит, возвращаются в регистре %eax, 64-битных целых типов – в регистрах %eax и %edx, вещественных типов – в регистре %st(0).
7. Если функция возвращает результат структурного типа, место под возвращаемое значение должно быть зарезервировано вызывающей функций. Адрес этой области памяти передается как (скрытый) первый параметр.

Отметим, что этот набор правил — это именно соглашения, которые «добровольно» выполняются в сгенерированном коде. Пока речь не заходит об интерфейсе с независимо скомпилированными сторонними модулями программист может в определенной мере модифицировать эти правила, существенно затрудняя задачу автоматического восстановления функций.

Декомпилятор ИСП РАН поддерживает все распространенные соглашения о передаче параметров. Кроме того, аналитик может выполнять тонкую настройку параметров соглашения о передаче параметров, например, изменяя множество модифицируемых или немодифицируемых регистров.

Для выделенных на предыдущем шаге функций применяется ряд эвристических алгоритмов для определения используемого соглашения о передаче параметров. Например, если в эпилоге функции используется инструкция `ret N`, значит, что данная функция использует соглашения stdcall или pascal.

Конечно же, аналитик имеет возможность задавать соглашение о передаче параметров вручную. Особенно важно это в тех случаях, когда вызываемая функция неизвестна в точке вызова при косвенных вызовах.

В ряде случаев найти функции, которые могут быть вызваны по указателю в данной точке программы, возможно и при статическом анализе программы. Несмотря на то, что такое множество может быть неполным, его может быть достаточно, чтобы извлечь информацию о типе соглашений о связях и о параметрах. В более сложных случаях возможно использование трасс, собранных при выполнении данной программы. В случае использования трассы будет найдено множество функций, вызванных в данной точке при данном наборе параметров, или при данном множестве наборов параметров. Найденных функций, опять-таки, может оказаться достаточно для выявления соглашения о передаче параметров.

На следующем этапе обработки функций определяется размер области параметров функций. Размер области параметров определяется с помощью отслеживания значения регистра %esp методами абстрактной интерпретации. Если используется соглашение о связях stdcall, размер области параметров содержится в функции в инструкции `ret N`.

По результатам предыдущих этапов выделяются параметры функции и переменные в области локальных параметров. Для каждой ячейки памяти в области локальных переменных строятся *DU-графы*. Каждый DU-граф получает свое символическое имя. Таким образом, одна ячейка памяти может быть отображена в несколько переменных в программе на Си. Для регистров также выполняется построение DU-графов. DU-графы регистров, не загружаемых из памяти и не сохраняемых в память, также отображаются в переменные в программе на Си, что соответствует переменным исходной программы на Си, которые компилятор языка Си разместил на регистрах. Пример DU-графа представлен на рис. 2. В квадратных скобках обозначена строка программы, которой соответствует конструкция, `def` - это определение переменной, `use` – использование переменной. В представленном примере переменная `a` исходной программы в восстановленной программе может быть представлена двумя переменными.

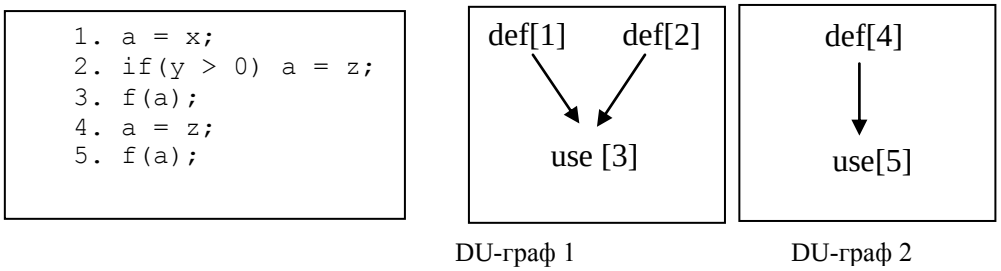

Рис. 2. Пример DU-графа.

4.3 Восстановление структурных конструкций языка высокого уровня. Одним из результатов предыдущих фаз анализа ассемблерного листинга программы является разбиение потока инструкций ассемблерного листинга на отдельные функции и выявление точек входа в функции и возврата из функций.

Инструкции ассемблерной программы рассматриваются как представление нижнего уровня (Low-level intermediate representation) [14]. В частности, в представлении низкого уровня отсутствуют структурные управляющие конструкции. Вся передача управления выполняется с помощью условных и безусловных переходов. Для восстановления управляющих конструкций в декомпиляторе ИСП РАН сначала строится граф потока управления программы. По графу потока управления строится дерево доминаторов, затем дуги графа потока управления классифицируются на «прямые», «обратные» и «косые».

На основании этой информации уже выполняется непосредственно структурный анализ, то есть восстановление высокоуровневых управляющих конструкций [13]. Поиском в глубину в графе выделяются шаблоны основных структурных конструкций, которые затем организуются в иерархическую структуру.

4.4 Восстановление типов выявленных объектов языка высокого уровня.

Новизной предлагаемого подхода является восстановление типов данных. Задача автоматического восстановления типов данных на настоящее время – одна из наименее проработанных с теоретической точки зрения задач в области декомпиляции. Ее можно условно разделить на подзадачу восстановления базовых типов данных языка, таких как *char*, *unsigned long* и т. п., и на подзадачу восстановления производных типов, таких как типы структур, массивов и указателей.

Восстановление типов данных выполняется итеративно, и каждая итерация выполняется в 2 этапа.

1) восстановление базовых типов языка (char, int…) и
2) восстановление производных типов языка (структуры, массивы, указатели)

Восстановление базовых типов данных выполняется на основе алгоритма, суть работы которого заключается в следующем. Алгоритм восстанавливает базовые типы из конечного множества базовых типов данных. Каждый базовый тип характеризуется кортежем из трех атрибутов *<ядро, размер, знак>*. По ассемблерным инструкциям строится множество ограничений на типы объектов, где под *объектами* понимаются ячейки, хранящие значения (регистры процессора, ячейки памяти в стеке, косвенные адресуемые ячейки памяти и т.д.). По ассемблерной программе строится дерево зависимостей использования типов данных. По этому дереву строится система уравнений над типами данных объектов. К системе уравнений применяются правила преобразования, аналогичные правилам продвижения констант при выполнении оптимизирующих преобразований. В результате применения таких правил на каждом проходе по системе уравнений множество значений, которое может принимать кортеж, характеризующий тип объекта, уточняется до тех пор, пока не будет найдена неподвижная точка системы. После нахождения неподвижной точки для каждого объекта имеется кортеж атрибутов типа данных *<ядро, размер, знак>*. По полученному кортежу восстанавливается тип данных. В

случае, если имеется неоднозначность, привлекается аналитик, который в диалоговом режиме выбирает тип данных из предлагаемых альтернатив.

Для восстановления производных типов данных выполняется обработка всех присутствующих в программе операций обращений к памяти. Каждая операция доступа к памяти представляется в виде ($base + offset + \sum_{j=0}^{n} C_j x_j$), где $C_j x_j$ – мультипликативные компоненты адресного выражения. При компиляции из языка Си мультипликативная составляющая генерируется при доступе к элементам массива. Обычно в программах на Си справедливо, что элементы массива `a[i]` и `a[0]` имеют один и тот же тип для массива `a`, следовательно, мультипликативная составляющая адресного выражения несущественна для восстановления типов элементов массива и может быть опущена из дальнейшего рассмотрения. Следует отметить особо, что для массивов с размером элементов 1 байт (например, строки или символьные буферы) имеется неопределенность, вызванная тем, что в ассемблерном листинге адрес начала массива и индекс могут быть неразличимы. В этом случае применяются специальные эвристики для различения базы и смещения, а так же могут использоваться трассы выполнения программы.

После того, как все мультипликативные составляющие адресных выражений опущены, ассемблерная программа содержит конечное число константных смещений относительно базовых выражений. Эти смещения можно рассматривать как смещения полей структур. Пары (*base* + *offset*) рассматриваются аналогично скалярным переменным на стеке, в области статических данных или на регистрах, аналогично тому, как это описано выше. Структурный тип целиком может быть восстановлен, если собрать вместе все смещения по одному базовому выражению.

Предположим, что C_j – это константа. Константы, используемые в адресном выражении, можно использовать для получения информации о размере элементов массива. Пусть m – это $\min_{j \in (0;n)} C_j$, тогда можно предположить, что m – это размер элемента массива. Если в индексных выражениях используются арифметические выражения типа `a[2*i]`, то размер массива равен $\hat{m} = НОД(m_1, ..., m_n)$, где m_i – это минимум i-го адресного выражения обращения к массиву `a`.

Массивы структур можно восстанавливать автоматически, используя ряд следующих эвристик. Пусть m – это размер элемента массива. Если есть доступ к памяти того же массива в виде ($base + offset_1$) и ($base + offset_2$), и $|offset_1 - offset_2| < m$, то элементы массива на самом деле структуры.

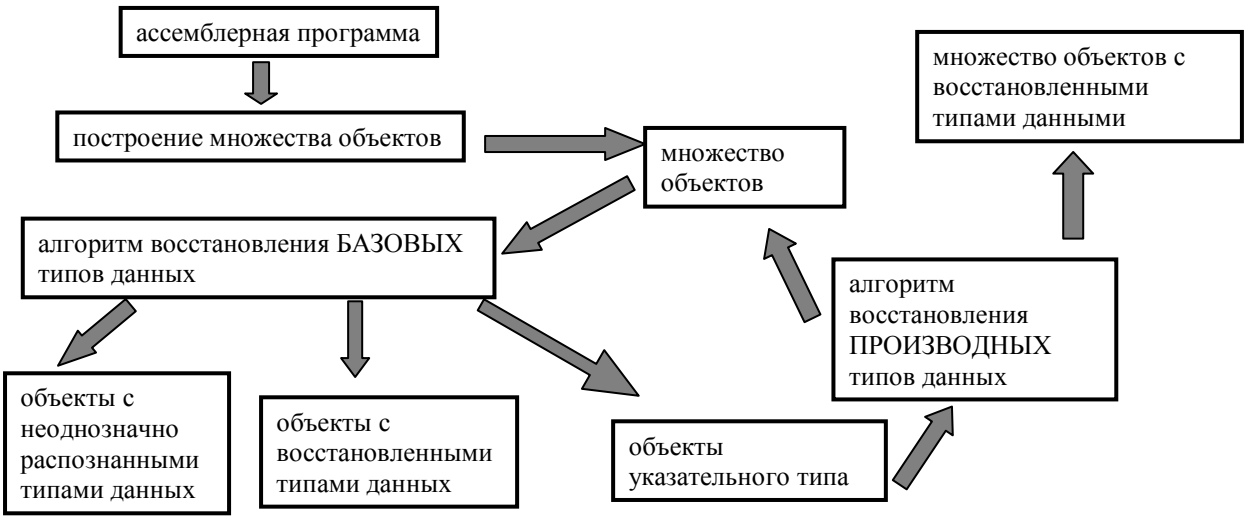

Рис 3. Схема работы полного алгоритма восстановления типов.

Заключение.

В данной работе рассмотрена задача декомпиляции как восстановления программы на языке высокого уровня по программе на языке ассемблера или в машинных кодах. Дано краткое описание основных шагов процесса декомпиляции программы. Также в работе представлено сравнительное тестирование существующих декомпиляторов и указаны их сильные и слабые стороны.

Представлено описание экспериментального декомпилятора ИСП РАН, который позволяет восстанавливать все высокоуровневые конструкции языка Си, включая типы данных.

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. **Фаулер М.** Рефакторинг: улучшение существующего кода. Символ-Плюс, 2008 г.
2. **Щеглов К.Е.** Обзор алгоритмов декомпиляции//Электронный журнал «Исследовано в России». http://zhurnal.ape.relarn.ru/articles/2001/116.pdf
3. AT&T Assembly Syntax. http://sig9.com/articles/att-syntax
4. Интерактивный дизассемблер и отладчик Ida Pro http://www.idapro.ru/
5. Boomerang Decompiler Home Page. http://boomerang.sourceforge.net/
6. DCC Decompiler Home Page. http://www.itee.uq.edu.au/~cristina/dcc.html
7. REC Decompiler Home Page. http://www.backerstreet.com/rec/
8. Hex-Rays Decompiler SDK. http://www.hex-rays.com/
9. Tool Interface Standards (TIS). Executable and Linkable Format (ELF). http://www.x86.org/intel.doc/tools.htm
10. Tool Interface Standards (TIS). Portable Executable Formats (PE). http://www.x86.org/intel.doc/tools.htm
11. **Гуильфанов И.** FLIRT – Fast Library Identification and Recognition Technology. http://www.idapro.ru/description/flirt/
12. ADM SimNow Simullator. http://developer.amd.com/cpu/simnow/Pages/default.aspx
13. **C. Cifuentes, D. Simon, A. Fraboulet**. Assembly to High-Level Language Translation. Technical Report 439. Department of Computer Science and Electrical Engineering. The University of Queensland. 1998.
14. **Steven S. Muchnik**. Advanced Compiler Design And Implementation. Morgan Kaufmann, 997.