Е. Н. Трошина, А. В. Чернов

Восстановление типов данных в задаче декомпилирования в язык С

Декомпиляция — одна из сложнейших задач обратной инженерии. Одной из подзадач декомпиляции является задача восстановления типов данных. В работе подробно рассматриваются методы восстановления типов данных языка С, как базовых, так и производных.

Создание и разработка сложных программных систем различного назначения часто ведется посредством интеграции отдельных компонентов, 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 С. 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 С и С++, 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 наиболь-

шую практическую значимость. Язык C++ можно считать расширением языка C, добавляющим в него концепции более высокого уровня относительно языка C. Поскольку при обратной инженерии в целом и при декомпиляции в частности уровень абстракции представления программы повышается, можно считать, что программы на языке C являются промежуточным уровнем при переходе от программы на языке ассемблера к программе на языке C++ . 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, т. е. программы, полученные на выходе декомпилятора, должны как можно больше и полнее использовать высокоуровневые конструкции языка.

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 полноту декомпиляции. Восстановление программы из низкоуровневого представления без восстановления типов данных не сильно повышает уровень представления программы, и может сводить на нет эффект от декомпиляции из-за трудности понимания восстановленного кода. Данная работа посвящена одной из задач декомпиляции — восстановлению типов данных, другая задача декомпиляции — восстановление структурных конструкций языков высокого уровня — рассматривалась в работе [1], которая была опубликована в № 4 (22). Комплексное решение задачи декомпиляции в язык C представлено в диссертации на соискание ученой степени кандидата физико-математических наук по специальности 05.13.11 — «Математическое и программное обеспечение вычислительных машин и компьютерных сетей» [2].

Задачу восстановления типов данных условно можно разделить на подзадачи восстановления:

1) **базовых типов данных** языка, таких как char, unsigned long и т. п.;

2) **производных типов данных**, таких как типы структур, массивов и указателей.

Изложение материала в данной работе выполнено в несколько упрощенной форме. Более строгое описание представленных моделей восстановления базовых данных представлено в работах [2], [3] и производных типов —

в работах [2], [4]. Описание инструментальной среды по декомпиляции программ представлено в работах [2], [5] и [6].

Данная статья имеет следующую структуру. В первом разделе представлено описание методов восстановления типов данных. Во втором — описание метода восстановления базовых типов данных, далее — описание восстановления производных типов данных. Следующий раздел посвящен экспериментальным проверкам и описанию работы модуля восстановления типов данных декомпилятора TyDec, который разрабатывается авторами.

Обзор работ по восстановлению типов данных

Один из подходов к восстановлению типов заключается в использовании методов математической логики. Подход впервые был предложен А. Майкрофтом (A. Mycroft) [7].

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 неприменим для автоматического восстановления типов данных.

Другой подход к задаче анализа бинарной программы представлен в работах Г. Балакришнана (G. Balacrishnan) [8] и [9], 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 работах М. Ю. Гусенко [10]. В качестве операций над типизированными данными рассматриваются инструкции процессора, а в качестве значений типизированных данных — 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, т. е. указателей на указатель, так как за один

проход восстановить всю глубину косвенности затруднительно. Также можно утверждать, что представленный метод не позволит восстановить рекурсивно-определенный структурный тип. Следовательно, он также не позволяет решать задачу восстановления типов данных с высоким уровнем качества.

Следует также отметить, что одной из фундаментальных работ по декомпиляции в язык С считается диссертационная работа К. Цифуентес (C. Cuffuentes). Однако в ее работах [11], [12] восстановление типов переменных практически не рассматривается.

На практике все декомпиляторы, кроме расширения Hex-Rays [13] к интерактивному дизассемблеру Ida Pro [14], вообще не восстанавливают даже базовые типы переменных, а в выражениях используют явное приведение типов, что делает декомпилированные программы сложными для понимания и модификации.

Восстановление базовых типов данных

На входе дана ассемблерная программа, по которой надо восстановить типы переменных, для декомпилированной программы в язык С.

Представление типов данных. Ограничим множество автоматически выводимых типов данных языка С только базовыми типами и указателями на базовые типы. Пусть множество T_0 {`unsigned char`, `char`, `unsigned short`, `short`, `unsigned int`, `int`, `unsigned long`, `long`, `unsigned long long`, `long long`, `float`, `double`, `long double`, `void`} — это множество базовых скалярных типов и «псевдо» тип void. Будем рассматривать ключевое слово void в качестве некоторого специального типа данных, об использовании которого будет сказано ниже.

Пусть множество $T_1 = \{\tau^* | \tau \in T_0\}$ — это множество, состоящее из указателей на базовые скалярные типы и обобщенного указателя *void**. Тогда все множество автоматически выводимых типов данных — это $T = \bigcup_{i=0}^{1} T_i$.

Представим каждый тип данных из множества T в виде совокупности трех характеристик:

ядро `core`$\in$ {`int`, `pointer`, `float`};

размер `size`$\in$ {1, 2, 4, 8};

знак `sign`$\in$ {`signed`, `unsigned`}.

Специальный тип `void` будем представлять как

$$\texttt{void} = \begin{cases} \texttt{void}^{\texttt{core}} = \varnothing \\ \texttt{void}^{\texttt{size}} = \{1, 2, 4\}. \\ \texttt{void}^{\texttt{sign}} = \varnothing \end{cases}$$

С точки зрения перечисленных выше трех составляющих некоторые типы из множества T в конкретных компиляторах языка С могут быть неразличимы. Тем не менее, для обеспечения независимости модели от конкретной платформы мы их сохраним во множестве T. Тогда, учитывая определение корректной декомпиляции, данное выше, из неразличимых для конкретного декомпилятора типов можно выбирать любой.

Различить в программе на ассемблере, с каким типом данных мы работаем — с вещественным или с целым — нетрудно, так как наборы инструкций для работы с вещественными и целыми типами различны. Учитывая ограничения на входные данные, сформулированные выше, мы рассматриваем только работу с регистрами, размером не более, чем 4 байта. А значит, распознавание работы с 8-мибайтными типами данных сводится к распознаванию шаблонных конструкций из нескольких инструкций процессора. Восстановление составных типов данных, таких как структуры, массивы и т. д. в данной работе не рассматривается.

Будем считать, что работа с указательным типом ведется как с беззнаковым 32-хбитным целым числом. Тогда, если размер типа меньше 4-х байтов, то тип не может быть указательным. Для типа данных «указатель» размер и знаковость относятся к тому типу данных, на который он указывает.

Например, если тип

$$T = \{\texttt{unsigned short*}\} = [\texttt{unsigned short*}],$$

то его представление в виде трех составляющих следующее:

$$T = \begin{cases} \tau^{\texttt{core}} = \{\texttt{pointer}\} \\ \tau^{\texttt{size}} = \{2\} \\ \tau^{\texttt{sign}} = \{\texttt{unsined}\} \end{cases}.$$

И, наоборот, если, например, имеем представление некоторого типа,

$$T=\begin{cases}\tau^{\text{core}}=\{\texttt{pointer}\}\\ \tau^{\text{size}}=\{1\}\\ \tau^{\text{sign}}=\{\texttt{unsined}\}\end{cases},$$

то тип $T=[\texttt{unsigned char}]$.

В предлагаемой модели не поддерживается распознавание таких конструкций как указатель на указатель и более сложных, а также не рассматриваются указатели на вещественные типы. Итак, основными задачами предлагаемого алгоритма распознавания типов являются:

1) определение знаковости типа, например `short` и `unsigned short`;

2) определение того, является ли тип указательным или целым, например `unsigned int` и `char*`;

3) определение размера типа, например, `short` и `int`.

Объекты. В работе [9] предложено понятие *a-loc* абстракции, которое, по сути, эквивалентно понятию переменной в языке С. Однако *a-loc* абстракция не дает возможности отслеживать перемещения переменных по регистрам и не позволяет разделять несвязанные использования одной переменной. Поэтому в данной работе объектами для анализа по восстановлению типа являются web [15]. Web представляет собой компонент связности двудольного графа «определения переменной — использование переменной». Отличие web от переменных, регистров и других подобных конструкций в том, что web учитывает особенность повторного использования ячеек памяти и регистров независимым образом, так как одна и та же ячейка памяти или регистр часто используются в различных фрагментах программы независимо. Таким образом, в ассемблерной программе web строятся по:

1) регистрам процессора, например, %eax;

2) ячейкам памяти по фиксированным адресам (это были глобальные переменные в исходной программе на языке С);

3) ячейкам памяти по фиксированным смещениям в текущем стековом кадре, например, –2(%ebp) — в исходной программе на языке С это локальные переменные функций и их параметры;

4) ячейкам памяти, адресуемым по смещениям регистра %esp, и ячейкам памяти, неявно адресуемым при занесении значений в стек, например при использовании команды push.

В дальнейшем под объектом будем понимать web, построенный по описанным выше компонентам ассемблерной программы. Каждый объект obj_i характеризуется некоторым типом данных $T_i(\tau_i)$ из рассматриваемого множества T, т. е. $T_i \in T$. Тип T_i назовем *«идеальным типом»*. Это именно тот тип, который изначально был у переменной, отображенной в объект obj_i, в исходной программе на языке С. Как было сказано выше, при корректной декомпиляции восстановленный тип может отличаться от идеального типа. Поэтому для объекта obj_i будем искать тип T_i, т. е. $obj_i{:}T_i$, где $T_i \in T$, который назовем *«искомым типом»*. Искомый тип в декомпилируемой программе должен совпадать с идеальным типом синтаксически или быть семантически эквивалентным ему.

Например, пусть параметр <`Type`> в функции `func` (см. листинг) принимает значение {`unsigned int`}. Тогда «идеальный тип» для всех объектов этой функции — это {`unsigned int`}. А «искомый тип» — это любой тип из множества {`int`, `unsigned int`, `long`, `unsigned long`}.

Искать «искомый тип» $T_i \in T$ будем через последовательное приближение. Каждый тип данных можно представить в виде 3-х составляющих. Будем считать, что каждый объект

$$obj_i \text{ имеет тип } obj_i{:}T_i, T_i \in T$$

где $T_i=\begin{cases}\tau_i^{\text{core}}\\ \tau_i^{\text{size}}\\ \tau_i^{\text{sign}}\end{cases}$.

Тогда будем восстанавливать тип $T_i \in T$ посредством последовательного приближения 3-х его составляющих характеристик $\{\tau_i^{\text{core}}, \tau_i^{\text{size}}, \tau_i^{\text{sign}}\}$.

Дерево зависимостей использования типов данных. Построим дерево зависимостей использования типов для исследуемой программы на языке ассемблера. Дерево зависимостей использования типов — это дерево,

у которого «узел» есть операция, полученная на основе ассемблерной инструкции, или объект, «лист» — это объект, а «дуга» — это зависимость использования узлов. Назовем узел, построенный на основе ассемблерной инструкции, «узел-инструкция», а все остальные узлы назовем «узел-объект». Назовем родительский узел-объект узла-инструкции «результат операции». Узлы-потомки узла-инструкции назовем «операндами».

Таким образом, каждый узел-инструкция характеризуется узлами-объектами «результат операции» и «операндами». Каждый узел-объект характеризуется узлом-инструкцией, для которой он является результатом операции, и узлом-инструкцией, для которой он является операндом. Очевидно, что для листьев отсутствует характеристика узел-инструкция, для которой этот объект является результатом операции, а для корня аналогично отсутствует узел-инструкция, для которой он является операндом.

Дерево состоит из узлов-инструкций 5 типов:

1) **следование,** такие узлы будем обозначать «FN»; их потомки не имеют локальной зависимости по использованию и должны соответствовать в декомпилированной программе в язык С отдельному оператору;

2) **вычисление выражений,** этим узлам соответствуют операции вычисления выражения, например операция сложения двух объектов;

3) **копирования,** такие узлы будем обозначать «CN»; им соответствует операции передачи данных в ассемблерном листинге между объектами;

4) **вызов подпрограмм,** обозначим такие узлы «CALL»; этот узел соответствует вызову пользовательской или библиотечной функции;

5) **проверка условия,** этот узел соответствует инструкциям проверки условий при условном переходе.

Для каждого узла-инструкции дерева зависимостей типов выпишем уравнение вида

$$obj_i{:}t_{n,1} < op > obj_j{:}t_{n,2} => obj_k{:}t_{n,3}$$

где obj_i, obj_j, obj_k — это узлы-объекты;

$t_{n,1}, t_{n,2}, t_{n,3}$ — это типы соответствующих объектов в n-м уравнении;

$<op>$ — это операция, соответствующая узлу-инструкции дерева зависимостей типов;

obj_i и obj_j — это операнды узла-инструкции, соответствующего инструкции $<op>$;

obj_k — это результат операции этого же узла-инструкции.

Совокупность уравнений для всех узлов объединим в систему уравнений. Каждый тип $t_{n,i}$ соответствует некоторому типу T_j, и, следовательно, его также можно разложить на 3 составляющие: $\tau_{n,j}^{\mathtt{core}}$ — ядро типа, $\tau_{n,j}^{\mathtt{size}}$ — его размер и $\tau_{n,j}^{\mathtt{sign}}$ — знак типа данных. Таким образом, между полученной системой и построенным деревом зависимостей использования типов есть взаимно однозначное соответствие. Так как каждый тип $t_{n,i}$ соответствует некоторому типу T_j, то для каждого типа T_j можно выписать все типы объекта obj_j из уравнений. Тогда можно сказать, что тип T_j представлен типом t_{i_1,k_1} в уравнении i_1, типом t_{i_2,k_2} в уравнении i_2 и т. д. Обозначим всех представителей типа T_j во всех уравнениях множеством $\{t_{i_1,k_1},...,t_{i_m,k_m}\}$, т. е. $T_j = \{t_{i_1,k_1},...,t_{i_m,k_m}\}$.

Рассмотрим отдельно правую и левую части уравнений. Тогда можно сказать, что множество $\{t_{j_1,l_1},...,t_{j_m,l_m}\}$ — это множество представителей некоторого типа T_i по всем левым частям уравнений системы и назовем множество $\{t_{j_1,l_1},...,t_{j_m,l_m}\}$ представителями типа T_i по левой части системы. Аналогично для правой части системы. Тогда тип $T_j = t_{i_1,k_1},...,t_{i_m,k_m}$ — это все представители типа T_j по всей системе, $T_j^{\mathtt{left}}\{t_{n_1,l_1},..., t_{n_m,l_m}\}$ и $T_j^{\mathtt{right}}\{t_{j_1,l_1},..., t_{j_m,l_m}\}$ — это все представители типа T_j по левой и правой частям системы, соответственно.

Очевидно, что для любого типа верно, что

$$T_j = T_j^{\mathtt{left}} \cup T_j^{\mathtt{right}}.$$

Ограничения. Рассмотрим некоторые ограничения, которые можно наложить, учитывая особенности имеющихся инструкций в ассемблерной программе:

1. $C_{\mathrm{reg}}| =$ — регистровое ограничение, оно влияет на составляющую «ядро», *т. е.* `core` и «размер», т. е. `size`.

Например, если некоторый объект *obj*:*T* построен по регистру %dx, то можно сказать, что «тип» этого объекта `int`, а «размер» у него **1 байт** или **2 байта**, т. е. $T=\{\tau^{\text{core}},\tau^{\text{size}},\tau^{\text{sign}}\}$, то $\tau^{\text{core}}=\{\text{int}\}$, а $\tau^{\text{size}}=\{1,2\}$.

2. $C_{\text{cmd}}|=$ — командное ограничение, оно влияет на составляющую «ядро», т. е. `type`, «размер», т. е. `size` и «знак», т. е. `sign`.

Например, если имеем команду `movswl` $obj_i{:}T_i$ $obj_j{:}T_j$, то можно сказать, что, если $T_i=\{\tau_i^{\text{core}},\tau_i^{\text{size}},\tau_i^{\text{sign}}\}$, то $\tau_i^{\text{sign}}=\{\text{signed}\}$, а, следовательно, $\tau_i^{\text{core}}=\{\text{int}\}$.

3. $C_{\text{flags}}|=$ — флаговое ограничение, оно влияет на составляющую типа «знак», т. е. `sign`.

Например, если имеем следующий фрагмент кода на ассемблере

```
cmpl  -12(%ebp), %eax
jae    L3
```

то, пусть объект *obj*:*T* построен по регистру %eax, тогда можно сказать, что $\tau^{\text{sign}}=\text{unsigned}$.

4. $C_{\text{env}}|=$ — ограничение окружения, оно влияет на все три составляющие типа. Это ограничение, которое возникает в результате использования библиотечных функций, так как прототип библиотечных функций считается известным.

Функция слияния. При решении уравнений потребуется использовать функцию слияния. Использования простого пересечения двух множеств не достаточно, так как в случае пустого пересечения множеств слияемых объектов не сохранятся их значения. А так как здесь рассматривается итеративный алгоритм, то, как раз в случае пустого пересечения слияемых множеств их значения надо объединять, так как на следующих итерациях алгоритма будет получено множество, пересечение с которым будет непустое. В противном случае мы имеет конфликт, для разрешения которого все равно требуется полная история возможных значений слияемых множеств. Следовательно, назовем множество S результатом слияния множеств S_1 и S_2, и обозначим $S=\overline{\{S_1,S_2\}}$, если

$$S=\begin{cases}S_1\cap S_2, \text{если } S_1\cap S_2\neq\varnothing\\ S_1\cup S_2, \text{если } S_1\cap S_2=\varnothing\end{cases}.$$

Пусть t_1 и t_2 — это представители некоторого типа $T_n\in T$, следовательно, как сказано в пункте 3.1, их можно разложить на три составляющие, т. е. $t_2=\{\tau_1^{\text{core}},\tau_1^{\text{size}},\tau_1^{\text{sign}}\}$ и $t_2=\{\tau_2^{\text{core}},\tau_2^{\text{size}},\tau_2^{\text{sign}}\}$.

Пусть t — это представитель некоторого типа $T_n\in T$, t_1 и t_2 — это представители того же типа $T_n\in T$. Назовем t результатом слияния t_1 и t_2, и обозначим $t=\overline{\{t_1,t_2\}}$, если

$$t=\{\tau^{\text{core}},\tau^{\text{size}},\tau^{\text{sign}}\},$$

где $\tau^{\text{core}}=\tau_1^{\text{core}},\ \tau_2^{\text{type}}$;
$\tau^{\text{sign}}=\tau_1^{\text{sign}},\ \tau_2^{\text{sign}}$.

Тогда, если $T_k=\{t_{i_1,j_1},\ldots,t_{i_n,j_n}\}$, выполним слияние по всем множествам трех составляющих для соответствующих трех составляющих типа T_k, т. е. если

$$T=\begin{cases}\tau_i^{\text{core}}\\ \tau_i^{\text{size}}\\ \tau_i^{\text{sign}}\end{cases},$$

а

$$t_{i_1,j_1}=\begin{cases}\tau_{i_1,j_1}^{\text{core}}\\ \tau_{i_1,j_1}^{\text{size}}\\ \tau_{i_1,j_1}^{\text{sign}}\end{cases},\ldots,t_{i_n,j_n}=\begin{cases}\tau_{i_n,j_n}^{\text{core}}\\ \tau_{i_n,j_n}^{\text{size}}\\ \tau_{i_n,j_n}^{\text{sign}}\end{cases},$$

то

$$\tau_k^{\text{core}}=\{\tau_{i_1,j_1}^{\text{core}},\ldots,\tau_{i_n,j_n}^{\text{core}}\},$$

и

$$\tau_k^{\text{sign}}=\{\tau_{i_1,j_1}^{\text{sign}},\ldots,\tau_{i_n,j_n}^{\text{sign}}\}.$$

После того, как слияние выполнено, переопределим все составляющие подтипов типа T_k значением функции слияния, т. е. теперь $T_k=\{t_{i_1,j_1}:=T_k,\ldots,t_{i_n,j_n}:=T_k\}$. Будем называть левой функцией слияния слияние представителей типа по левой части уравнения, т. е. $T_j^{\text{left}}=\overline{\{t_{n_1,l_1},\ldots,t_{n_m,l_m}\}}$.

Аналогично $T_j^{\text{right}}=\overline{\{t_{n_1,l_1},\ldots,t_{n_m,l_m}\}}$ — правая функция.

Правила обхода дерева. Для дерева зависимостей использования типов определим 2 типа обхода:

1) обход снизу вверх, или восходящий обход;

2) обход сверху вниз, или нисходящий обход.

При обходе **снизу вверх** информация о типах объектов распространяется от операндов к результату операции. В терминах системы уравнений назовем соответствующий проход *продвижением слева направо.* При обходе дерева **сверху вниз** информация о типах объектов распространяется от результатов операции к ее операндам, что в терминах системы уравнений соответствует *продвижению справа налево.*

Так как в данной модели отсутствует межпроцедурный анализ, то будем различить узлы дерева зависимостей следующих типов: это узлы, непосредственно продвигающие информацию, и узлы, не продвигающие информацию о типах данных непосредственно. То есть узел, непосредственно продвигающий информацию, — это такой узел, что при обходе дерева снизу вверх в результате операции появляется новая информация о типе данных от операндов, и при обходе сверху вниз в операндах появляется новая информация о типах от результата операции.

Каждый узел дерева, который может непосредственно распространять информацию о типе данных, соответствует одному из следующих типов операций:

1) бинарная;
2) унарная;
3) копирования;
4) обращение к памяти.

Узел **следования** не распространяет непосредственно информации о типах данных между своими потомками.

Узел **вычисления выражений** — это узел, непосредственно распространяющий информацию о типах данных между своими результатом и потомками. Это узел может быть как бинарной, так и унарной операцией.

Узел **копирования** является узлом, непосредственно распространяющим информацию о типах данных между своими операндами и результатом операции. Узел копирования соответствует операции копирования. Первый операнд — это объект, из которого копируем. Второй операнд — это объект, в который копируем. Результат операции — это объект, который передается следующей инструкции и содержит скопированные данные. Учитывая возможности неявного расширения или усечения регистров, второй операнд и результат операции могут различаться. Например, если имеем следующий фрагмент кода на языке ассемблера:

```
movzbw   (%eax), %dx,
```

то для узла копирования, соответствующего команде movzbw, первым операндом будет объект, соответствующий обращению к памяти по адресу, записанному в регистре %eax, вторым операндом — объект, соответствующий регистру %dx, а результатом операции — объект, соответствующий регистру %edx.

Узел **вызова подпрограмм** не распространяет информацию о типах данных своих потомков. Следовательно, этот узел не является узлом, непосредственно распространяющим информацию.

Узел **проверки условий** является узлом**,** непосредственно распространяющим информацию. Этот узел соответствует унарной операции.

Каждый узел, непосредственно распространяющий информацию о типе данных, может соответствовать операции обращения к памяти, если один из его потомков — это операция обращения к памяти.

Рассмотрим правила распространения информации о типах при обходе дерева **снизу вверх.**

Бинарную операцию рассмотрим на примере вычитания. Пусть уравнение имеет вид

$$e_1{:}t_{e_1} - e_2{:}t_{e_2} => e_{\text{res}}{:}t_{e_{\text{res}}},$$

а

$$t_{e_{\text{res}}} = \left\{\tau_{e_{\text{res}}}^{\text{core}}, \tau_{e_{\text{res}}}^{\text{size}}, \tau_{e_{\text{res}}}^{\text{sign}}\right\},$$

$$t_{e_1} = \left\{\tau_{e_1}^{\text{core}}, \tau_{e_1}^{\text{size}}, \tau_{e_1}^{\text{sign}}\right\},$$

следовательно, можно условно обозначить, что $\tau_{\text{res}}^{\text{core}} = \tau_{e_1}^{\text{core}} - \tau_{e_2}^{\text{core}}$, и $\tau_{\text{res}}^{\text{sign}} = \tau_{e_1}^{\text{sign}} - \tau_{e_2}^{\text{sign}}$.

Определим правила распространения информации об использовании типов данных для составляющей типа «ядро», т. е. для τ^{core}, следующим образом:

$\tau_{e_1}^{\text{core}}{:}\texttt{pointer} - \tau_{e_2}^{\text{core}}{:}\texttt{pointer} \Rightarrow \tau_{\text{res}}^{\text{core}}{:}\texttt{int};$

$\tau_{e_1}^{\text{core}}:\text{pointer}-\tau_{e_2}^{\text{core}}:\text{integer}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{pointer};$

$\tau_{e_1}^{\text{core}}:\text{pointer}-\tau_{e_2}^{\text{core}}:\text{float}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{error};$

$\tau_{e_1}^{\text{core}}:\text{float}-\tau_{e_2}^{\text{core}}:\text{pointer}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{error};$

$\tau_{e_1}^{\text{core}}:\text{float}-\tau_{e_2}^{\text{core}}:\text{float}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{float};$

$\tau_{e_1}^{\text{core}}:\text{float}-\tau_{e_2}^{\text{core}}:\text{integer}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{error};$

$\tau_{e_1}^{\text{core}}:\text{integer}-\tau_{e_2}^{\text{core}}:\text{float}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{error};$

$\tau_{e_1}^{\text{core}}:\text{integer}-\tau_{e_2}^{\text{core}}:\text{integer}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{integer};$

$\tau_{e_1}^{\text{core}}:\text{integer}-\tau_{e_2}^{\text{core}}:\text{pointer}\Rightarrow\tau_{\text{res}}^{\text{core}}:\text{error}.$

Рассмотрим операцию над множествами «пересечение сверху»: $S=S_1\prod S_2$, т. е.

$$S=\{s|s=\max(s_1,s_2),s_1\in S_1,s_2\in S_2\},$$

где $(s_1,s_2)\in S_1\times S_2$, т. е. (s_1,s_2) — это все члены декартового произведения множеств S_1 и S_2.

Для составляющей типа «размер», т. е. для τ^{size}, определим правила распространения информации о типах так:

$$\tau_{\text{res}}^{\text{size}}=\tau_{e_1}^{\text{size}}\prod\tau_{e_2}^{\text{size}}.$$

Для составляющей типа «знак», т. е. для τ^{sign}, правила распространения информации о типах использования данных будут определены таким образом:

$\tau_{e_1}^{\text{sign}}:\text{unsigned}-\tau_{e_2}^{\text{sign}}:\text{unsigned}\Rightarrow\tau_{\text{res}}^{\text{sign}}:\text{unsigned};$

$\tau_{e_1}^{\text{sign}}:\text{unsigned}-\tau_{e_2}^{\text{sign}}:\text{signed}\Rightarrow\tau_{\text{res}}^{\text{sign}}:\text{unsigned};$

$\tau_{e_1}^{\text{sign}}:\text{signed}-\tau_{e_2}^{\text{sign}}:\text{signed}\Rightarrow\tau_{\text{res}}^{\text{sign}}:\text{signed}.$

Для *унарной* операции правила распространения информации такое: если унарная операция имеет вид

$$e_1:t_{e_1}<op>=>e_{\text{res}}:t_{e_{\text{res}}},\text{ то }t_{e_{\text{res}}}:=t_{e_1}.$$

Пусть операция *копирования* имеет вид:

$$e_1:t_{e_1}->e_2:t_{e_2}=>e_{\text{res}}:t_{\text{res}},$$

тогда правило распространения информации для нее такое: $t_{e_{\text{res}}}:=\overline{t_{e_1},t_{e_2}}$. Для операции *обращения к памяти* правило распространения информации определим таким образом: пусть есть некоторое уравнение

$$e_1:t_{e_1}\otimes e_2:t_{e_2}=>e_{\text{res}}:t_{\text{res}},$$

где $\otimes$ — это либо бинарная операция, либо операция копирования.

Если по адресу, записанному в одном из операндов, выполняется обращение к памяти, например, в операнде $e_1:t_{e_1}$, то

$$\tau_{\text{res}}^{\text{core}}:=\tau_{e_1}^{\text{core}}\cup\{\text{pointer}\},$$

$$\tau_{\text{res}}^{\text{sign}}:=\tau_{e_1}^{\text{sign}}\text{ и }\tau_{\text{res}}^{\text{size}}:=\tau_{e_1}^{\text{size}}.$$

Рассмотрим правила распространения информации о типах при обходе дерева **сверху вниз.**

При обходе дерева зависимостей типов сверху вниз для таких составляющих типа данных, как «ядро», т. е. core, и «знак», т. е. sign может возникать альтернативная информация. Если в логике предикатов рассматривать уравнение вида $A=B\vee C$, то, зная, что $A=\text{true}$, можно точно сказать, что либо $B=\text{true}$, либо $C=\text{true}$, либо и $B=\text{true}$, и $C=\text{true}$. Таким образом, можно сказать, что значение «true» является альтернативным для переменных B и C, потому что они одновременно не могут принимать значение «false». По аналогии рассмотрим альтернативное значение для составляющих типа данных. Обозначим альтернативное значение как

$$\tau_m^{\text{class}}(?)[n],$$

где $\text{class}\in\{\text{core},\text{size},\text{sign}\}$.

Альтернативное значение соответствует значению некоторой составляющей типа данных. Альтернативные элементы должны быть парными. Это означает, что если во множестве некоторой составляющей типа m есть альтернативный элемент $\tau_m^{\text{class}}(?)[n]$, то во множестве той же самой характеристики типа n есть альтернативный элемент $\tau_n^{\text{class}}(?)[m]$. Следовательно, определим альтернативный элемент как такой элемент, что при слиянии, либо и $\tau_m^{\text{class}}(?)[n]$, и $\tau_n^{\text{class}}(?)[m]$ будут принадлежать результирующему множеству, либо обязательно один из них, т. е. либо $\tau_n^{\text{class}}(?)[m]\in\tau_n^{\text{class}}$ либо $\tau_m^{\text{class}}(?)[n]\in\tau_m^{\text{class}}$, либо и $\tau_n^{\text{class}}(?)[m]\in\tau_n^{\text{class}}$, и $\tau_m^{\text{class}}(?)[n]\in\tau_m^{\text{class}}$ после выполнения слияния. Если в результате слияния оба из альтернативных элементов пары пропадают из соот-

ветствующих множеств, то их надо явно добавить во множество $\tau_m^{\mathtt{class}}$ и во множество $\tau_n^{\mathtt{class}}$.

Рассмотрим правила распространения информации для *бинарной* операции.

Пусть *i*-е уравнение системы имеет вид

$$e_1{:}t_{e_1} \otimes e_2{:}t_{e_2} => e_{res}{:}t_{e_{res}},$$

где $\otimes$ — некоторая бинарная операция, тогда для характеристики «ядро», т. е. для $\tau^{\mathtt{core}}$ применяется правило:

если $\mathtt{pointer} \in \tau_{\mathrm{res}}^{\mathtt{core}}$, то

$$\tau_{e_1}^{\mathtt{core}} := \tau_{e_1}^{\mathtt{core}} \cup \left\{ \mathtt{pointer}_{(e_1)}^{\mathrm{res}}(?)[e_2] \right\},$$

$$\tau_{e_2}^{\mathtt{core}} := \tau_{e_2}^{\mathtt{core}} \cup \left\{ \mathtt{pointer}_{(e_2)}^{\mathrm{res}}(?)[e_1] \right\},$$

если $\mathtt{pointer} \notin \tau_{\mathrm{res}}^{\mathtt{core}}$, то

$$\tau_{e_1}^{\mathtt{core}} := \overline{\tau_{e_1}^{\mathtt{core}}, \tau_{\mathrm{res}}^{\mathtt{core}}}, \tau_{e_2}^{\mathtt{core}} := \overline{\tau_{e_2}^{\mathtt{core}}, \tau_{res}^{\mathtt{core}}}.$$

Для характеристики типа «размер», т. е. для $\tau^{\mathtt{size}}$, правила такое $\tau_{e_1}^{\mathtt{size}} := \tau_{\mathrm{res}}^{\mathtt{size}} \prod \tau_{e_1}^{\mathtt{size}}$, а $\tau_{e_2}^{\mathtt{size}} := \tau_{\mathrm{res}}^{\mathtt{size}} \prod \tau_{e_2}^{\mathtt{size}}$.

Для характеристики типа «знак», т. е. для $\tau^{\mathtt{sign}}$, правила распространения информации аналогичны правилам для характеристики типа «ядро».

Если $\{\mathtt{unsigned}\} \in \tau_{\mathrm{res}}^{\mathtt{sign}}$, то

$$\tau_{e_1}^{\mathtt{sign}} := \tau_{e_1}^{\mathtt{sign}} \cup \left\{ \mathtt{unsigned}_{(e_1)}^{\mathrm{res}}(?)[e_2] \right\},$$

$$\tau_{e_2}^{\mathtt{sign}} := \tau_{e_2}^{\mathtt{sign}} \cup \left\{ \mathtt{unsigned}_{(e_2)}^{\mathrm{res}}(?)[e_1] \right\},$$

если $\{\mathtt{unsigned}\} \notin \tau_{\mathrm{res}}^{\mathtt{sign}}$, то

$$\tau_{e_1}^{\mathtt{sign}} := \overline{\tau_{e_1}^{\mathtt{sign}}, \tau_{\mathrm{res}}^{\mathtt{sign}}}, \tau_{e_2}^{\mathtt{sign}} := \overline{\tau_{e_2}^{\mathtt{sign}}, \tau_{res}^{\mathtt{sign}}}.$$

Для *унарной* операции правила распространения информации аналогичны правилам для нее при проходе снизу вверх, т. е. для *унарной* операции правила распространения информации такое: если унарная операция имеет вид

$$e_1{:}t_{e_1} <op> => e_{\mathrm{res}}{:}t_{e_{\mathrm{res}}}, \text{ то } t_{e_1} := t_{e_{\mathrm{res}}}.$$

Для операции *копирования* правило распространения информации определено так: если операция копирования имеет вид

$$e_1{:}t_{e_1} -> e_2{:}t_{e_2} => e_{\mathrm{res}}{:}t_{\mathrm{res}},$$

то

$$t_{e_1} := t_{e_1} \cup t_{e_{\mathrm{res}}} \text{ и } t_{e_2} := t_{e_2} \cup t_{e_{\mathrm{res}}}.$$

Для операции *обращения к памяти* используется правила, аналогичные для бинарной операции, в случае если $\{\mathtt{pointer}\} \in \tau_{\mathrm{res}}^{\mathtt{core}}$.

Отметим особо то, что правила распространения информации таковы, что «обход» нельзя нарушать. То есть нельзя начать обходить дерево сверху вниз, и не закончив, начать выполнять восходящий обход, так как в этом случае результат работы алгоритма может быть некорректным из-за того, что не все ограничения на использования объекта будут учтены.

Алгоритм нахождения типов для объектов по построенной системе уравнений. Пусть исходная программа имеет *N* объектов. Тогда множество *OBJ* мощности *N* — это множество, которое содержит все объекты исходной программы и только их, т. е.

$$OBJ = \{obj_1, ..., obj_n\},$$

где obj_i — это объект исходной программы.

Для каждого объекта $obj_i \in OBJ$ существует «идеальный тип» $T_i \in T$, а также существует «искомый тип» $T_i \in T$. Будем считать, что «искомый тип» $T_i \in T$ для объекта $obj_i \in OBJ$ найден корректно, если «идеальный тип» $T_i \in T$ является его надмножеством, т. е., если $T_i = \{\mathtt{int}\}$, а $T_i = \{\mathtt{int}, \mathtt{unsigned\ int}\}$, то $T_i \in T$ найден корректно. Также будем считать, что «идеальный тип» найден точно, если имеется полное совпадение, т. е. если $T_i = \{\mathtt{int}\}$ и $T_i = \{\mathtt{int}\}$, то $T_i \in T$ найдет точно.

Следовательно, определим множество $\mathtt{Type} = \{T_1, ..., T_n\}$, мощности *N* — это множество, содержащее все искомые типы объектов исходной программы и только их. Пусть функция `Tree make_tree(program)` — это функция, которая по программе на ассемблере строит дерево зависимостей типов. Пусть построенное дерево отображается в систему, описанную там же и состоящую из *M* уравнений, функцией `Equation make_equation(Tree)`. Определим процедуру `init(Equation)`, которая инициализирует все типы объектов в системе уравнений начальным значением `void`, т. е. для всех типов *t* в системе уравнений. Затем на полученную систему уравнений *Equation* накладываются ограничения, это выполняет процедура `set_constraint(Equation)`.

Определим процедуры
`spread_information_lr(Equation)`
и
`spread_information_rl(Equation)`,
которые по уравнению выполняет распространение информации об использовании типов данных в соответствии с правилами: первая — слева направо, а вторая — справа налево соответственно. Функция `Type(obj)` возвращает множество всех представителей типов объекта $obj \in OBJ$ по всем уравнениям из множества `Equation`. Процедуры
`boolean make_left_join(Type(obj))`,
`boolean make_right_join(Type(obj))`
и
`boolean make_full_join(Type(obj))`
выполняют соответственно левое, правое и полное слияние типа объекта $obj_i \in OBJ$ и возвращают «true», если в результате слияния $T_i \in T$ изменилось, и «false» — в противном случае.

После того, как для всех объектов $obj \in OBJ$ множества представителей типа стали неподвижными в результате последовательного распространения информации о типах по уравнениям и слияния, выполняется подборка типа из множества всех рассматриваемых типов *T*. Это выполняется процедурой *T* `match_type(Type(obj))`.

Ниже представлен алгоритм вычисления типов для программы program.

Здесь следует отметить, что выполнять слияние требуется каждый раз после завершения обхода дерева, так как правила распространения информации не симметричны для восходящего и нисходящего обходов. Если выполнять слияние после полного прохода по дереву, т. е. после прохода сверху вниз и снизу вверх, то, возможно, в результирующем множестве и после слияния останутся лишние элементы.

Восстановление производных типов данных

Задача алгоритма восстановления производных типов данных — извлечь максимум информации о возможных производных типах данных из ассемблерного листинга. Мы будем рассматривать следующие производные типы.

```
Tree:= make_tree(program);
Equation:= make_equation(Tree);
init (Equation);
set_constraint(Equation);
flag:=true;
while (flag) do begin
        flag:=false;
        for all equation: Equation do
                 spread_information_lr(equation);
        for all obj:OBJ do
                 flag ||= make_left_join(Type(obj));
        for all equation: Equation do
                 spread_information_rl(equation);
        for all obj:OBJ do
                 flag ||= make_right_join(Type(obj));
        for all obj:OBJ do
                 flag ||= make_full_join(Type(obj));
end
for all obj:OBJ do
        obj:=match_type((Type(obj));
```

Структуры. Структура — это совокупность элементов, располагаемых в памяти последовательно. Структура характеризуется своим размером и занимает в памяти область этого размера. Между полями структуры, а также в ее конце компилятор может резервировать память для обеспечения корректного выравнивания элементов структуры. В низкоуровневом ассемблерном коде имена полей, очевидно, отсутствуют, а весь доступ к полям ведется через смещения относительно адреса начала структуры. Поля структуры при декомпиляции можно восстановить, построив множество всех смещений, использованных в программе при обращении к областям памяти, в которых предположительно размещается структура некоторого, одного и того же типа. Для этого необходимо отслеживать типы базовых указателей в декомпилируемой программе, т. е. для каждых двух обращений к памяти по базовым адресам вычислять, гарантированно ли эти два обращения к памяти имеют один и тот же тип. Как было сказано выше, такую задачу можно решать с помощью метода продвижения типов по потоку данных, аналогичного методу продвижения констант.

Поскольку метод является консервативным, для некоторых пар обращений к памяти отождествление типов может быть не установлено, хотя, на самом деле, типы были тождественны. Тогда в декомпилированной программе возникнут клоны одного и того же структурного типа, возможно, с различным распознанным набором полей.

Очевидно, что гарантировать восстановление типов всех полей структуры при декомпиляции в общем случае невозможно, так как декомпилятор не имеет информации о полях, к которым не было обращений в декомпилируемом фрагменте кода. Такие поля представляются так же, как и память, зарезервированная под выравнивание, в виде массивов типа char необходимого размера.

Поля структуры могут быть произвольного типа: базового, указательного, массивового, структурного. Однако, случай, когда один структурный тип вложен в другой структурный тип, может оказаться неотличим от случая, когда поля вложенной структуры непосредственно находятся в объемлющем структурном типе. Например,

```
struct t1
{
   int t1;
   int t2;
};
   struct t2
{
   struct t1 tt;
   int t3;
};
```

```
struct t
{
   int t1;
   int t2;
   int t3;
};
```

В обоих случаях будет сгенерирован идентичный ассемблерный код. Поэтому структуры, вложенные в другие структуры, как правило, не могут быть автоматически выделены. Существуют некоторые эвристики, которые в ряде случаев могут помочь обнаружить вложенные структурные типы, однако в данной работе они рассматриваться не будут в силу своего частного характера.

Стоит отметить, что и область параметров функции может рассматриваться как структура, размещенная в стеке, на начало которой указывает регистр %ebp. Соответственно, автоматическое обнаружение значений структурных типов, размещенных в стековой области (то есть, другими словами, локальных переменных и параметров функций структурного типа) представляется затруднительным.

Поскольку локальные переменные и параметры функций предоставляют начальную информацию для восстановления типов при декомпиляции, они рассматриваются особенным образом, а не в общем контексте восстановления структурных типов.

Массивы. Массив — это совокупность однотипных элементов, размещаемая в памяти последовательно. В отличие от структур, доступ к элементам которых ведется с помощью константных смещений относительно начала области памяти, доступ к элементам массива производится с помощью индексных выражений. Например, для массива int m[32] и операции доступа m[j] будет сгенерировано адресное выражение, подобное m + j * 4. Операция умножения в адресных выражениях — отли-

чительная особенность обращений к массивам.

С другой стороны, поскольку все элементы массива имеют один и тот же тип, тип `m[0]` и `m[j]` совпадает, следовательно, вычисление смещения элемента массива ничего не дает для определения его типа и поэтому может быть отброшено. Таким образом, после отбрасывания вычисления адреса элемента массива тип каждого обращения к памяти однозначно определяется по базовому адресу и смещению относительно этого базового адреса.

Мультипликативные составляющие адресного выражения дают информацию о размерах элементов массива. Так, пример, приведенный выше, показывает, что константа 4 — это на самом деле размер элемента. В случае многомерных массивов адресное выражение имеет более сложную форму, в частности, в него входит несколько мультипликативных слагаемых, но размер элемента массива может быть определен как минимальная мультипликативная константа в адресном выражении.

Некоторые сложности могут возникать, если в индексном выражении в программе на языке высокого уровня уже используется умножение, например, `m[2*j]`. Тогда константа 2 в результате окажется в константном множителе адресного выражения, что дало бы нам неправильный размер массива. Чтобы избавиться от пользовательских мультипликаторов, необходимо взять наибольший общий делитель от всех мультипликативных констант, найденных во всех адресных выражениях, относящихся к одному и тому же массиву.

Если же все обращения к массиву используют один и тот же множитель, т. е., например, `m[2*j]`, то с точки зрения алгоритма восстановления производных типов, элемент массива окажется структурой из двух полей, причем тип первого поля будет восстановлен, а второе поле, поскольку к нему отсутствуют обращения, будет представлено как массив типа char требуемого размера.

Операции доступа к элементам массива с константным индексом, например, `m[5]`, транслируются в ассемблерной программе в доступ по константному смещению относительно начала массива. Для одиночного доступа такая ситуация будет неотличима от работы со структурой, но собирая информацию об использовании блока памяти по всей программе, неоднозначность выбора между полем структуры и элементом массива с константным индексом можно в ряде случаев устранить.

Основной сложностью представляется определение размера массива, т. е. количества элементов в нем. Язык С не предполагает никакого контроля границ массивов, поэтому во многих случаях информация о размере массива даже не попадает в ассемблерную программу. Одним из возможных подходов было бы вычисление диапазона значений для индексных переменных с помощью известных методов частичных вычислений, однако такой подход, опирающийся только на статический анализ программы, в большинстве случаев дает слишком низкую точность. Поэтому нам представляется более предпочтительным анализировать размещение элементов в памяти. В некоторых случаях, когда нам известен размер блока памяти, из размера блока памяти может быть выведен размер массива. В частности, если массив располагается между другими элементами (например, полями структуры или локальными переменными), размер массива может быть оценен сверху по ограничению адресов.

Формальное описание алгоритма. Допустим, что все обращения к памяти можно представить в виде:

$$\left(b+o+\sum_{j=0}^{n} C_j x_j\right),$$

где b (базовый адрес) — это изначально регистр, а впоследствии — объект;

o — смещение;

C_j — это константы.

Такое предположение справедливо в поставленных выше ограничениях на использование языка С в исходной программе. Без ограничения общности можно считать, что $C_j < C_{j+1}$. Пусть $m = \min_{j=0}^{n} C_j = C_0$ и $M = \min_{j=0}^{n} C_j = C_n$.

В дальнейшем выражения, описывающие доступ к памяти, будут преобразованы к виду

(*obj* + *offset*). При этом *obj* может быть не тем же самым объектом, что *b*, и даже может иметь другой базовый адрес. Такие ситуации могут возникнуть при восстановлении вложенных структурных типов. *offset* также может не совпадать со смещением *o* в исходном адресном выражении.

Присваивание меток `Assign_labels`. Пусть *l* — это метка, выбранная из перечислимого множества меток. Будем присваивать метки следующим элементам ассемблерной программы:

1) доступам к памяти: $l_i:(b_i, o_i, C_0, ..., C_n)$;

2) возвращаемым значениям подпрограмм (содержимое регистра %eax).

Выражения, вырабатывающие значения указательных типов в программе, являются строгим подмножеством всех помеченных элементов программы, так как некоторые помеченные элементы могут вырабатывать значения неуказательных типов.

Метки являются исходным материалом для восстановления типов. Другими словами, изначально каждая операция обращения к памяти имеет свой уникальный предварительный тип (метку). Далее эти уникальные предварительные типы (метки) объединяются во множества эквивалентности, из которых уже строятся окончательные типы.

Построение множества меток в регистрах (`Build_reg_label_sets`). Пусть $LS_{in}(reg, n)$ — это множество меток, соответствующее тем значениям, которые могут находится в регистре *reg* в точке, непосредственно предшествующей *n*-й строке ассемблерной программы. Аналогично, пусть $LS_{out}(reg, n)$ — это множество меток, соответствующее тем значениям, которые могут находиться в регистре в точке, непосредственно следующей за *n*-й строкой ассемблерной программы. Для простоты обозначения будем опускать квалификатор *in* для множества меток *LS*.

Анализ достижимости меток (`label-set analysis`) строит множество меток *LS* для каждого регистра и для каждой строки ассемблерной программы. Это прямой итеративный анализ потока данных, в котором в качестве функции слияния используется объединение множеств. Так как множество присваиваемых меток конечно, то анализ достижимости меток завершается за конечное число итераций. Анализ достижимости меток оперирует с регистрами, следовательно, не возникает проблемы алиасинга, так как регистры не могут быть синонимизированы.

Построение множеств эквивалентных меток (`Build_label_equiv_sets`). Основываясь на вычисленном множестве достижимых меток, между метками устанавливается отношение эквивалентности следующим образом:

$$\frac{\exists l_1, l_2, reg, n: (l_1, l_2) \subseteq LS(reg, n)}{l_1 \equiv l_2} \quad (1)$$

$$\frac{l_1:(b_1, o_1, C_{1,o}, ..., C_{1,n}), l_2:(b_2, o_2, C_{2,o}, ..., C_{2,n}), b_1 \equiv b_2}{l_1 \equiv l_2} \quad (2)$$

Равенство констант $C_{1,i}$ и $C_{2,i}$ в правиле 2 не требуется. Определенное таким образом отношение рефлексивное, транзитивное и симметричное по построению. Следовательно, все метки разбиваются на классы эквивалентности, и произвольная метка из каждого класса выбирается в качестве его представителя для определения производного типа данных всего множества. Каждый класс эквивалентности соответствует своему типу данных в восстанавливаемой программе на языке С.

Для каждой операции доступа к памяти $(b_i, o_i, m_i, ..., M_i)$ база b_i может быть заменена любым представителем класса эквивалентности, членом которого является b_i.

Две метки l_1 и l_2 имеют *общую базу*, если $l_1:(b_1, o_1, ...)$, $l_2:(b_2, o_2, ...)$ и $b_1 \equiv b_2$.

Построение агрегированных множеств (Build_aggreg_sets). Пусть t_j это — представитель *j*-го класса эквивалентности. Определим отношения *агрегирования в массив* для двух меток, обозначенное как $AA(l_1, l_2)$, следующим правилом вывода:

$$\frac{l_1:(t_1, o_1, C_{1,o}, ..., C_{1,n}), l_2:(t_1, o_2, C_{2,o}, ..., C_{2,n}), |o_1 - o_2| < C_{1,o}}{AA(l_1, l_2)}$$

Однако, для отношения *AA* в некоторых случаях может быть нарушено свойство транзитивности. Например, рассмотрим объявление типа

```
struct s1
   {
     struct s2
     {
         int f1;
         int f2;
     }   a[1];
     struct s3
     {
         int f3;
         int f4;
     }   b[1];
};
```

Операция доступа к полю **f1** имеет вид $(b,0,8)$, где b — это некоторый базовый адрес структуры, *0* — смещение поля **f1** относительно начала структуры, *8* — размер структуры struct s2, так как поле **f1** расположено в массиве структур этого типа. Аналогично, операция доступа к полю **f2** имеет вид $(b,4,8)$, к полю **f3** — $(b,8,8)$, к полю **f4** — $(b,12,8)$. Согласно правилу вывода 3 все они окажутся связанными отношением *AA*.

Отношение *AA* назовем *конфликтным*, если для него нарушена транзитивность, т. е., если существуют две метки l_1 и l_2, попавшие в один класс транзитивного замыкания отношения *AA*, что $|o_1 - o_2| \geq C_{1,o}$.

Для устранения конфликта разобьем конфликтный класс транзитивного замыкания на непересекающиеся подклассы так, чтобы

1) в один подкласс попали только метки, для которых $|o_1 - o_2| < C_{1,o}$;

2) для любых двух подклассов K_1 и K_2 $|\min_{K_1}(o) - \min_{K_2}(o)| \geq C_{1,o}$, т. е. разность минимальных смещений полей двух подклассов должна быть не меньше константы $C_{1,o}$ (размера массива), индуцировавшей данное разбиение.

Отношение *AA* с дополнительным разбиением устранения конфликтов разбивает множество всех меток на *множества агрегации*. Пусть множество

$K = \{l_1, l_2, ..., l_m\}$ — это какое-либо множество агрегации.

Пусть $o = \min_{j=1}^{m} o_j$.

Обозначим $t' = \langle b, o \rangle$ как метку для массивов вложенных структур, где $\langle b,o \rangle$ означает вычисление смещения от базы, но без разыменования.

Тем самым смещение *o* полагается началом вложенной структуры, а все остальные смещения в множестве *K* пересчитываются в смещения относительно *o*, т. е. для всех

$$l_i : (b, o_1, C_o, ..., C_m) \in K, l_i' = (t', o_i - o, C_{i,o}, ..., C_{i,m}).$$

Кроме того, поскольку агрегация в массив структур выполнена, константа C_o удаляется. В результате множество *K'* состоит из меток $l_i' = (t', o_i - o, C_{i,o}, ..., C_{i,m})$. Агрегирование не изменяет базу меток, поэтому если они до агрегирования имели общую базу, то и после него они также будут иметь общую базу.

Теперь, поскольку множество всех меток изменилось, в частности, за счет удаления констант C_0 для нового множества меток можно построить новое множество множеств агрегации меток и выполнить в нем новый шаг агрегации.

Этот процесс продолжается до тех пор, пока можно выполнить очередной шаг агрегации.

Восстановление структур (`Reconstruct_structs`). В итоге на основании вычисленной информации восстанавливаются структурные типы. Пусть *S* — это множество всех меток. Оно разбивается на классы эквивалентности $S_1, S_2, ..., S_m$, индуцируемые отношением «иметь общую базу», т. е. в одном множестве находятся все метки, имеющие одну базу. Пусть O_i — это множество смещений, непосредственно относящихся к общей базе для элементов множества S_i. Это множество смещений рассматривается как множество смещений полей нового структурного типа t_i, соответствующего множеству меток S_i.

Обновление множества объектов (`Update_object_sets`). Для всех полей нового типа создаются объекты и добавляются в множество рабочих объектов алгоритма восстановления базовых типов. Теперь задача алгоритма восстановления базовых типов состоит в отображении типов полученных объектов в базовые типы языка С.

Наконец, алгоритм, восстанавливающий производные типы данных, может быть реализован, как представлено в листинге ниже.

```
def composite_reconstruction()
    assign_labels()
    build_reg_label_sets()
    build_label_equiv_sets()
    build_aggreg_sets()
    reconstruct_structs()
    update_object_sets()
```

Этот алгоритм сходится, так как работает над конечным множеством меток и все используемые в его работе операторы монотонны.

Предлагаемый алгоритм восстановления производных типов данных языка С имеет ряд недостатков, как-то:

1) не обрабатываются объединения и операции приведения типов указателей;

2) не во всех случаях имеется возможность восстановить размер массива;

3) вложенные структуры могут быть восстановлены только в том случае, если они представимы в виде массива.

В итоге имеем полный алгоритм, который восстанавливает типы данных языка С и является композицией итеративного попеременного применения алгоритма восстановления производных типов данных и восстановления базовых типов языка С над итеративно обновляемыми данными.

Реализация и экспериментальная проверка

На рис. 1 представлена схема работы модуля восстановления типов данных.

Начальное множество объектов для анализа строится по листингу ассемблерной программы, как это описано выше. Далее полученное множество объектов поступает на вход модулю, который реализует алгоритм восстановления базовых типов данных. В результате одной итерации работы модуля на выходе сформированы три группы объектов.

1. Объекты, типы которых восстановлены. Этим объектам в исходной С программе соответствовали переменные базовых типов данных, и эти переменные участвовали в таких операциях, по которым однозначно восстанавливается тип переменной.

2. Объекты, типы которых не восстановлены. Для восстановления типов данных этих объектов нужно учесть информацию о типах, которая была получена на предыдущей итерации работы модуля восстановления базовых типов данных.

3. Объекты, тип которых распознан как указательный. Объекты, типы которых не восстановлены, опять подаются на вход алгоритму восстановления базовых типов данных. Объекты, тип которых распознан как указательный, соответствуют косвенным объектам и подаются на вход модулю, в котором реализован алгоритм восстановления производных типов данных. Модуль восстановления производных типов данных для множества косвенных объектов строит скелеты производных типов, а также создает новые объекты для полей структурных типов и первого элемента массива. Множество новых объектов объединяется с множеством объектов, тип которых не восстановлен, и новое множество объектов подается на вход алгоритму восстановления базовых типов данных. Объекты, соответствующие полям структур и первому элементу массива, могут также быть косвенными. Производный

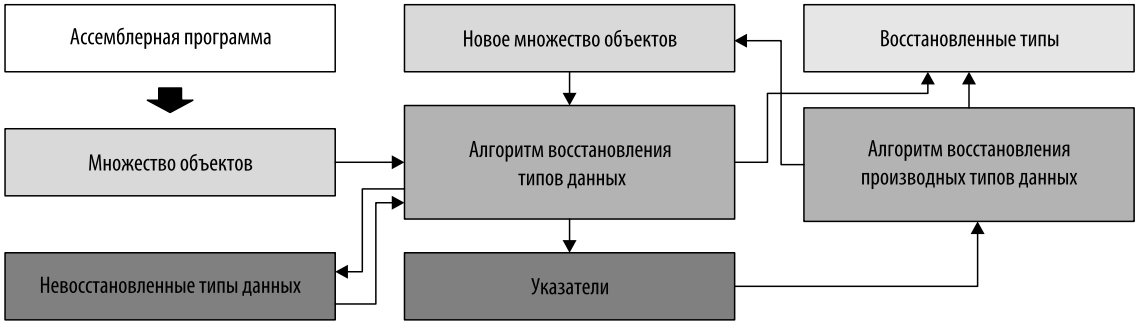

Рис. 1. Организация работы модуля восстановления типов данных

тип данных считается восстановленным, если для всех полей структуры восстановлены все типы ее полей вне зависимости от уровня косвенности, а для массива восстановлен тип его элементов. Как отмечалось выше, размер массива удается восстановить не во всех случаях.

Алгоритм завершает работу, когда множество новых объектов достигает неподвижной точки, т. е. при следующей итерации работы всего алгоритма восстановления типов данных оно не изменяется. Если при достижении неподвижной точки множество новых объектов не пусто, это означает, что при восстановлении этих типов обнаружен конфликт. При наличии конфликта восстанавливается тип данных union. Алгоритм всегда завершает работу, так как все переменные указательного типа в языке С имеют конечный уровень косвенности. В случае рекурсивно определенного производного типа данных алгоритм восстановления производных типов данных также завершается. Алгоритм восстановления базовых типов данных оперирует конечной решеткой, элементами которой являются тройки атрибутов, с конечной монотонной функцией слияния, определенной как почленное пересечение множеств троек атрибутов. В табл. 1 приведены результаты ручной проверки качества восстановления типов данных. Колонка «CLOC» содержит количество строк кода в исходном коде на языке С, колонка «ALOC» содержит количество строк кода в соответствующем ассемблерном коде.

Результаты статического восстановления типов данных представлены в табл. 2. Детальное восстановление типов данных на примере функции isnow представлено в табл. 3 (с. 116). Так как переменная char * endp используется только как аргумент функции getfield и никогда не разыменовывается, в результате только статического анализа она восстановлена как 4-хбайтная переменная типа int. Однако по результатам динамического анализа тип может быть восстановлен как указатель void*.

Колонка «BT» табл. 3 показывает количество переменных базового типа данных на стеке; переменные, расположенные на регистрах, не учитываются. Колонка «ExactBT» показывает количество переменных, тип которых был восстановлен точно, а колонка «CorrBT» показывает количество переменных, тип которых был восстановлен корректно. Колонка «FailBT» показывает количество переменных,

Таблица 1

Пример ручной проверки восстановления типов данных

Пример	CLOC	ALOC	Описание
35_wc	107	245	`cnt()` функция утилиты wc (file wc.c)
36_cat	27	104	`raw_cat()` функция утилиты cat (file cat.c)
37_execute	486	1167	`execute()` функция утилиты bc (file execute.c)
38_day	173	346	`isnow()` функция утилиты calendar (file day.c)

Таблица 2

Статистика ручной проверки восстановления типов данных

Пример	BT	Exact BT	Corr BT	Fail BT	PTRs	Exact PTRs	Corr PTRs	Fail PTRs
35_wc	7	1	6	0	2	1	1	0
36_cat	7	4	3	0	1	0	0	0
37_execute	1	0	1	0	0	0	0	0
38_day	9	8	1	0	4	2	1	1

Таблица 3

Детальное восстановление типов функции isnow

Адрес объекта	Восстановленный тип	Исходный тип
[ebp + 8]	[unsigned] int, void* (profile-based)	char *endp
[ebp + 12]	int*	int *monthp
[ebp + 16]	int*	int *dayp
[ebp + 20]	[unsigned] int*	int *varp
[ebp – 28]	[unsigned] int*	int flags
[ebp – 24]	int	int day
[ebp – 20]	int	int month
[ebp – 16]	int	int v1
[ebp – 12]	int	int v2

тип которых не удалось восстановить ни точно, ни корректно. Колонка «PTR» содержит количество переменных типа «указатель на переменную базового типа». Колонки «EcaxtPTR», «CorrPTR», «FailPTR» показывают количество переменных указательного типа, тип которых удалось восстановить точно, корректно и не удалось восстановить ни точно, ни корректно соответственно.

На рис. 2 и рис. 3 показано графическое представление восстановления производных типов данных. На рис. 2 представлено восстановление структуры sturct shorts программы lalr.c. На рис. 3 представлено восстановление структуры struct tm из файла <time.h> программы day.c. Здесь следует заметить, что некоторые поля, например, поле tm_sec и другие, восстановлены как «pad», т. е. в виде заглушек, потому что в программе к ним не было обращений.

На рис. 4 представлен результат работы декомпилятора *TyDec* по восстановлению типов данных для примеров 59_lalr и 38_day. Представлено две программы. Слева расположено окно, отображающее входную программу,

```
typedef struct shorts {
   struct shorts *next;
   short value;
   } shorts;
   short *lookaheads;
```

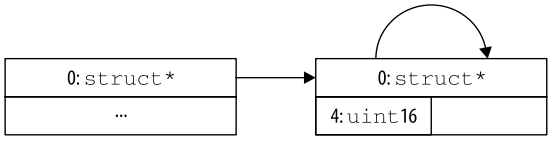

Рис. 2. Пример восстановления структурного типа программы lalr.c

```
struct tm {
   int   tm_sec, tm_min, tm_hour;
   int   tm_mday, tm_mon, tm_year;
   int   tm_wday, tm_yday, tm_isdst;
   long  tm_gmtoff;
   char  *tm_zone;
};
```

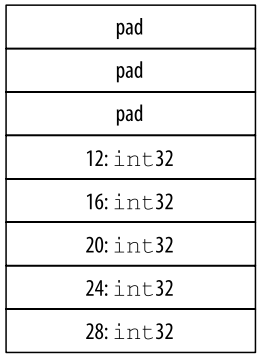

Рис. 3. Пример восстановления структурного типа программы day.c

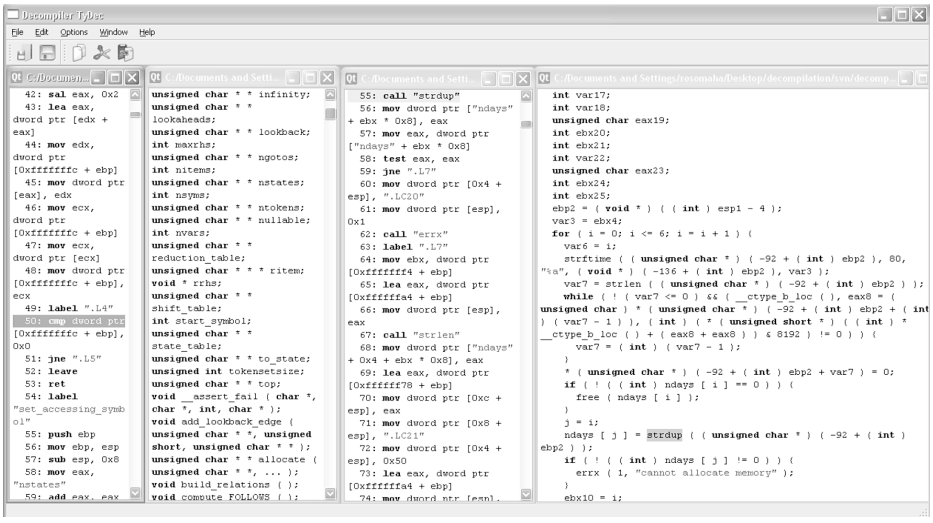

Рис. 4. Результат работы компоненты восстановления типов данных

а справа от него находится окно, отображающее восстановленную программу. Как можно заметить, на рисунке представлено успешное восстановление знаковости типов, указателей более одного уровня косвенности, а также показано восстановление операций доступа к элементам массива.

СПИСОК ЛИТЕРАТУРЫ

1. *Деревенец Е. О., Трошина Е. Н.* Структурный анализ в задаче декомпиляции // *Прикладная информатика.* № 4. С. 87–99.

2. *Трошина Е. Н.* Диссертация на соискание ученой степени кандидата физико-математических наук по специальности 05.13.11 — Математическое и программное обеспечение вычислительных машин, комплексов и компьютерных сетей // Научная библиотека МГУ им. М. В. Ломоносова. 20.11.2009.

3. *Dolgova K., Chernov A.* Automatic Type Reconstruction in Disassembled C Programs // Proceedings of IEEE 15th Working Conference on Reverse Engineering 2008. Antwerp, Belgium. October 2008. P. 202–206.

4. *Troshina K.* and *Chernov A.* High-Level Composite Type Reconstruction During Decompilation from Assembly Programs // Proceedings of 7th Perspectives of System Informatics. Akademgorodok. Novosibirsk. Russia. 15–19 June 2009. P. 292–299.

5. *Troshina K., Chernov A., Derevenets Y.* C Decompilation: Is It Possible? // Proceedings of International Workshop on Program Understanding. Altai Mountains, Russia. 19–23 June 2009. P. 18–27.

6. *Долгова К. Н., Чернов А. В.* Автоматическое восстановление типов в задаче декомпиляции // *Программирование* [журнал Российской академии наук]. № 2. 2009. Март—апрель. С. 63–80.

7. *Mycroft A.* Type-Based Decompilation // European Symp.on Programming. 1999. Pp. 208–223.

8. *Balacrishnan G., Reps T.* DIVINE: Discovering variables in executables // *Verification, Model Checking, and Abstract Interpretation.* 2007. P. 1–28.

9. *Balakrishnan G., Ganai M.* PED: Proof-guided Error Diagnosis by Triangulation of Program Error Causes // Proc. of Software Engineering and Formal Methods (SEFM), 2008.

10. *Гусенко М. Ю.* Декомпиляция типов данных исполняемых программ // *Безопасность информационных технологий.* 1998. С. 83–88.

11. *Cifuentes C., Fraboulet A.* Assembly to high-level language translation // Int. Conf.on Softw. Maint., 1998. P. 223–237.

12. *Cifuentes C., Emmerik M., Lewis B., Ramsey N.* Experience in the Design, Implementation and Use of a Retargetable Static Binary Translation Framework // Technical Report, 2002.

13. Декомпилятор Hex-Rays // Hex-Rays Decompiler SDK. URL: http://www.hex-rays.com.

14. Интерактивный дизассемблер Ida Pro. URL: http://www.idapro.ru.

15. *Muchnick S.* Advanced Compiler Design and Implementation // Morgan Kaufmann Publishers, 1997.