М. А. Потапов, Е. А. Шатохин, Некоторые методы автоматизированного анализа и управляемого преобразования программ, *Автомат. и телемех.*, 2008, выпуск 8, 176–186

Автоматика и телемеханика, № 8, 2008

PACS 02.70.-c

© 2008 г. **М.А. ПОТАПОВ, канд. физ.-мат. наук,**
Е.А. ШАТОХИН
(Институт автоматизации проектирования РАН, Москва)

НЕКОТОРЫЕ МЕТОДЫ АВТОМАТИЗИРОВАННОГО АНАЛИЗА И УПРАВЛЯЕМОГО ПРЕОБРАЗОВАНИЯ ПРОГРАММ

Рассматриваются принципы и методы создания программных систем, облегчающих анализ и преобразование структуры программ. При реализации масштабных проектов невозможно иметь полное представление о структуре программы без использования специальных систем. Такие системы содержат средства анализа исходной программы и в результате автоматизированного преобразования создают другую программу, обладающую заданными свойствами. В качестве примеров рассматриваются задачи быстрого автоматического дифференцирования и задачи обфускации ("затемнения", запутывания) программ.

1. Введение

История развития математических исследований, являющихся основой современных методов автоматизированного анализа и управляемого преобразования программ, восходит к временам Ньютона, Лейбница, Эйлера и Вейерштрасса. Целенаправленно и интенсивно эти проблемы начали изучаться с появлением электронно-вычислительных машин. Одной из первых работ, где были рассмотрены вопросы автоматизированного программирования и введен целый ряд терминов и понятий (компьютерная (машинная) алгебра, вычислительный граф (схема) алгоритма), широко используемых в наше время [1], стала [2]. Важным продвижением в развитии компьютерной алгебры стала статья [3]. В нашей стране в последние годы ряд аспектов этой тематики получил современное освещение и развитие в [4,5] и др. Практическая реализация идей компьютерной алгебры и автоматизированного программирования в различных приложениях началась сравнительно недавно – с 80-90-х годов.

Вручную, как правило, затруднительно анализировать зависимости данных в программах с большим числом операций. К настоящему времени разработаны средства, облегчающие анализ и преобразование структуры программ для различных классов задач. Эти программные системы используют возможности современных языковых средств и соответствующих трансляторов. Такие системы содержат средства анализа исходной программы и в результате автоматизированного преобразования создают другую программу, обладающую заданными свойствами.

Применение таких средств оказывается полезным в самых разных задачах, например:

1. При выполнении быстрого автоматического дифференцирования функций. Использование графа алгоритма позволяет быстро и с высокой точностью вычислять

производные функций, заданных в виде программного кода. Методы быстрого автоматического дифференцирования обладают реальными преимуществами по сравнению с традиционными конечно-разностным и символьным дифференцированием.

2. Для обфускации ("затемнения", запутывания) программ при решении проблем информационной безопасности.

3. При разработке эффективных алгоритмов и их программных реализаций для параллельных вычислительных систем [5].

В данной статье рассматриваются вопросы использования свойств структуры алгоритмов при реализации быстрого автоматического дифференцирования и в задачах обфускации программ. Задача быстрого автоматического дифференцирования является одной из центральных при управлении динамическими системами в режиме реального времени. Сравнительно новая проблема обфускации программ важна при управлении автоматизированным преобразованием программ с целью создания максимальных препятствий для их несанкционированного использования.

2. Основные понятия

Код (программа) на некотором языке программирования обычно описывает не один алгоритм, а некоторое семейство алгоритмов [5]. При выполнении кода реализуемый алгоритм зависит от того, как срабатывают условные операторы и т.д. Это в свою очередь определяется входными данными программы.

Пусть при реализации алгоритма выполняется конечный набор операций из некоторого множества M ("множество элементарных операций"). M может быть, например, множеством арифметических операций, может включать в себя математические функции из стандартных библиотек используемых языков программирования и т.д. Операции из M нередко имеют не более двух аргументов.

Пусть при конкретных входных данных программа описывает некоторый алгоритм. Сопоставим этому алгоритму ориентированный граф $G = (V, E)$ следующим образом [5].

В данном алгоритме выполняется конечная последовательность операций из M. (Одна и та же операция из M может встретиться и несколько раз.) Элементы этой последовательности можно перенумеровать. В качестве множества вершин V графа возьмем множество номеров элементов. Ниже будем говорить для краткости, что в вершинах графа G находятся операции алгоритма.

Для каждой пары вершин (a, b) из V построим ребро (a, b) в том и только в том случае, если результат операции в вершине a является аргументом для операции в вершине b.

Если аргументами операции в некоторой вершине b являются начальные данные алгоритма, будем считать их результатом специальной операции ввода данных (in) из M. Тогда начальным данным алгоритма (и только им) соответствуют вершины G, не имеющие ни одной входящей дуги. Такие вершины будем называть *входными*. Соответственно, вершины, не имеющие ни одной исходящей дуги, будем называть *выходными* [5]. Среди выходных вершин есть те, результаты операций в которых являются искомыми результатами работы алгоритма.

Построенный таким образом ориентированный ациклический мультиграф G в [5] называется графом информационной зависимости реализации алгоритма при фиксированных входных данных, или, короче, *графом алгоритма*. В англоязычной литературе этому понятию соответствует выражение "*computational graph*" [6]. Граф алгоритма является одним из представлений "явной схемы", предложенной Л.В. Канторовичем [2].

Еще раз отметим: граф алгоритма строится для **фиксированных** входных данных программы. При других входных данных он может быть иным. В общем случае граф алгоритма не является ни деревом разбора соответствующей программы, ни control flow графом и т.д. [6].

3. Быстрое автоматическое дифференцирование

3.1. Общие сведения

Быстрое автоматическое дифференцирование (fast automatic differentiation, FAD) – один из методов вычисления производных функции, заданной в виде кода на каком-либо языке программирования [1]. Процесс вычисления значения функции для конкретных входных данных представляет собой конечную последовательность операций из множества M. Выражения для производных этих операций известны заранее. Тогда, применяя правило дифференцирования сложной функции, можно получить значение производной искомой функции.

Пример 1. Пусть функция $z(x)$ определяется так: $z = f(g(h(x), k(x)))$, где f, g, h, k – операции из M (см. рис. 1). Необходимо найти производную функции z в точке x_0. (Предположим, что производные функций f, g, h, k существуют в соответствующих точках.) Обозначим: $h_0 = h(x_0)$, $k_0 = k(x_0)$, $g_0 = g(h_0, k_0)$.

В "прямом" методе быстрого автоматического дифференцирования ("forward mode") последовательность действий такова [6]:

1) $\mathrm{d}h = h'(x_0)\mathrm{d}x$;
2) $\mathrm{d}k = k'(x_0)\mathrm{d}x$;
3) $\mathrm{d}g = (g_h(h_0, k_0)h'(x_0) + g_k(h_0, k_0)k'(x_0))\mathrm{d}x$;
4) $\mathrm{d}z = f'(g_0)(g_h(h_0, k_0)h'(x_0) + g_k(h_0, k_0)k'(x_0))\mathrm{d}x$.

Это соответствует проходу по графу алгоритма от входных вершин к выходным, т.е. в "прямом" направлении: в таком же порядке выполняются операции при вычислении $z(x_0)$.

В "обратном" методе ("reverse mode") проход по графу алгоритма осуществляется в противоположном направлении – от вершины, соответствующей искомой функции, до входных вершин:

1) $\mathrm{d}z = f'(g_0)\mathrm{d}g$;
2) $\mathrm{d}z = f'(g_0)(g_h(h_0, k_0)\mathrm{d}h + g_k(h_0, k_0)\mathrm{d}k)$;
3) $\mathrm{d}z = f'(g_0)(g_h(h_0, k_0)h'(x_0)\mathrm{d}x + g_k(h_0, k_0)\mathrm{d}k)$;
4) $\mathrm{d}z = f'(g_0)(g_h(h_0, k_0)h'(x_0) + g_k(h_0, k_0)k'(x_0))\mathrm{d}x$.

В обоих случаях погрешность полученного значения производной определяется только погрешностью вычисления производных элементарных операций и погрешностью вычисления суммы и произведения. Способы вычисления производных для операций из M с требуемой точностью предполагаются известными заранее. Поэтому погрешность значения $z'(x_0)$ возникает только за счет накопления ошибок округления при выполнении сложения и умножения.

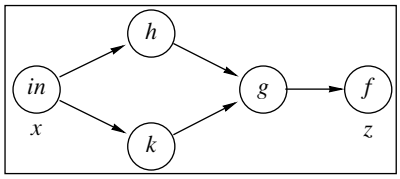

Рис. 1. Граф алгоритма вычисления $z(x)$.

Рассмотрим более общий случай. Пусть значение функции $z : R^p \to R$ вычисляется в следующем процессе:

$$(1) \qquad \begin{cases} u_k = F_k\left(u_{k_1}, \ldots, u_{k_{s(k)}}\right), \quad p < k \leqslant n; \quad 1 \leqslant k_1, \ldots, k_{s(k)} < k, \\ z(u_1, \ldots, u_p) = u_n, \end{cases}$$

$u_1, \ldots, u_p$ – независимые переменные (входные данные алгоритма), F_k – операции из M, $s(k)$ – число аргументов операции F_k.

Пусть процесс (1) проведен для точки $u = (u_1, \ldots, u_p)$, пусть также все частные производные операций F_k существуют в точке u. Необходимо найти $\nabla z(u)$ – градиент функции z в этой точке.

Граф алгоритма (1) имеет n вершин. Сопоставим каждой вершине число a_k. В вершину с номером k $(p < k \leqslant n)$ входят дуги только из вершин $k_1, \ldots, k_{s(k)}$. Вершины $1, 2, \ldots, p$ являются входными (см. раздел 2).

"*Прямой*" метод в данном случае можно реализовать так. Пусть w – произвольный p-мерный вектор. Положим $a_k = w_k$, $k = 1, \ldots, p$. Для каждой не входной вершины $(p < k \leqslant n)$ выполним следующие операции:

1) найдем все частные производные функции F_k в соответствующих точках;

2) вычислим a_k:

$$a_k = \sum_{i=1}^{s(i)} \frac{\partial F_k}{\partial u_i} a_{k_i}.$$

При этом порядок обхода вершин графа может быть разным. Главное, чтобы вершина № k обрабатывалась **после** того, как обработаны все вершины для аргументов F_k (т.е. $k_1, \ldots, k_{s(k)}$).

После выполнения этих операций a_n равен скалярному произведению ∇z в данной точке на вектор w. Если w – j-й единичный координатный вектор, то a_n – значение производной функции z по u_j в точке u.

Операции в вершине № k выполняются за время не более $C_1 T_k$, где T_k – время выполнения операций по вычислению значения F_k, C_1 – константа, не зависящая от n и p. Поэтому общее время вычисления одной компоненты градиента функции z "прямым" методом не превосходит $C_1 T$, где T – время вычисления $z(u)$. Все компоненты градиента будут определены за p таких проходов по графу алгоритма. Общее количество операций над числами при определении ∇z в "прямом" методе пропорционально p.

В случае, когда $z : R^p \to R^m$, "прямой" метод позволяет за один проход по графу алгоритма определить значения частных производных всех m компонент искомой функции по данному аргументу.

"*Обратный*" метод можно реализовать следующим образом. Положим $a_n = 1$, $a_k = 0, k < n$. Для всех $k = n-1, n-2, \ldots, p+1$ выполним следующие операции:

1) найдем все частные производные функции F_k в соответствующих точках;

2) для всех вершин, соответствующих аргументам операции F_k (т.е. для вершин с номерами $k_1, \ldots, k_{s(k)}$), выполним:

$$a_{k_i} = a_{k_i} + \frac{\partial F_k}{\partial u_{k_i}} a_k, \quad i = 1, \ldots, s(k).$$

Порядок обхода вершин графа алгоритма должен быть таким, чтобы вершина № k обрабатывалась **раньше** вершин, соответствующих аргументам F_k.

После выполнения указанных выше операций a_j – значение производной функции z по u_j в точке u. Таким образом, за один проход по графу алгоритма определены **все компоненты градиента** функции z. Операции в вершине № k выполняются за

время не более C_2T_k, где C_2 – константа, не зависящая от n и p. Поэтому общее время вычисления ∇z "обратным"' методом не превосходит C_2T, где T – время вычисления $z(u)$. Отметим, что явной зависимости от количества начальных данных (p) для этой оценки нет.

В [5] рассматривается способ определения градиента, аналогичный "обратному" методу быстрого дифференцирования (только в матричных обозначениях). Имеет место следующее утверждение (см. [5]).

Утверждение 1. Пусть множество M включает в себя операции сложения, вычитания, умножения двух чисел и вычисления обратной величины. Пусть также (для простоты) эти операции эквивалентны по сложности реализации.

Тогда независимо от величины p одновременное вычисление значения функции z и ее градиента ∇z в одной и той же точке может быть осуществлено с затратами по числу операций, не превосходящими затрат на четырехкратное вычисление значений функции.

3.2. Быстрое автоматическое дифференцирование и другие методы вычисления производных

В настоящий момент в программных системах реализуются следующие способы вычисления производных:

1) символьное дифференцирование;

2) дифференцирование с использованием разностных аппроксимаций ("конечно-разностное" дифференцирование);

3) быстрое автоматическое дифференцирование.

Применение символьного подхода дает возможность получить производную функции на некотором подмножестве области определения этой функции. Конечно-разностные методы и быстрое автоматическое дифференцирование позволяют вычислить производную функции в конкретной точке.

Вычисление градиента функции $z : R^p \to R$ в точке x с помощью разностных методов требует не менее $p+1$ вычислений самой функции z. (Нужно найти значения функции z в точке x и еще в p точках вида $(x_1, \ldots, x_k + h_k, \ldots, x_p)$, h_k – приращение k-го аргумента.) Время, необходимое для определения $z(x)$ и $\nabla z(x)$ разностными методами, можно оценить так: $T(z, \nabla z) \geqslant (p+1)T$, где T – время вычисления $z(x)$.

В [7] рассматриваются функции, для которых вычисление производных методами быстрого автоматического дифференцирования требует как минимум в 1,5 – 2 раза меньше времени, чем при использовании разностных методов. "Обратный" метод быстрого дифференцирования позволяет получить $z(x)$ и $\nabla z(x)$ в данной точке за время $T(z, \nabla z) < CT$, где C не зависит от p. В проведенных тестах $2 < C < 6$ (результаты получены для разработанной авторами программной реализации методов быстрого автоматического дифференцирования).

Методы как символьного, так и быстрого автоматического дифференцирования обрабатывают последовательность элементарных операций исследуемой функции. Однако в случае быстрого дифференцирования эта последовательность строится для *конкретных* значений аргументов. При применении символьных методов необходимо иметь такую последовательность для *всех* допустимых значений аргументов из области определения исследуемой функции.

Поскольку при программной реализации быстрого автоматического дифференцирования не нужно хранить всю структуру функции, а только ее часть (граф алгоритма), системам быстрого дифференцирования требуется для работы, как правило, меньше оперативной памяти, чем системам, реализующим символьное дифференцирование.

Важные преимущества быстрого автоматического дифференцирования по сравнению с символьными методами проявляются при вычислении производных функций, заданных рекурсивно. Рассмотрим следующий пример.

Пример 2. Пусть функция $y(x)$ задается следующим образом:

$$y = f(x,m),\ x \in (0,1),\ m = 0,1,2,\ldots \qquad \text{где } h(u) = \begin{cases} 4u(1-u),\ u < 0{,}5, \\ 1 - 0{,}75u^2,\ u > 0{,}5, \\ -1,\ u = 0{,}5. \end{cases}$$
$$\begin{cases} f(x,0) = x, \\ f(x,i) = h(f(x,i-1)),\ i \geqslant 1, \end{cases}$$

Глубину рекурсии m считаем фиксированной, но заранее неизвестной. (Например, пусть m задается пользователем.) При каждом m функция $y(x)$ дифференцируема на $(0,1)$ за исключением конечного числа точек (их не более 2^m).

Системы быстрого дифференцирования не испытывают трудностей с вычислением производной функции $y(x)$. При каждом m последовательность элементарных операций для $y(x)$ конечна – можно построить соответствующий граф алгоритма и определить по нему $y'(x)$, как описано выше.

Однако поскольку аргумент функции $y(x)$ и глубина рекурсии m заранее не известны, для систем символьного дифференцирования данный пример может оказаться трудным: необходимо будет найти выражение для $y(x)$ при **произвольных** m и x. Это может привести к большему расходу памяти и более низкой скорости работы по сравнению с системами быстрого дифференцирования.

3.3. Особенности реализации быстрого автоматического дифференцирования в программных системах

Преобразование кода вычисления функции ("пользовательского кода") в код, вычисляющий еще и ее производную, может быть осуществлено автоматически с помощью специальных программных средств. При этом, как правило, применяется один из следующих методов [8].

1. Непосредственное преобразование исходного кода (source transformation). При таком подходе в нужных местах пользовательской программы явно добавляются вызовы функций для записи и обработки графа алгоритма. Это обычно делается с помощью специализированных компиляторов (или их front-end'ов), препроцессоров и т.д.

2. Использование перегрузки (overloading) операторов и функций. Разработчик пользовательской программы при этом изменяет типы данных в соответствующем участке кода. Для этих новых типов данных операции (арифметические и др.) переопределены. При обработке такой программы компилятор сам добавляет в нее вызовы функций для построения и обработки графа алгоритма. Компилятор при этом может быть любым, не обязательно специализированным для реализации автоматического дифференцирования.

Рассмотрим следующий пример.

Пример 3. Пусть исходный фрагмент пользовательского кода на языке C++ имеет такой вид:

```
double a, b, c;
...// ввод исходных данных
a=cos(b/c);
b=a*c;
```

Граф алгоритма для данного фрагмента показан на рис. 2.

Допустим, необходимо продифференцировать функцию, заданную этим участком программы, по аргументу c.

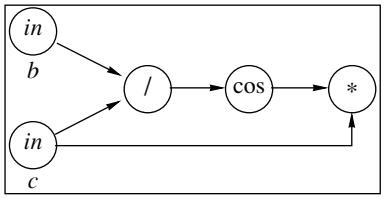

Рис. 2. Граф алгоритма для фрагмента программы.

После применения средств непосредственного преобразования кода этот фрагмент может выглядеть, например, так (функции вида `createXXXNode()` создают новые узлы графа алгоритма для соответствующих операций: умножения, деления и др.):

```
double  a, b, c, t1;
...// ввод исходных данных
CNode *ndA, *ndB,*ndC, *nd1;
CGraph* g=createComputationalGraph(); // создать новый граф алгоритма
ndB=g->createInputNode(b);
ndC=g->createInputNode(c);
t1=b/c;
nd1=g->createDivNode(ndB, ndC, t1);
a=cos(t1);
ndA=g->createCosNode(nd1, a);
b=a*c;
ndB=g->createMulNode(ndA, ndC, b);
// теперь граф построен
g->reverseDiff(ndB,ndC); /* выполнить "обратный"метод быстрого
дифференцирования для графа алгоритма */
```

Для использования средств, основанных на перегрузке операторов и функций, разработчику достаточно изменить исходный фрагмент кода таким образом:

```
CADouble a, b, c; // эти переменные имели раньше тип double.
...// ввод исходных данных
CGraph* g;
beginSection(g); // указать начало нужного участка кода
a=cos(b/c); // здесь все выглядит так же, как и раньше
b=a*c;
endSection(); // указать конец нужного участка кода
// теперь граф алгоритма построен
g->reverseDiff(b,c); /* выполнить "обратный"метод быстрого
дифференцирования для графа алгоритма */
```

Тип данных `CADouble` определяется в программной системе (библиотеке классов), реализующей быстрое автоматическое дифференцирование. Для этого типа данных операции умножения, деления и др. переопределены (“перегружены”). Компилятор при обработке преобразованного фрагмента кода сам заменяет их на вызовы функций для построения узлов графа алгоритма и т.д.

Заметим, что в случае использования перегрузки операторов код вычисления значения функции, которую надо продифференцировать, внешне не изменился (a=cos(b/c); b=a*c;). Это удобно при разработке и отладке пользовательской про-

граммы: вся работа с графом алгоритма скрыта в реализациях перегруженных операций и не загромождает код. Программные средства на основе непосредственного преобразования кода таким достоинством, как правило, не обладают.

Применение перегрузки также дает возможность обрабатывать программу любым компилятором для данного языка программирования. Это особенно важно, если используемый компилятор лучше оптимизирует код или, например, лучше соответствует стандартам данного языка программирования, чем те или иные специализированные компиляторы, реализующие методы быстрого дифференцирования.

На наш взгляд, использование непосредственного преобразования кода имеет лишь одно преимущество по сравнению с перегрузкой операций: последняя возможна не во всех языках программирования. Однако в таких широко используемых языках, как C++, C#, Fortran 90 (как и в более поздних версиях этого языка), Ada и т.д., перегрузка операторов и функций допустима.

В настоящее время методы быстрого автоматического дифференцирования с помощью непосредственного преобразования кода реализованы в таких программных системах, как:

– *ADIC* и *ADIFOR*. Системы разработаны сотрудниками Argonne National Laboratory и Rice University. Используются в проекте NEOS [9], при поиске оптимальной формы поверхности летательных аппаратов в NASA Langley Research Center, в компании Boeing [10] и т.д.;

– *TAPENADE*. Система разработана "TROPICS Team". Используется в проекте "Tapenade" по решению оптимизационных задач в режиме online [11];

– и т.д.

С использованием перегрузки операций методы быстрого дифференцирования реализованы в следующих программных системах:

– *ADOL-C*. Система применяется в проектах: NEOS, GasNetOpt (Zuse Institut Berlin) и т.д. [7];

– *FADBAD++*, *TADIFF* и др. Системы предназначены, в основном, для демонстрации возможностей методов быстрого дифференцирования;

– и т.д.

Разработанная авторами система, реализующая методы быстрого автоматического дифференцирования, ориентирована на язык C++ и использует перегрузку операторов и функций. Основные компоненты системы:

1) подсистема управления памятью;

2) определения специальных типов данных, реализации перегруженных операторов и функций;

3) набор функций-обработчиков для разных типов узлов графа алгоритма.

Специализированная подсистема управления памятью – важная часть данного программного комплекса. Она отвечает за хранение представления графа алгоритма и других данных в оперативной памяти или на жестком диске, предоставляет средства для выделения, освобождения и доступа к памяти, занятой этими данными. От эффективности реализации этого компонента в значительной степени зависит производительность системы в целом, т.е. скорость работы полученного после преобразования программного кода [12].

О назначении специальных типов данных и переопределенных операций с ними уже говорилось выше, в комментариях к примеру 3.

Функции-обработчики вызываются для узлов графа алгоритма при выполнении автоматического дифференцирования и т.д. (В примере 3 они не показаны, так как вызываются внутри `reverseDiff()`.) В этих функциях реализуются все те операции,

которые необходимо выполнить в каждом узле графа алгоритма при применении "прямого" или "обратного" метода (см. раздел 3.1).

Проведенные тесты показывают, что данная система по скорости работы получаемого с ее помощью кода не уступает существующим аналогам (ADOL-C, FADBAD++), а в ряде случаев и превосходит их (в 2–3 раза и более).

4. Обфускация программ и структура алгоритмов

Обфускация (obfuscation) – один из методов защиты программного кода, позволяющий усложнить процесс анализа и модификации этого кода посторонними лицами ("злоумышленниками"). При обфускации защищаемая программа преобразуется в эквивалентную ей (т.е. обе эти программы при одних и тех же входных данных дадут один и тот же результат). На практике, как правило, целесообразно применять такие методы защиты не для всего кода программы, а только для некоторых ("критических") его участков [13]. Это связано, например, с тем, что обфускация может привести к значительному увеличению времени работы защищаемой программы.

В настоящее время одним из наиболее перспективных направлений в области затруднения анализа и несанкционированного использования программ является разработка технологий встраивания кодовой последовательности ("пароля") в сам процесс работы защищаемого фрагмента программы. При этом преобразование программного кода желательно выполнить так, чтобы никакая часть кодовой последовательности (в исходной или преобразованной форме) не фигурировала в качестве данных, с которыми работает программа.

Рассмотрим $G = (V, E)$ – граф алгоритма для защищаемого фрагмента. В общем случае этот граф зависит от входных данных, т.е. является параметризованным графом [5]. Пусть x – набор входных данных для рассматриваемого фрагмента кода. От x может зависеть не только множество вершин графа, но и множество дуг: $G(x) = (V(x), E(x))$.

Пусть по $G(x)$ с учетом кодовой последовательности построен $G_1(x) = (V_1(x), E_1(x))$ – граф, тоже зависящий от параметра x, такой что при каждом x $G(x)$ является подграфом $G_1(x)$. При каждом x число дуг N_1 графа $G_1(x)$ должно быть велико по сравнению с N – количеством дуг $G(x)$. В идеале N_1 должно расти экспоненциально с ростом N. Кодовая последовательность в данном случае определяет выбор подграфа графа $G_1(x)$. Верному ее значению при каждом x должен соответствовать $G(x)$.

Таким образом, с учетом заданной разработчиком кодовой последовательности производится преобразование фрагмента кода, соответствующего $G(x)$, в фрагмент, соответствующий $G_1(x)$. Кодовая последовательность в данном случае определяет структуру защищаемого фрагмента программы. При таком подходе нет необходимости в проверках правильности введенного пользователем пароля. Структура защищенного фрагмента программы такова, что он будет работать *при любом* введенном пароле, но работать правильно будет тогда и только тогда, когда пароль совпадает с "заложенной" в данный фрагмент программы кодовой последовательностью.

Одним из вариантов реализации такой технологии может быть метод **управляемого преобразования программного кода**. В процессе работы фрагмента программы фиксируется (в специальном, трудном для анализа злоумышленником формате) последовательность операций, необходимых для обработки конкретных исходных данных (граф алгоритма). При этом сразу осуществляется достраивание $G(x)$ до $G_1(x)$: добавление новых операций, изменение связей между уже существующими и т.д.

Затем созданная последовательность операций выполняется. То, как это происходит, зависит от пароля, введенного пользователем. Правильный результат будет получен только тогда, когда этот пароль верен. В противном случае будет выбран

другой подграф $G_1(x)$ и реализован иной алгоритм с иными выходными данными. Стоит отметить, что это относится не только к конечным результатам работы фрагмента (значениям соответствующих переменных), но и к промежуточным. Поясним это на примере.

Пример 4. Пусть в защищаемом фрагменте кода вычисляется значение некоторой функции $H(a, b)$ по следующему алгоритму:

1) $c = F(a, b)$;

2) $d = G(c, a)$;

3) $H(a, b) = d$.

Если структура функций F и G достаточно сложна для анализа, то логику их работы можно попытаться восстановить, составив и исследовав таблицу их значений для различных аргументов. Но при применении предлагаемой здесь методики эти значения окажутся измененными, если пользователь указал неверную кодовую последовательность. Таким образом, анализ промежуточных операций (функций F и G) оказывается затруднительным, что, в свою очередь, повышает сложность анализа функции H в целом.

Для реализации изложенного выше метода необходимо иметь возможность преобразовать исходный текст защищаемого фрагмента программы так, чтобы вместо арифметических, логических и других операций и вызовов функций стандартных библиотек вызывались определенные разработчиком специальные функции. Заметим, что такое преобразование текста программы используется и рассмотренными в предыдущем разделе средствами быстрого автоматического дифференцирования. В обоих случаях необходимо:

1) зафиксировать и, возможно, преобразовать структуру зависимостей данных в фрагменте программы (граф алгоритма);

2) обработать построенный граф тем или иным способом.

Как и при выполнении быстрого автоматического дифференцирования, реализовать это можно с помощью программных средств на основе непосредственного преобразования исходного фрагмента или на основе перегрузки операторов и функций (см. раздел 3.3). Последний вариант представляется более предпочтительным по тем же причинам, что и для быстрого дифференцирования. Кроме того, как уже говорилось выше, применение перегрузки позволяет компилировать и компоновать полученную программу любыми из доступных средств для данного языка программирования. Это дает возможность использовать специальные компиляторы и т.д., реализующие другие методы обфускации, что могло бы еще усилить сложность анализа защищаемой программы.

В рамках данного подхода можно также обеспечить поддержку разных паролей для разных наборов исходных данных. (Это возможно, так как при работе преобразованного фрагмента программы граф алгоритма строится для конкретных исходных данных, а структура используемых реализаций перегруженных операторов и функций определяется выбранным паролем.) Такая технология особенно полезна, если с защищаемой программой работают люди с разными правами доступа: каждый из них сможет получить верный результат работы программы для тех и только для тех исходных данных, для которых он имеет на это право.

В настоящее время проводится тестирование прототипа системы, реализующей данный метод обфускации программ.

5. Заключение

В данной статье рассматривались принципы и методы создания программных систем, облегчающих анализ и преобразование структуры программ для двух раз-

личных приложений. Первое из них можно отнести к классу задач вычислительной математики, второе к области системного программирования. Такие, казалось бы, совершенно различные группы задач объединяет тот факт, что для эффективного решения каждой из них важно использовать сведения о структуре зависимостей обрабатываемых данных.

Все это говорит о том, что использование свойств структуры алгоритмов является универсальным фактором для создания средств автоматизированного проектирования программных систем. Это может быть связано с тем, что, по мнению авторов книги [5], граф алгоритма представляет собой основу ("ядро") конструктивной идеи разработчика соответствующей программы. При этом структура этого "информационного ядра" далеко не всегда до конца ясна и прозрачна даже для самого разработчика. На практике, при реализации масштабных проектов невозможно иметь полное представление о графе алгоритма без использования специальных систем. Поэтому средства анализа и управления преобразованием структуры алгоритмов могут стать ценным инструментом при создании эффективных программ для самых разных приложений.

СПИСОК ЛИТЕРАТУРЫ

1. *Rall B.* Automatic Differentiation: Techniques and Applications // Lecture Notes Comput. Sci. V. 120. Berlin: Springer Verlag, 1981.
2. *Канторович Л.В.* Об одной математической символике, удобной при проведении вычислений на машинах // Докл. АН СССР. Т. 113. 1957. С. 738–741.
3. *Матиясевич Ю.В.* Вещественные числа и ЭВМ // Кибернетика и вычисл. техника. 1986. № 2. С. 104–133.
4. *Воеводин В.В.* Информационная структура алгоритмов. М.: Изд-во МГУ, 1997.
5. *Воеводин В.В., Воеводин Вл.В.* Параллельные вычисления. СПб.: БХВ-Петербург, 2004.
6. *Iri M.* History of automatic differentiation and rounding error estimation // Proc. Automat. Different. Algorithms: Theory, Implementation and Application. Philadelphia, PA, 1991. P. 3–16.
7. *Griewank A.* Evaluating Derivatives: Principles and Techniques of Algorithmic Differentiation. Philadelphia: SIAM, 2000.
8. *Bischof C., Buecker M.* Computing Derivatives of Computer Programs // Proc. Modern Methods Algorithm. Quantum Chemistry. Forschungszentrum Juelich, 2000. P. 315–327.
9. *Hovland P., Norris B., Smith B.* Making Automatic Differentiation Truly Automatic: Coupling PETSc with ADIC. Preprint. Argonne National Laboratory, 2002.
10. *Green L.L.* Applications of Automatic Differentiation at NASA Langley Research Center // Proc. Automat. Different. Workshop. Shrivenham, UK, 2003. P. 18–34.
11. *Hascoet L., Pascual V., Dervieux D.* Automatic Differentiation with TAPENADE. Springer Verlag, 2005.
12. *Alexandrescu A.* Modern C++ Design: Generic Programming and Design Patterns Applied. Addison Wesley, 2001.
13. *Collberg C., Thomborson C., Low D.* A Taxonomy of Obfuscating Transformations. Technical report. Department of Computer Science, University of Auckland, 1997.

Статья представлена к публикации членом редколлегии О.П. Кузнецовым.

Поступила в редакцию 29.06.2006