Терехов Андрей Николаевич

ИСТОРИЯ ОДНОЙ ИДЕИ

Аннотация

В статье рассматривается эволюция идеи учета последующего контекста, возникшая в середине 50-х годов в задаче автоматического перевода с русского языка на английский, её пробная реализация в середине 70-х годов на примере синтеза эффективной объектной программы в компиляторах и, наконец, её современная реализация и применение в задачах поиска решений методом сначала-в-ширину (*breadth-first*). Для реализации этой идеи применялись различные техники, наиболее эффективной из которых оказалась техника использования BDD (двоичных диаграмм решений). Конечным, хотя и несколько неожиданным результатом, явилось утверждение, что для широкого класса рекурсивно-переборных задач метод решения сначала-в-ширину выигрывает у традиционного механизма возвратов (метод сначала-в-глубину).

Ключевые слова: учет последующего контекста, методы сначала-в-ширину и сначала-в-глубину, рекурсивно-переборные задачи, BDD, оптимизация объектного кода.

1. ПРЕДЫСТОРИЯ

Я окончил математико-механический ЛГУ в 1971 году, причем первым защитил диплом в первом выпуске новой кафедры математического обеспечения ЭВМ. Моим научным руководителем был доктор физико-математических наук Григорий Самуилович Цейтин, под руководством которого большая группа студентов и сотрудников ВЦ ЛГУ занимались реализацией языка Алгол 68 для ЕС ЭВМ. Темой моей дипломной работы был поиск цепочки приведений (преобразований типов) в Алголе 68. В 1974 году было опубликовано «Пересмотренное сообщение об Алголе 68», в котором язык был существенно упрощен, но в первом варианте языка приведения и эквивалентность видов (в том числе рекурсивных) были очень трудными задачами.

Г.С. Цейтин предложил мне поступить в аспирантуру, честно предупредив, что у него уже было довольно много аспирантов, но ни один пока не защитился. Мы стали обсуждать возможную тему диссертационной работы. Мне уже тогда казалось, что тема анализа алгоритмических языков (как видонезависимого, так и видозависимого) практически исчерпана, поэтому я попросил предложить мне какую-нибудь тему по синтезу (генерации) кода. Здесь у нас было новое, прорывное решение. Г.С. Цейтин существенно улучшил идею Бранкара и Леви [1], которые предлагали во время синтеза анализировать стек объемлющих инструкций, чтобы понять, какой именно вариант генерации кода будет оптимальным в данном конкретном случае. Такое решение приводило к перебору огромного числа вариантов. Вместо этого Г.С. Цейтин предложил элегантную идею запросов и ответов [2], существенно упростившую схему синтеза.

В те годы вообще были популярны синтаксически ориентированные схемы трансляции, были «горячие головы», утверждавшие, что и синтез рабочей программы можно реализовать таким образом. Разумеется, на самом деле генерация эффективного кода требует много разных механизмов. Например, если в одной конструкции значение идентификатора ABC возникло на регистре, а затем в совершенно другой конструкции снова нужно использовать значение этого идентификатора, то лучше его использовать на регистре (если значение ABC «долежало» на этом регистре, то есть регистр не был использован для других целей), а не считывать его из памяти. Вряд ли эту ситу-

ацию можно описать синтаксическим образом. Еще более трудной является следующая ситуация. Пусть значение какого-либо идентификатора или промежуточного результата вычислений возникло на регистре. Оно либо долежит на регистре до момента использования, либо будет выгружено в память, если регистр понадобится для других целей (как в известном анекдоте про вероятность встретить динозавра на Невском проспекте). Если долежит – замечательно, если же не долежит, то лучше сразу выгрузить это значение в память, иначе команда выгрузки будет размножена в ветвях будущих условных предложений или (что ещё хуже) циклов.

Таким образом, возникает задача предсказания будущего контекста. И вот тут Г.С. Цейтин вспомнил, что лет 15 назад он уже продумывал решение этой задачи, правда, в совершенно другой области – в задаче перевода с одного естественного языка на другой. Г.С. Цейтин руководил лабораторией математической лингвистики, а всё, что он делал с нами в ВЦ ЛГУ, было для него, так сказать, хобби, правда, высокопрофессиональным (как и все, что он делал в других областях). В середине 50-х годов он пытался реализовать перевод с русского языка на английский. Разумеется, на ЭВМ того времени это были только эксперименты, но многие задачи были сформулированы уже тогда. Рассмотрим пример. «Иван разбил чашку», «Вася разбил сад». В первом случае слово «разбил» имеет отрицательный характер, во втором – положительный, с совершенно другим смыслом. Уже тогда Г.С. Цейтин понял, что смысл слов надо кодировать не простыми значениями, а «условными», представленными в виде дерева вариантов. Слабость тогдашней техники (особенно малые размеры памяти) не позволили ему даже провести эксперименты. Спустя 15 лет Г.С. Цейтин предложил мне этим заняться.

2. ПОСТАНОВКА ЗАДАЧИ

Задача эффективного программирования с учетом вариантов, возникающих в дальнейшем, может быть решена с помощью двухпроходной схемы синтеза: во время первого просмотра входного текста порождается рабочая программа, содержащая несколько вариантов для каждого спорного случая; во время второго (обратного) просмотра происходит сравнение вариантов и выбор лучшего из них. В частности, решение поставленной выше задачи о выгрузке регистра в память может быть следующим:

– в первом просмотре для каждого значения, которое находится в регистре и не используется сразу же после получения, программируем условную выгрузку этого регистра в память. Далее помечаем каким-либо образом те значения в регистрах, которые из-за нехватки регистров должны быть выгружены в память до своего использования;

– во втором просмотре исключаем из текста ненужные выгрузки.

Заметим, что решение оказалось столь простым благодаря тому, что к моменту использования значения в регистре уже известно, что этот регистр не был использован для другой цели. В более сложных задачах, связанных с повышением эффективности рабочей программы, могут возникнуть ситуации, в которых представление значения не выбрано окончательно к моменту его использования. Если при появлении подобной альтернативы мы всегда будем начинать независимое программирование двух (или более) вариантов, его придется снова раздроблять на подварианты и т. д., и в результате число рассматриваемых вариантов быстро выйдет за разумные границы. Однако поиск наилучшего варианта при учете всех таких альтернатив может быть облегчен за счет следующих двух обстоятельств:

а) две разные альтернативы могут оказаться «ортогональными» друг другу, то есть изменения в порождаемой программе, обусловленные выбором решения по одной из альтернатив, не будут зависеть от решения по другой альтернативе;

б) каждое принимаемое решение касается программирования определенной конструкции и, вообще говоря, не влияет на программу после окончания этой конструкции.

Предлагаемый метод как раз и рассчитан на то, чтобы описывать порождение рабочей программы с учетом альтернатив, разрешаемых при дальнейшей работе, но без явного разветвления генерируемой программы на много вариантов.

Для того чтобы иметь возможность учитывать информацию о последующем тексте программы, вводятся особого вида логические переменные – предсказатели. Если при программировании некоторой конструкции нужно выбрать один из двух вариантов в зависимости от последующих конструкций, то в транслируемой конструкции вводится новый предсказатель и затем допускаются ветвления по этому предсказателю. Условие, задающее действительное значение предсказателя, не задается явно. Вместо этого все порождаемые команды получают определенную оценку «стоимости». Транслятор выбирает значения предсказателей таким образом, чтобы стоимость порождаемой программы была минимальной.

Подробное обсуждение задачи приведено в [3].

3. УСЛОВНЫЕ ЗНАЧЕНИЯ

Некоторые операторы в программе могут находиться внутри условных предложений и выполняться или не выполняться в зависимости от значений определенных предсказателей. Среди таких операторов могут оказаться и присваивания, и, таким образом, на выходе из условного предложения, управляемого предсказателем, окажется, что значение некоторой переменной зависит от этого предсказателя. Далее эта зависимость может распространяться на результаты операций с таким значением, на условные предложения, управляемые результатом такой операции и т. д. Поэтому считается, что любое значение, над которым производится действие, может быть условным, например, иметь вид:

```
if A then x elif B then y else z fi.
```

Однако учет этих условностей не должен усложнять написание макроопределений порождения рабочей программы, и для автора макроопределений эти значения должны выглядеть как простые. Система, реализующая описываемый способ синтеза (далее будем говорить просто система), использует специальный способ представления этих значений в виде двоичного дерева; каждый узел соответствует некоторому предсказателю, а две дуги, выходящие из этого узла, соответствуют значениям предсказателя **true** и **false**. В висячих вершинах этого дерева находятся конкретные значения. Операции над такими деревьями (арифметические операции, присваивания, сравнения и т. п.) выполняются при помощи специальных процедур.

Для правильного исполнения условных присваиваний вводится специальная переменная, образующая текущее условие. Значением этой переменной является условное логическое выражение, то есть двоичное дерево описанного выше вида, в висячих вершинах которого находятся логические значения. Перед началом генерации программы эта переменная получает значение **true**. Другие значения она получает в условных предложениях и циклах.

Кроме того, значение текущего условия изменяется, если обнаруживается, что при некотором сочетании значений предсказателей программирование не может продолжаться (например, из-за отсутствия свободных регистров); в этих случаях к текущему условию конъюнктивно присоединяется условие продолжимости генерации.

Значение текущего условия влияет на исполнение присваиваний и генерацию рабочей программы. Если при текущем условии **B** исполняется присваивание **x:=y**, то фактически **x** присваивается значение **if B then y else x fi** (технически эта операция несколько сложнее, поскольку **B**, **x**, **y** могут быть условными). Если же при текущем условии **B** генерируется текст рабочей программы, то он должен быть снабжен пометкой, что в окончательную программу он войдет, лишь если значение **B** в конечном счете окажется **true**. Фактически такие пометки в рабочей программе достаточно ставить лишь в тех точках, где текущее условие изменяется.

Рассмотрим условное предложение
if A then S fi.

Пусть перед входом в это предложение значением текущего условия было **B**. Тогда **S** исполняется при значении текущего условия **B&A**. Условное предложение

```
if A then S1 else S2 fi
```

в принципе реализуется так же, как

```
bool AA=A; if AA then S1 fi;
if not AA then S2 fi.
```

Рассмотрим цикл **while A do S od**. Предположим, что известно целое число **N**, ограничивающее сверху число повторений цикла. Тогда для данного цикла можно определить эквивалентное выражение S_N следующим образом:

в качестве **S0** берем **skip**;
в качестве **Sk+1** берем **if A then S; Sk fi**.

Способ реализации таких предложений уже описан. На самом деле не требуется определять такое **N** заранее. При выполнении цикла мы будем входить во вложенные условные выражения достаточное число раз, пока текущее условие не окажется тождественным **false**. Этот процесс может, вообще говоря, привести к бесконечному циклу, но это будет означать, что такой бесконечный цикл возможен и при некоторых конкретных, а не условных значениях используемых величин, то есть в этом случае макроопределение не должно оканчивать работу.

4. ПРЕДСТАВЛЕНИЕ УСЛОВНЫХ ЗНАЧЕНИЙ

Опишем подробнее представление условных значений в памяти машины и процедуры, работающие с этими значениями. Как уже было сказано, условные значения представляются двоичными деревьями; вид таких деревьев можно описать следующим образом:

```
mode узелX= struct (int
предсказатель, значениеX да, нет);
mode значениеX = union (X, ref узелX);
```

такие описания даются для каждого вида **X**, допускаемого реализацией.

В начале работы системы вся память, отведенная для хранения деревьев, разбивается на ячейки, доступные для хранения значений видов **узелX**; эти ячейки связываются в список свободных ячеек. Имеется процедура, доставляющая очередную свободную ячейку, а также процедура «сборки мусора», которая вызывается при исчерпании списка свободных ячеек.

Система обрабатывает все идентификаторы простых переменных, а также элементы массивов как значения видов **значениеX**. Такая замена видов происходит автоматически.

Основным способом выполнения различных действий над значениями, представленными в виде двоичных деревьев, является копирование двоичного дерева в свободных ячейках памяти с выполнением соответствующего действия на каждой висячей вершине этого дерева. В результате такого действия вместо висячей вершины может быть, в частности, вставлена копия другого дерева, полученная аналогичным процессом. Например, для того, чтобы получить сумму двух значений, заданных в виде двоичных деревьев, мы можем копировать дерево первого слагаемого, выполняя в каждой его висячей вершине следующее действие: запоминаем число, стоящее в этой вершине, затем копируем второе слагаемое с увеличением числа в каждой его висячей вершине на запомненное число из первого слагаемого, получившееся дерево вставляем на место этого числа в копию первого слагаемого. Фактически такое копирование не обязательно будет копированием полного дерева. Именно, при копировании первого дерева мы запоминаем в стеке путь от его корня к текущей вершине и, таким образом, имеем список значений всех предсказателей, по которым производилось ветвление на этом пути. Когда в висячей вершине первого дерева мы начнем копирование второго дерева, то оно будет выполняться уже при определенных значениях предсказателей из первого дерева; поэтому, встречая во втором дереве ветвление по предсказателю, значение которого уже записано в стеке, мы не копируем узел с та-

ким ветвлением, а просто сразу идем по нужной ветви и рассматриваем встретившуюся там вершину вместо данного узла. Такое неполное копирование может применяться и к первому копируемому дереву, если текущее условие конъюнктивно содержит некоторые предсказатели или их отрицания. Список таких предсказателей выделяется из текущего условия посредством специальной операции и добавляется к началу стека, используемого для копирования.

Рассмотрим, как используется описанная техника при действиях над условными значениями.

Для присваивания переменной значения сначала копируется дерево текущего условия, после чего висячие вершины со значением **true** заменяются копией дерева источника, а висячие вершины со значением **false** – копией дерева значения, именуемого получателем. Полученное дерево объявляется новым значением переменной. Таким образом, при тех значениях предсказателей, при которых текущее условие есть **false**, переменная своего значения не изменяет.

В описываемой системе существенно используются условные значения, представленные в виде деревьев. Каждое действие с условными значениями связано с копированием деревьев, подстановкой деревьев вместо висячих вершин и т. д. При этом, если не принять специальных мер, объем памяти, отведенной для хранения деревьев, может оказаться недостаточным даже для сравнительно простых случаев.

Очевидно, что задача упрощения деревьев с произвольными конкретными значениями включает в себя упрощение деревьев с булевскими значениями, то есть задачу минимизации булевской формулы. Известно, однако, что полная минимизация булевской формулы в общем виде является трудоемкой задачей. Исходя из этого, в данной реализации не предполагалась полная оптимизация деревьев. Использовались лишь сравнительно несложные алгоритмы, позволяющие заметить следующие ситуации:

a) дерево эквивалентно константе;

b) дерево может быть преобразовано к виду:

if A then x elif B then x elif...elif произвольное дерево fi

Здесь **x** обозначает конкретное значение. **A**, **B**,... обозначают предсказатели или их отрицания.

В частности, если в таком виде представлено значение текущего условия, причем **x** есть **false**, то для генерируемого в данный момент варианта значения определенных предсказателей зафиксированы таким образом, что **A**, **B**, ... все имеют значение **false**. Это используется для упрощения копирования деревьев. Для приведения дерева к виду b) используется следующий алгоритм.

Строится таблица из двух столбцов, каждая строка этой таблицы соответствует одному предсказателю. Сначала таблица заполняется кодом «пусто». Далее обходится дерево, которое нужно оптимизировать. В каждой висячей вершине со значением **A** для каждой строки делается следующее. Если значение соответствующего предсказателя определено в стеке обхода дерева, то в нужный столбец, а в противном случае – в оба столбца записывается:

если было «пусто», то **A**;
если было **A**, то значение не изменится;
в других случаях пишется код «*».

После окончания обхода дерева просматривается таблица. Если найдется строка, в обоих столбцах которой записаны значения, отличные от кода «*», то результирующее значение будет либо константой (если эти два значения совпадают), либо будет состоять из одного узла, обе ветви которого – конкретные значения, взятые из этой строки таблицы.

Если найдется строка, в одном из столбцов которой записано конкретное значение, то перестраиваем дерево так, чтобы в его корне стояло ветвление по предсказателю, соответствующему этой строке таблицы. Одна из ветвей, исходящих из этого узла будет указанным конкретным значением, на второй ветви расположен «остаток» исходного дерева.

Может оказаться несколько строк таблицы, содержащих конкретное значение в

одном из столбцов таблицы (и тогда, как нетрудно догадаться, это конкретное значение будет всегда одним и тем же). В этом случае в корень дерева помещается узел, построенный для одной из тех строк, в начале остатка дерева становится узел для другой такой строки и т. д.

После того, как все такие строки исчерпаны, окончательный остаток дерева, присоединяемый к последнему из построенных узлов, получается копированием исходного дерева, выполняемым с учетом значений тех предсказателей, которые вошли в начальные узлы.

Заметим, что в данной реализации построение таблицы, используемой в алгоритме оптимизации деревьев, может быть совмещено с выполнением копирования. Эквивалентность дерева константе также выясняется во время копирования: при копировании очередного узла проверяется, являются ли обе ветви, исходящие из этого узла, совпадающими конкретными значениями. При положительном ответе этот узел не копируется, вместо него рассматривается конкретное значение. Таким образом, обнаруживается не только эквивалентность дерева константе, но также и эквивалентность константам его поддеревьев.

В заключение отметим, что этот алгоритм, конечно, не дает полной оптимизации деревьев, например,

а) не будет замечено, что дерево не зависит от некоторого предсказателя, хотя его и содержит;

б) не будут найдены эквивалентные поддеревья.

Для того чтобы проверить изложенные выше методы, была предпринята экспериментальная реализация на инструментальной ЭВМ «Одра 1204». Эта небольшая ЭВМ на мат-мехе ЛГУ предоставлялась в режиме индивидуального использования. В её матобеспечение входила практически полная реализация Алгола 60, а мы реализовали удобную систему для диалоговой отладки и исправления текстов [4]. Вся оперативная память, доступная для динамического распределения, составляла примерно 5000 ячеек, что, конечно, не позволяло проводить масштабные эксперименты. Тем не менее, все алгоритмы были реализованы и проверены на реальных примерах [5]. В это время ожидался массовый выпуск ЕС ЭВМ, поэтому мы решили отложить дальнейшие исследования на некоторое время.

5. НЕОЖИДАННОЕ ПРОДОЛЖЕНИЕ

В конце 1975 года диссертация была переплетена, основные результаты были опубликованы в академическом журнале [3]. В начале 1976 года в Академгородке (г. Новосибирск) состоялась большая международная конференция, на которой я выступил с докладом о своих результатах.

В те годы Академгородок был настоящей Меккой советского программирования. А.П. Ершов создал там одну из самых сильных в нашей стране школ системного программирования, часто ездил за границу (что тогда было большой редкостью), имел широкий круг друзей и коллег, которые считали за честь принять его приглашение на конференцию в Академгородок.

Одним из таких ученых, часто приезжавших в Академгородок, был Джекоб Шварц, по книге которого мы в университете учили функциональный анализ [6]. «На старости лет» Джекоб решил переквалифицироваться в программиста и создал первый в мире язык сверхвысокого уровня SETL, в котором элементарными данными были множества, деревья, списки и т. д. Одну из наиболее успешных реализаций SETL осуществили как раз в Академгородке (Давид Левин).

Так вот, когда я закончил свой доклад, Джекоб Шварц, сидевший, как всегда, на первом ряду, спросил меня: «А как Ваш метод соотносится с языками искусственного интеллекта Planner и Conniver?». Я даже не слышал о таких языках, переспросил Г.С. Цейтина, сидевшего в зале, но и он ничего об этом не знал. Интернета тогда не было, а «железный занавес» – был. Вечером я напросился к Джекобу в компанию на ужин в единственной тогда гостинице «Золотая долина». Обслуживание там было типично советским, поэтому за 2 часа

Джекоб рассказал мне об этом новом для нас направлении. Я очень расстроился, мне показалось, что моя диссертация «накрылась медным тазом», по крайней мере, предсказатели, перебор вариантов, поиск оптимального решения в AI-языках (Artificial Intelligence) присутствовали. Позже Джекоб прислал мне бандероль с подборкой статей на эту тему [7–10], ксероксов тогда не было, поэтому эти статьи были зачитаны до дыр сотрудниками мат-меха ЛГУ.

После внимательного изучения этих материалов оказалось, что все не так плохо, более того, выяснилось, что для задач искусственного интеллекта, где тогда царствовал экспоненциально сложный механизм возврата (бэктреккинга), мы предложили совершенно другой, часто более эффективный метод решения.

Диссертацию пришлось переплетать заново, был добавлен короткий параграф «Приложение метода к построению языков искусственного интеллекта», и все закончилось благополучно.

6. ЧЕМ МОЖНО ЗАМЕНИТЬ МЕХАНИЗМ ВОЗВРАТОВ В ЗАДАЧАХ ПОИСКА РЕШЕНИЯ

Впервые механизм возвратов был четко сформулирован, по-видимому, в языке PLANNER [8]. Мы рассмотрим его, пользуясь «алголоподобными» методами описания.

Пусть обходится дерево решений. Есть средства для фиксации точек ветвления

```
If   ok   then вариант1 else вариант2 fi
```

и есть оператор, указывающий на невозможность продолжения варианта (**fail**). Система, интерпретирующая текст на таком языке, встретив оператор ветвления, запоминает состояние памяти и запускает первый вариант. Если встретится **fail**, то система восстанавливает содержимое памяти в момент последней точки ветвления и запускает альтернативный вариант; если в данной точке ветвления вариантов больше нет, то рассматривается предыдущая точка ветвления.

В качестве примера рассмотрим задачу восьми ферзей (установить 8 ферзей на шахматной доске так, чтобы они не били друг друга).

```
[8] int a;

for i to 8 do a[i]:=0 od;
proc p= (int i) void:
```

co устанавливает ферзя в **i** столбце на такую позицию, в которой он не бьет **1, ... , i-1** ферзей (последняя занятая позиция в i столбце фиксируется в **a[i]**); если такой позиции нет, то **a[i]:=0** и выдается **fail co**

```
for i to 8 do
m: p(i);

 if ok then
if i=8 then print (a) else go to m fi od
```

Программа достаточно коротким и наглядным способом задает путь поиска решений, причем автор программы может совершенно не заботиться о технических деталях.

Однако механизм возвратов оказался хотя и удобным для авторов программ, но чрезвычайно неэффективным средством. Многократные запоминания состояний памяти, необходимость повторного вычисления многих участков программы (например:

```
if ok then A1 else A2 fi;
if ok then B1 else B2 fi;
if ok then C1 else C2 fi;
```

здесь операторы **B1** и **B2** выполняются по 2 раза, операторы **C1** и **C2** – по 4 раза и т. д.) заставили признать, что PLANNER поощряет «плохую практику программирования» [9].

Поэтому PLANNER так и не был реализован в полном объеме. G.J. Sussman (один из авторов первой реализации MICRO-PLANNER [10]) сделал попытку преодолеть указанные трудности. Он предложил язык CONNIVER [9], в котором многие действия, выполняемые автоматически в языке PLANNER, должен указывать программист. В частности, автор программы определяет, значения каких переменных нужно запоминать в точках ветвления, задает явно контексты для выполне-

ния операторов и т. д. Это позволило резко улучшить эффективность работы, но практически явилось шагом назад; как и в традиционных языках программирования, программист должен сам заботиться о многих технических деталях, что затемняет сущность решения задачи.

Нужно заметить, что G.J. Sussman сделал несколько интересных предложений и не связанных явно с повышением эффективности механизма возвратов.

Оператор **fail** был заменен на процедуру, вычисляющую текущую стоимость варианта. Если написать **work(S)**, то текущая стоимость увеличится на **S**; очевидно, что **work(∞)** эквивалентно **fail**.

Левосторонний обход дерева решений был заменен «параллельным» сравнением вариантов (если текущая стоимость варианта превысила некоторый порог, то вариант приостанавливается и запускается следующий вариант, причем порог постепенно растет).

В диссертации я написал, что, по нашему мнению, описанные трудности в реализации языков искусственного интеллекта можно преодолеть с помощью введенного нами метода синтеза рабочей программы. Предлагалось следующее решение.

1. Язык искусственного интеллекта строится на основе описанной в данной работе системы синтеза рабочей программы.

2. Точки ветвления можно задавать, например, следующим образом:

```
if ok then A1 else A2 fi.
```

3. Определяется явная функция стоимости варианта (аналогично **work** в CONNIVER) и точки, в которых должно происходить сравнение вариантов.

4. Никакого запоминания состояния памяти не предполагается. Варианты проверяются все одновременно, что отражается в специальной форме представления значений (описанные в данной работе двоичные деревья). Каждый оператор выполняется только столько раз, сколько требуется по алгоритму программы; никаких перевычислений не будет (разумеется, элементарные операции усложнятся). Можно рассчитывать, что в обрабатываемых значениях будут отражаться далеко не все варианты (ввиду локальности и ортогональности вариантов).

Механизм возврата является классической реализацией «поиска-в-глубину». Предложенный нами механизм можно сопоставить с «поиском-в-ширину». Мы постепенно спускаемся по дереву решений, просматривая все узлы слева направо без повторов и возвратов. История прохода отражается в специальном представлении всех обрабатываемых значений. В худшем случае, когда все варианты зависят друг от друга (как в рассмотренной выше задаче о 8 ферзях), мы меняем «шило на мыло», заменяя полный перебор по дереву решений на обработку столь же сложных условных значений. К счастью, большинство практических задач сводятся к выбору из локализированных, мало зависящих друг от друга вариантов. Это и обеспечивает, по крайней мере, теоретическое преимущество нашего метода.

7. ДОЛГОЖДАННОЕ ПРОДОЛЖЕНИЕ

Я успешно защитил кандидатскую диссертацию, меня сразу же назначили исполняющим обязанности заведующего лабораторией системного программирования ВЦ ЛГУ. Собственно, эти обязанности я уже исполнял несколько лет, но в те времена завлабом в Университете мог быть только доктор наук, поэтому мне, даже не кандидату наук, шансов занять эту позицию не было никаких. Таким образом, моей защите начальство обрадовалось больше меня, и сразу на меня свалилось огромное количество административных работ. Ранее директору ВЦ ЛГУ Шауману А.М. (кстати, одному из самых любимых мною начальников), видимо, было просто неудобно возлагать на меня обязанности, не давая никаких прав. В эти же годы мы были по горло заняты завершением работ и сдачей заказчику (НИЦЭВТ) нашего главного детища – транслятора с языка Алгол 68 для ЕС ЭВМ. Все старшие товарищи, начинавшие проект, когда мы были еще студентами, не

выдержали еженедельных командировок в Москву, где мы отлаживали транслятор на IBM 360 исключительно в ночное время, так что завершал этот проект я уже руководителем.

В декабре 1980 года ректор ЛГУ В.Б. Алесковский поручил мне сотрудничество с Оборонным отделом Обкома КПСС, наши военные испытывали большие трудности с внедрением цифровой вычислительной техники и обратились за помощью в Университет. Появился новый круг обязанностей (и возможностей), новые интересные задачи, время летело. В эпоху перестройки государство перестало платить по оборонным заказам, пришлось срочно искать западных заказчиков. С этой абсолютно новой задачей мы также успешно справились, но практически потеряли возможность заниматься фундаментальными исследованиями в той же мере, как это было в счастливых для нас семидесятых годах.

Все эти годы я продолжал «краем глаза» следить за публикациями в области поиска решения на дереве вариантов, но к своему удивлению, ничего похожего на свою кандидатскую диссертацию не обнаружил.

Один раз мое сердце дрогнуло. Это было в середине 90-х годов. По заказу итальянской компании ITALTEL мы создавали технологию проектирования гибких кристаллов (FPGA) и столкнулись с задачей минимизации булевских формул. Ещё когда я был студентом мат-меха, нас учили, что это пример сложной задачи, для которой возможны лишь решения, связанные с полным перебором, то есть имеющие экспоненциальную сложность. На военной кафедре на лекциях по электронике нас учили, как получить почти оптимальную форму (диаграммы Вейча, метод Квайна-МакКласки), но итальянцы показали нам работы американских учёных из Беркли по BDD (двоичным диаграммам решений), которые эту задачу решают очень быстро. Фокуса тут никакого нет, теория, как всегда, права, просто для формул ограниченной длины (скажем, содержащих до 1 млн. переменных, что с избытком покрывает практические потребности), применяя хэширование с длинной сверткой и большими таблицами, можно найти поддерево в дереве за 1 сравнение. Я не буду в этой обзорной статье углубляться в математику этого вопроса [11], скажу только, что все довольно просто. Я сильно обозлился на себя и на своих сотрудников, что мы не только не придумали этого сами, но и прозевали столь интересный результат.

В любом случае, техника BDD была существенно сильнее тех простых методов оптимизации, которые я предложил в своей кандидатской диссертации. Мне захотелось повторить ту работу на базе новой техники. Кроме того, мощности машин и их память фантастически выросли и, таким образом, база для поиска сначала-в-ширину стала обретать реальные очертания.

Я несколько раз предлагал эту тему своим аспирантам, но интереса она не вызвала, а мне неудобно было настаивать, все-таки речь шла о моей кандидатской, защищенной много лет назад. Помог случай. Один из лучших теоретиков среди моих учеников Дмитрий Булычев поручил своему дипломнику Д. Иванову разработать библиотеку операций над BDD на языке OCaml (на этом новом очень удобном языке мы собираем большую библиотеку средств анализа программ, операций на графах и т. д.). Дипломник со своей задачей справился, но не предоставил ни одного содержательного примера (к сожалению, стремление минимизировать свои усилия – типичная черта современных молодых специалистов). Мне как заведующему кафедрой пришлось разбираться в этом споре между научным руководителем и студентом. Разумеется, я всецело был на стороне научного руководителя, но и «гробить» хорошего студента не хотелось.

Я предложил Д. Иванову за оставшиеся до защиты 2 недели реализовать несколько основных операций над условными значениями, базируясь на его библиотеке BDD. Результат превзошел все ожидания, Д. Иванов получил «красный» диплом, а я пригласил его к себе в аспирантуру. Не все шло гладко, например, я потратил много времени на объяснение, что висячими вер-

шинами BDD могут быть не только логические значения, но и произвольные значения, например, целые числа. Для меня это было совершенно естественно, но действительно противоречило каноническим определениям. Долго мы не могли подобрать хорошую модельную задачу, которая должна быть достаточно сложной, но в то же время иметь локальные варианты, не полностью зависящие друг от друга. Хорошую задачу предложил М. Бульонков. Это задача составления расписания для школы или другого учебного заведения. Есть набор предметов, учителей, которые их могут преподавать, ограниченный набор аудиторий и множество ограничений – пожеланий учителей, например, чтобы не было «дырок» между занятиями, чтобы были свободные дни и т. д.

В качестве модельной мы использовали более простую подзадачу – задачу о назначениях:

Есть N потребителей и K ресурсов, каждый потребитель может использовать только один ресурс из специфичного для этого потребителя набора ресурсов. Если потребитель использует ресурс, то другой потребитель его использовать уже не может. Требуется назначить каждому потребителю свой ресурс.

Для этого простого варианта существует полиномиальный алгоритм, но мы проводим сравнение для переборных алгоритмов поиск-в-глубину и поиск-в-ширину. Это же просто пример, далеко не для всех переборных задач существуют полиномиальные решения.

Приведем результаты сравнения этих методов в зависимости от количества ресурсов (см. табл. 1).

На наш взгляд, результаты говорят сами за себя.

Чтобы окончательно убедиться в преимуществах нашего подхода, мы провели эксперименты с более сложной переборной задачей. На шахматной доске расставлена позиция. Требуется поставить мат за заданное количество ходов. Оказалось, что поиск-в-ширину и здесь выигрывает, причем увеличение числа ходов на 1 увеличивает коэффициент выигрыша в 2–3 раза. Увеличение числа фигур на доске также существенно повышает выигрыш нашего метода.

8. ЗАКЛЮЧЕНИЕ

На мой взгляд, анализируя историю развития идеи решения переборных задач, изложенную в данной статье, можно сделать следующие выводы.

1. Идеи (как и рукописи) не горят.

2. Программисты-практики не должны относиться к теории как к абсолюту. Большинство практических задач имеют такие особенности и ограничения, которые позволяют найти более эффективные решения. Студентов кафедры системного программирования СПбГУ я учу, что главным для нас должен быть лозунг ad hoc – для данного конкретного случая.

Табл. 1

N	Поиск-в-глубину (ms)	Поиск-в-ширину (ms)
11	141	187
12	687	374
13	3682	594
14	21138	1171

Литература

1. *Branquart P., Cardinael J. P., Lewi J.* An optimised translation process and application to ALGOL 68. Part 1: General Principles // MBLE Lab. de Rech., Report R209, Bruxelles, September 1972.

2. Алгол 68. Методы реализации. Соавторы: *Балуев А.Н., Братчиков И.Л., Гиндыш И.Б., Крупко Н.А., Терехов А.Н., Цейтин Г.С.* (всего 12 чел.). Изд. ЛГУ, 1976.

3. *Терехов А.Н., Цейтин Г.С.* Средства эффективного синтеза объектной программы // Программирование, 1975. № 6.

4. *С.Н. Баранов, А.Н.Терехов, Г.С. Цейтин* Инструкции к программе DICO/ Методические материалы по программному обеспечению ЭВМ. Серия 4, Вып. 5. Изд. ЛГУ, 1974.

5. *Терехов А.Н.* Распределение регистров в рабочей программе // Программирование, 1977. № 1.

6. *Данфорд Н., Шварц Дж.Т.* Линейные операторы (том 1) Общая теория. М.: Наука, 1974.

7. *Bobrow D. G., Raphael B.* New programming languages for AI research // Xerox Palo Alto Research Center, August 1973.

8. *Hewitt C.* Description and theoretical analysis (using schemata) of Planner: A language for proving theorems and manipulating models in a robot // AI Memo No. 251, MIT Project MAC, April 1972.

9. *McDermot J., Drew V., Sussman G. J.* The Conniver reference manual // AI Memo No. 259, MIT Project MAC, May 1972.

10. *Sussman G. J., Winograd T.* Micro-Planner reference manual // AI Memo No. 203, MIT Project MAC, July 1970.

11. *Randell E. Bryant.* Symbolic Boolean manipulations with ordered binary decision diagrams. ACM Computer Surveys, 24(3): 293–318, September 1992.

Abstract

The article covers the evolution of the idea of taking subsequent context into account. This idea emerged in the middle of 1950th in the automated Russian-English translation task. The article describes its trial implementation for the task of efficient object program in compilers in 1970th, and its modern implementation for the task of width-first searching problem decisions. Different techniques to implement this idea were applied, the most effective being the technique that uses BDD (binary decision diagrams). Finally, the advantages of the width-first method, as compared to traditional mechanism of backtracking (depth-first method) for wide class of recursive search tasks are claimed.

Терехов Андрей Николаевич, доктор физико-математических наук, профессор, заведующий кафедрой системного программирования СПбГУ, генеральный директор ГУП «Терком», andrey.terekhov@at-software.com