ЛОГИКА И СЕМАНТИЧЕСКОЕ ПРОГРАММИРОВАНИЕ
(Вычислительные системы)
1992 год Выпуск 146

УДК 519.68

КОМПИЛЯТОР ФУНКЦИОНАЛЬНО-ЛОГИЧЕСКОГО ЯЗЫКА "Флэнг"

А.В.Манцивода, В.А.Петухин

В в е д е н и е

Как известно, одним из главных упреков, направляемых в сторону нестандартных языков (таких, как "Лисп" и "Пролог"),является обвинение в том, что их реализации никогда не достигнут достаточной для решения большинства серьезных задач эффективности и уж, по крайней мере, не сравнятся по производительности с императивными языками программирования. Основной целью данной работы можно считать опровержение этого тезиса. Ниже будет показано, что при достаточном усложнении процесса трансляции нестандартных языков можно получать код, сравнимый по скорости с кодами, генерируемыми, например, транслятором Turbo Паскаля. Будут приведены несколько примеров, когда программа, написанная на нестандартном языке, исполняется даже *быстрее*, чем соответствующая программа на Turbo Паскале.

В статье рассматриваются основные аспекты реализации функционально-логического языка "Флэнг" [1]. Во "Флэнге" объединены наиболее выразительные средства, присущие функциональным языкам программирования и "Прологу". Наиболее подходящий стиль работы с "Флэнгом" - интегрированный, поскольку как логические, так и функциональные черты могут проявиться не только в рамках одной программы, но даже внутри одного определения

(см. примеры ниже). Но при желании на "Флэнге" можно работать и в чисто функциональном, и в чисто логическом стилях.

Общность языка делает разработку компилятора с "Флэнга" задачей более трудной, чем, скажем, разработку компилятора с "Пролога" или "Лиспа". С другой стороны, поскольку "Флэнг" является обобщением "Пролога", то и компилятор с "Флэнга" будет содержать внутри себя прологовский компилятор.

Основной идеей, позволившей значительно повысить эффективность компиляции "Флэнга", явилось введение в процесс компиляции нового шага - глобального анализа программы. Информация, полученная от глобального анализа, позволяет применить ряд существенных оптимизаций генерируемого кода.

Проблема эффективной реализации логических и функционально-логических языков программирования привлекает внимание многих специалистов [2, 4-7]. Наш подход к компиляции близок к подходам, реализованным в системах Aquarius Prolog [5] и Parma [6]. Основными отличиями являются: во-первых, "Флэнг" - обобщение "Пролога", содержащее дополнительные средства, также нуждающиеся в трансляции Флэнг-компилятором; кроме того, Aquarius и Parma транслируют Пролог-программы в промежуточный код, близкий коду компьютера, мы же предпочитаем работать с модификацией абстрактной машины Уоррена (WAM) [2]; наконец, в качестве целевого компьютера мы используем IBM PC, обладающий довольно слабым процессором. Хорошо известно, что оптимизации, отлично работающие на одной архитектуре, не всегда хороши для другой.

Компилятор, транслирующий Флэнг-программы в код IBM PC, был разработан в системе TurboC. Производительность генерируемого кода очень высока. Например, некоторые программы выполняются на "Флэнге" в 20 раз быстрее, чем оттранслированные с помощью компилятора Arity/Prolog.

1. Описание "Флэнга"

"Флэнг" [1] - функционально-логический язык, предназна - ченный для символьной обработки данных, решения комбинаторных проблем, для алгебраических вычислений, а также для обучения студентов методам искусственного интеллекта. На "Флэнге" легко пишутся программы в функциональном, логическом и смешанном стилях. Причина - функциональный и логический подходы к програм - мированию объединены в языке естественным образом.

"Флэнг" может рассматриваться как расширение "Пролога",но на самом деле он развился из функционального программирования. Невозможно разделить функциональную и логическую части языка - в языке реализована одна идея для обоих стилей.В основе "Флэнга" лежит понятие недетерминированной функции. Недетерминированные функции являются следующим обобщением "обычных" функций:

- допускается вычисление функций с аргументами, содержащими свободные (логические) переменные;

- используется стратегия поиска в глубину: если система не может вычислить цель (функцию), она возвращается обратно (применяет бэктрэкинг) для поиска альтернативных путей вычисления.

Это обобщение понятия функции включает в себя обычные прологовские отношения - они являются функциями лишь с одним значением - true (истина).С другой стороны, мы можем использовать недетерминированные функции как обычные и, таким образом, писать чисто функциональные программы. Рассмотрим несколько примеров определений на "Флэнге". Начнем с чисто функциональных определений. Первое из них - *факториал*:

```
0! <=>  1;
N! <=   N > 0, N * (N-1)!;
```

Теперь логические определения:

родитель(*Петя*, *Вася*) <= *истина*;

родитель(*Вася*, *Коля*) ⇐ *истина*;

дед(X,Y) ⇐ *родитель*(X,Y),*родитель*(Y,Z).

Программа, реализующая быструю сортировку Хоара, является примером определения, написанного в интегрированном стиле:

разбить(X, [], [], []) ⇔ true;

разбить(X, [Y|Z], [Y |W1], W2) ⇐ X ⇐ Y ➡
разбить(X,Z,W1,W2);

разбить(X, [Y|Z],W1, [Y|W2]) ⇐ *разбить*(X,Z,W1,W2);

сорт([],X) ⇔ X;

сорт([X|Y],Z) ⇐ *разбить*(X,Y,W1,W2)
сорт(W1, [X|*сорт*(W2,Z)]).

Отношение *разбить* определено в логическом духе. Определение же *сорт* - функциональное. Тем не менее правая часть его второго правила содержит логические переменные W1 и W2,отсутствующие в голове правила.

Следующая программа - функционально-логическое определение:

предок(*Пр*, *Пот*) ⇐ *родитель*(*Пр*, *Пот*) [*Пр*, *Пот*];

предок(*Пр*, *Пот*) ⇐ *родитель*(*Пр*,X),
[*Пр*|*предок*(X, *Пот*);

Функция *предок* возвращает список родственников, находящихся в генеалогическом дереве между *предком* и *потомком*. Во время вычисления этой *функции* Флэнг-система иногда использует возврат.

Несколько замечаний о программах. Атом " ⇐ " играет в "Флэнге" ту же роль, что и ":-" в "Прологе"; " ➡" - атом для обозначения отсечения; " ⇔ " - сокращение для "⇐ ➡" (эквивалентного ":!" в "Прологе"). Функции *факториал*, append и reverse имеют чисто функциональные определения. Определение функции *сорт* смешанное - для вычисления этой функции возврат не нужен,но переменные W1 и W2 - логические,и система использует унификацию для работы с ними. Отношения *родитель* и *дед* определяются так же, как в "Прологе". Функция *предок* неде-

терминированная в самом общем смысле (система при ее вычисле - нии может использовать бэктрэкинг).

Второй важный тип функций, введенных во "Флэнг", - *алгебраическая функция*. Алгебраические функции введены в язык для облегчения написания программ в области аналитических вычислений. Между недетерминированными и алгебраическими функциями имеется следующее отличие: когда Флэнг-системе не удается редуцировать недетерминированную цель, она включает механизмы возврата (бэктрэкинга); если же цель алгебраическая, то система оставляет цель нетронутой. Это может быть проиллюстрировано с помощью следующего простого примера. Определим функцию f: f(0) ⇐ 1. Эта функция определена только на 0. Различия между случаями, когда f - недетерминированная и алгебраическая, видны из таблицы:

f - *недетерминированная*	f - *алгебраическая*
Запрос f(0)	*Запрос* f(0)
Ответ 1	*Ответ* 1
Запрос f(1)	*Запрос* f(1)
Ответ fail	*Ответ* f(1)

Следующий пример демонстрирует диалог между пользователем и Флэнг-системой, когда "+", "*" объявлены алгебраическими:

Запрос (a+1) * b
Ответ (a+1) * b

Правило (X+Y) * Z ⇐ X * Z + Y * Z;
Запрос (a+1) * b
Ответ a * b + 1 * b

Правило 1 * X ⇐ X;
Запрос (a+1) * b
Ответ a * b + 1

и так далее.

Алгебраические функции одновременно обладают чертами и функции,и структуры данных.Используя это, во "Флэнге" можно определять конструкторы со специальными свойствами, например,конструктор *упорядоченного* списка. Для этого опишем атом " " как алгебраическую функцию и как инфиксный правоассоциативный оператор. Программа сортировки будет состоять только из одного простого и логически чистого правила:

$$X.Y.Z \Leftarrow X > Y \rightarrow Y.X.Z.$$

Теперь

Запрос 3.2.1.6.5.4.9.8.7.0.nil
Ответ 0.1.2.3.4,5,6,7,8,9.nil

Идея алгебраической функции играет во "Флэнге" примерно ту же роль,что и идея *несвободной алгебры* в языке "Миранда",разработанном Д.Тернером [7]. Тем не менее концепция алгебраической функции кажется ближе к ядру "Флэнга", чем несвободные алгебры - к ядру "Миранды".

2. Основные шаги компиляции

В этом пункте мы рассмотрим основные шаги компиляции Флэнг-программ. Компилятор выполняет следующие действия:

- трансляцию Флэнг-программы в стандартизированную форму;
- глобальный анализ потоков данных;
- трансляцию стандартизированной Флэнг-программы в промежуточный код абстрактной Флэнг-машины (FAM);
- трансляцию из промежуточного кода в код компьютера.

Первый шаг процесса компиляции - *трансляция Флэнг-программы в стандартизированную форму*. Основной наследственный недостаток абстрактной Флэнг-машины - неумение работать с вложенными вызовами функций. Таким образом, прежде чем транслировать Флэнг-программу, компилятор вынужден преобразовать ее, избавившись от термов вида f(...g(...)...), где вызов функ-

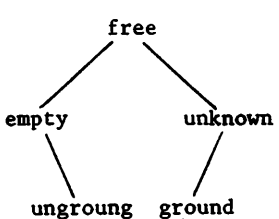
ции g является аргументом функции f. В определении факториала компилятор преобразует выражение X * (X-1)!. Процедура избавления от вложенных вызовов известна под названием flattening (*выравнивание*). Если дан терм f(...g(...)...), то компилятор преобразует его в V is g(...),...,f(...V...),где V - новая переменная. Переменные типа V обладают специфическими свойствами, допускающими важные оптимизации.

3. Глобальный анализ потоков данных

Глобальный анализ Флэнг-программ содержит следующие шаги:

- анализ аргументов;
- анализ точек возврата в программе;
- анализ и разделение функций и предикатов;
- выбор способа возвращения значения через "выходные" параметры.

Первый шаг - анализ аргументов - дает информацию, которая чрезвычайно важна для оптимизаций. Для всех функций компилируемой Флэнг-программы анализатор вычисляет типы аргументов. Мы можем выбрать множество различных алгоритмов для вычисления типов, обладающих различными уровнями сложности (некоторые из них описаны в [5-6]). В действующей версии Флэнг-компилятора мы используем следующую простую решетку типов:



Здесь free - аргумент является свободной переменной; это означает, что все вхождения переменной свободны; unground - аргумент есть терм со свободными переменными; ground - в аргументе

вхождения свободных переменных нет; unknown - различные вхождения аргумента имеют различные типы, при этом хотя бы одно вхождение имеет тип free.

В нашем компиляторе используется быстрый, но в то же время достаточно мощный алгоритм вычисления типов. Этот алгоритм напоминает *абстрактную интерпретацию* и может быть охарактеризован как "трассирование свободных переменных". Информация, полученная от этого шага анализа, очень важна. К примеру, когда все переменные в программе имеют либо тип ground,либо тип free, то программа может быть вычислена без использования следа (Trail).

Во время анализа точек возврата компилятор для каждой функции выясняет,является ли данная функция детерминированной или нет. Информация от предыдущих шагов анализа о том,имеет ли та или иная переменная тип ground или unground,позволяет произвести более утонченный анализ точек возврата и еще сократить их число.

Система также анализирует для каждой функции, используется ли где-нибудь в программе возвращаемое ею значение. Этот анализ позволяет не тратить время на возвращение ненужных значений в тех случаях, когда функция играет роль предиката.

Последняя часть глобального анализа выбирает методы возвращения значений выходных (output) параметров. Во "Флэнге" имеются два пути возвращения значений. Во-первых, могут использоваться свободные переменные (как в "Прологе"). Во-вторых, сами функции также могут иметь значение. Для обоих случаев компилятор может применить один из двух различных способов возвращения значений. Это возврат *значения* и возврат *ссылки*. Первый способ проще и генерирует меньшее количество ссылок. Но его применение делает невозможным использование такой важной оптимизации,как хвостовая рекурсия. Поэтому используется следующая схема. Компилятор распознает все случаи, когда хвостовая рекур-

сия может быть применена, и только для этих случаев применяется возврат ссылки.

4. Архитектура абстрактной Флэнг-машины

Абстрактная Флэнг-машина (FAM) является расширением и модификацией абстрактной машины Уоррена [2]. Она была разработана авторами как основа Флэнг-компилятора.

Основными сегментами памяти FAM являются:

- *стек (локальный стек)*,
- *куча (глобальный стек)*,
- *след* (Trail),
- *сегмент для регистров*.

Состояние FAM зависит от следующих регистров:

P - указателя на программу;

E - указателя на последнюю запись активации в локальном стеке;

B - указателя на последнюю точку возврата;

A - вершины локального стека;

Tr - вершины следа;

H - вершины кучи;

M - состояния унификации (write/read);

S - ссылки на текущую структуру (при унификации);

$R_1,\ldots,R_n$ - регистров для передачи параметров.

Для упрощения инструкций FAM мы будем также использовать регистр BP, хотя он никак не используется при реальных вычислениях и может быть охарактеризован как регистр времени компиляции.

Мы не предполагали разрабатывать абстрактную машину, полностью независимую от архитектуры реального компьютера. Проблема заключается в том, что тип архитектуры влияет не только на эффективность FAM-инструкций. Различные виды архитектуры приводят к различным оптимизационным принципам. Тем не менее

имеется инвариантная часть FAM, общая для всех типов архитектур.

В компиляторе была реализована модификация FAM, наиболее приспособленная для IBM PC. Поскольку процессор IBM PC имеет очень маленькое число регистров и эти регистры специализированные, Флэнг-компилятор не держит основные FAM-регистры в оперативной памяти, а использует для них регистры процессора PC. Например, FAM-оператор P находится в hardware-регистре IP, E находится в BP, A - в SP, H - в BX. Другие регистры распределяются в оперативной памяти.

Для повышения эффективности порождаемого кода в случае арифметических вычислений система использует еще один регистр - T (аккумулятор). Он находится в регистрах AX и CX процессора PC. Кроме того, множество FAM-инструкций расширено инструкциями, работающими с регистром T.

Из вышесказанного, в частности, следует, что в случае IBM PC такая известная оптимизация, как минимизация числа регистров $R_1,\ldots,R_n$ весьма несущественна. С другой стороны, возможность группировать вместе инструкции, имеющие вхождение одной и той же переменной, очень важна, поскольку она позволяет использовать регистр T максимально эффективно.

5. Компиляция в абстрактную Флэнг-машину

Компиляция в FAM выполняется независимо для каждого правила Флэнг-программы с использованием информации, полученной от глобального анализа. Чтобы описать процесс компиляции, нам нужно ввести понятия *правой* и *левой* частей правила. Левая часть правила включает голову правила и все цели вплоть до первого вызова функции, определенной пользователем

```
X! <= X > 0, V1 is X-1,    V2 is V1!, V3 is X * V2, V3;
      ------------------    ---------------------------
          левая                    правая
```

Модуль Флэнг-компилятора, транслирующий правило, последовательно выполняет следующие шаги:

- анализ унификаций в голове правила;
- генерирование подходящих try-инструкций для правила;
- генерирование инструкций для левой части правила;
- генерирование инструкций, порождающих запись активации в локальном стеке;
- генерирование инструкций для правой части правила.

Важное свойство компилятора - способность избегать установки точек возврата в коде детерминированных функций. Флэнг-компилятор минимизирует число точек возврата и генерирует только необходимые части точек возврата, когда это возможно.

Утонченная работа с точками возврата позволяет использовать три различных типа возврата (бэктрэкинга):

- *ветвление* - нет необходимости восстанавливать значения, точка возврата также не генерируется;
- *близкий возврат* - система восстанавливает состояние кучи и следа, но не регистры;
- *дальний (стандартный) возврат* - обычный полный бэктрэкинг.

Чтобы проиллюстрировать процесс выбора подходящего типа возврата, введем следующее понятие. Будем говорить, что унификация *усложненная*, если она влечет рост кучи или следа.Используя информацию,предоставленную глобальным анализом, компилятор правила подсчитывает число усложненных унификаций. Пусть теперь выполняется одно из следующих условий:

- нет усложненных унификаций;
- правило детерминированное и содержит только одну усложненную унификацию, которая может быть выполнена после всех других.

Тогда в коде для этого правила используется *ветвление* и установку точки возврата (если она необходима) можно оттянуть до того момента,когда вычисление левой части правила закончено.

6. Оценка производительности Флэнг-компилятора

В этом пункте мы сравним производительность Флэнг-системы с наиболее известными Пролог-системами, а также с программами, написанными на "Паскале" в системе TurboPascal.

Сравнения показывают, что производительность Флэнг-системы очень высока. Откомпилированный код не только значительно быстрее кода, генерируемого не слишком эффективным компилятором системы Arity/Prolog, но и кода, созданного системой TurboProlog, чей язык был специально "испорчен",в угоду эффективности, введением обязательных описаний типов данных.Наш компилятор не имеет подобной информации от пользователя.Но он имеет ту же самую и много другой информации от глобального анализа.

Сравнение откомпилированных Флэнг-программ с соответствующими программами, написанными на "Паскале",показывает, что для некоторых классов задач Флэнг-программы имеют примерно ту же скорость, что и Паскаль-программы. Основная проблема в том,что в "Паскале" доступны деструктивные средства, такие, как прис - ваивание. Они допускают иные чрезвычайно эффективные стратегии вычислений. Когда эти стратегии используются в большой степени, текущая версия Флэнг-системы генерирует более медленный код. Но мы надеемся, что использование ряда мощных оптимизаций (таких, как оптимизация последнего использования структуры) поможет значительно улучшить ситуацию.

Опишем задачи, для которых проводились измерения. Первая задача - алгебраическая версия программы сортировки (*алг_сорт*): X.Y.Z ⇐ X > Y ➡ Y.X.Z.

Пролог-программа, имеющая ту же стратегию вычислений:

```
алг_сорт(X, [Y|Z], Res): - X   Y,!,алг_сорт(X,Z,U),
        алг_сорт(Y,U,R),
алг_сорт(X,Y,  [X|Y]).
```

Обычный список играет в этой программе роль ".". Сортировался список [300, 299, 298,...,2,1].

Тот же список сортировался программами быстрой сортировки (*быст_сорт*). Программа на "Паскале" существенно использует преимущества оператора присваивания и других свойств императивных языков, поэтому она примерно в два раза быстрее.

Приведем еще несколько программ, на которых проводились сравнения. Начнем с функции Аккермана:

```
akkermann(0,N) <=> N+1;
akkermann(M,0) <=> akkermann(M-1,1);
akkermann(N,M) <= akkermann(N-1, akkermann(N,M-1)).
```

Функция fun6 позволяет оценить эффективность вызовов функций и работы со стеком. В зависимости от параметра X оценка числа вызовов - 6^X:

```
fun6(0) <=> 1;
fun6(X) <= fun6(X-1), fun6(X-1), fun6(X-1), fun6(X-1),
           fun6(X-1), fun6(X-1).
```

Вычислялось значение fun6(7).

Функция *простые_числа*(N) возвращает спикок N первых простых чисел:

```
остаток(Num, X) <= Num < X => Num;
остаток(Num, X) <= остаток(Num-X, X);
провер(Pri, 1). <=> true;
провер(Pri, Den) <= остаток(Pri, Den) < 0,
        провер(Pri, Den-1);.
простые_числа(1) <=> [];
простые_числа(X) <= провер(X,X-1) => [X|
        простые_числа (X-1)];
простые_числа(X) <= простые числа(X-1).
```

Вычислялось выражение *простые_числа (200)*.

Наконец, определение функции *много_возвратов* демонстрирует пользу глобального анализа точек возврата:

```
много_возвратов(0) <=> true;
много_возвратов(X) <= возврат(1,1000) =>
        много_возвратов(X-1);
возврат(X, Dim) <= X > Dim;
возврат(X, Dim) <= возврат(X+1, Dim).
```

Вычислялось значение *много_возвратов(100)*. В процессе вычислений система проводит 100 тыс. бэктрэкингов.

Основные результаты сравнений приведены ниже. Время выполнения дано в секундах. Сравнения проводились на компьютере IBM PC XT с тактовой частотой 4.77 МГц:

Программа	"Флэнг"	"Паскаль"	TurboProlog	Arity
алг_сорт	*19.88*	*20.93*	*34.11*	*69..60*
быст_сорт	*11.05*	*6.04*	*20.21*	
akkermann	*18.95*	*19.68*	*23.34*	
fun6	*25.35*	*29.48*	*35.37*	*201.55*
простые_числа	*6.24*		*14.17*	
много_возвратов	*8.9*		*31.17*	

З а к л ю ч е н и е

Подведем некоторые итоги. Основной из них заключается в том, что программы, написанные на языках высокого уровня,могут быть откомпилированы в чрезвычайно быстрый код. Для важного класса задач эффективность кода практически не уступает коду, сгенерированному компиляторами с императивных языков ("Си","Паскаль"). Достаточно мощные системы, поддерживающие языки высокого уровня,могут быть развиты даже на относительно маленьких и слабых компьютерах.

Сейчас мы развиваем коммерческую версию Флэнг-системы.Она содержит не только быстрый компилятор, который был описан выше, но поддерживает и ряд дополнительных технологий искусствен-

ного интеллекта, в частности, такой мощный метод дискретной оптимизации,как удовлетворение ограничений на конечных областях [3].

Литература

1. MANTSIVODA A. Flang: A Functional-Logic Language // Proc. of Int.conf on Processing Declarative Knowledge.-Kaiserslautern, July 1991.

2. WARREN D.H.D. An Abstract Prolog Instruction Set.Technical Note 309 SRI International, Menlo Park, CA, October 1983.

3. HENTENRYCK P., van. Constraint Satisfaction in logic Programming. The MIT Press, Cambridge, 1989.

4. BOLEY H. A relational/functional Language and its Compilation into the WAM. SEKI Report SR-90-05, University of Kaiserslautern,1990.

5. ROY P.L.,van. Can Logic Programming Execute as Fast as Imperative Programming? PhD Dissertation, University of California at Berkeley, November *1990.*

6. TAYLOR A. High Performance Prolog Implementation. PhD. Dissertation, Basser Department of Computer Science, University of Sydney, June 1991.

7. TURNOR D. An Onorviow of Mitanda //CIOPLAN Naticoo.Vol. 21, № 12.

Поступила в ред.-изд.отд.

6 июля 1992 года