К. А. Пыжов

ВНУТРЕННИЕ ПРЕДСТАВЛЕНИЯ СРЕДНЕГО УРОВНЯ ДЛЯ КОМПИЛЯТОРОВ ЯЗЫКА SISAL*

ВВЕДЕНИЕ

Функциональные языки и языки однократного присваивания имеют семантику, базирующуюся на понятии «значение», а не «ячейка памяти», и это дает им ряд преимуществ. В частности, такой подход позволяет трансляторам функциональных языков осуществлять эффективное распараллеливание вычислений. Отсутствие побочных эффектов делает анализ зависимостей по данным и возможностей параллелизма гораздо более легкой задачей, чем в случае с императивными языками. В частности, отпадает необходимость трудоемкого анализа зависимостей по данным для применения тех или иных оптимизирующих преобразований, а также для распараллеливания и векторизации. Во внутреннем представлении потоковых языков зависимости по данным представлены в явном виде. К слову, предлагаются различные механизмы, позволяющие императивным языкам использовать преимущества концепции однократного присваивания. Например, в последнее время в компиляторах для императивных языков широкое распространение получила SSA-форма [1, 2], которая вносит в представление программы положительные стороны языков однократного присваивания. SSA-форма позволяет эффективно удалять избыточные зависимости по данным и, тем самым, расширяет и упрощает применение оптимизирующих преобразований.

К функциональным языкам однократного присваивания, свободным от побочных эффектов, относится Sisal [8]. Sisal — довольно известный потоковый язык, предназначенный для параллельных вычислений. От языка VAL отличается поддержкой ошибочных значений, допустимостью рекурсии, потоковым типом данных. Кроме того, Sisal имеет наглядный синтаксис, сходный с языком Pascal. Все эти особенности делают язык Sisal удобным для записи больших программ, содержащих сложные математические расчеты. Однако с точки зрения компилятора

*Работа частично поддержана Российским фондом фундаментальных исследований (грант РФФИ № 07-07-12050).

концепция однократного присваивания и семантика, базирующаяся на значениях, создают специфические трудности, связанные прежде всего с необходимостью использования больших объемов динамической памяти и управлением параллельно исполняемыми задачами.

Нужно отметить два основных источника неэффективности. Первый — это необходимость создания большого количества динамических объектов. Для эффективного использования памяти необходимо организовать хорошее управление этими объектами во время исполнения программы. Второй источник неэффективности возникает как раз из-за отсутствия побочных эффектов: изменение элементов структурного объекта (такого как массив, например) семантически предполагает создание новой копии всего объекта. Такое копирование может быть достаточно «дорогим» и значительно влиять на эффективность кода. Однако существует возможность во многих случаях избегать таких копирований. Одним из подходов к решению этой задачи является создание промежуточного представления среднего уровня, которое наследует потоковую структуру первого представления, но при этом содержит информацию о размещении объектов в памяти [6]. Такая информация позволяет организовать оптимизацию использования динамических объектов. Во многих случаях удается избежать необходимости копирования всего объекта, ограничиваясь модификацией какой-либо его части. Понятно, что такого рода преобразования требуют выполнения определенных условий использования объектов и выполнения соответствующего анализа программы.

В данной статье представлено краткое описание некоторых проблем, возникающих при трансляции функциональных программ и методов их решения.

В первой и второй частях статьи содержится обзор различных методов, применяющихся для оптимизации работы с памятью и управления многозадачностью в компиляторах функциональных языков.

В третьей части дано краткое описание внутренних представлений, разработанных для нового компилятора языка Sisal 3.1, разрабатываемого в Лаборатории конструирования и оптимизации программ ИСИ СО РАН.

1. ТЕХНОЛОГИИ ОПТИМИЗАЦИИ РАБОТЫ С ПАМЯТЬЮ

В Ливерморской лаборатории в рамках работы над созданием оптимизирующего транслятора языка Sisal 1.0 было разработано проме-

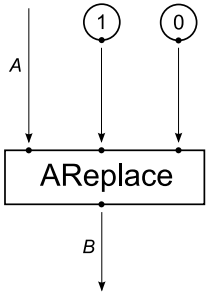

Рис. 1. Операция AReplace

жуточное представление IF2 (наряду с представлением верхнего уровня IF1), расширяющее представление IF1 операциями распределения и управления памятью для объектов [5]. Если представление IF1 не несет никакой информации о размещении частей объектов в памяти, то IF2 делает некоторые предположения: например, требование размещения элементов одномерного массива в последовательных ячейках памяти. Операции со скалярными значениями остаются неизменными.

Пример 1. На рис. 1 показано потоковое представление операции AReplace языка Sisal, выполняющей замену элемента массива (Первый элемент массива заменяется на 0). Операция задается следующим выражением языка Sisal:

```
B := A[1:0].
```

В семантике представления IF1 операция AReplace создает новую копию массива A с измененным первым элементом. Соответственно, необходимо создание нового динамического объекта для новой копии массива. Однако если AReplace является последним использованием массива A, мы можем записать новое значение в область памяти, используемую для первого элемента этого массива, и использовать память, отведенную для A, для размещения массива B. В графе IF2 дуга, представляющая массив A, будет помечена специальным свойством, указывающим, что к этому массиву возможно обращение «по ссылке», т.е. обращение не только к значению, но и к соответствующей области памяти. При этом для новой копии массива будет использован тот же самый буфер динамической памяти.

Кроме того, в представление IF2 добавляются искусственные дуги

зависимостей. Дело в том, что в некоторых случаях бывает удобно задержать выполнение оператора над большой структурой данных до тех пор, пока все остальные ссылки на эту структуру не будут вычислены. Такая задержка позволит операции обращаться к данным по ссылке, избегая таким образом копирования. В качестве примера рассмотрим представление для фрагмента программы на рис. 2. Если задержать исполнение оператора AElement до завершения операции ASize, то AElement сможет работать с массивом A по ссылке, без создания новой копии. Для обозначения этой задержки в граф вводится дуга искусственной потоковой зависимости, которая проводится от вершины ASize к вершине AElement.

```
function length(N:integer; L:integer  returns integer, integer)
   let
      A:= for I in 1, N
             returns array of I
          end for
   in
      size(A),
      A[L]
   end let
end function
```

В Стенфордском университете был разработан back-end транслятор для IF1, в котором основное внимание уделялось динамическому распределению памяти для массивов и процедуре освобождения динамической памяти [4]. Механизм освобождения памяти был основан на подсчете ссылок (reference counting). Разработанный менеджер памяти имел такие механизмы, как передача параметров-структур по указателям и разделение доступа к структурам. Для упрощения задачи динамического подсчета ссылок на объекты была реализована двухпроходная схема кодогенерации: на первом проходе по графу IF1 происходила генерация информации о последних использованиях структур (само представление IF1 никак не определяет порядок исполнения своих элементов), на втором проходе осуществлялась непосредственно кодогенерация.

В 1989 г. в университете Колорадо совместно с Ливерморской лабораторией был разработан компилятор OSC (Optimizing Sisal Compiler) [8]. Процесс трансляции состоял из следующих фаз:

1) font-end трансляция в представление IF1;
2) построение единого IF1 графа программы;
3) build-in-place анализ и трансляция в IF2;

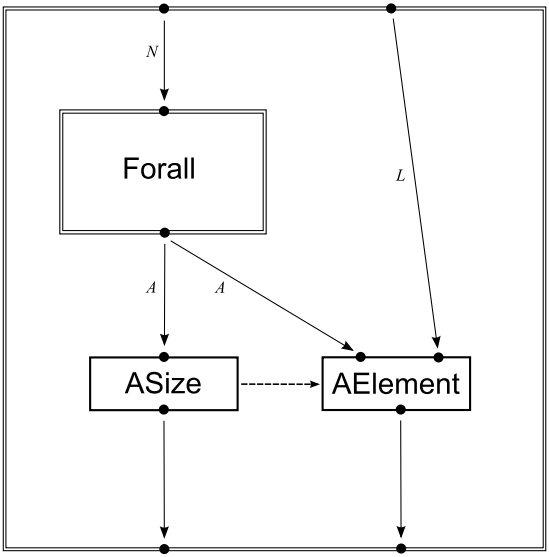

Рис. 2. Искусственные дуги зависимости в IF2

4) update-in-place анализ;
5) кодогенерация.

После построения единого графа программы на представлении IF1 производятся такие оптимизации, как межпроцедурная протяжка значений, подстановка функций, вынос инвариантного кода, удаление общих подвыражений, свертка константных вычислений, удаление мертвого кода.

Задачей build-in-place анализа является решение проблемы конструирования объектов путем предварительного распределения буферов памяти для массивов вне зависимости от того, является ли размер массива известным во время компиляции. Результатом build-in-place анализа является трансляция программы в представление IF2.

Затем программа в представлении IF2 подвергается update-in-place анализу для решения проблемы модификации объектов. Выявляются операции преобразования, которые могут быть выполнены локально. Анализ включает три фазы. Сначала в графы IF2 добавляются специальные вершины, помечающие операторы, дублирующие объекты. Целью оставшихся шагов является удаление ненужных дублирований.

Первая фаза включает также аннотирование каждой дуги, несущей агрегатный тип, значением счетчика ссылок в худшем случае (это возможно на стадии компиляции, поскольку все агрегатные объекты являются ациклическими). Вторая фаза переставляет, где это возможно, вершины в каждом графе, вводя при этом искусственные дуги зависимости. На этом шаге устраняются также счетчики ссылок, оказавшиеся ненужными в результате перестановки вершин. Наконец, заключительная фаза удаляет все избыточные операции дублирования, введенные на первом шаге.

В качестве кодогенератора в компиляторе OSC использовался транслятор оптимизированного представления IF2 в эквивалентную программу на языке C.

2. РАСШИРЕНИЕ ЯЗЫКА SISAL ДЛЯ УПРАВЛЕНИЯ МНОГОЗАДАЧНОСТЬЮ

Практика показывает, что SISAL-программы содержат значительный потенциальный параллелизм. Проблема заключается в преобразовании этого внутреннего параллелизма в множество эффективных задач для многопроцессорной системы. Во-первых, важно выбрать правильную стратегию выделения задач и управления ими во время исполнения программы. Во-вторых, следует производить оптимизацию последовательного кода задач, которая позволит SISAL-коду быть сравнимым по производительности с последовательными программами на таких языках, как C и Fortran.

Но трансляция потока вычислений для исполнения на традиционных «фон-неймановских» архитектурах вызывает трудности. Концепция однократного присваивания и отсутствия побочных эффектов кардинально отличается от концепции последовательного исполнения традиционных архитектур. Осознание необходимости эффективной трансляции функциональных языков для исполнения на таких архитектурах подтолкнуло к идее комбинирования двух концепций (кстати, ставилась и обратная задача — трансляция последовательных языков на потоковые архитектуры. Но эта идея широкого распространения не получила).

SISAL допускает различные модели исполнения. Например, возможны потоковая, макропотоковая, векторная, модель систолических массивов. С точки зрения компилятора конструкциями, вводящими параллелизм, являются параллельные циклы и вызовы функций. Однако при этом не предполагается никаких возможностей для программиста явно

задавать распараллеливание и выделение задач в программе. Большинство компиляторов потоковых языков выделяют независимо исполняемые блоки кода, исходя из итеративных конструкций и вызовов функций, содержащихся в программе. Управление задачами и их синхронизация являются исключительно задачами компилятора. Императивные языки, напротив, предоставляют программисту широкий набор средств для явного описания параллельных частей программы (например, широко распространенная технология OpenMP). Как известно, в потоковых языках распараллеливание базируется на выявлении независимых вычислений (вершин потокового графа). Любые конструкции, не связанные между собой потоком данных, могут быть исполнены параллельно. Если же они связаны, выполнение зависимой конструкции возможно лишь тогда, когда будут вычислены все конструкции, поставляющие данные этой вершине. Исходя их этого, строится стратегия выделения задач и управления ими.

Было предложено расширение языка Sisal [3], позволяющее пользователю задавать процессы в Sisal-программе в явном виде. Явное определение процесса может находиться в любом месте Sisal-программы и включать любые функции и выражения. Выглядит определение процесса следующим образом:

```
process <process-name>
    <Sisal statement>;
    <Sisal statement>;
    ...
    ...
    function f1(...)
    ...
    ...
end process
```

Для управления процессами предусмотрены директивы fork и join. Конструкция fork выглядит так:

```
fork(P_1, P_2, ... P_n);
```

где P_i — имя процесса. Директива fork разрешает разделение n процессов и выполнение их на разных процессорах, если это позволяют ресурсы. Конструкция join имеет вид:

```
join(P_1, P_2, ... P_m);
```

Конструкция join вызывает принудительную синхронизацию процессов: уже завершившиеся процессы будут остановлены до завершения всех процессов из списка $P_1, P_2, ...P_m$.

Компилятор помечает процессы, участвующие в директиве fork, для

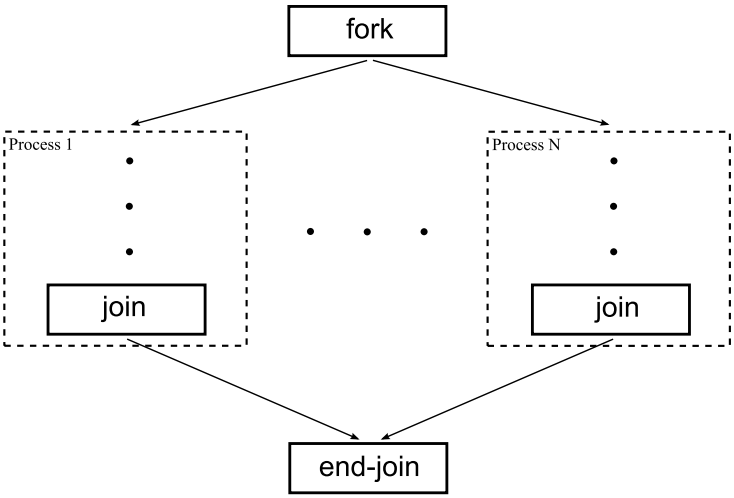

Рис. 3. Синхронизация процессов

последующего параллельного исполнения. В конец каждого процесса из директивы join добавляется специальная вершина, обозначающая, что процесс участвует в синхронизации. Выходы всех таких вершин для всех процессов, обозначенных директивой join, подаются на вход дополнительной «end-join» вершине. Присутствие этой вершины гарантирует одновременное завершение этих процессов (рис. 3).

3. ПРОМЕЖУТОЧНЫЕ ПРЕДСТАВЛЕНИЯ ДЛЯ КОМПИЛЯТОРА ЯЗЫКА SISAL 3.1

В Лаборатории конструирования и оптимизации программ ИСИ СО РАН разрабатывается компилятор, реализующий язык Sisal 3.1. Новая версия языка имеет несколько нововведений, призванных сделать язык более наглядным, гибким и удобным для использования на практике [10]. Однако дело осложняется отсутствием законченной рабочей версии компилятора, а также достаточно широкой тестовой базы. Понятно, что эти проблемы взаимосвязаны.

Первое внутреннее представление (представление front-end транслятора) IR1 является ориентированным ациклическим иерархическим графом потока данных в программе [7]. Каждой дуге приписан тип

значения, которое она несет. Вершины обозначают операции над своими аргументами (входами), результаты которых находятся на выходах вершины. Структурированность вычислений отражается иерархичностью графов IR1. Представление IR1 является полностью машинно-независимым.

Следующее промежуточное представление IR2 представляет собой потоковый граф, подобный IR1, с некоторыми дополнительными конструкциями. В частности, каждой дуге графа приписывается переменная. Дугам, идущим из одного порта, приписывается одна и та же переменная. С использованием этой дополнительной информации осуществляется анализ представления IR2 с целью оптимизации динамической памяти.

Первым этапом построения IR2 является конвертация потокового графа IR1 в аналогичный граф IR2. С точки зрения структуры граф IR2, полученный сразу после трансляции IR1 → IR2, является изоморфным графу IR1, поэтому собственно создание вершин и дуг представления IR2 является тривиальным.

На следующем шаге для дуг, выходящих из каждого порта, создается уникальная переменная, описывающая некоторую область памяти (динамическую или статическую, в зависимости от типа переменной). При этом сохраняется семантика однократного присваивания. Переменные наследуют типы языка Sisal, приписанные к соответствующим дугам графа IR1. Кроме того, каждая переменная описывается такими машинно-ориентированными характеристиками, как двоичный тип, размер и т. д. Таким образом, представление IR2 является машинно-зависимым.

Следующей фазой является оптимизация IR2, которая выполняет оптимизацию распределения переменных агрегатных типов (тем самым удаляя избыточные копирования), а также выносит инвариантные вычисления из графов циклов.

Далее следует трансляция IR2 в IR3. Для каждой вершины представления IR2 вычисляется приоритет ее исполнения (на основе потока данных). Затем для вершин строятся последовательности операторов IR3, которые располагаются согласно приоритетам исполнения вершин IR2.

Представление IR3 является высокоуровневым императивным представлением. В процессе трансляции IR2 в IR3 из конструкций IR2 выделяются простые операторы. Переменные IR2 наследуются.

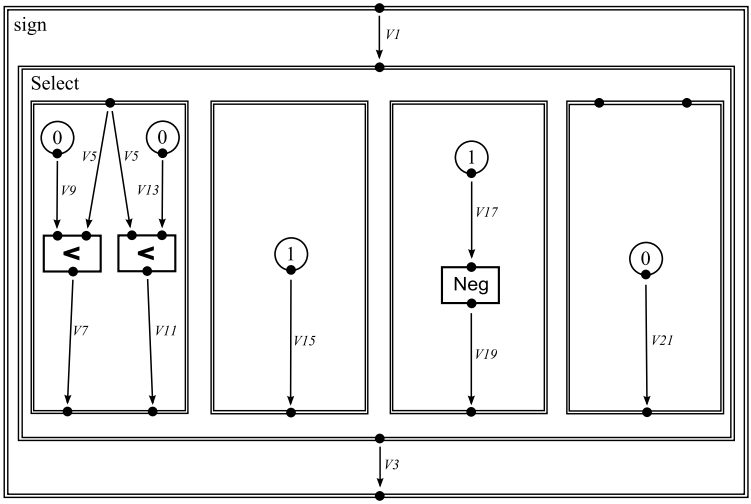

Рис. 4. Граф IR2

На этапе оптимизации IR3 выполняются такие преобразования, как протяжка копий, удаление мертвого кода, удаление бесполезных присваиваний.

В качестве кодогенератора используется транслятор IR3 в программу на языке C#. В дальнейшем предполагается реализация кодогенератора, осуществляющего трансляцию представления IR3 в псевдокод IL платформы Microsoft .NET.

Пример. На рис. 4 показано представление IR2 для следующего фрагмента Sisal-программы (функции, вычисляющей знак числа):

```
function sgn(N: integer returns integer)
  if     N > 0 then  1
  elseif N < 0 then -1
  else               0
  end if
end function
```

Распечатка представления IR3, полученного для функции *sign* после трансляции IR2 → IR3, выглядит так:

```
  0    entry "function sgn[integer]" (V_1(I32)  returns V_3(I32));
       {
  1        V_5(I32) = V_1(I32);
  2        V_5(I32) = V_1(I32);
```

```
 3        V_9(I32) = 0x0(I32);
 4        V_13(I32) = 0x0(I32);
 5        V_7(BOOL) = (V_9(I32) < V_5(I32));
 6        V_11(BOOL) = (V_5(I32) < V_13(I32));
 7        if (V_7(BOOL) == true(BOOL))
          {
10            V_15(I32) = 0x1(I32);
11            V_3(I32) = V_15(I32);
          }
          else
          {
12            if (V_11(BOOL) == true(BOOL))
              {
15                V_19(I32) = 0x1(I32);
16                V_17(I32) =  - V_19(I32);
17                V_3(I32) = V_17(I32);
              }
              else
              {
18                V_21(I32) = 0x0(I32);
19                V_3(I32) = V_21(I32);
              }
          }
20        return;
      }
```

Содержимое IR3 после оптимизации:

```
 0    entry "function sgn[integer]" (V_1(I32)  returns V_3(I32));
      {
 5        V_7(BOOL) = (0x0(I32) < V_1(I32));
 6        V_11(BOOL) = (V_1(I32) < 0x0(I32));
 7        if (V_7(BOOL) == true(BOOL))
          {
11            V_3(I32) = 0x1(I32);
          }
          else
          {
12            if (V_11(BOOL) == true(BOOL))
              {
17                V_3(I32) =  - 0x1(I32);
              }
              else
              {
19                V_3(I32) = 0x0(I32);
              }
          }
20        return;
      }
```

ЗАКЛЮЧЕНИЕ

В статье показаны некоторые проблемы, связанные с оптимизацией программ, написанных на функциональных языках. Приведен обзор технологий, применяемых для оптимизации работы с памятью. Показаны преимущества использования внутренних представлений среднего уровня, ориентированных на оптимизацию распределения динамической памяти. Также дано краткое описание промежуточного представления среднего уровня, разработанного и реализованного для компилятора языка Sisal версии 3.1.

Важнейшим направлением дальнейшей разработки компилятора языка Sisal 3.1 является поддержка параллелизма вычислений. Для этого необходимо реализовать выделение параллельных регионов в IR2 и поддержку параллельного кода на уровне IR3.

В направлении оптимизирующих преобразований планируется реализация оптимизации ошибочных значений. Это позволит избежать проверок ошибочности значений для каждой операции и значительно повысит производительность получаемого кода.

Другим важным направлением работы видится создание полноценного кодогенератора для платформы .NET.

СПИСОК ЛИТЕРАТУРЫ

1. Allen R., Kennedy K. Optimizing Compilers for Modern Architectures. — Morgan Kaufmann, 2002.
2. Muchnik S. Compiler Design and Implementation. — Morgan Kaufmann, 1997.
3. Vijayaraghavan V., Kavi K., Shirazi B. Control Flow Extensions To The Dataflow Language SISAL. — IEEE, 1991.
4. Gharachorloo K., Sarkar V., Hennessy J. A Simple and Efficient Implementation Approach for Single Assignment Languages. — ACM, 1988.
5. Welcome M., Skedzielewski S., Yates R., Ranelletti J. IF2: An Applicative Language Intermediate Form with Explicit Memory Management. — Manual M-195. — University of California Lawrence Livermore National Laboratory, 1986.
6. Cann D. C. Compilation Techniques for High Performance Applicative Computation — PhD thesis, Colorado State University, 1989.
7. Стасенко А. П. Система интерфейсов транслятора во внутреннее представление IR1 // Методы и инструменты конструирования и оптимизации программ. — Новосибирск, 2005. — С. 229–238.
8. Стасенко А. П., Синяков А. И. Базовые средства языка Sisal 3.1. — Новосибирск, 2006. — 60 с. — (Препр. / РАН. Сиб. Отд-е. ИСИ; № 110).