В. А. Маркин, С. А. Маркина

СИСТЕМА ДЛЯ БЫСТРОГО ПРОТОТИПИРОВАНИЯ РАСПАРАЛЛЕЛИВАЮЩЕГО КОМПИЛЯТОРА ПРОГРЕСС-2. ЯДРО СИСТЕМЫ. СЦЕНАРИЙ СИСТЕМЫ[*]

ВВЕДЕНИЕ

Данная статья является продолжением статьи [1], описывающей систему для быстрого создания прототипа распараллеливающего компилятора. Система создается как инструмент для манипулирования программами; что означает пошаговое преобразование программ — начиная с исходного текста программы на одном из языков высокого уровня, переводим программу в одно из внутренних представлений и далее пропускаем программу через различные проходы-блоки. В конце концов, мы можем получить текстовое представление распараллеленной программы либо исполняемый код для запуска программы на одной из целевых архитектур. Теоретические размышления и предпосылки создания подобной системы можно найти в работах [2,3] по проекту ПРОГРЕСС, в рамках которого и начиналась работа над описанной далее системой.

Центральным понятием работы системы является СЦЕНАРИЙ прототипа создаваемого компилятора. В сценарии определяется множество и порядок исполнения функциональных и инструментальных компонент системы. Компонентом системы может быть либо front-end с одного из языков высокого уровня, обход дерева внутреннего представления системы либо анализ свойств программ. Компонент может быть интерактивным, ориентированным на взаимодействие с пользователем, иметь внешний графический интерфейс. Такие компоненты, например, могут служить для визуализации одного из теоретико-графовых внутренних представлений программ. Однотипные компоненты могут быть объединены в функциональные блоки. Функциональные и инструментальные блоки реализуются как внешние библиотеки (dll либо элементы ActiveX) с учетом заданных интерфейсов взаимодействия с системой, а точнее с Внутренним Представлением Систе-

[*] Работа выполнена при финансовой поддержке Российского Фонда Фундаментальных Исследований (гранты № 02-07-90409 и 03-07-06-112).

мы Прогресс-2. Регистрация либо загрузка соответствующих компонентов происходит во время запуска системы, либо во время начала работы над заданным СЦЕНАРИЕМ.

Исходя из выше сказанного, система рассматривается как КОНСТРУКТОР для построения прототипа компилятора, кирпичиками которого являются различные функциональные и инструментальные компоненты. С помощью заданного СЦЕНАРИЯ пользователь сможет построить, исполнить либо исследовать прототип построенного компилятора.

Компоненты системы могут быть реализованы как отдельные процедуры внешней библиотеки, взаимодействующие с заданными объектами внутреннего представления системы. Либо компонент может быть определен как тип СООБЩЕНИЯ плюс множества ОБРАБОТЧИКОВ-ДЕЙСТВИЙ на данное сообщение для определенных типов объектов внутреннего представления. Для активизации данного типа компонента достаточно передать сообщение объекту внутреннего представления, обычно вершине какого-либо графового представления программы.

ЯДРО И ВНУТРЕННЕЕ ПРЕДСТАВЛЕНИЕ СИСТЕМЫ

Требования и общая модель

Основным компонентом системы является расширяемое внутреннее представление системы (ВП), которое служит для связи между компонентами. Главными целями при его проектировании являлись *универсальность (общность), расширяемость и гибкость*. Данное представление должно было бы позволить переводить во внутреннее отображение многие широко распространенные языки программирования, а также обобщить различные теоретико-графовые внутренние представления, разработанные за последнее время.

Модель разработанного внутреннего представления можно разделить на две составляющие: Языковая составляющая и Графовая составляющая.

Языковая составляющая ВП — это множество синтаксических и семантических объектов, иерархически упорядоченных, отображающих большинство общих конструкций входных языков. Все объекты распределяются на области видимости, тем самым ВП включает в себя таблицы объектов для каждой области видимости и операторную часть. Операторная часть представлена в виде абстрактного синтаксического графа, узлы которого делятся на семантические и управляющие вершины. К семантическим вер-

шинам относятся if-вершины, loop-вершины, или операторные вершины. К управляющим относятся вершины-директивы сценария, или дополнительные вершины ВП. Множество конструкций, соответствующих большинству конструкций императивных языков программирования, расширено операторами с параллельной семантикой исполнения.

К графовой составляющей ВП относятся различные теоретико-графовые представления программ, такие как управляющие граф, ГПЗ, граф зависимостей по данным и т.д. Пользователь может реализовать свое графовое представление и встроить в систему, при условии, что типы вершин графа будут унаследованы от базового типа Node_Object, который, в свою очередь, унаследован от основного класса OMA_Object.

Базовые классы и механизм передачи сообщений

Базовым классом для всех объектов ВП системы является класс OMA_Object. Основным атрибутом данного класса является указатель на объект типа OMA_ObjectType(*). Классы, наследованные от класса OMA_ObjectType, определяют типы различных объектов ВП и содержат локальные кучи объектов OMA_Object данного типа, что в частности позволяет динамически вести контроль выделения и удаления памяти. Внутренняя реализация класса (*) позволяет вести регистрацию всех наследуемых от него типов, динамического создания объектов и контроль типов.

При рассмотрении интерфейса класса OMA_Object:

```
class  PUBLIC OMA_Object: public OMA_Message{
public:
  static OMA_ObjectType *Type();
...
private:
  OMA_Act *act;
public:
  OMA_ObjectType *ObjectType() const
 …
//
  void Act(OMA_Message &msg)
OMA_Act *SetAct(OMA_Act *new_act=NULL)
…
static void ClassAct(OMA_Object *obj, OMA_Message &msg);};
```

помимо указателя на тип объекта можно выделить дополнительную составную часть атрибутов и методов, связанных с обработкой СООБЩЕНИЙ

(объекты, наследованные от класса OMA_Message). Данные СООБЩЕНИЯ могут быть реализацией какого-либо функционального компонента в виде прохода объектов ВП. При этом определенным типам объектов ВП устанавливается определенный обработчик выше указанного СООБЩЕНИЯ.

Рассмотрим использование механизма передачи сообщений на пример вычисление арифметического выражения. Допустим, что входное арифметическое выражение

```
(5+10-25)*2
```

было переведено во ВП в виде бинарного дерева. Вершины данного дерева являются экземплярами класса

```
class Exp_Node {        // должен быть наследован от OMA_Object или, если мы хотим воспользоваться средствами визуализации графовых представлений от Node_Object

// Предположим второе

…

Uint type; // 0 — если числовая константа 1- если сложение 2 — минус и т.д.

OMA_Variant val;

// значение числовой константы Описание класса OMA_Variant смотри ниже

…

}
```

Переменная

```
Exp_Node* Root //указатель на вершину
```

указывает на вершину.

Определяем новый тип сообщение

```
Class Calc_Exp: public OMA_Message{

…

  OMA_Variant curVal;

}
```

и определяем обработчик сообщений для типа вершин Exp_Node для сообщения Calc_Exp.

```
Static void OnCalcExp(Exp_Node* c, Calc_Exp & msg)
{ if (!c->type) msg.curVal=val;
    else
  {     c->son[0]->Act(msg);
            OMA_Varinat left=msg.curVal;
            c->son[1]->Act(msg);
        if (c->type==1) msg.curVal=left+msg.curVal;
        else
        … // Для остальных операций
  }
}
```

Теперь необходимо подключить обработчик сообщений к типу вершин Exp_Node

```
Exp_Node::Type()->SetAct(OnCalcExp, Calc_Exp::Type());
```

и запустить проход, передав сообщение вершине дерева

```
Calc_Exp msg;
Root->Act(msg);
```

После обработки сообщения в поле msg.curVal мы получим значение нашего арифметического выражения.

Введение механизма передачи СООБЩЕНИЙ позволяет легко переносить и модифицировать уже реализованные библиотечные компоненты.

Допустим, что кем-то реализован алгоритм. Описаны множество типов объектов ВП, с которыми реализация оперирует. Тогда, во-первых, данный алгоритм сможет обрабатывать введенные новые типы объектов ВП (при условии, что они наследованы от описанных в реализации алгоритма). А во-вторых, можно просто подключить свой обработчик сообщений для определенного типы вершины, не переписывая всю реализацию.

Дополнительные классы, типы данных, системные библиотеки

Основной целью при проектировании ВП ставилась его расширяемость. Поэтому ВП организовано в виде иерархии классов и библиотеки работы с ними. При разработке новых функциональных компонентов системы пользователь может либо по-своему интерпретировать зарезервированное поле, присутствующее в любом из классов, либо наследовать от уже существующих классов новые типы объектов.

Если говорить о языковой составляющей ВП, то базовым классом для всех внутренних объектов является класс UIFFEO_Object (унаследованный от базового класса OMA_Object), а для объектов операторной части — класс UIFFE_Object. Любой объект ВП содержит атрибуты, необходимые для визуализации программы. Для каждого объекта хранятся координаты его текстового прообраза — это номер строки и номер литеры в строке по текстовому файлу, для начальной и конечной литеры конструкции в тексте исходной программы. К атрибутам визуализации также можно отнести идентификатор исходной программы, вид объекта, цикла, безусловного перехода и информацию о переименование объектов в тексте исходной программы.

Помимо основных объектов в ВП введены системно зависимые объекты, так называемые директивы компилятора. Они могут быть установлены пользователем между запуском различных частей компилятора и указывать системе на вызов каких-либо библиотечных алгоритмов, функций и т.д. (в основном это объекты интерпретатора Сценария).

Помимо реализации типов объектов ВП в системной библиотеке реализованы различные структуры данных: массивы, неограниченные массивы, упорядоченные списки, древовидные структуры данных и т.д. Данные структуры и их реализации спроектированы с учетом поддержки механизма передачи сообщений, что может упростить реализацию новых функциональных компонентов.

Дополнительно, в системной библиотеке реализован класс OMA_Variant, инкапсулирующий в себя различные типы данных: integer, uint, float, double, char и т.д., а также указатель на базовый класс OMA_Object. Вариантный тип может быть использован, когда тип объекта явно не определен, а типизация происходит в период выполнения программы. Более детально это будет видно при описании Языка Сценариев.

Системная библиотека, которая автоматически подключается при запуске интерпретатора Сценариев, предоставляет также функции для работы с командной строкой, файловой системой, окнами и консолями, функции для

обработки ошибок, для работы с внешними библиотеками, хэш-таблицами и механизмы обработки текстовых файлов.

СЦЕНАРИЙ

Для создания своего собственного прототипа компилятора, пользователю необходимо определить СЦЕНАРИЙ его работы. Иными словами, пользователю необходимо задать набор программных библиотек, которыми он будет пользоваться, определить множество типов объектов внутреннего представления и определить множество и порядок исполнения функциональных компонентов (далее ПРОХОДОВ). Это может быть либо парсер, либо алгоритм оптимизации или преобразования дерева, либо компонент визуализации, например, Графа Программных Зависимостей. Ниже приведен возможный сценарий возможного прототипа компилятора “Конвертор с языка Паскаль в С”:

```
lib Pas=libPascal;libMy;Gr=libGraph;
pass Pas.Translate {
  UIFFE_Assign=libMy.OnTranslate_UIFFE_Assign;
};
pass Gr.BuildGraph;
pass Gr.VisCGraph;
$Module:=Pas.PASP_OpenFile(@1);
$Module:=PAS.PASP_TextParse($Module);
if ($Error ) {
        $OutputError(ErrMes);
}
else
{
          $Root:=Gr.Build_CGraph($Module);
          $Window_Title:="Control Graph";
          Gr.Out_Graph($Root);
          Run $Module(PAS.Translate); (***)
          IF ($Error) {
                  $OutputError($ErrorMes) ;
          }
}
end.
```

Далее разберем пример подробнее.

Декларация библиотек и сообщений

В данном примере можно увидеть основные объекты и операторы Языка Описания Сценариев (ЯОС). В начале сценария пользователю необходимо определить, какими программными библиотеками он будет пользоваться. Данные библиотеки он может взять из предложенных в системе либо написать свою, которую можно включить в сценарий. В нашем примере можно видеть, что используются три библиотеки: libPascal.dll (библиотека парсеров с языков Паскль и С), libMy.dll (дополнительная библиотека с измененным обработчиком сообщений) и libGraph.dll (библиотека работы с графовыми представлениями). Для задания используемых библиотек используется блок **lib**. Синонимы PAS и GR, заданные при объявлении библиотек, могут в дальнейшем использоваться пользователем для сокращения записи вызываемых функциональных и инструментальных библиотек. При подключении библиотеки интерпретатор автоматически вызывает процедуру *<Lib_Name>_Start* , в которой программист, создающий библиотеку, может инициализировать необходимые для работы библиотечных функций данные. Помимо этого, при завершении сценария интерпретатор вызывает для каждой загруженной библиотеки процедуру *<Lib_Name>_Finish*, где можно вставить вызовы деструкторов используемых объектов.

С помощью декларации **pass** пользователь объявляет используемые в дальнейшем проходы из библиотек. Проходы — это вид СООБЩЕНИЙ, описанный выше. Для проведения необходимых инициализирующих действий для проходов можно определить библиотечную функцию *<Pass_Name>_Init*, которая будет вызываться интерпретатором при объявлении прохода. Кроме этого, пользователь во время декларации библиотечного прохода может переопределить обработчик сообщений для конкретных типов объектов ВП системы. Так, в выше приведенном примере предлагается для сообщения Translate, определенного в библиотеке libPascal, имеющей синоним PAS, переопределить обработчик сообщений для объектов ВП типа UIFFE_Assign (присваивание) на другой, из пользовательской библиотеки libMy. Теперь, например, на выходе в С-коде вместо текста

```
A= b+c;
```

будем генерировать, к примеру, следующее:

```
A= b+ c; // Input data b & c; Output data A.
```

Тем самым, мы, не затрагивая всю реализацию С-кодогенератора и дерева разбора, поменяли лишь обработчик сообщений для вершин типа

UIFFE_Assign (присваивание). А все остальные обработчики остались библиотечными.

Здесь необходимо указать, что специальные библиотечные функции должны иметь определенную сигнатуру.

- Функция инициализации библиотеки — ***void <LibName>_Init ()***
- Функции деинициализации библиотек — ***void <LibName>_Finish ()***
- Функции инициализации СООБЩЕНИЙ — ***void <PASS_Name>_Init()***
- Обработчики СООБЩЕНИЙ —
  `void <Message_Handler>(OMA_Object *c, OMA_Message &msg)`

  Уже внутри реализации обработчика необходимо проверять конкретные типы входных параметров.
  Рассмотрим пример реализации переопределенного обработчика из нашего скрипта:

```
__declspec(dllexport)  void  OnTranslate_UIFFE_Assign(OMA_Object  *c,
OMA_Message &msg)
{
      if (msg.IsOf(Translate::Type()))
           OnTranslate((UIFFE_Assign*)c,(Translate&)msg);
      else
           c->Act(msg);
}
```

  Здесь OnTranslate — это статическая процедура с конкретными действиями обработчика. В начале происходит проверка типа сообщения msg. Если данный объект типа Translate — это сообщение-проход, отвечающее за кодогенерацию, то выполняем свои действия. Иначе объекту OMA_Object передаем сообщение msg. Конечно, здесь необходимо было бы дополнительно проверить тип объекта *c* на его принадлежность к объектам типа UIFFE_Assign, но в этом нет необходимости, если мы точно знаем, что данный обработчик определен только для объектов типа UIFFE_Assign.
- Функциональный и инструментальный компонент —

  `OMA_Variant <Func_Name>(OMA_Variant& in)`

  Как видно из сигнатуры, тип входного параметра и результата имеют вариантный тип. Данный подход позволяет создавать практически различные компоненты, не усложняя реализацию интерпретатора, но на-

кладывает определенные требования к документированию библиотек системы. Необходимо точно указать, какой тип передается на входе в вариантной переменной, и какой — на выходе. В нашем скрипте, например, компонент Gr.Build_CGraph (построение управляющего графа) на входе должен получить указатель на объект типа UIFFEO_Module (класс, инкапсулирующий в себе указатели на объекты транслируемого модуля и указатель на вершины дерева разбора). А на выходе получаем указатель на вершину управляющего графа типа, наследованного от класса Node_Object.

Другими объектами ЯОС являются переменные. Имена переменных начинаются со знака $. Все переменные имеют вариантный тип OMA_Variant. Нет необходимости явно объявлять переменную. Она создается в момент первого присваивания значения переменной. Переменные можно изменять не только в скрипте, но и внутри библиотечных компонентов с помощью библиотечных функций:

```
OMA_Variant Get_Engine_Variable(const char* name)
OMA_Variant Set_Engine_Variable(const char* name,
OMA_Variant& val)
```

Вторая функция возвращает старое значение переменной, если она существует, либо создает новую переменную с именем name и возвращает новое значение.

В нашем примере переменная $Window_Title используется компонентом Build_CGraph для указания имени GUI окна компонента. Если переменная не задана, то задается заголовок по умолчанию (надпись “Graph”).

Все переменные делятся на

- пользовательские, обычные переменные;
- системные переменные, существующие по умолчанию. Например, переменная $Error содержит код последней ошибки, а $ErrMes — текстовое представление;
- аргументы командной строки. Имеют вид @n, где n — номер аргумента начиная с единицы.

Операторная часть

Остальные конструкции ЯОС представляют операторную часть языка. Ниже описаны эти операторы.

Оператор вызова библиотечного компонента. В нашем примере это вызов процедур Pas.PASP_OpenFile, PASP_TextParse, Gr.Build_CGraph, Gr.Out_Graph. Первые два компонента отвечают за открытие текстового файла и разбор Паскаль-программы. Последние два используются для построения Управляющего графа программы и вывода его на экран. Все такие компоненты имеют одинаковую сигнатуру

```
OMA_Variant <Proc_Name>(OMA_Variant& in).
```

Видно, что для передачи и возврата данных используется вариантный тип, поэтому при описании создаваемого компонента пользователь должен будет явно указать: что скрывается за вариантными параметрами. Данное требование должно облегчить процесс создания новых функциональных компонентов. Например, известно, что компонент визуализации произвольного графа Gr.Out_Graph на входе должен получить указатель на вершину типа Node_Object. Соответственно, если мы хотим воспользоваться данным компонентом, то все вершины нашего графового представления должны наследоваться от типа Node_Object.

Функциональный компонент, как видно выше, может возвращать значение, которое может быть присвоено переменной сценария. Переменная сценария всегда имеет вариантный тип, и имя ее начинается со знака $. Переменные сценария служат для передачи данных между компонентами либо для вывода внутрисистемной информации.

С помощью **оператора IF_ELSE** можно определять поток управления в сценарии.

И последний оператор — это **оператор RUN**. Семантика данного оператора определяется как передача определенного сообщения объекту ВП. В нашем примере в строке (***) объекту $Module передается сообщение Translate, где переменная $Module содержит указатель на объект типа UIFFEO_Module, что можно рассматривать как корень языкового компонента ВП.

ЗАКЛЮЧЕНИЕ

Данная публикация дает краткое представление об основных компонентах системы и ее архитектурных особенностях. Более детальное описание с примерами можно найти на сайте разработчиков системы `http://pco.iis.nsk.su/progress`.

На текущем этапе работы идет создание программных библиотек для системы, их наполнение различными функциональными компонентами. Уже готовы такие компоненты системы, как парсеры с языка Паскаль и С, кодогенераторы в язык С, построение управляющего графа программы и его визуализации. Набор данных библиотек постоянно пополняется как авторами системы, так и некоторыми сторонними пользователями.

СПИСОК ЛИТЕРАТУРЫ

1. Маркин В.А., Маркина С.А. Проект системы для быстрого прототипирования распараллеливающего компилятора. Универсальное внутреннее представление системы // Современные проблемы конструирования программ. —Новосибирск, 2002.
2. Kasyanov V.N., Evstigneev V.A. The PROGRESS program manipulation system // Proc. of the Intern. Conf. “Parallel Computer Technologies”. — Obninsk, 1993. — Vol.3. — P. 651–656
3. Kasyanov V.N., Evstigneev V.A., Gorodniaia L. The system PROGRESS is a tool for parallelizing compiler prototyping // Proc. of 8-th SIAM Conf. on Parallel Processing for Scientific Computing (PPSC-97). — Minneapolis, 1997. — P. 301–306.