УДК 004

ИНСТРУМЕНТАЛЬНАЯ ПОДДЕРЖКА ПРОЦЕДУРНО-ПАРАМЕТРИЧЕСКОГО ЯЗЫКА ПРОГРАММИРОВАНИЯ

Косов П.В.
научный руководитель д-р техн. наук Легалов А.И.
Институт космических и информационных технологий Сибирского федерального университета

Введение

При разработке программного обеспечения необходимо учитывать множество факторов, среди которых одним из наиболее значимых является возможность эволюционного наращивания функциональных возможностей программы. Доминирующее в настоящее время объектно-ориентированное программирование (ООП) позволило повысить гибкость при разработке эволюционно расширяемых программ, однако использование только возможностей данной парадигмы зачастую оказывается недостаточным. Поэтому для безболезненного наращивания программ широко используются методы, базирующиеся на паттернах проектирования. Это ведет к написанию дополнительного кода, напрямую не связанного с решаемой задачей.

Для повышения эффективности при разработке эволюционно расширяемых программ была предложена процедурно-параметрическая парадигма программирования [1], обеспечивающая на основе процедурного подхода гибкое расширение процедур и данных. Возможности процедурно-параметрического полиморфизма могут быть встроены в ОО языки программирования, обеспечивая дополнительную гибкость языка. Для исследовательских целей на базе ЯП Оберон-2 был разработан язык O2M [2].

O2M привнес в Оберон-2 процедурно-параметрические конструкции и позволил детально рассмотреть принципы их построения. Однако данные ППП конструкции перекрыли возможности ООП элементов, имеющихся в Оберон-2. Исходя из этого, было решено создать новый язык программирования для исследования достоинств и недостатков предлагаемой парадигмы, сохранив семантику предшественника. Избавление от ОО конструкций и изменение ряда других элементов привело к созданию языка программирования Alien [3]. В работе рассматривается использование данного языка совместно с генератором кода в язык LLVM [4].

Особенности LLVM

Программы на языке процедурно-параметрического программирования Alien транслируются в байткод LLVM. LLVM — (Low Level Virtual Machine) это система, состоящая из инструментов, позволяющих анализировать, трансформировать, оптимизировать и отлаживать программы. Данная система используется для многих целей, из которых можно выделить следующее:

- Системный компилятор для платформ Apple и FreeBSD;
- Использование инструментов LLVM во многих реализациях GPGPU;
- Удобство использования для реализации современных компиляторов многих языков программирования, благодаря развитому и хорошо задокументированному API;
- Расширенный анализ исходного кода;

- Также инструменты LLVM используются при разработке спецэффектов в художественных фильмах, при разработке видеоигр (например Sony Playstation 4) и т.д.

В основе LLVM лежит представление любой программы на любом языке в одном и том же промежуточном представлении кода (intermediate representation, IR), над которым возможно производить различные операции во время компиляции, компоновки и выполнения. Из данного представления генерируется оптимизированный машинный код для целого ряда платформ (x86, x86-64, ARM, PowerPC, SPARC, MIPS и т.д.). Следует отметить, что для данного IR возможно не только статическое выполнение бинарного файла, но и динамическое выполнение (JIT-компиляция).

Особенности трансляции процедурно-параметрических программ в LLVM

Транслятор языка Alien разработан с учетом необходимости разделения части, отвечающей за построение семантической модели программы, от части, отвечающей за кодогенерацию. В настоящее время разрабатывается два кодогенератора – в C++ и в байткод LLVM, оба используют одну семантическую модель. Транслятор разработан на C++ с учетом требований кроссплатформенности (может выполняться как на Windows, так и на Linux).

В качестве аргумента транслятору передается путь к *.prj файлу, который содержит список модулей для компиляции с текстом программ на ЯП Alien.

Компилятор состоит из следующих основных программных объектов:

- класс Scanner используется для лексического анализа программы;
- класс Module – содержит описание переменных, новых типов, выполняемых инструкций и т.д. в каждом модуле;
- Класс Project – содержит настройки проекта (имя и путь к файлу проекта, директории файлов описания и т.д.);
- Класс Parser – выполняет синтаксический анализ, также содержит экземпляр класса Scanner и класса Module (для выполнения лексического анализа и формирования описания модуля). Данный класс содержит метод запуска процесса трансляции – Parser::Do(Project*, Module*).
- Класс Generator – абстрактный класс, являющийся интерфейсом для всех генераторов кода.

В функции main() после проверки валидности передаваемых аргументов происходит создание и инициализация проекта, затем компиляция всех модулей из списка в переданном *.prj – файле, в функции CompileModule(…). В этой же функции, при завершении выполнения метода Parser::Do(…) происходит вызов генераторов кода, использующих уже сформированное абстрактную синтаксическую модель программы.

Спецификой ППП является возможность гибкого изменения состава параметрических обобщений – типов данных, объединяющих в единую категорию множество альтернативных понятий [5]. Обобщения можно создавать как на основе уже существующих понятий, так и использовать эволюционное расширение, добавляя новые структуры в уже существующее обобщение. Все включаемые в обобщение типы характеризуются признаком (индексом), который контролируется с помощью дополнительной инструментальной поддержки.

Обобщенные записи формируются следующим образом [6]:

```
ТипОбобщение = CASE [TYPE] [ OF [LOCAL] [СписокСпециализаций] ]
[ ELSE Специализация ] END .
СписокСпециализаций =
((СписокПризнаков ":" (Тип | NIL) ) | Тип)
{ "|" ((СписокПризнаков ":" (Тип | NIL) ) | Тип) } .
СписокПризнаков = идент [ "," идент ] .
Специализация = (идент ":" (Тип | NIL) ) | Тип.
```

Пример применения обобщенных записей:

```
(* 1. Прямоугольник со сторонами x и y *)
Rectangle = RECORD x, y: INTEGER END
(* 2. Треугольник со сторонами a, b и c *)
Triangle = RECORD a, b, c: INTEGER END
(* 3. Обобщение фигуры *)
Figure = CASE OF trian: Triangle | rect: Rectangle END
(* 4. Круг радиуса r *)
Circle = RECORD r: INTEGER END
(* 5. Расширение обобщения *)
Figure += circ: Circle
```

Кодогенерация в байткод LLVM данной структуры происходит следующим образом: для строк №1 и №2 в байткоде LLVM формируются обычные структуры:

```
%struct.Rectangle = type { i32, i32 }
%struct.Triangle = type { i32, i32, i32 }
```

для строки №3 формируется код LLVM, аналогичный следующей структуре на языке программирования C++:

```
struct Figure {int rank; void *t;};
%struct.Figure = type { i32, i8* }
```

где void *t – указатель на подключаемую специализацию, int rank – признак специализации. Также при создании данного обобщения формируется связь между признаком и типом. Строка №5 создает еще одну связь между признаком и типом.

```
(* 6. Объявление переменной *)
VAR c(circ) : Figure;
```

При разборе данной строки переменной «c» присваивается тип Figure с признаком, соответствующем обобщенному типу Circle, и с указателем также на структуру типа Circle.

Данная строка отобразится в промежуточном представлении LLVM следующим образом:

1) Выделение памяти под переменную «c» (тип Figure).

```
%c = alloca %struct.Figure, align 8
```

2) Затем выделяется память под специализацию (тип Circle).

```
%tc = alloca %struct.Circle, align 4
```

3) Определение адреса поля обобщения со значением ранга

```
%2 = getelementptr inbounds %struct.Figure* %c, i32 0, i32 0
```

4) По данному адресу помещается значение, соответствующее типу Circle (в данном случае «3»)

```
store i32 3, i32* %2, align 4
```

5) Изменение внутреннего представления типа специализации на void*

```
%3 = bitcast %struct.Circle* %tc to i8*
```

6) Определение адреса указателя на специализацию (поле «void*» переменной «c»)

```
%4 = getelementptr inbounds %struct.Figure* %c, i32 0, i32
```

7) Сохранение указателя на специализацию по выше полученному адресу.

```
store i8* %3, i8** %4, align 8
```

Далее рассмотрим обращение к полю специализации

```
c.r = 5;
```

При реализации данной конструкции, происходит обращение к полю «r» указателя «t». Данное обращение состоит из нескольких этапов:

1) Получение адреса поля типа «void*» переменной «c», хранящего адрес специализации

```
%5 = getelementptr inbounds %struct.Figure* %c, i32 0, i32 1
```

2) Загрузка в регистр данных с этого адреса

```
%6 = load i8** %5, align 8
```

3) Изменение внутреннего представления типа данных на тип специализации (предварительно этот тип был определен компилятором с помощью поля обобщения, содержащего значение ранга специализации). Данная команда — единственное отличие от случая обращения к полю обычной структуры, не находящейся в обобщении

```
%7 = bitcast i8* %6 to %struct.Circle*
```

4) Получение адреса поля «r» переменной «c»

```
%8 = getelementptr inbounds %struct.Circle* %7, i32 0, i32 0
```

5) Сохранение по адресу поля «r» переменной «c» значение (в данном случае - 5)

```
store i32 5, i32* %8, align 4
```

В заключение следует отметить, что, благодаря поддержке со стороны компилятора, расширение типов данных в процедурно-параметрической парадигме программирования осуществляется без изменения ранее написанного кода и без увеличения сложности исполняемой программы.

Список использованных источников

1. Легалов А.И. Процедурно-параметрическая парадигма программирования. Возможна ли альтернатива объектно-ориентированному стилю? - Красноярск: 2000. Деп. рук. № 622-В00 Деп. в ВИНИТИ 13.03.2000. - 43 с.
2. Легалов А.И. Швец Д.А. Процедурный язык с поддержкой эволюционного проектирования. – Научный вестник НГТУ, № 2 (15), 2003. С. 25-38.
3. Легалов А.И., Бовкун А.Я., Легалов И.А. Расширение модульной структуры программы за счет подключаемых модулей. / Доклады АН ВШ РФ, № 1 (14). – 2010. – С. 114-125.
4. Lattner C. LLVM and Clang. Advancing compilers and tools. Proceedings of the 9th Central and Eastern European Software Engineering Conference 2013 (материал размещен по адресу: http://2013.secr.ru/2013/files/047_lattner.pdf).
5. Легалов И. А. Применение обобщенных записей в процедурно-параметрическом языке программирования. / И.А. Легалов // Научный вестник НГТУ. – 2007. – № 3 (28). – С. 25-38.
6. Legalov A., Kosov P. Evolutionary software development using procedural-parametric

programming / Proceedings of the 9th Central and Eastern European Software Engineering Conference 2013 // ACM, New York, 2013.