УДК 004.432

РЕАЛИЗАЦИЯ МЕТАПРОГРАММНОГО СЛОЯ ДЛЯ ОБЪЕКТНО-ОРИЕНТИРОВАННОЙ МОДЕЛИ ЯЗЫКА ЛИСП

METAPROGRAMMING LAYER IMPLEMENTATION FOR AN OBJECT-ORIENTED MODEL OF THE LISP LANGUAGE

Галкина Е.В. / Galkina E.V.

Московский государственный университет им. М.В.Ломоносова, факультет вычислительной математики и кибернетики, Москва, Ленинские Горы 1, стр. 52, 119234
Lomonosov Moscow State University, Faculty of Computational Mathematics and Cybernetics, Moscow, Leninskie Gory 1, str. 52, 119234

Аннотация. В работе рассматривается реализация метапрограммного слоя в рамках вычислительной модели диалекта языка Лисп, представленного в библиотеке InteLib. В результате работы в диалект была добавлена возможность определять макросы и осуществлять их вызов с дальнейшей макроподстановкой и вычислением результирующего выражения.

Ключевые слова: мультипарадигмальное программирование, языки программирования, функциональное программирование, объектно-ориентированное программирование, метапрограммирование

Вступление.

В библиотеке InteLib [1, 2] реализован метод непосредственной интеграции [3], в основе которого лежит моделирование средствами базового языка синтаксиса и семантики альтернативного языка. В качестве базового языка используется язык C++. На данный момент в рамках проекта InteLib реализованы библиотеки классов, в том числе и моделирующие изобразительный стиль языка Лисп [4]. Основной структурой данных в языке Лисп и его диалектах являются S-выражения [5]. S-выражение представляет собой либо атом, либо точечную пару, составленную из S-выражений.

Библиотека InteLib предоставляет набор классов языка C++, реализующих S-выражения как гетерогенные структуры данных. Перегрузка операторов в языке C++ даёт возможность конструировать объекты S-выражений, используя представление, синтаксически схожее со списочным представлением в Лисп-подобных языках. Например, списку (2 4 6) соответствует конструкция языка C++ (L| 2, 4, 6).

Основой реализации вычислительной модели являются стек действий, стек результатов и стек фреймов. Работа первых двух из них описана в [3]. Стек фреймов используется для оптимизации передачи фактических параметров при вызове функции и автоматической очистки после вычисления результата.

В предыдущих версиях библиотеки InteLib лексическое связывание реализовывалось с помощью контекстов — однонаправленных списков соответствий между символом и его текущим значением. Основным недостатком такого решения являлась низкая скорость поиска нужного значения в контексте, поэтому в последней версии библиотеки механизм лексического связывания был полностью заменён на новый, основанный на использовании позиционных параметров [6]. В данной статье предложен способ реализации метапрограммного слоя (макросы) для диалекта InteLib Lisp с использованием этого механизма.

Основной текст.

Реализация лексического замыкания в рассматриваемой вычислительной модели языка Лисп, представленной в библиотеке InteLib, основывается на использовании позиционных параметров, позволяющих получить прямой доступ к значению, переданному в качестве аргумента функции или спецформы в стеке. Замена происходит при создании лексического замыкания. На позиционные параметры заменяются символы, находящиеся в списке формальных параметров.

Для представления позиционных параметров используется отдельный класс (IntelibTokenReplace), инкапсулирующий позицию в стеке, на которой будет находиться соответствующее значение после вычисления фактического

параметра. Позиционный параметр является разновидностью специального атома, который имеет своё правило вычисления. Результатом вычисления позиционного параметра является значение, содержащееся в соответствующем стековом фрейме.

Методы для замены символов на позиционные параметры предоставляются специально разработанным классом. Контекст, в котором происходит замена реализован в виде векторов соответствий между символом и его номером в стековом фрейме. Векторы разбиты на уровни, которые учитываются при обработке вложенных форм. Эти уровни определяют, в каком именно стековом фрейме будет находиться значение фактического параметра. При заполнении вектора соответствий каждому символу в качестве позиции приписывается его номер в списке аргументов или списке инициализации и номер текущего уровня. Также класс инкапсулирует список, содержащий символы, которые должны игнорироваться в процессе замены (например, параметры вложенных λ-выражений).

Атомарный объект обрабатывается по-разному в зависимости от его типа. Позиционные параметры заменяются на объекты класса, сопоставляющие символ его значению. Символы, содержащиеся в векторе соответствий на каком-либо уровне, заменяются на позиционные параметры, остальные не изменяются. Если выражение представляет собой вызов некоторой спецформы, то для дальнейшей обработки вызывается метод, отдельно определённый для каждой из них и зависящий от семантики формы.

Лексическое замыкание представляется классом SExpressionCompiledLambda. Полями класса, представляющего λ-выражение, являются количество обязательных параметров λ-функции, флаг, отвечающий за наличие дополнительных параметров и тело λ-функции с позиционными параметрами.

Создание объекта этого класса начинается с обработки списка параметров λ-выражения. Символы из этого списка нумеруются, в результате чего создаётся вектор соответствий символа и позиции, который сохраняется для

дальнейшего использования при обработке тела выражения. Символ, отвечающий за дополнительные аргументы, если их количество не фиксировано, обрабатывается таким же образом.

Класс SExpressionCompiledLambda наследуется от класса SExpressionFunction, т. е. лексическое замыкание является функциональным объектом, и в его классе определён метод, реализующий применение этого объекта к параметрам. Работа этого метода зависит от наличия необязательных аргументов. Во время его вызова количество фактических параметров известно, а их значения вычислены и находятся в стеке результатов, то есть текущий стековый фрейм сформирован. В случае, если количество параметров не фиксировано, формируется список дополнительных параметров, который добавляется в текущий стековый фрейм как единый аргумент. Затем в стек действий заносится команда для вычисления тела λ-выражения.

Более подробное описание реализации механизма лексического связывания, а также его использование при определении функций и вызове LET-форм для локального связывания переменных представлены в статье [6].

В диалектах языка Лисп как данные, так и код программы, представляются в виде S-выражений. Это свойство позволяет формировать вычисляемое выражение также с помощью самой программы. Наиболее естественным средством для программного формирования выражений (метапрограммирования) являются макросы. С их помощью можно определить формирование и вычисление произвольного выражения или целой программы.

Спецформа DEFUN предназначена для определения функций, но перед вызовом пользовательских функций всегда происходит вычисление аргументов. В диалектах Лиспа присутствуют встроенные спецформы, аргументы которых не вычисляются перед их вызовом (например, формы AND и OR). Определить пользовательские спецформы с помощью DEFUN невозможно, но при необходимости их можно определить в виде макросов. Кроме того, в реализациях диалектов языка Лисп встроенные спецформы также могут быть определены как макросы.

Для определения пользовательского макроса используется форма DEFMACRO. Она принимает те же аргументы, что и форма DEFUN: имя макроса, список формальных параметров и тело макроса:

(DEFMACRO <name> (<arg1> … <argn>) <body>)

Вызов макроса совпадает с вызовом функции, но его вычисление отличается от вычисления вызова обычной функции.

Во-первых, при вызове макроса не происходит предварительного вычисления аргументов. Они подставляются в тело макроса в том виде, в котором были переданы при вызове.

Во-вторых, тело макроса вычисляется в два этапа. Первый этап называется расширением или раскрытием макроса (macro expansion). На этом этапе осуществляется вычисление тела макроса с фактическими параметрами. Полученная форма обычно имеет более сложную структуру, чем исходная форма вызова. На втором этапе производится вычисление полученной при расширении формы. Её значение возвращается в качестве результата всего макровызова.

Третьим отличием является контекст вычисления макроса и функции. При расширении макроса действует контекст его определения, а вычисление полученной формы происходит в контексте выражения, содержащего макровызов, поэтому связи из контекста определения макроса уже не действуют.

Определение макроса, как и функции, может быть рекурсивным. В таком случае после каждого макрорасширения могут получиться выражения, также содержащие макровызов, который будет обработан аналогичным образом.

Проиллюстрируем разницу между вычислением функции и макроса на примере обмена значений двух символов. Реализуем сначала функцию SWAP-FUNC:

```
(DEFUN SWAP-FUNC (X Y)
        (LET ((TMP X))
                        (SETQ X Y)
                        (SETQ Y TMP)))
```

Пусть изначально значения символов A и B были равны 0 и 1 соответственно. Попробуем поменять их значения с помощью функции:

```
>>> (SWAP-FUNC A B)
0
>>> A
0
>>> B
1
```

Значения исходных переменных не изменились, поскольку вычисление происходило в контексте определения функции, откуда нет доступа к самим символам A и B. Аргументы функции были сначала вычислены, поэтому в качестве фактических параметров вызова выступали значения 0 и 1.

Для решения этих проблем реализуем обмен значений в виде макроса:

```
(DEFMACRO SWAP-MACRO (X Y)
        (LIST 'LET (LIST (LIST 'TMP X))
                        (LIST 'SETQ X Y)
                        (LIST 'SETQ Y 'TMP)))
```

В результате расширения макровызова (SWAP-MACRO A B) будет получено выражение:

```
(LET ((TMP A))
        (SETQ A B)
                (SETQ Y TMP))
```

Это выражение будет вычислено в контексте макровызова, в результате чего значения символов A и B действительно поменяются местами.

Таким образом, макрос в языке Лисп представляет собой функциональный объект, который может быть применён к аргументам, которые не вычисляются перед вызовом. Для поддержки метапрограммирования в диалекте InteLib Lisp

было необходимо представить макрос в виде такого объекта, а также реализовать механизм макрорасширения.

Для представления объектов, обладающих описанными выше свойствами, в библиотеке используется абстрактный класс SExpressionForm. В классе присутствует виртуальный метод Call, который вызывается при применении формы к аргументам. Он определяется наследниками класса в соответствии с семантикой отдельных спецформ. Метод ReplaceTokens используется для обработки спецформы во время процесса замены символов на позиционные параметры.

В качестве реализации макроса как функционального объекта был добавлен класс LExpressionCompiledMacro, являющийся наследником класса SExpressionForm. Он, в свою очередь, инкапсулирует объект класса SExpressionCompiledLambda. Создание объекта класса LExpressionCompiledMacro происходит при вызове спецформы DEFMACRO. Её аргументы обрабатываются так же, как и при вызове формы DEFUN [6]. На основе объекта класса SExpressionCompiledLambda, полученного в результате обработки, создаётся объект класса LExpressionCompiledMacro. Такой подход обусловлен тем, что определение макроса аналогично определению обычной функции, а расширение макроса можно представить как вызов функции с невычисленными параметрами.

Далее рассмотрим реализацию вызова макроса. Первый этап представляет собой макрорасширение: подстановку аргументов в исходном виде в тело макроса и его вычисление. Для этого сначала вычисляется количество фактических параметров вызова, и в стек действий помещается команда для вызова функции с таким числом аргументов. Затем в стек помещаются команды для квотирования параметров и тела макроса (quote_parameter). Они нужны для того, чтобы подготовить стеки действий и результатов к вызову тела макроса. Описание команд вычислителя приведено в работе [3]. После этого необходимо произвести само макрорасширение, а именно выполнить несколько шагов вычисления, пока в стеке результатов не окажется готовое к дальнейшему

вычислению выражение. Для этого перед заполнением стека действий сохраняется актуальная на тот момент позиция, а после заполнения стека в цикле вызывается метод, производящий один шаг вычислений, пока не будет достигнута сохранённая ранее позиция. После этого на вершине стека результатов будет находиться результат первого этапа вычисления макроса.

Второй этап вызова макроса состоит в следующем: полученная форма извлекается из стека и вычисляется заново. Если при этом был встречен рекурсивный вызов макроса, то макрорасширение выполняется заново описанным выше способом. После окончания всех вычислений в верхней позиции стека результатов будет находится выражение, являющееся результатом исходного макровызова.

Таким образом, предложенная реализация сохраняет семантику вычисления макросов языка Лисп.

Заключение и выводы.

Была предложена реализация метапрограммного слоя (определение и вычисление макросов) в объектно-ориентированной вычислительной модели языка Лисп, основанная на более эффективном механизме лексического связывания. В диалект InteLib Lisp добавлена спецформа, позволяющая определять пользовательские макросы и применять их в дальнейшем в соответствии с семантикой языка.

Литература:

1. И. Г. Головин, А. В. Столяров. Объектно-ориентированный подход к мультипарадигмальному программированию // Вестник МГУ, сер. 15 (ВМиК), 2002. – No 1. – С. 46-50.

2. Официальный сайт проекта InteLib [Электронный ресурс]. URL: http://www.intelib.org (дата обращения 01.11.2017).

3. А. В. Столяров. Импорт вычислительной модели языка Scheme в объектно-ориентированное окружение // Сборник статей молодых учёных факультета ВМК МГУ, 2008. – Вып. 5. – С. 119-130.

4. G. L. Steele. Common Lisp the Language, 2nd edition. – Digital Press, 1990. – 1029 p.

5. J. McCarthy. Recursive Functions of Symbolic Expressions and Their Computation by Machine // Communications of the ACM, 1960. – No 3(4). – P. 184-195.

6. Е. В. Галкина. Об одной реализации лексических замыканий языка Лисп // Сборник статей молодых учёных факультета ВМК МГУ, 2016. – Вып. 13. – С. 17-29.

Abstract

The article describes one possible implementation of metaprogramming layer for an object-oriented model of Lisp dialect presented in the InteLib library. The DEFMACRO special form has been added to the dialect, allowing user to create their own macros. Macros are represented as class objects that incapsulate lexical closure and provide a way of calling it with given parameters. The implementation of macro call is based on using positional parameters when creating lexical closure and consists of two sequential steps: macro expansion and evaluation of the resulting form.

Key words: multi-paradigm programming, programming languages, functional programming, object-oriented programming, metaprogramming

Статья отправлена: 04.11.2017 г.