УДК 004.432

Об одной реализации лексических замыканий языка Лисп

© 2016 г. Е. В. Галкина

catherine.galkina@gmail.com

Кафедра алгоритмических языков

1. Введение

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++ [4]. На данный момент реализованы библиотеки классов, моделирующие изобразительный стиль языков Lisp [5], Scheme [6], Refal [7] и Prolog [8]. Следует уточнить, что речь идёт лишь о подмножествах данных языков, включающих в себя их основные особенности, поскольку предполагается, что они будут использоваться в качестве второстепенных языков программирования для решения определённых подзадач.

Основной структурой данных в языке Lisp и его диалектах являются S-выражения [9]. 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 [3]. Общие свойства, характерные для всех S-выражений, описаны в базовом абстрактном классе. Атомарные S-выражения разделены на основные типы (например, целые числа, числа с плавающей точкой, строки и т. д.) и представлены классами-наследниками.

Точечные пары также представлены классом-наследником S-выражения. В соответствии с определением, поля класса являются ссылками на составляющие подвыражения.

Для удобства использования S-выражений введён класс «умного указателя» [10] на объект класса S-выражения, обладающего дополнительной функциональностью. Подробное описание его методов и дополнительных структур данных для работы с S-выражениями приведено в [11].

Перегрузка операторов в языке C++ даёт возможность конструировать объекты S-выражений, используя представление, синтаксически схожее со списочным представлением в Lisp-подобных языках. Например, списку `(2 4 6)` соответствует конструкция `(L| 2, 4, 6)`, а точечная пара `("ab" . "cd")` представляется в виде `(L| "ab" || "cd")`.

Реализация вычислений S-выражений основана на работе с континуацией. Континуацию можно определить как последовательность действий, которые необходимо выполнить до завершения вычислений. Каждое элементарное вычисление рассматривается как переход от одной континуации к другой.

В библиотеке InteLib континуация представляет собой виртуальную вычислительную машину, её реализация вынесена в отдельный класс. Методами этого класса реализуется основной механизм вычислений S-выражений. Особенности диалектов InteLib Lisp и InteLib Scheme (см. [3], стр. 128–129) учитываются в классах-наследниках.

Основой реализации континуации 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. Его элементы представляют собой структуры, хранящие позицию результата вычисления функции, количество переданных параметров и текущую позицию в стеке действий на момент вызова.

В предыдущих версиях библиотеки лексическое связывание реализовывалось с помощью контекстов — однонаправленных списков соответствий между символом и его текущим значением. Для вычисления значения символа производился линей-

ный поиск по этому списку. При применении функционального объекта или вычислении LET-форм и других форм с предварительной инициализацией создавался временный контекст, который использовался во время вычисления тела формы.

Новый механизм лексического связывания основан на использовании так называемых позиционных параметров — специальных символов, позволяющих получить прямой доступ к нужному значению параметра в стеке.

Замена происходит при создании лексического замыкания, а именно при определении функций, использовании λ-выражений и LET-форм. На позиционные параметры заменяются символы, находящиеся в списке формальных параметров функции или в списке инициализации LET-формы.

Для реализации указанной замены символов используется специально разработанный класс, инкапсулирующий контекст замены и предоставляющий методы для обработки различных видов спецформ, списочных структур и отдельных символов.

2. Основы реализации используемых классов

Для представления позиционных параметров используется отдельный класс, 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 (все остальные атомарные S-выражения вычисляются сами в себя). Результатом вычисления позиционного параметра является значение, содержащееся в соответствующем стековом фрейме.

Методы для замены символов на позиционные параметры при создании замыкания и обработке LET-форм 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 стеко-

вом 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).

Основным методом класса является метод, обрабатывающий произвольное S-выражение. 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. Также замена не происходит для всех динамически связываемых символов и символов, которым уже было сопоставлено некоторое вычисленное значение.

Если выражение представляет собой вызов некоторой спецформы, то для дальнейшей обработки вызывается метод, отдельно определённый для каждой из них и зависящий от семантики формы. Поскольку для большинства спецформ процесс замены символов сводится к рекурсивному обходу списочных структур, были добавлены методы для рекурсивной обработки списков.

Также в классе присутствует метод для обработки LET-форм, работа которого будет подробнее описана при рассмотрении этого механизма.

3. Создание лексического замыкания

В диалекте InteLib Lisp лексическое замыкание создаётся либо при определении пользовательской функции с помощью спецформы `DEFUN`, либо при применении к λ-выражению формы `FUNCTION` (или сокращённо `#'`), реализующей функциональную блокировку. Заметим, что сами λ-выражения не являются функциональными объектами и рассматриваются как обычные списки, то есть вычисление на верхнем уровне выражения ви-

да `(LAMBDA (X) (* X 10))` не приведёт к созданию замыкания (но в диалекте InteLib Scheme такой синтаксис используется). Пользовательские функции в InteLib Scheme определяются с помощью формы `DEFINE`. Таким образом, лексическое замыкание создаётся в одном из четырёх случаев (`<args>` — аргументы, `<body>` — тело функции):

- определение функции в реализации InteLib Lisp: `(DEFUN F (<args>) <body>)`;
- использование функциональной блокировки в реализации InteLib Lisp: `#'(LAMBDA (<args>) <body>)`;
- определение функции в реализации InteLib Scheme: `(DEFINE (F <args>) <body>)`;
- вызов формы `LAMBDA` в реализации InteLib Scheme: `(LAMBDA (<args>) <body>)`.

Лексическое замыкание представляется отдельным классом, экземпляр которого создаётся при вычислении указанных выше форм с использованием одинаковых механизмов в обоих диалектах.

Полями класса, представляющего λ-выражение, являются количество обязательных параметров λ-функции, флаг, отвечающий за наличие дополнительных параметров, тело λ-функции с позиционными параметрами и указатель на объект, отвечающий за работу с динамическими переменными (см. § 5). 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. Если замыкание создавалось при определении пользовательской функции, то этот объект устанавливается в качестве функционального значения символа, являющегося именем новой функции.

Лексическое замыкание является функциональным объек-

том, и в его классе определён метод, реализующий применение этого объекта к параметрам. Работа этого метода зависит от наличия необязательных аргументов. 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.

Рассмотрим обработку вложенных λ-форм. Возможна ситуация, когда в списке параметров внешней и внутренней формы присутствуют одинаковые символы:

```
#'(LAMBDA (X Y) #'(LAMBDA (X) (LIST X Y)))
```

На основе списка аргументов внешнего λ-выражения формируется вектор, сопоставляющий символам `X` и `Y` позиции 1 и 2. Эти символы должны заменяться на соответствующие позиционные параметры в теле выражения. Если производить замену так же, как в общем случае, то в выражении `(LIST X Y)` символ `X` будет рассматриваться как аргумент внешней формы, что приведёт к неверному вычислению. Поэтому во время процесса замены при наличии вложенной λ-формы её аргументы добавляются в отдельный список, содержащий символы, которые должны игнорироваться. Они будут обработаны непосредственно при вычислении вложенного выражения, т. е. после применения внешнего выражения к аргументам.

4. Обработка LET-форм

LET-формы предназначены для создания локальных связей переменных внутри других форм и на самом деле представляют собой синтаксически видоизменённые λ-выражения [12].

Переменные формы `LET` связываются с новыми значениями одновременно, в то время как список инициализации формы `LET*` обрабатывается последовательно. Разницу в их вычислении покажем на примере:

```
>>> (LET ((X 1) (Y (* X 2))) (LIST X Y))
Error: symbol has no value: X
>>> (LET* ((X 1) (Y (* X 2))) (LIST X Y))
(1 2)
```

Как и в случае лексического замыкания, LET-формы обрабатываются одинаково в обоих диалектах. Обработанная LET-форма представляет собой объект специального класса, который затем размещается в стеке для вычисления.

Создание этого объекта начинается с добавления нового уровня для вектора соответствий и прохода по списку инициализации. Отличие формы `LET` от `LET*` состоит в том, что при обработке выражения из списка инициализации предыдущие символы из этого списка не учитываются. Для этого была добавлена возможность создавать отдельный «скрытый» уровень. Такой уровень создаётся при обработке списка инициализации формы `LET`, а перед началом обработки тела формы он становится текущим уровнем. Для формы `LET*` текущий уровень устанавливается сразу.

Поскольку в качестве нового значения может использоваться произвольное S-выражение, сперва обрабатывается оно (в нём некоторые символы могут быть заменены на позиционные параметры в случае вложенного вызова или использования формы `LET*`). Затем связываемый символ добавляется в вектор соответствий, и ему приписывается очередная позиция. После того, как все пары в списке инициализации были обработаны, т. е. вектор соответствий был целиком сформирован, в теле формы происходит замена символов на позиционные параметры.

В результате этого процесса получается набор объектов, в которых присутствуют позиционные параметры: формы инициализации и тело LET-формы. Из них создаётся новый объект, который является разновидностью специального атома. Также параметрами конструктора являются флаг, отличающий формы `LET` и `LET*` и указатель на объект класса для ра-

боты с динамическими переменными (см. § 5).

Общая схема вычисления этого атомарного объекта состоит в следующем: создаётся новый стековый фрейм, в стек действий помещается команда для вычисления тела формы, а затем — команды для вычисления выражений из списка инициализации. Эти выражения вычисляются последовательно, и полученные значения помещаются в стек результатов. Перед вычислением тела LET-формы из этих значений должен быть сформирован стековый фрейм, но при его формировании существенную роль играет разница между формами `LET` и `LET*`.

При вычислении формы `LET` стек действий заполняется таким образом, чтобы сначала вычислялись все инициализирующие выражения, и только после этого размер стекового фрейма изменялся в соответствии с их количеством (для изменения размера текущего фрейма введена специальная команда).

Описанный подход неприменим для вычисления формы `LET*`: проблема состоит в том, что символы связываются с новыми значениями последовательно, и уже вычисленные значения могут использоваться во всех последующих выражениях. Таким образом, необходимо обеспечить их наличие в текущем стековом фрейме на нужной позиции. Для этого размер фрейма увеличивается после вычисления каждого из них, что делает возможным вычисление позиционного параметра, поскольку на соответствующей позиции находится нужное значение. После получения значения последней по счёту формы инициализации размер текущего фрейма становится равным количеству символов в списке инициализации, и вычисление тела происходит так же, как и для формы `LET`.

5. Динамическое связывание символов

В диалекте Intelib Lisp переменные по умолчанию считаются статическими, однако есть возможность использования динамических переменных.

Для их объявления предназначена форма `DEFVAR`:

(DEFVAR *name value*)

Параметр *name* — имя переменной (символ), *value* — начальное значение.

Значение переменной, объявленной таким образом, определяется во время вычисления (динамически), в отличие от статических переменных, значение которых зависит от контекста места её определения. Вычисления будут происходить поразному для статических и динамических переменных, в случае если переменная свободна в вычисляемом выражении, но при этом является формальным параметром в более внешней форме [12]. Приведём пример:

```
>>> (DEFVAR *X* 100) ; динамическая переменная
*X*
>>> (SETQ Y 0)       ; статическая переменная
Y
>>> (DEFUN F (*X* Y) (G))
F
>>> (DEFUN G () (LIST *X* Y))
G
>>> (G)
(100 0)
>>> (F 1 1)
(1 0)
>>> *X*          ; значение динамической переменной
100              ; восстановлено
```

Для корректной работы в рамках нового механизма связывания динамические переменные должны обрабатываться особым образом. Во-первых, при создании замыкания и при обработке форм LET и LET* динамически связанные символы, встречающиеся в списке формальных параметров или списке инициализации, не должны заменяться на позиционные параметры. Во-вторых, при связывании динамического символа с новым

значением необходимо запланировать восстановление старого значения.

В рассматриваемой реализации для работы с динамическими переменными введён специальный класс. Методы этого класса предоставляют возможность запланировать восстановление исходных значений динамических переменных и установить их новые значения на время вычисления тела LET-формы или λ-выражения. Экземпляр этого класса создаётся при обработке LET-форм и при создании замыкания в рамках модели вычисления выражений на InteLib Lisp. Список формальных параметров или список инициализации просматривается на наличие динамических символов, и эти символы добавляются в вектор, который сохраняется в объекте данного класса. Позиция символа в векторе соответствует его позиции в списке формальных параметров или списке инициализации. В метод, обрабатывающий атомарные объекты в процессе замены на позиционные параметры, была добавлена проверка на тип связывания символа.

Таким образом, динамически связанные символы не заменяются на позиционные параметры, а сохраняются в объекте специального класса. Указатель на него передаётся в конструктор объекта, соответствующего обработанной LET-форме или лексическому замыканию, и обращение к нему происходит при их вычислении. Если вычисление производится в рамках диалекта Intelib Scheme, такой объект не создаётся, и указатель всегда остаётся нулевым.

В момент начала этих вычислений с сохранёнными динамическими символами связаны именно те значения, которые будет необходимо восстановить. Поэтому перед добавлением команды для вычисления тела замыкания или LET-формы в стек действий помещаются команды для связывания сохранённых динамических переменных с их исходными значениями.

Далее необходимо заполнить стек таким образом, чтобы перед вычислением тела динамические переменные связывались

с новыми значениями. В качестве этого значения должен устанавливаться элемент текущего фрейма с номером, равным позиции символа в векторе, т. е. именно то значение, которое было передано в виде фактического параметра или было использовано в списке инициализации.

Сделать это напрямую при вычислении обработанной LET-формы невозможно, поскольку формирование фрейма, содержащего новые значения, ещё не закончено. Для решения этой проблемы была введена новая команда континуации, аргументом которой является указатель на объект используемого класса для работы с динамическими переменными. Она помещается в стек действий сразу после команды для вычисления тела LET-формы и перед планированием вычисления форм инициализации.

На момент выполнения этой инструкции формы инициализации уже вычислены, и полученные значения находятся в текущем стековом фрейме. Позиции динамических переменных можно получить из сохранённого объекта, т. е. доступна вся необходимая информация для установления новых значений динамических переменных. Для этого осуществляется проход по вектору, в котором сохранены символы. Позиция каждого символа в векторе совпадает с позицией в текущем стековом фрейме значения, с которым он должен связываться. Оно устанавливается в качестве нового значения символа и будет использоваться при вычисления этого символа в теле LET-формы. При применении лексического замыкания к аргументам используется такой же механизм.

Таким образом, к моменту вычисления тела замыкания или LET-формы динамические переменные уже имеют новые значения, а сразу после вычисления происходит восстановление значений, с которыми они были связаны в объемлющем блоке.

Заметим, что в диалекте InteLib Scheme динамически связываемые символы отсутствуют, и введённые механизмы не влияют на работу вычислительной модели языка Scheme.

6. Заключение

Целью данной работы являлась реализация лексического замыкания в рамках вычислительной модели Lisp-подобных языков, представленной в библиотеке InteLib.

В ходе работы в библиотеку были добавлены классы и методы для обработки и вычисления λ-выражений и LET-форм, а также разработан механизм динамического связывания переменных в рамках нового подхода.

Предложенный подход предоставляет прямой доступ к значениям фактических параметров, что даёт выигрыш в скорости до 30 % по сравнению с прямым поиском в контексте, реализованном в предыдущих версиях библиотеки.

7. Литература

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

[2] Официальный сайт проекта InteLib. — `http://www.intelib.org`.

[3] *А. В. Столяров.* Импорт вычислительной модели языка Scheme в объектно-ориентированное окружение // Сборник статей молодых учёных факультета ВМК МГУ — 2008 г. — № 5 — стр. 119–130.

[4] *B. Stroustrup.* The C++ Programming Language (Special ed.). — Addison-Wesley, Reading, MA, 2000.

[5] *G. L. Steele.* Common Lisp the Language, 2nd edition. — Digital Press, 1990.

[6] *R. K. Dybvig.* The Scheme Programming Language. — Prentice Hall, New York, 1996.

[7] *V. Turchin.* REFAL-5 Programming Guide and Reference Manual. — The City College of New York, New England Publishing Co., Holyoke, 1989.

[8] *I. Bratko.* Prolog Programming for Artificial Intelligence, 3rd edition. — Pearson Education / Addison-Wesley, 2001.

[9] *J. McCarthy.* Recursive Functions of Symbolic Expressions and Their Computation by Machine // Communications of the ACM — April 1960 — № 3(4) — pp. 184–195..

[10] *Jeff Alger.* C++ for real programmers. — Academic Press, Orlando, FL, 2000.

[11] *Andrey V. Stolyarov.* A framework of heterogenous dynamic data structures for object-oriented environment: the S" expression model // V. Stefanuk and Kenji Kaijiri, eds., Knowledge-Based Software Engineering. Proceedings of the 6th JCKBSE — vol. 108 of Frontiers in Artificial Intelligence and Applications — pp. 75–82. IOS Press, Protvino, Russia, August 2004.

[12] *Э. Хювёнен, И. Сеппянен.* Мир Лиспа. В 2-х т. — М.: Мир, 1990.