Московский государственный университет имени М.В. Ломоносова
Факультет вычислительной математики и кибернетики
Кафедра алгоритмических языков

Галкина Екатерина Владимировна

О расширении возможностей одной абстрактной Лисп-машины

ВЫПУСКНАЯ КВАЛИФИКАЦИОННАЯ РАБОТА

Научный руководитель:
к.ф.-м.н., доцент
А. В. Столяров

Москва, 2016

Аннотация

Работа посвящена расширению функциональности вычислительной модели диалектов языка Лисп, представленной в библиотеке InteLib. В ходе работы в имеющиеся диалекты InteLib Lisp и Intelib Scheme была добавлена возможность использования многозначных функций, реализован механизм динамических нелокальных переходов. Предложен новый подход к созданию лексического и динамического связывания переменных.

Оглавление

1 Введение

1.1 Методы многостилевого программирования

Качество программного продукта и эффективность работы программиста во многом зависят от выбранного языка программирования, а именно от того, насколько адекватны изобразительные средства выбранного языка по отношению к конкретной решаемой задаче. При решении частных подзадач может оказаться удобным использование средств языков программирования, отличных от основного.

Мультипарадигмальное программирование позволяет использовать различные выразительные средства для решения частных подзадач в рамках одного проекта. Каждая из основных парадигм имеет свою сферу применения. Например, для символьной обработки текста наиболее удобны функциональные языки, для решения переборных задач — логические, для создания интерактивных приложений — императивные. Кроме того, во многих проектах необходимо взаимодействовать с базами данных, запросы к которым декларативны. Использование разных парадигм при решении соответствующих подзадач позволяет в целом повысить качество программы и упростить процесс её создания, но может быть невозможным в рамках базового языка или повлечь за собой слишком высокие накладные расходы. Таким образом, возникает необходимость в использовании нескольких языков программирования, которым соответствуют различные парадигмы, в рамках одного проекта.

При интеграции разнородных языков неизбежно возникают технические трудности. В статье [1] приведены следующие подходы к решению проблемы многостилевого программирования и описаны их основные преимущества и недостатки:

- применение нескольких систем программирования;
- создание нового языка;
- расширение существующего языка;

- непосредственная интеграция.

Метод непосредственной интеграции является частным случаем расширения существующего языка программирования. В его основе лежит моделирование средствами заранее выбранного базового языка синтаксиса и семантики альтернативных языков.

В статье [1] в качестве базового языка предлагается использовать объектно-ориентированный язык программирования, а также приводятся требования, которым должен удовлетворять этот язык. Во-первых, для построения модели альтернативного языка необходимо наличие в базовом языке понятия класса, поддерживающего инкапсуляцию, наследование и полиморфизм. Во-вторых, для синтаксически схожего представления конструкций альтернативного языка базовый язык должен предоставлять возможность переопределения стандартных операций. Кроме того, этот язык должен быть достаточно известным, иметь библиотеки для различных областей и реализации на нескольких платформах, что делает допустимым его использование в большинстве проектов.

В библиотеке InteLib [1, 2] реализован метод непосредственной интеграции [3]. В качестве базового языка используется объектно-ориентированный язык C++ [4], как отвечающий всем перечисленным выше требованиям. На данный момент в рамках проекта InteLib 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.

1.2 Представление и вычисление S-выражений в библиотеке InteLib

Основной структурой данных в языке Лисп и его диалектах являются S-выражения [9] (s-expressions, symbolic expressions). S-выражение представляет собой либо атом (константа определённого типа, символ и т. п.), либо точечную пару, составленную из S-выражений. Это индуктивное определение можно записать в виде БНФ [13]:

```
<S-выражение> ::= атом | (<S-выражение> . <S-выражение>)
```

По историческим причинам за левым и правым подвыражением в точечной паре закрепились названия CAR и CDR соответственно. Далее будем придерживаться этой терминологии.

Библиотека InteLib предоставляет набор классов языка C++, реализующих S-выражения как гетерогенные структуры данных [3]. Общие свойства, характерные для всех возможных S-выражений, представлены в базовом абстрактном классе `SExpression`.

Атомарные S-выражения разделяются на основные типы: целые константы, константы с плавающей точкой, строковые константы и метки. Символьные константы представляют собой строки единичной длины. Метки однозначно идентифицируются своим адресом и используются для представления уникальных объектов, т. е. S-выражений, существующих в единственном экземпляре. Например, на основе этого типа данных реализован объект пустого списка, булевские значения и символы (в контексте диалектов Лиспа). S-выражения перечисленных типов представлены классами `SExpressionInt`, `SExpressionFloat`, `SExpressionString`, `SExpressionLabel` и `SExpressionChar` соответственно. Все они являются наследниками базового класса `SExpression`.

Точечные пары представлены классом `SExpressionCons`, также наследуемым от `SExpression`. В соответствии с определением S-выражения, поля класса являются ссылками на два других S-выражения: CAR и CDR исходного.

Для удобства использования S-выражений был введён класс `SReference`. Объект этого класса представляет собой «умный указатель» [14] на объект класса `SExpression`, обладающий дополнительной функциональностью (подсчёт ссылок на объект, интерфейс для работы с S-выражениями и т. п.). Классы `LReference` и `SchReference` являются его наследниками и отражают различия в представлении и вычислении S-выражений в рамках диалекта InteLib Lisp и InteLib Scheme соответственно.

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

Подробное описание работы с S-выражениями, методов класса `SReference` и дополнительных структур данных в библиотеке InteLib приведено в работе [10].

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

Особенность континуаций в языке Scheme состоит в том, что они являются полноправными (*first-class*) объектами, т. е. могут быть переданы как аргументы, возвращены из функции или сохранены в какой-либо структуре

данных. Это свойство может использоваться в том числе при реализации сопрограмм, поиска с возвратом и многозадачности. Примеры такой работы с континуацией в языке Scheme приведены в статье [15].

В библиотеке InteLib континуацию можно рассматривать как виртуальную вычислительную машину, а её реализация вынесена в отдельный класс `IntelibContinuation`. Именно методами этого класса реализуется основной механизм вычислений S-выражений. Различия между диалектами InteLib Lisp и InteLib Scheme (см. [3], стр. 128–129), очевидно, влекут за собой и различия в реализации вычислительных моделей языков. Эти особенности учитываются в классах `LispContinuation` и `SchemeContinuation`, наследуемых от `IntelibContinuation`.

Основой реализации класса `IntelibContinuation` являются стек действий, стек результатов и стек фреймов. Работа первых двух из них описана в статье [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. Перед вызовом функции в стек помещается новый фрейм, который автоматически удаляется, когда в стеке отложенных действий кончаются команды, добавленные туда после его создания.

Работа с объектом `IntelibContinuation` начинается с помещения в стек действий команды `just_evaluate`, а параметром является подлежащее вычислению S-выражение. Эти действия реализованы в методе `Evaluate` классов `LReference` и `SchReference`, в котором и создаётся континуация, соответствующая используемому диалекту.

Все дальнейшие вычисления происходят пошагово. Поместив команду для вычисления выражения в стек действий, метод `Evaluate` вызывает метод континуации `Step`, производящий один шаг вычислений. На каждом таком шаге из стека действий извлекается очередная команда, а дальнейшая работа вычислительной модели зависит от её интерпретации. Например, в стек действий могут быть добавлены новые инструкции, в стек результатов — новые значения. Если очередной командой оказалась команда вызова функции от заданного числа аргументов (далее будем обозначать её `CALL/n`, где `n` — количество аргументов), то из стека результатов будут извлечены сначала `n` значений параметров функции, а затем сама функция. Из полученных объектов формируется вызов функции, а дальнейшие действия зависят от её семантики.

Таким образом, каждый шаг вычислений в общем случае приводит к изменению стеков, т. е. происходит переход к другой континуации. Вычисление

S-выражения считается завершённым, когда в стеке действий не остаётся команд для выполнения.

1.3 Постановка задачи

Целью работы является расширение функциональности вычислительной модели диалектов языка Лисп, реализованной в библиотеке InteLib.

Для достижения поставленной цели необходимо решить следующие задачи:

1. реализовать в рамках имеющихся диалектов средства для осуществления динамических нелокальных переходов (`CATCH`, `THROW` и `UNWIND-PROTECT` для диалекта InteLib Lisp и `DYNAMIC-WIND` для диалекта InteLib Scheme);

2. разработать и реализовать механизмы лексического и динамического связывания, исключающие линейный поиск;

3. реализовать в рамках имеющихся диалектов функции и формы, позволяющие использовать возврат нескольких значений (`VALUES`, `VALUES-LIST` и `MULTIPLE-VALUE-CALL` для диалекта InteLib Lisp и `VALUES` и `CALL-WITH-VALUES` для InteLib Scheme).

2 Нелокальные переходы и исключения

2.1 Механизм исключений в языке Lisp

В языке Lisp механизм исключений реализован в виде форм `CATCH` и `THROW`. Их вызовы выглядят следующим образом [5]:

`(CATCH` *tag form ...*`)`
`(THROW` *tag result*`)`

Параметр *tag* — выражение, результат вычисления которого может быть произвольным объектом, который будет использован в качестве метки формы `CATCH`. Остальными параметрами `CATCH` являются формы, которые вычисляются по очереди, и в качестве результата возвращается результат последней из них. Если же во время периода выполнения одной из этих форм была вызвана функция `THROW` с соответствующей меткой, то вычисление прерывается на этой форме, и результатом работы `CATCH` будет значение параметра *result*.

При вызове формы `THROW` управление передаётся в точку вызова ближайшей формы `CATCH` с соответствующей меткой. Если метка не найдена, вырабатывается ошибка. Сравнение объектов-меток производится в соответствии с семантикой предиката `EQ` [11].

```
>>> (catch 'tag (print 1) (print 2) (throw 'tag 'exception) (print 3))
1
2
EXCEPTION
>>> (catch 'tag (throw 'another-tag 0))
Error: no catcher for tag: ANOTHER-TAG
```

Спецформа UNWIND-PROTECT позволяет контролировать развёртку стека после завершения вычисления формы-аргумента [5]:

(UNWIND-PROTECT *protected-form cleanup-forms ...*)

При вызове этой формы сначала вычисляется выражение *protected-form*, и затем, вне зависимости от способа его завершения, будут вычислены выражения *cleanup-forms*. UNWIND-PROTECT гарантирует выполнение форм *cleanup-forms* перед выходом, как в случае нормального выхода, так и в случае генерации исключения. В качестве результата вычисления возвращается результат выполнения защищённой формы *protected-form*. Все результаты выполнения форм чистки *cleanup-forms* игнорируются. Покажем это на примере:

```
>>> (setq x 0)
>>> (unwind-protect x (setq x (+ x 1)))
0
>>> x
1
>>> (catch 'tag (unwind-protect (throw 'tag 'exception) (setq x (+ x 1))))
EXCEPTION
>>> x
2
```

2.2 Нелокальные динамические переходы в языке Scheme

В языке Scheme нелокальные динамические переходы реализуются функцией DYNAMIC-WIND, которая принимает три аргумента, каждый из которых в свою очередь является функцией без аргументов [12]:

(DYNAMIC-WIND *before thunk after*)

Возвращаемым значением является результат вызова *thunk*. Вызовы функций *before* и *after* происходят согласно следующим правилам: *before* вызывается перед началом периода выполнения *thunk*, *after* — сразу после него. Из-за наличия в языке Scheme функции CALL-WITH-CURRENT-CONTINUATION (далее — CALL/CC), которая позволяет «законсервировать» текущую континуацию, этот период может быть не целостным временны́м промежутком. В работе [12] рассматривается несколько вариантов. Началом периода вычисления функции считается

- начало вычисления тела вызываемой функции;

- обращение к континуации, сохранённой (с использованием `CALL/CC`) во время жизни данной функции, если текущие вычисления ведутся вне его.

Этот период заканчивается

- при возврате из функции;
- при обращении к континуации, сохранённой вне периода существования данной функции, если текущие вычисления ведутся во время её вызова.

При обращении к континуации, сохранённой во время «вложенных» вызовов `DYNAMIC-WIND` (т. е. функция использовалась повторно в теле *thunk*) будут вызваны все функции *before*, от самой внешней к самой внутренней, а после завершения периода вызова — все *after* в обратном порядке.

Если не происходит обращений к сохранённым континуациям, работа функции сводится к тривиальному случаю, а именно к последовательным вызовам трёх функций и возвращению результата вычисления второй из них.

Приведём несколько примеров работы функции `DYNAMIC-WIND`:

```
>>> (define path ())
>>> (define (before) (set!  (append path '(b))))
>>> (define (after) (set! path (append path '(a))))
>>> (define (thunk1) (set! path (append path '(1))))
>>> (dynamic-wind before thunk1 after) ; функции before, thunk1 и after
                                       ; выполнились последовательно
>>> path
(B1 1 A1)
>>> (define cont 0)
>>> (define (thunk2) (call/cc (lambda (c) (set! cont c))))
>>> (dynamic-wind before thunk2 after)
>>> (set! path ())
>>> (cont 'foo)                        ; функции before и after были
FOO                                    ; вызваны при применении
                                       ; сохранённой континуации
>>> path
(B1 A1)
```

2.3 Идея применённого решения

Как было сказано ранее, вычисления в библиотеке InteLib представляют собой изменение текущей континуации. Для реализации нелокальных переходов необходимо иметь возможность указать, куда должен быть совершён переход, т. е. восстановить одно из предыдущих состояний континуации. Кроме того, требуется сохранять дополнительную информацию (метку формы `CATCH`, формы очистки в `UNWIND-PROTECT` и их аналоги для `DYNAMIC-WIND`).

Для этих целей было решено использовать стек фреймов. В структурах, из которых он состоит, сохраняются позиции стека действий и стека результатов на момент вызова функции, которой соответствует фрейм. Для хранения дополнительной информации можно использовать поле `context` и добавлять в стек «псевдофреймы». При этом может возникнуть ситуация, когда при очистке стеков во время выхода из функции будет удалён фрейм с дополнительной информацией вместе с текущим фреймом. Чтобы избежать этого, необходимо дополнительно помечать псевдофреймы и удалять их только после выхода из форм, во время вычисления которых они могут использоваться.

Такой механизм позволяет реализовать все описанные выше формы. Реализация исключений в рамках диалекта InteLib Lisp сводится к сохранению и, при генерации исключения, поиску соответствующей метки. Для корректной работы формы `UNWIND-PROTECT` формы очистки также сохраняются в псевдофрейме, и в случае нелокального выхода происходит их поиск и планируется их вычисление. Кроме того, для реализации функции `DYNAMIC-WIND` необходимо в классе континуации модифицировать метод, обеспечивающий замену континуации.

2.4 Необходимые изменения в библиотеке

В имеющуюся реализацию класса континуации были внесены изменения, необходимые для создания общих механизмов, используемых для реализации динамических нелокальных переходов в рамках диалектов InteLib Lisp и InteLib Scheme.

Во-первых, в структуру, представляющую элемент стека фреймов, было добавлено поле, позволяющее отличить псевдофреймы от всех остальных. Очистка стека происходит в методе `DropExhaustedFrames`. В него была добавлена проверка на тип фрейма, а также булевский флаг в качестве аргумента, позволяющий удалить все использованные фреймы (независимо от их типа) при выходе из форм, во время вычисления которых могли происходить нелокальные переходы.

Во-вторых, в класс `IntelibContinuation` были добавлены следующие методы:

```
void InsertPseudoFrame(SReference data);
int LocatePseudoFrame(bool up, int begin) const;
void UnwindStack(SReference tag, SReference res);
void HandleLispCatch();
```

В методе `InsertPseudoFrame` создаётся новый псевдофрейм, его полю `context` присваивается аргумент, а остальные поля инициализируются так

же, как и при создании фреймов перед вызовами функций. Полученная структура помещается в стек следующим образом: если стек пуст или на его вершине находится другой псевдофрейм, то новый объект помещается на вершину стека; во всех других случаях он располагается под текущим фреймом, чтобы не потерять информацию о переданных в функцию аргументах.

Метод `LocatePseudoFrame` является приватным методом класса и производит поиск псевдофреймов. Аргументы задают направление и начало просмотра стека. Метод возвращает позицию первого найденного псевдофрейма или `-1`, если таких при просмотре на нашлось.

Следующие два метода используются для реализации механизма исключений в рамках диалекта InteLib Lisp.

Метод `UnwindStack` используется при генерации исключения. Его аргументами являются метка формы `CATCH` и выражение, возвращаемое функцией `THROW`. В методе происходит поиск псевдофреймов, пока не будет найдена соответствующая метка. Помимо этого сохраняются все формы очистки, если они встретились во время поиска. Псевдофрейм с меткой содержит информацию о состоянии континуации на момент вызова соответствующей формы `CATCH`, в том числе позиции стеков действий и результатов, которые устанавливаются как текущие. После этого планируется вычисления форм очистки и возвращение в качестве результата выражения, указанного в функции `THROW`.

Форма `CATCH` не вычисляет свои аргументы, поэтому для получения значения метки необходимо это значение вычислить вручную, и только после этого сохранить. Для этих целей был введён метод континуации `HandleLispCatch`. К моменту его вызова стек результатов содержит вычисленное значение метки и список форм из вызова `CATCH`. Работа метода сводится к извлечению этих значений из стека, добавлению псевдофрейма с меткой и планированию вычисления форм.

Кроме того был модифицирован метод `ReplaceContinuation`, применяющийся при вызове ранее сохранённой континуации в диалекте InteLib Scheme. Внесённые в него изменения будут рассмотрены при описании реализации функции `DYNAMIC-WIND`.

2.5 Реализация механизма исключений для InteLib Lisp

Сначала рассмотрим реализацию формы `UNWIND-PROTECT`. Простейший вариант её работы состоит в последовательном вычислении нескольких форм с возвращением значения первой из них. Эти действия планируются в стеке действий с использованием команд `evaluate`, `evaluate_progn` и `drop_result`, описанных в статье [3]. Конфигурация стека, приводящая к та-

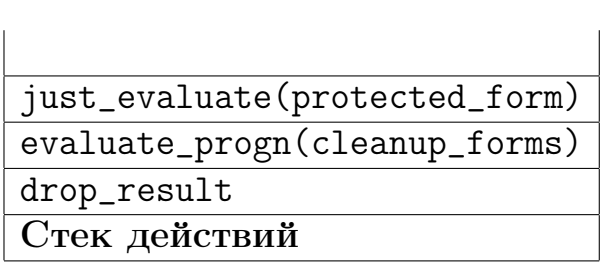

Рис. 2.1: Содержимое стека после вызова UNWIND-PROTECT

ким вычислениям, приведена на рис. 2.1. Охранные выражения также должны вычисляться при генерации исключения во время вычисления защищённой формы, если форма `CATCH` с соответствующей меткой находится в объемлющем блоке.

Подобный механизм используется и для реализации формы `CATCH`. Её метка также должна сохраняться в псевдофрейме для дальнейшего использования. Согласно семантике формы, сначала необходимо произвести вычисление этой метки, и только после её сохранения возможно планировать вычисление остальных аргументов.

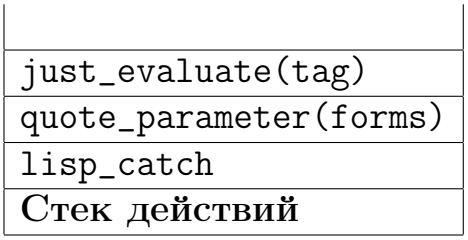

Рис. 2.2: Содержимое стека после вызова CATCH

Стек действий заполняется таким образом, чтобы в ходе дальнейшей работы сначала вычислялось значение метки, затем в стек результатов заносились остальные аргументы в исходном виде, а после вызывалась новая команда континуации `lisp_catch`. Исходная конфигурация стека действий, приводящая к такой последовательности вычислений, указана на рис. 2.2. Когда на очередном шаге работы в стеке действий встречается эта команда, вызывается метод континуации `HandleLispCatch`. При вызове этого метода из стека результатов извлекаются ещё не вычисленные аргументы формы `CATCH` и результат вычисления метки. Этот результат сохраняется в новом псевдофрейме. Далее планируется вычисление остальных параметров формы `CATCH`. Помимо этого во фрейме сохраняются позиции вершин стеков на момент вызова, следовательно, по метке можно определить ту конфигурацию стеков, к которой необходимо вернуться при генерации исключения. Если исключение не будет вызвано при вычислении аргументов формы `CATCH`, этот фрейм будет удалён после выхода из формы.

Сохранённые при вызове форм `CATCH` и `UNWIND-PROTECT` выражения используются только при развёртке стека, а именно при генерации исключения функцией `THROW`. Семантика этой функции подразумевает просмотр и изменение стеков, поэтому для её реализации был введён метод континуации `UnwindStack`. В этот метод в качестве аргументов передаются параметры функции `THROW`: метка и результирующее значение.

При вызове функции `THROW` производится проход по стеку фреймов начиная с вершины. Во время прохода осуществляется поиск соответствующей метки. Если метка не найдена, генерируется сообщение об ошибке. Позиция найденной метки сохраняется. Также из всех встреченных псевдофреймов сохраняются охранные выражения, которые могли быть туда добавлены, если исключение было сгенерировано во время вычисления аргумента формы `UNWIND-PROTECT`, а соответствующая форма `CATCH` является частью объемлющего блока.

Далее происходит модификация стеков. В качестве новых позиций вершин стека действий и стека результатов устанавливаются позиции, сохранённые в найденном фрейме с соответствующей меткой. При этом все элементы стеков, находящиеся выше, автоматически уничтожаются. Таким образом, континуация переходит в такое же состояние, которое было во время момента вызова формы `CATCH`. Далее в стек действий добавляются инструкции для вычисления охранных выражений, удаления результатов их вычисления и сохранения результирующего значения, переданного из формы `THROW`, в качестве последнего результата.

После выполнения этих команд в континуации происходит автоматическая очистка стека фреймов, в ходе чего удаляется вся информация, сохранённая ранее для осуществления нелокального перехода.

2.6 Реализация нелокальных динамических переходов

Функция `DYNAMIC-WIND` должна иметь возможность сохранить свои аргументы *before* и *after* для дальнейшего использования. Они могут быть повторно вызваны, только если текущая континуация была сохранена с помощью `CALL/CC` и в дальнейшем применена в качестве функции. Таким образом, для хранения *before* и *after* следует использовать структуру данных в самой континуации. Из-за вложенных вызовов `DYNAMIC-WIND` возникает потребность сохранять несколько элементов в определённой последовательности. Для этого методу `InsertPseudoFrame` в качестве аргумента передаётся точечная пара, составленная из *before* и *after*.

`CALL/0`	
`drop_result`	
`CALL/0`	
`quote_parameter(after)`	
`CALL/0`	*before*
`drop_result`	*thunk*
Стек действий	**Стек результатов**

Рис. 2.3: Содержимое стеков после вызова DYNAMIC-WIND

Далее происходит заполнение стека действий и стека результатов. В итоге они принимают вид, показанный на рис. 2.3. Команды `drop_result` и `quote_parameter` описаны в работе [3]. Нетрудно видеть, что полученная конфигурация задаёт нужную последовательность вычислений (вызов трёх функций и возврат результата второй из них).

Замену континуации при вызове функции `CALL/CC` осуществляет метод `ReplaceContinuation`, который удаляет все три стека текущей континуации, заменяя их на соответствующие новые. Модификация метода состоит в следующем: замена стека фреймов происходит в последнюю очередь, а перед этим оба стека проверяются на наличие псевдофреймов. Дальнейшие вычисления должны происходить так, чтобы сначала были вызваны все функции *after* из текущей континуации от внутренней к внешней, затем — все *before* из новой континуации от внешней к внутренней. Для этого просмотр стеков происходит таким образом: поиск в новом стеке фреймов ведётся сверху вниз, начиная с вершины; после него просматривается стек текущей континуации в обратном порядке. Поиск производит метод `LocatePseudoFrame`. Из каждого найденного псевдофрейма изымаются сохранённые в нём функции (соответственно *before* и *after*), а в стек действий последовательно помещаются команды `drop_result`, `CALL/0` и `quote_parameter`. Приведённый ниже фрагмент кода поясняет на примере поиска в текущем фреймовом стеке, как именно происходит эта процедура:

```
int located;
//Go "down" the new frame stack and get all "befores"
int begin = other.frame_stack_pointer - 1;
while ((located = other.LocatePseudoFrame(false, begin)) >= 0) {
    PushTodo(drop_result);
    PushTodo(0);
    PushTodo(quote_parameter,
            (other.frame_stack[located].context).Car());
    begin = located - 1;
}
```

```
//Now go "up" the old frame stack and get all "afters"
begin = 0;
while ((located = LocatePseudoFrame(true, begin)) >= 0) {
    PushTodo(drop_result);
    PushTodo(0);
    PushTodo(quote_parameter, (frame_stack[located].context).Cdr());
    begin = located + 1;
}
```

3 Механизм лексического связывания

В предыдущих версиях библиотеки InteLib лексическое связывание реализовывалось с помощью контекстов — однонаправленных списков соответствий между символом и его текущим значением. Для вычисления значения символа производился линейный 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. Основным недостатком такого решения является низкая скорость поиска нужного значения в контексте.

3.1 Идея применённого решения

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.

В диалекте InteLib Lisp присутствуют символы с динамическим связыванием (динамические переменные), которые игнорируются при замене и обрабатываются отдельно с использованием специально разработанных механизмов.

3.2 Используемые классы и основы их реализации

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 (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 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).

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.

3.3 Создание лексического замыкания

В диалекте InteLib Lisp лексическое замыкание создаётся либо при определении пользовательской функции с помощью спецформы `DEFUN`, либо при применении формы `FUNCTION` (или сокращённо `#'`), реализующей функциональную блокировку [11], к λ-выражению. Заметим, что сами λ-выражения не являются функциональными объектами и рассматриваются как обычные списки, то есть вычисление на верхнем уровне выражения вида `(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 классом `SExpressionCompiledLambda`, 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. § 4). 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.

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.

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.

3.4 Обработка LET-форм

LET-формы предназначены для создания локальных связей переменных внутри других форм и на самом деле представляют собой синтаксически видоизменённые λ-выражения [11].

Разницу в вычислении форм `LET` и `LET*` 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)
```

В обоих случаях формы инициализации вычисляются последовательно, но в форме `LET` они связываются с символами одновременно сразу после вы-

числения всех их значений, в то время как переменные в форме LET* связываются последовательно после вычисления каждой формы.

Как и в случае лексического замыкания, LET-формы обрабатываются одинаково в обоих диалектах. Обработанная LET-форма представляет собой объект класса SExpressionCompiledLet, 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 работы с динамическими переменными (см. § 4). Создание объекта обработанной LET-формы реализовано в методе LetProcessing класса IntelibTokenReplace.

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 коли-

чеством (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`.

4 Динамическое связывание переменных

4.1 Виды связывания в языке Lisp

В языке Lisp переменные по умолчанию считаются статическими, однако есть возможность использования динамических переменных. Для их объявления предназначена форма `DEFVAR` [5]:

`(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. Статическая переменная видна только в теле той формы, в которой была объявлена (если она не является глобальной). Динамическая переменная, даже связанная локально (например, в LET-форме), видна во всех вложенных вызовах функций. После выхода из формы, в которой локально устанавливается новое значение динамической переменной, восстанавливается предыдущее связывание, если оно было. Таким образом, вычисления будут происходить по-разному для статических и динамических переменных, в случае, если переменная свободна в вычисляемом выражении, но при этом является формальным параметром в более внешней форме [11]. 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.

4.2 Реализация динамического связывания для InteLib Lisp

В рассматриваемой реализации для работы с динамическими переменными введён класс `IntelibBodyPrologue`. 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 `proceed_prologue`, 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. Описанные действия реализованы в методе `ProceedPrologue` класса `IntelibContinuation`.

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.

5 Многозначные функции

5.1 Многозначные функции в языке Lisp

Обычно в ходе вычисления выражений на языке Lisp из всех функций возвращается одно значение. Для того чтобы производить и принимать несколько значений сразу, необходимо использовать специально предназначенные для этого формы [5].

Для осуществления возможности передачи нескольких значений в функцию предназначена спецформа `MULTIPLE-VALUE-CALL`:

`(MULTIPLE-VALUE-CALL` *function form ...*`)`

Аргумент *function* вычисляется в функциональный объект, который необходимо применить к нескольким значениям. Эти значения генерируются при вычислении всех последующих аргументов, причём из каждого из них может быть возвращено произвольное количество значений при помощи функции `VALUES`:

`(VALUES` *arg ...*`)`

Затем все сгенерированные значения собираются и передаются в качестве аргументов полученному функциональному объекту. Результат этого вычисления возвращается в качестве результата вызова всей формы `MULTIPLE-VALUE-CALL`.

Если функция `VALUES` вызывается отдельно, то результатом вычисления является только первый аргумент.

Также для возврата нескольких значений может быть применена функция `VALUES-LIST`:

`(VALUES-LIST` *list*`)`

Её можно выразить через функцию `VALUES`:

`(VALUES-LIST` *list*`) == (APPLY #'VALUES` *list*`)`

Рассмотрим работу этих функций на примере:

```
>>> (multiple-value-call #'+ (values 1 2) (values-list '(3 4)))
10
>>> (* (values 3) (values 5 0))
15
```

5.2 Передача нескольких значений в континуацию в языке Scheme

Согласно [12], любая континуация в языке Scheme может принять в точности одно значение. Исключением являются континуации, созданные с помощью функции `CALL-WITH-VALUES`:

`(CALL-WITH-VALUES` *producer consumer*`)`

Её аргументы также являются функциями: *producer* — от нуля аргументов, *consumer* — от произвольного числа аргументов. Во время работы `CALL-WITH-VALUES` вызывает свой первый аргумент и создаёт континуацию, которая, получив одно или несколько значений, вызывает второй аргумент (*consumer*) с этими значениями.

Для передачи значений в континуацию используется функция `VALUES`:

`(VALUES` *obj ...*`)`

Приняв произвольное число аргументов, функция возвращает максимально возможное количество значений в континуацию. Заметим, что `VALUES` может быть вызвана и вовсе без аргументов, при этом результат такого вызова в [12] не оговаривается.

Приведём пример работы этих функций:

```
>>> (call-with-values (lambda () (values 4 5)) (lambda (a b) (+ a b)))
9
>>> ((lambda (a b) (+ a b)) (values 4 5))
Error: too few parameters
```

5.3 Идея применённого решения

Для реализации возврата нескольких значений в рамках обоих диалектов необходимо иметь возможность указывать и при дальнейших вычислениях определять, что текущая континуация может их принять. Кроме того, при использовании спецформы `MULTIPLE-VALUE-CALL` эти значения генерируются в нескольких формах, вычисляемых отдельно, и их суммарное количество заранее неизвестно, но при передачи их в функцию необходимо явно указать итоговое число аргументов.

Поскольку функция `CALL-WITH-VALUES` отличается лишь тем, что её аргументы уже вычислены, а значения генерируются только одной формой, для реализации возврата многих значений в обоих диалектах используются одинаковые механизмы, основанные на возможностях континуации.

Во-первых, чтобы обозначить, что текущая континуация принимает несколько значений, была введена новая инструкция, которая помещается в стек действий при вызове форм `CALL-WITH-VALUES` и `MULTIPLE-VALUE-CALL`. Аргументом этой команды является целое число — количество уже вычисленных значений, изначально равное нулю.

Во-вторых, реализация функций, генерирующих значения, вынесена в метод континуации. Опишем общую схему работы этого метода. В качестве параметров он принимает вектор аргументов и их количество. Эти аргументы должны быть возвращены в континуацию, если она может принять несколько значений. Для проверки этого свойства из стека действий достаётся верхняя инструкция. Если континуация принимает только одно значение, возвращается первый элемент из вектора аргументов. Если континуация принимает несколько значений, происходит процесс вычисления всех аргументов и сохранения их общего числа, используя для этого новую команду. Подробнее этот процесс будет рассмотрен при описании рабочей реализации. Когда в стеке действий останется только инструкция для применения функции к полученным значениям, их реальное количество уже будет известно: оно сохранено как параметр инструкции, извлечённой до этого. Таким образом, можно применить функцию к полученным аргументам, поместив в стек действий соответствующую инструкцию `CALL/n`.

5.4 Рабочая реализация

В вызове `MULTIPLE-VALUE-CALL` формы, генерирующие значения, вообще могут отсутствовать, тогда функция *consumer* будет вызвана без аргументов. У функции `CALL-WITH-VALUES` все аргументы обязательны, то есть как минимум одно значение будет сгенерировано при вызове аргумента *producer*.

Поэтому при вычислении этих форм первыми в стек действий помещаются команды, реализующие такое поведение: `CALL/0` и `CALL/1` соответственно. Далее стеки заполняются таким образом, чтобы при генерации значений все они могли быть помещены в стек результатов в качестве будущих аргументов функции *consumer*.

Нетрудно заметить, что для корректности вышеописанных действий необходимо, чтобы функция `VALUES` вызывалась хотя бы с одним аргументом (список-аргумент `VALUES-LIST` был не пуст). Доопределим работу данной функции при вызове без аргументов возвратом неопределённого значения. Для этого в классе `IntelibContinuation` уже имеется метод `ReturnUnspecified` и соответствующая команда `return_unspecified` [3].

`just_evaluate(`*consumer*`)`
`just_evaluate(`$form_1$`)`
`expand_multivalue(0)`
`just_evaluate(`$form_2$`)`
`expand_multivalue(0)`
...
`just_evaluate(`$form_n$`)`
`expand_multivalue(0)`
`CAll/0`
Стек действий

Рис. 5.1: Содержимое стека после вызова MULTIPLE-VALUE-CALL

`CALL/0`	
`expand_multivalue(0)`	*producer*
`CALL/1`	*consumer*
Стек действий	**Стек результатов**

Рис. 5.2: Содержимое стеков после вызова CALL-WITH-VALUES

Для реализации описанного подхода в класс континуации была добавлена команда `expand_multivalue`, используемая для агрегации количества значений. Она помещается в стек действий с нулевым параметром перед каждой командой для вычисления форм, генерирующих значения. После этого туда добавляется команда для вычисления функционального объекта, которому эти значения будут переданы. В случае функции `CALL-WITH-VALUES` этот объект уже вычислен, поэтому сразу помещается

в стек результатов для дальнейшего использования. Ещё одно отличие состоит в том, что значения генерируются не при вычислении произвольных форм, а при вызове функционального аргумента *producer*, не имеющего параметров. Для совершения этого вызова в стек помещается команда `CALL/0`. Таким образом, стеки будут иметь конфигурацию, показанную на рис. 5.1 и 5.2, для вызовов `(MULTIPLE-VALUE-CALL` *consumer* $form_1$ ... $form_n$`)` и `(CALL-WITH-VALUES` *producer consumer*`)` соответственно.

В обоих диалектах функции, генерирующие значения, реализованы одинаково: вектор их аргументов передаётся в метод континуации `MultipleValuesReturn` (в случае функции `VALUES-LIST` этот вектор предварительно создаётся из списка).

В случае, когда значения генерируются несколькими формами, а континуация может принять все полученные значения, дальнейшая работа метода зависит от последующих инструкций. Во время вычисления очередной формы на вершине стека находится инструкция `expand_multivalue`. Её параметр представляет собой целое число, равное количеству сгенерированных на данный момент значений (при вычислении самой первой формы оно равно нулю). К этому значению прибавляется количество новых значений, передаваемое в качестве аргумента методу `MultipleValuesReturn`, а сами значения помещаются в стек результатов. Далее, согласно рис. 5.1, возможны два варианта: либо в стеке находятся инструкции `just_evaluate` и `expand_multivalue` для вычисления следующей генерирующей формы, либо следующая инструкция отвечает за вызов функции (`CALL/0` или `CALL/1`). Если конфигурация стека отличается от упомянутых выше, то вырабатывается исключение.

Рассмотрим дальнейшую работу метода в первом случае, т. е. когда вычисление всех форм, генерирующих значения, ещё не закончено. Текущее количество уже имеющихся в стеке значений на данный момент известно, и параметр ближайшей к вершине стека действий команды `expand_multivalue` заменяется на это число, благодаря чему при получении значений, генерируемых следующей формой, их суммарное количество может быть снова обновлено.

Процесс продолжается, пока в стеке действий сразу после команды `expand_multivalue` не встретится команда `CALL/0` (или `CALL/1` при вычислении в рамках диалекта InteLib Scheme). Такая ситуация означает, что генерация всех значений завершена, их общее количество можно представить как сумму аргумента метода `MultipleValuesReturn` и параметра текущей инструкции, а все значения помещены в стек результатов. Таким образом, все данные, необходимые для вызова функции, уже находятся в стеке результатов. Для того, чтобы корректно осуществить этот вызов, достаточно лишь поменять команду на вершине стека действий, указав полученное суммарное количество параметров.

Заметим, что в диалекте InteLib Lisp в аргументах формы `MULTIPLE-VALUE-CALL`, генерирующих значения, для корректной работы в рамках описанной реализации всегда должны использоваться функции `VALUES` или `VALUES-LIST`, даже если возвращается только одно.

Реализация метода `MultipleValuesReturn` позволяет рассматривать механизм многозначного возврата в InteLib Scheme как частный случай этого механизма в InteLib Lisp. Когда из стека действий извлекается очередная инструкция, её код сравнивается с кодом `expand_multivalue`. Если эти коды совпадают, то текущая континуация может принять несколько значений, т. е. функция `VALUES` вызывается из первого аргумента функции `CALL-WITH-VALUES` (*producer*). Далее команда `CALL/1` заменяется на команду `CALL/n`, где `n` — количество аргументов, переданных в метод `MultipleValuesReturn`. Если извлечённая команда не совпадает с `expand_multivalue`, то это означает, что вычисления происходят в рамках континуации, принимающей только одно значение. Тогда в стек действий возвращается извлечённая команда, а в стек результатов помещается только первый аргумент. Таким образом, в диалекте InteLib Scheme семантика функций для передачи нескольких значений совпадает с описанной в [12].

6 Заключение

В рамках работы получены следующие основные результаты:

1. Диалекты InteLib Lisp и InteLib Scheme были снабжены функциями и спецформами для осуществления динамических нелокальных переходов.

2. В рамках библиотеки InteLib была предложена новая реализация лексических контекстов, исключающая поиск при вычислении значений переменных.

3. Для имеющихся диалектов были реализованы механизмы возврата нескольких значений и использования многозначных функций.

В качестве дальнейших перспектив работы можно указать реализацию на основе нового механизма лексического связывания других форм для локального связывания переменных (например, `LETREC`, `FLET`, `MACROLET` и `LABELS` в диалекте InteLib Lisp), а также реализацию иерархии числовых типов («numerical tower») [12] и соответствующих функций для работы с различными множествами чисел в диалекте InteLib Scheme.

Литература

[1] И. Г. Головин, А. В. Столяров. Объектно-ориентированный подход к мультипарадигмальному программированию // Вестник МГУ, сер. 15 (ВМиК). 2002. № 1. С. 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. № 5. С. 119–130.

[4] B. Stroustrup. The C++ Programming Language (Special ed.). Reading, MA, USA: Addison-Wesley Professional, 2000. 1030 p.

[5] G. L. Steele. Common Lisp the Language, 2nd edition. Digital Press, 1990. 1029 p.

[6] R. K. Dybvig. The Scheme Programming Language. New York, USA: Prentice Hall, 1996. 272 p.

[7] V. Turchin. REFAL-5 Programming Guide and Reference Manual. Holyoke, MA, USA: New England Publishing Co., 1989.

[8] I. Bratko. Prolog Programming for Artificial Intelligence, 3rd edition. New York, USA: Pearson Education, 2001. 678 p.

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

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

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

[12] R. Kelsey, W. Clinger, J. Rees. (eds.). Revised^5 Report on the Algorithmic Language Scheme // ACM SIGPLAN Notices. 1998. 33. № 9. P. 26–76.

[13] Cf. J. W. Backus. The syntax and semantics of the proposed international algebraic language of the Zuerich ACM-GRAMM conference. Proceedings of the International Conference on Information Processing. Preprint. New York, USA: International Business Machines Corp., 1959. 24 p.

[14] Jeff Alger. C++ for real programmers. Orlando, FL, USA: Academic Press, 1998. 388 p.

[15] D. Ferguson, D. Dwight. Call with Current Continuation Patterns // Pattern Languages of Programs Conference Proceedings. Monticello, IL, USA, 2001.